An Introduction

to C++

Abstract

The aim of the notes is to provide an introduction to the C++ programming language.

Author: Ian D Chivers

Email: ian_d_chivers@yahoo.co.uk, ian.d.chivers@kcl.ac.uk

Version: 9.00

Date: November 2006

Dedication

These notes are dedicated to the people who have implemented the gcc suite of software, to Dinkumware and to Microsoft.

Gcc and g++

The gcc home page is

- http://gcc.gnu.org/

The following comments are taken from their home page.

GCC, the GNU Compiler Collection, includes front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj,...).

We strive to provide regular, high quality releases, which we want to work well on a variety of native and cross targets (including GNU/Linux), and encourage everyone to contribute changes and help testing GCC. Our sources are readily and freely available via CVS and weekly snapshots.

Major decisions about GCC are made by the steering committee, guided by the mission statement.

GCC steering committee members

The steering committee consists of the following members. The best place to reach them is the gcc mailinglist.

> Per Bothner, Joe Buck (Synopsys), David Edelsohn (IBM), Kaveh R. Ghazi, Jeffrey A. Law (Red Hat), Marc Lehmann (Technische Universität Karlsruhe), Jason Merrill (Red Hat), David Miller (Red Hat), Mark Mitchell (CodeSourcery), Toon Moene (Koninklijk Nederlands Meteorologisch Instituut), Gerald Pfeifer (SUSE), Joel Sherrill (OAR Corporation), Jim Wilson (Specifix Inc)

Membership in the steering committee is a personal membership. Affiliations are listed for identification purposes only; steering committee members do not represent their employers or academic institutions.

Dinkumware

The Dinkumware home page is

- http://www.dinkumware.com/

They provide the only standard C++ library at this time.

Microsoft

You've all done a great job!

Table of Contents

1
Overview

‘The first thing we do, let’s kill all the language lawyers.’

Henry VI, part II

Aims

The aims of the chapter are to provide a background to the organisation of the course.

1 Overview

1.1 Aims

The aim of the notes is to provide an introduction to the C++ language. There is a brief coverage of object oriented programming and the standard library. You will also find chapters on IEEE 754, mixed language programming under Windows and one on using the Microsoft Foundation Class library for Windows programming under Developer Studio 6.

C++ can be thought of as having two components, the language itself and its associated library. The key features of the library are:

- I/O support
- strings
- containers or data structures
- algorithms
- numeric support
- internationalisation

This is the language as defined in the November 1997 standard document. Ratification took place the following year. Subsequent years have seen clarifications to the language as implementations mature and people develop greater familiarity with the language as they use it. You will see aspects of this with some of the examples in the notes.

1.2 Assumptions

It is assumed that the reader is familiar with using a computer system, in particular using an editor and working with files.g.

1.3 **Web** resources

The King's web server has copies of these notes.

Also have a look at

- http://www.accu.org/

which is dedicated to C and C++ programmers, but also has coverage of Java and C#.

The ACCU is a non-profit organisation devoted to professionalism at all levels in C, C++ and Java. They reviewed over 2000 books. That address is

- http://www.accu.org/bookreviews/public/index.htm

and the beginners's book home page is

- http://www.accu.org/bookreviews/public/reviews/0hr/beginner_s_c__.htm

I wouldn't classify the Stroustrup book as for beginner's myself though.

Also visit Dinkumware. Their home address is:

- http://www.dinkumware.com/

They provide the only standard C++ library at this time.

Their on-line manuals can be found at

- http://www.dinkumware.com/libraries_ref.html

These are an essential source of technical information about the C and C++ libraries.

1.4 Additional **material**

The following are some of the sources I recommend. It is useful to make the following distinctions:–

- introductory texts on C++
- reference texts on C++
- good programming practice in C++
- introductions to object oriented programming

Don't bother at all with older texts, as they won't address standard C++ in any real way.

- Deitel H.M., Deitel P.J., *C++ How to Program*, Prentice Hall.

  A very good introduction and modern in its approach. A cd is included which has Microsoft Visual C++ 6, Introductory Version with the fourth edition.

  The fifth edition is a major reworking and introduces classes and object oriented programming from chapter three.

- Seed, *An Introduction to Object-Oriented Programming in C++ with Applications in Computer Graphics*, Springer Verlag. 2nd Edition.

  If you are a beginner and wanted to just get one book then this is a very good book. Conventional programming book in style.

- Josuttis N.M., *The C++ Standard Library: A Tutorial and Reference*, Addison Wesley.

  Very comprehensive coverage of the library. If you are going to program in C++ then a book like this is essential.

- Stroustrup, *The C++ Programming Language*, Addison Wesley.

  Reference text. The designer of the language, and as with many programming languages it is essential to get a feel for the language by someone who has had a major influence on what the language is today. Don't bother with any edition other than the third. This is a massive improvement over the other two editions. I quite like it but someone has pointed out that it is aimed at so called *language lawyers*. The chapters on design and development and design of libraries are essential reading if you are going to be involved in larger scale projects. If you are going to use C++ seriously then you **must** get hold of this book.

- Scott Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, Addison Wesley, ISBN: 0321334876

  Here is an Amazon review.

  I'd recommend this book (and the subsequent "More Effective C++) to any serious C++ developer. It gives sage guidence in the most common areas in which novice C++ programmers make mistakes. If I were a C++ project manager, I would ensure all of my developers read this book at least once. Meyer's style isn't everyone's cup of tea - sometimes amusing, sometimes tedious; but you simply cannot fault the man on his mastery of C++ Whilst this book would almost certainly propel a C++ novice's code up to the next level of quality, experienced developers should know this stuff. However,

I've personally worked with several 'experienced' (> 2 years) C++ developers who admitted that they had more than one 'light-bulb moment' after reading Meyer's books. This is a testament to Meyer's readable style and ability to explain C++'s more esoteric elements (time for a coffee break, perhaps?). The book works well as a reference guide when developers have that "I know there's a better way to do this..." feeling. Overall, an excellent buy - well worth the money.

- Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison Wesley, 020163371X

  Here is an Amazon review.

  Scott Meyer's book tackles some of the more problematic areas of the language discussing do's and dont's. This is achieved to some part by offering an insight into the C++ compiler and how it processes code. The books goes way beyond this because it covers the most recent changes to the language and how they are supported by the compiler. It even offers help and advice (plus code fragments) in situations when your compiler doesn't yet support the latest C++ enhancement. This book is packed with information on every page and is a must to have for anyone using C++.

  I've got both books as a cd. A lot cheaper than the two books.

- Cline and Lomow, *C++ FAQs.*

  Good programming practice. This is a published book version of the on-line FAQs. It contains roughly 5 times as much material as the on-line version. A lot of *why* questions are answered here. About 30 uk pounds. Slighthly dated.

- Budd, *Data Structures in C++ using the Standard Template Library*, Addison Wesley.

  Approaches data structuring using the Standard Template Library.

- Stroustrup, *C++*, HOPL.

  History. Thoughts on the development of the language. With the benefit of hindsight, but who likes to be wrong!

- Kernighan, *C*, HOPL.

  History. One of the original C team. Again a must. Leave until one has a reasonable experience of C++.

- Barton J.B., Nackman L.R., *Scientific and Engineering C++, An Introduction with Advanced Techniques and Examples*, Addison Wesley.

  Aimed at someone with a Fortran background wanting to learn C++. The edition I have dated 1994 is out of date as some of the solutions they have would be done better using the features as defined in the C++ standard. Hopefully an updated version will become available that uses vectors and valarrays. The comments they make regarding Fortran are a little out of date too. Take up of Fortran 90 has improved in the five years since the publication of this book. Look at the paper by the Computational Science Education Project for a more up to date comparison. Details are given in previous chapter. This paper compares Fortran 77, Fortran 90, C and C++ using five crite-

ria. The criteria are numerical robustness, data parallelism, data abstraction, object oriented programming and functional programming.

- Ladd S.R., *C++ Components and Algorithms: A Comprehensive reference for designing and implementing algorithms in C++*, M&T Books.

  He is very cautious. Old C style used.

- Sedgewick R., *Algorithms in C++*, Addison Wesley.

  A C++ version of his algorithms. Again worth getting hold of to see how best to use C++. This book is not a direct hack from Pascal or Modula. The C++ home page contains a link to the Addison Wesley home page. Old C style.

- The standard. This is currently available electronically on the net.

  I find this useful. If you want a highly technical coverage of the language then this is it. The body of the standard is some 620 pages.

  ftp://ftp.research.att.com/dist/c++std/WP/CD2/

The on-line FAQs on C++ also contain much useful information. I'd get hold of these as they represent a very low cost source. Essentially just your time and effort getting hold of and printing them. I would definitely read the HOPL material at some stage to gain an insight. I would look at the various comp.lang. newsgroups too.

1.5 Compilers and Standards

I've tried to adhere to standard C++ in this course and notes. The standard was finally published in November 1997. The compilers in use currently may not be 100% standard conforming. Some of the examples given in the course may well not compile with your C++ compiler. I would expect problems in the following areas:–

- namespaces;
- the string data type with strings as objects;
- numeric support: complex, valarray, numerics, cmath and cstdlib;
- templates;
- exception handling;
- the standard template library:–
  - containers: vector, list, deque, queue, stack, map, bitset;

I have used a variety of compilers during the development of the notes and examples. These include:–

- DEC VAX running VMS – DEC's compiler
- DEC Alpha running VMS – DEC's compiler
- SUN UltraSparc running Solaris – g++
- SUN UltraSparc running Solaris - Sun's compiler
- Variety of pcs running DOS 6.x and Windows 3.x – Borland 4.5 and Microsoft 1.x and 2.x;
- Variety of pcs running Windows 95, Windows 98 and NT4 – Microsoft C++ Release 5.x, various betas of 6 and the final release of 6, and cygwin and gcc.

- Dell Workstation, running Wndows 2000 Professional and Developer Studio 6, Windows XP Professional and .Net, and various linux distributions (Redhat and Suse) and g++.

If you aren't going to buy Josuttis then I recommend getting hold of the documentation on the standard template library. I have this material up on the King's web server. Some of the examples in these notes will not compile or run with some of the older compilers.

1.5.1 Free and low cost compilers

If you have a pc running Windows then a straightforward option is to install cygwin. Their home address is

- http://www.cygwin.com/

Cygwin is a Linux-like environment for Windows. It consists of two parts:

- A DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux API functionality.
- A collection of tools, which provide Linux look and feel.

The Cygwin DLL works with all non-beta, non release candidate, ix86 32 bit versions of Windows since Windows 95, with the exception of Windows CE.

You will get g++ in a standard installation. It takes up about 150Mb on one sysem I have at home.

Another option is to visit the Microsoft site.

- http://msdn.microsoft.com/visualc/

Visual Studio 2005 express editions are available of a range of Microsoft programming languages including C++.

You can also try Borland.

- http://www.borland.com/products/downloads/download_cbuilder.html

Borland C++ Compiler version 5.5 is available as a free download.

Another option is to install a version of Linux. I've used Redhat and SuSe and can recommend both. It is very easy to install one of these on a pc.

Alternatively you can install in dual boot mode and have both Windows and Linux available.

1.5.2 Contents of the standard

The following provides a brief coverage of the C++ standard:–

1 General, 2 Lexical conventions, 3 Basic Concepts, 4 Standard Conversions, 5 Expressions, 6 Statements, 7 Declarations, 8 Declarators, 9 Classes, 10 Derived Classes, 11 Member Access Control, 12 Special Member Functions, 13 Overloading

14 Templates, 15 Exception Handling, 16 Preprocessing directives, 17 Library Introduction, 18 Language Support Library, 19 Diagnostics Library, 20 General Utilities Library, 21 Strings Library, 22 Localisation Library, 23 Containers Library, 24 Iterators Library, 25 Algorithms Library, 26 Numerics Library, 27 I/O Library, A Grammar Summary, B Implementation Quantities, C Compatibility, D Compatibility Features

The following has the original 1997 standard available:-

- ftp://ftp.research.att.com/dist/c++std/WP/CD2/

If you want to look at it online then I'd recommend the Adobe Acrobat portable document format (pdf). Acrobat readers are available at a number of sites, and there is no charge. The document can be searched using the Acrobat reader. It is also installed in all of the Public Access Workstation Rooms (PAWS) at King's. If you have access to a postscript printer and are going to be working with C++ seriously then I'd think about printing the standard. Most of the texts in print do not address all aspects of the standard, they couldn't possibly. I'd look at the chapters on the various libraries. You need to know what is in the language. Many texts will provide examples based on strings and complex arithmetic. These examples are out of date as they are now supported within the language.

The latest draft is available at

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf

C++ projects under the standard can be found at

- http://www.open-std.org/JTC1/SC22/WG21/docs/projects#14882

1.6 Old and New

You will find both old C style and new standard C++ styles examples throughout the notes. Whilst you should learn and use the new style it is inevitable that you will have to become familiar with both styles in the short term. Firstly because you will have to work with code and examples written in the old style, secondly because you will have to use libraries written and compiled in the old style, thirdly all compilers don't fully support the standard!

The latter part of the notes looks at examples that better highlight the language in actual use. The last two or three years have seen the evolution of a style of programming in C++ that is regarded as safe. The examples try to illustrate this approach.

1.7 Coda

Be prepared to devote some time to learning C++. You can't gain an understanding of 50 years of program language development and a complex language like C++ without some effort on your part.

The course material and examples have been developed with the following guidelines:–

- to look at a set of commonly occurring problems that need to be solved in most programming languages, e.g.
    - data types
    - expression syntax
    - control structuring
    - array handling
    - file i/o
- to look at the object oriented features of C++
- to look at the features in the language that support generic programming
    - templates
    - the functionality offered by the standard template library

Some of the major strengths of C++ lie in the latter areas, i.e. its support for object oriented programming, the functionality provided by the standard template library, and its support for generic programming through the template facility.

Modifying existing programs is a good place to start, and you can often cannibalise the examples to solve some of the problems that you have. To develop further you have to get hold of one of the additional books and extend your knowledge beyond the information provided in these notes.

The consensus seems to be:–

- 20–30 hour course – basic syntax of C++;
- 3-6 months writing programs – basic grounding in C++;
- 2-3 years, no access to C++ class library guru, getting started;
- 2-3 years, access to class library guru, competent C++ programmer;

Be patient, Rome wasn't built in a day.

As with other langauges work is already under way for the next version of the standard. This will comprise corrections at least.

1.8 Course Details

The course is organised as a mixture of lectures and practicals as the most effective way to learn a programming language is by using it. Practice is essential. Think about how you learn French, German, etc.

It is important for you to read the notes between the timetabled sessions and also try completing the examples and problems set.

1.9 Problems

1. Do you have a user number and password for the King's system? If not please visit Computer Reception.

2. Once you have an account log into the system. Details of how to do this are given in a separate hand-out.

3. Create a file using whichever editor you wish to use. textedit is probably the easiest. vi is not for the faint hearted.

4. Start a web browser and locate the course material on the King's web server.

5. If you have your own pc running Windows then install cygwin. This will provide a very good compiler at just the time and effort taken to download and install.

I have downloaded it and have it on both cd and memory stick. It is about 150 Mb. You can take a copy, just ask me.

6. If you have your own pc then you may like to install gvim. This is a version of vi with a graphical interface. It also does colour coding of a lot of programming languages.

2
Introduction to Programming Languages and Object Oriented Programming

‘We have to go to another language in order to think clearly about the problem.’

Samuel R. Delany, Babel–17

Aims

The primary aim of this chapter is to look at some of the languages used in the sciences. There is a look at the developments from a historical view point; a comparison of their features and a look at future developments.

2 An Introduction to Programming Languages and Object Oriented Programming

The intention of this chapter is to examine, from the viewpoint of languages of use in scientific problem solving, of the background of programming languages and their development. It is essential that you develop an understanding of why there are so many programming languages and their strengths and weaknesses. No one language is suitable for solving all the problems that you will come across. You need to chose the right tool for the job. Think about diy around the home. If the only tool you have is a hammer then everything seems to be seen as a nail.

See the bibliography for a broader coverage.

2.1 Fortran 66, 1966

The original was designed by a team lead by Backus at IBM, in the late 50's. It is therefore quite old. This is the date of the first standard. The language quickly established itself as the language of first choice for numeric programming.

2.2 Pascal, 1975, ANSI & BSI 1982, ISO 1983, Extended Pascal 1991?

Very successful attempt at a *teaching* language. Note that it precedes both C and Fortran 77. Pascal still is the most widely taught programming language in computer science departments, as the introductory programming language. The following summarises the survey done by Dick Reid taken from a number of editions:–

Language	**20th**	**18th**	**15th**	**13th**
Pascal	140	144	151	157
C++	101	100	87	34
Ada	82	82	74	73
C	58	56	51	39
Scheme	50	49	51	50
Modula	32	32	32	35
Java	15	-	-	-
Modula-2	14	15	15	13
Fortran	9	9	9	8
SML	8	7	6	6
Turing	4	4	5	6
Miranda	4	4	4	3
Smalltalk	4	4	4	1
Eiffel	3	3	3	3
Oberon	3	3	2	2
ISETL	2	2	2	2
ML	2	2	2	1
Modula-3	2	2	2	2
ObjPascal	2	2	2	1
Ada95	2	-	-	-
Haskel	2	1	1	1
Beta	1	1	1	1
Oberon-2	1	1	1	-
Orwell	1	1	1	1
Prolog	1	1	1	1
Simula	1	1	1	1
Blue	1	-	-	-

The first edition was May 1990. New editions come out about every six months. He doesn't keep past editions. I've put up the complete survey, which includes the institutions, at:–

`http://www.kcl.ac.uk/kis/support/cc/fortran/dickreid20.txt`

What is interesting is the following:–

`http://www.kcl.ac.uk/kis/support/cc/fortran/sdickreid20.txt`

which is a sorted list by language. Where is your institution?

2.3 Fortran 77, 1978

Modest attempt to update the language. Still largely Fortran 66. Added BLOCK IF, better array subscripting and corresponding do looping. Given the knowledge of the time a very conservative update to the language.

2.4 C, K&R 1978, Standard 1989.

C was developed by Kernighan and Ritchie at Bell Labs. Bell Labs was the research laboratory of the Bell Telephone company in the US. It was originally written for a PDP 11 under UNIX. It is based on the cpl, bcpl, b family of languages and these are typeless languages. It was designed as a systems implementation language and was used to rewrite some 95% of the UNIX operating system. Only 5% or so ended up being written in assembler. The UNIX tools are written in C and are a very good example of what C is best at: the construction of sharp, small tools. There are little or no facilities in the language for the construction of larger code suites. Given the date of the publication of K&R there was the opportunity to have tidied the language up somewhat. Look at the features of some of the other languages covered here and the dates to see what is meant by this statement.

The X library is written in C, is over 15 years old and still leaks memory.

2.5 Modula 2, 1982, Standard 1996?

Wirth's next language after Algol W and Pascal. Attempt to produce a professional programmers Pascal. Very many good features. Let down by the delay in getting standardised. Introduced modules (hence the name), got rid of some of the idiosyncratic syntactic sugar of Pascal, had the idea of separate definition and implementation. Rivals Ada without much of the complexity of Ada for real time applications.

Numeric work in Modula 2 isn't very attractive. Explicit type casts are required in mathematical expressions. The proposed standard alleviates some of the problems in this area.

2.6 Ada, ISO 8652: 1987

Attempt to produce a powerful and expressive language by the American Department of Defence. Given the very large defence spending budget, even a 1% gain from the adoption of a better programming language will be repaid. Gaining ground, from a slow start. See also Ada 95 later. First draft report was 1980.

2.7 C++, 1986, Standard November 1997

Attempt by Stroustrup to produce an object oriented version of C. He had been exposed to Simula early on and realised the benefits of a language like that. Simula is a product of the 1960's. The first version of Simula was available in 1967. Object oriented programming is not new! Simula was widely used for discrete event simulation.

2.8 Oberon 2, Late 1980's, early 1990's.

Very clean and simple OOP language. Partly driven by the visit of Wirth and Gutnecht to Xerox PARC, and Wirth having to take over the operating system course at ETH. Name arises from the Voyager probe taking pictures of Uranus. Oberon is the largest moon. Oberon was simpler than Modula 2. Oberon 2 tidied up a bit. First operational system by the late 1980s. Oberon replaced Modula 2 in 1989 for teaching purposes at ETH. Ported to

a Apple MAC, SUN, DEC, IBM RS6000 and MS/DOS by 1991. Free versions available from the ftp server at ETH.

Let down badly by continual development and lack of a standard. It is a research vehicle for Wirth and the CS department at ETH in Zurich. Some of the problems here may be remedied in the near future with the progress being made on the standardisation front.

2.9 Fortran 90, 1991.

Modern language. Limited OO capability. Good information hiding and powerful mathematical capability. The language of first choice for people involved in numeric programming.

2.10 Eiffel, 1988

Date is the publication of *Object Oriented Software Construction*, Meyer, Prentice Hall. Modern object oriented language. Meyer is a very keen exponent of the benefits of object oriented programming. Meyer's book on object oriented software construction is an extremely good introduction to OO programming.

Eiffel is an attempt to produce an industrial strength OOP language. I hope to be able to make available an Eiffel compiler for the alpha in the near future.

Another extremely worthwhile acquisition is *Software Development Using Eiffel: There Can Be Life Other Than C++*, Richard Wiener. He highlights some of the weaknesses of C++. If you are going to attempt a reasonable size application in C++ using OO techniques then you should read this book. He clearly highlights some of the major pitfalls. He provides both academic and commercial courses on C++ and Eiffel.

2.11 Ada, ISO 8652: 1995

Latest standard. Major changes from the 1987 standard are in the areas of:–

- Support for 8 and 16 bit character sets;
- Object oriented programming with run-time polymorphism;
- Extension of access types;
- Efficient data oriented synchronisation;
- library units;
- interfacing to other languages;

There are several so called *Specialised Needs Annexes*. These are:–

- Annex C, *Systems Programming*
- Annex D, *Real-Time Systems*
- Annex E, *Distributed Systems*
- Annex F, *Information Systems*
- Annex G, *Numerics*
- Annex H, *Safety and Security*

2.12 Java

Bill Joy (of Sun fame) had by the late 1980's decided that C++ was too complicated and that an object oriented environment based upon C++ would be of use. At round about the same time James Gosling (mister emacs) was starting to get frustrated with the implementation of an SGML editor in C++. Oak was the outcome of Gosling's frustration.

Sun over the next few years ended up developing Oak for a variety of projects. It wasn't until Sun developed their own web browser, Hotjava that Java as a language hit the streets. And as the saying goes *the rest is history*.

Java is a relatively simple object oriented language. Whilst it has its origins in C++ it has dispensed with most of the dangerous features. It is OO throughout. Everything is a class.

It is interpreted and the intermediate byte code will run on any machine that has a Java virtual machine for it. This is portability at the object code level, unlike portability at the source code level – which is what we expect with most conventional languages.

It has built in garbage collection – no dispose!

It has no pointers – everything is passed by reference!

It is multithreaded, which makes it a delight for many applications.

It has a extensive windows toolkit, the so called AWT that was in the original release of the language and Swing that came in later. It achieves much of what Visual Basic offers but within the framework of a far more powerful language. Development environments are becoming widely available to aid in this task.

Finally it is fun!

Major drawback is the rapid development of the language and the large number of different versions. Further compounded by the different virtual machines available.

2.13 Visual Basic

This language is a development by Microsoft to enable visual user interfaces to be programmed easily. It has subject to continual development and offers one of the easiest ways of developing a windows style program for a pc running Windows 3.x, 95, 98 and NT.

2.14 Language Comparison

The following page has a simple language feature comparison. The emphasis is on highlighting the strengths and weaknesses of languages used mainly in the scientific area.

The following symbols are used:–

Y	supported
y	supported with qualification, e.g. may be achieved using a different mechanism
-	not supported
?	unable to verify adequately at the time of writing

In all cases please see the more detailed coverage that follows. The table needs to be brought up to date. The later developments include:-

- Fortran - Fortran 95, ISO TR 15580, 15581, Fortran 2003, ISO TR Enhanced Modules.
- Java - Sun have made a number of changes since the version described in these notes.
- C# - Microsoft have released C# - pronounced C Sharp.

Simple Program Language Feature Comparison

	Fortran			Pascal			C	C++	Ada		Java
	66	77	90	Modula 2 Oberon 2							
				P	M2	O2	C	C++			
Year	66	78	91	75 82	82 96?		78 89	86 97	87	95	199?
Feature											
Independent compilation	Y	Y	Y	-	-	-	Y	Y	-	-	-
Separate compilation	-	-	Y	-	Y	Y	-	Y	Y	Y	Y
Concrete data types	-	-	Y	Y	Y	Y	Y	Y	Y	Y	[1]
Abstract data types	-	-	Y	-	Y	Y	-	Y	Y	Y	Y
Dynamic arrays	-	-	Y	-	-	-	Y	Y	Y	Y	Y
Modules See below	-	-	Y	-	Y	Y	-	y	y	y	y
Numeric Polymorphism See below	-	-	Y	-	?	Y	-	y	y	y	-
General Polymorphism	-	-	Y	-	?	Y	-	Y	Y	Y	Y
Pointers	-	-	Y	Y	Y	Y	Y	Y	Y	Y	[2]
Procedure variables	-	-	-	Y	Y	Y	Y	Y	?	?	?
Inheritance single/mult	-	-	-	-	-	S	-	M	-	S	S
Dynamic binding	-	-	-	-	-	Y	-	Y	-	Y	Y
Operator overloading	-	-	Y					Y	Y	Y	-
Threads Tasking	-	-	-	-	Y	?	-	-	?	Y	Y
Exception Handling	-	-	-		?	?	-	Y	?	Y	Y

These are some of the major features that we need to look at when comparing programming languages and looking at the development of programming languages.

[1] – Against the spirit of object oriented programming.

[2] – Replaced by references

Dates

Fortran: The dates are the dates of the standards.

Pascal: Preliminary version 1968. Major development and first operational compiler 1970. 1973 publication of first revised report. User Manual and Report 1975.

Modula: Defined experimentally 1975. Lilith research project 1977. First implementation of Modula 2 1979. Technical report March 1980. First distributed compiler March 1981. *Programming in Modula 2* 1982.

Oberon 2: 1988, N. Wirth. *The Programming Language Oberon*, Software Practice and Experience, 1991, Mossenbeck and Wirth, *The Programming Language Oberon 2*. 1991, Reisser , *The Oberon System, User Guide and Programmer's Manual*. 1993, Mossenbeck, *Object Oriented Programming in Oberon 2*.

C: 1969, M Richards: *BCPL A Tool for Compiler Writing and Systems Programming*. 1970, Ken Thompson, B. 1978, K and R. *The C Programming Language*,. 1989 ANSI C Standard.

C++: C with classes, 1979-1983 From C with Classes to C++, 1982-1985 Release 2.0: 1985-1988 Stroustrup, *The C++ Programming Language*, 1st edition, 1986 Stroustrup, *The C++ Programming Language*, 2nd edition, 1991. Standard 1997.

Ada: 1980 original definition. Standard 1987. Latest standard is 1995.

Java: Exact date not well defined. Hopefully the standardisation effort will make life easier!

2.15 Language Features

It is illuminating to look at the languages from the viewpoint of their features and functionality.

2.15.1 Independent Compilation

The ability to break a problem down into parts and work on one part at a time. No checking between compilation units.

2.15.2 Separate Compilation

As above with checking *across* compilation units. Major step forward in the construction of more complex programs. Lint helps out with C. Forcheck is useful for Fortran 66 and 77 programmers.

2.15.3 Concrete Data Types

The ability for the user to define data types that mapped directly onto their problem. A major step forward. User has to know about the implementation however.

2.15.4 Abstract Data Types

The twin concept of data types and procedures that manipulated the data. Hiding the internals from the calling routine.

2.15.5 Dynamic arrays

Arrays that are allocated dynamically at run time.

2.15.6 Numeric and General Polymorphism

Numeric Polymorphism

In the simplest case the ability to have so called mixed mode arithmetic expressions, e.g. mix integers and reals (both of one or more underlying representations) without casting between one type and another. The explicit type casting required in some languages means that they are not widely used for numeric programming.

At the next level the ability to call in built functions with numeric data of one or more types. This has been in Fortran from a very early stage.

Finally the ability to create one's own functions that accept numeric data of a variety of numeric types.

Languages that support OOP have to offer the last kind of polymorphism.

General Polymorphism

OO programming languages have to offer this functionality.

2.15.7 Modules

The primary purpose of modules is to provide the ability to group related functions and procedures. This is a powerful program decomposition tool. They normally have a well controlled mechanism for making visible what the external, calling routine needs to have access to.

Terminology varies with programming languages, and so does the exact functionality that these different languages support.

Classes and packages are two terms also used.

2.15.8 Pointers and References

Pointers in a programming language extend the range of problems that can be solved considerably. Multi-dimensional structures are easily programmed using pointers, e.g. linked lists, queues, trees, quad-trees, oct-trees etc.

The major problem is that the user is provided with very little help if they are programmed incorrectly. It is assumed that you know what you are doing.

In Java all objects are accessed via an object reference. When you see an object variable in a Java program you actually see a reference to an object. We will look into the concept of references in much greater detail throughout the course.

2.15.9 Procedure Variables

An elegant way of extending the expressive power of a language. Quite old.

2.15.10 Inheritance

The first of the two major step towards OO programming. Allows the user to extend an existing type without having to know what is going on.

2.15.11 Dynamic Binding

The second of the two major features of OO programming. Provided in a limited sense via procedure variables in older languages.

2.15.12 Operator Overloading

Syntactic sugar in many ways and over valued. Very useful to people with numeric problems. Given that C++ has 47 operators it poses problems of readability and comprehensibility with most other areas. The restrictions that C++ has in this area will be looked at later.

2.15.13 Threads/Multitasking

Multitasking and/or threads are needed in a programming language when solving problems in the areas of real-time systems, equipment interfacing, embedded systems and parallel programming.

2.15.14 Exception Handling

Exception handling is needed in a programming language when solving problems in the areas of real-time systems, equipment interfacing, embedded systems, parallel programming and robust systems.

2.16 Some Important Milestones in Program Language Development

We look here at some of the major steps forward in the way we approach problem solving using programming languages. Often people adopted the following methodologies without having features in a programming language that actually supported them.

2.16.1 Structured Programming

Structured programming in its narrowest sense concerns itself with the development of programs using a small but sufficient set of statements and in particular control statements. It has had a great effect on program language design and most languages now support a minimal set of control structures. In a broader sense it subsumes other objectives including simplicity, comprehensibility, verifiability, modifiability and maintenance of programs.

The ideas are very well covered in the Dahl, Dijkstra, Hoare text. This is essential reading.

2.16.2 Stepwise Refinement

The original ideas are very well expressed a a paper by Wirth entitled *Program Development by Stepwise Refinement*, published in 1971. Essential reading.

2.16.3 Data Structuring, Concrete vs Abstract Data Types

With a concrete data structure you have to know how the data structure is organised explicitly. With abstract data types the internals are hidden from you and all you see are a set of procedures that provide the functionality you want.

2.16.4 Information Hiding – Modules

A major step forward in the development of programming languages. The paper by Parnas addresses the idea of information hiding and the module concept is one that a number of languages now offer to help in this area.

2.17 Terminology of Object Oriented Programming

The following provides a link between conventional programming language terminology and that used in object oriented programming.

Class	Extensible abstract data type
Object	Instance of a class
Message	Procedure call, dynamically bound
Method	Procedure of a class.

See Mossenbeck for a good treatment of this. We'll come back to the whole area of object oriented programming after a coverage of the basics of C++.

2.18 Parallel Developments

With the increasing availability of computers with multiple processors at ever decreasing costs the importance of languages that support parallel computation increases. Two languages that offer support in this area are based on Fortran 90 and C++. A brief coverage is given below.

2.18.1 Parallel Fortran – Fortran 95, Fortran 2003, Fortran 2006 and Coarray Fortran

There are a number of ways of doing parallel programming in Fortran, and these include

- MPI - a message passing library
- Open MP - a variant of Fortran
- Coarray Fortran - a proposal for the Fortran 2008 standard.

MPI is probably the most widely used at this time.

2.18.2 Parallel C++

Similar developments are in the pipeline for C++.

2.19 Object Oriented Programming

Object oriented programming is characterised by two main concepts:–

- inheritance;
- dynamic binding;

The major thing we need to consider is how to extend the functionality of an existing program. To make sense of the benefits of OOP we need to have a good understanding of the strengths and weaknesses of the traditional programming paradigm. We need to look at the way our appreciation of how to use programming languages developed and what we needed from them as the problems we tackled became more complex.

To quote Friedman:–

> Object oriented programming makes good on the promise of structured programming. It implements in a very practical way the principles of program decomposition, data abstraction and information hiding. It ties together and provides a framework for abstraction and structure at all levels; data program and control.
>
> ...
>
> OOP picks up where structured programming methodology leaves off. Dijkstra's concept of structured programming, Wirth's stepwise refinement and Parnas's information hiding all contribute to a software development milieu that promised to become increasingly systematic and scientific. OOP, to a great extent, fulfils that promise. It takes the concepts of data abstraction, modular decomposition, and information hiding and refines them in a cohesive world view, data objects are active entities. Instead of passing data to procedures the user asks objects to perform operations on themselves. A complex problem is viewed as a network of objects that communicate with each other.

The benefits of OOP come with programming in the large. If the problems you have don't warrant it you may never need to devote the time and effort to gain complete mastery of a powerful and complex language like C++.

2.20 Object Oriented Languages

The ideas are not new.

2.20.1 Simula – 1967

The seminal text on OOP is *Simula BEGIN*, Birtwistle, Dahl, Myhrhaug and Nygaard. The book is very well written if a little dated today. I'd recommend it if you are involved in discrete event simulation. This was what the language was used for whilst I worked at Impe-

rial College before I came to King's in 1986. They are the first people to use the concept of a class. Algol 60 based. Stroustrup got many of his ideas from Simula.

2.20.2 Smalltalk – 1978

The text I recommend is *Smalltalk 80, The Language and its Implementation*, Goldberg and Robson. They worked at the Xerox Palo Alto Research Centre (Xerox Parc) Learning Research Group. Ideas are drawn from Simula and the vision of Alan Kay. Steve Jobs was heavily influenced by and the Apple Macintosh owes a big debt to the Xerox Parc people. We take windowing systems for granted these days on many of the systems we work with from the pc with Windows, to Unix workstations with their X windows interfaces and of course the first to bring them to the mass market – the Apple Mac.

Regarded as a pure OO system by most people with everything an object.

Wirth has spent a number of periods at Xerox Parc and that is reflected in Oberon 2.

2.20.3 C++

In Stroustrup's words ...*C++ is a general purpose programming language; its core application domain is systems programming in the broadest sense*...

It is of course also used in a wide range of other application domains, notable graphics programming. It is enormously popular, and there are a very large number of jobs advertised for people with C++ skills. You are unlikely to be out of work if you get to be a good C++ programmer.

C++ supports inheritance through class derivation. Dynamic binding is provided by virtual class functions.

2.20.4 Eiffel

Object Oriented Software Construction is dated 1988. As stated earlier the first four chapters address OOP. The latter chapters look at Eiffel in some depth. The text is a relatively easy read.

Achieves much of the power and expressiveness of C++ without the complexity. As is has its origins in Ada is is also a language that offers far greater support for error protection and safe code.

I'm informed that the Channel Tunnel software uses Eiffel!

I am looking at getting hold of an Eiffel compiler at this time, but am unsure as to what platform it might be available on.

2.20.5 Oberon 2

The language has its origins in a visit that Gutnecht and Wirth made to Xerox PARC in 1985. They decided to design and implement a new operating system. In their words ... *the ultimate goal was to create a system for personal workstations that was not only powerful and convenient for practical use but also describable and explicable in a convincing way*... They had originally decided to use Modula 2 but made the decision to strip out some of the features of that language and add a very small number of features. The outcome was Oberon. The language was defined in 1986.

The object code size of the outer core of the Oberon system is 200K, and comprises

- a kernel
- a dynamic loader and garbage collector
- a file system

- drivers for disk, diskette, asynchronous and synchronous communication, printer and a bit mapped display;
- local network services;
- support for texts and fonts;
- a window subsystem;
- a text editor;
- the Oberon compiler;

Educational versions of the system are available from the ETH ftp server. I'd recommend 8Mb of memory and an 80486 with 1024*756 display. I've tried at home on a 20 MHz 80386 with 5 Mb of memory – *a bit slow*... Well worth a look at.

Programming in Oberon, Reiser and Wirth, is a good introductory text and combined with *OOP in Oberon 2*, Mossenbeck, you have sufficient information to get started.

There is a lot of documentation that comes with the system and this can be printed.

The system integrates very well with both Windows on the PC and Apple macs. Versions for other platforms are available.

If your are familiar with Pascal or Modula 2 then I'd recommend Oberon very highly to see what OOP has to offer. Very low cost in time, effort and money.

2.20.6 Ada 95

Whilst I do not have access yet to an Ada 95 compiler from what I've read it looks a very good language. The standard is available from a number of ftp servers, and there is also a look at the changes from the original version to 95 available on the web. A text I'd recommend is *Programming in Ada 95*, Barnes. This is well written. Ada 95 is the first language that has been standardised with OOP in mind.

If you want a job in the defence industries, or see yourself working with real time embedded systems then this is certainly a language to consider looking at.

2.20.7 Java

Java is a recent OO language. It is unusual in that it is the product of one company, Sun, rather than the subject to formal language standardisation like the majority of the other languages covered in this chapter. It is hoped that it will pass out of Sun's hands in the near future into the mainstream of language standardisation. Freely available if you have internet access. IBM have taken it on board in a big way and so have Microsoft. They both realise the potential earning capacity of Java and the internet.

Due to long file names and multithreading requirements needs operating systems like Windows 95 and above.

2.21 Other Languages

There are languages that offer limited support for OOP. The two that follow fit into this category. I don't have completely up to date information on what is likely to be in the Modula 2 standard at this time.

2.21.1 Fortran 90

Through the functionality provided via user defined data types and modules it offers support for *object based* programming. See Dupee's MSc thesis for a good coverage of what Fortran 90 has to offer here.

2.21.2 Modula 2

The original language was a major advance over Pascal. It corrected many of the deficiencies of Pascal in a straightforward way. From what I've seen of the draft standards it will be a powerful and expressive language.

Standard versions of the language look like being quite expensive and it is unlikely that we would be able to make available a compiler on any platform given the proposed cuts in expenditure by the various government funding bodies.

2.22 The OO Approach

We will look at two approaches here. The work of Meyer and Rumbaugh et al are both well regarded, and we will cover both briefly.

2.22.1 Meyer's Approach

Meyer identifies seven steps that lead towards object oriented solutions. These are given below:–

- object based modular structure – systems are modularised on the basis of the data structures;
- data abstraction – objects should be described as implementations of abstract data types;
- automatic memory management – unused objects should be deallocated by the underlying language systems, without programmer intervention;
- classes – every non simple type is a module, and every high level module is a type;
- inheritance – a class may be defined as an extension of restriction of another;
- polymorphism and dynamic binding – program entities should be permitted to refer to objects of more than one class and operations should be permitted to have different realisations in different classes;
- multiple and repeated inheritance – it should be possible to declare a class as heir to more than one class and more than once to the same class;

You may disagree with one or more of the above and some of the above are capable of arousing very strong views in the programming language world!

Essential reading for large scale systems.

2.22.2 Rumbaugh et al

This book concentrates on OO modelling, rather than using a particular programming language for OO programming. If you have a background in the relational database area then much of the coverage should be quite familiar. They present a methodology for object oriented development – the Object Modelling Technique or OMT. They identify four stages:–

- analysis: build a model of the real world situation;
- system design: make the high level decisions about the overall architecture;
- object design: build a design model (based on the analysis model) with implementation details;
- implementation: translate into an implementation using a particular programming language, database, or hardware implementation;

and three kinds of models to describe the system:–

- the object model: describes the static structure of the objects and their relationships;
- the dynamic model: describes the aspects of the system that change with time;
- the functional model: describes the data value transformations within the system;

and the three models are orthogonal, with cross links.

This book is a must for large scale systems.

2.22.3 Practical Steps

The two major practical steps are:–

- identify the classes; within this discriminate between:–
  - an *is a* relationship, e.g. where one class is a sub-type of another;

    Consider the concept of a point. It has two attributes, an x and y position. So we could write point(x,y). Now consider the concept of a pixel – a point with the added attribute of colour. Now we could write pixel(point,colour), Thus pixel is-a point.
  - a *has a* relationship, e.g. where a class is made up of an object of another class;

    Consider the concepts of engine and car. In this case a car has-a engine.
- define the interfaces to the classes.

Inheritance commits you to much more than becoming a client. As a client you are protected against future changes in the implementation of a class. When you inherit you gain access to the implementation and all that goes with it.

The above have to be done before any code is written. Programming is an iterative process and it is inevitable that you will need to cycle through the design and implementation stages as you write code, i.e. it will be obvious that you will need to go back and redesign and reimplement base classes in the light of experience.

2.23 Simple Example

Consider putting together a graphical drawing system. We are interested in shapes and the concepts of moving and drawing. We can do this in a very straightforward way using an OO approach.

Firstly we have a base abstract class shape with two associated procedures, one to move and one to draw.

- shape(x,y)

  move(shape s)

  draw(shape s)

Secondly we then derive other shapes from them. We provide two derived classes with two associated procedures:–

- square(shape,side)

  move(shape s)

  draw(shape s)

- rectangle(shape,length,breadth)

  move(shape s)

  draw(shape s)

Consider the following pseudo-code segment

```
...
square s(50,50,10);
rectangle r(100,100,10,20);
  move(s)
  draw(s)
  move(r)
  draw(r)

return(0)
}
```

We can now add another shape, e.g. circle, and still have things work *without* recompiling the old code. We just compile the new move and draw procedures and link them in. The method resolution is handled by dynamic binding. We will look into this in greater depth later in the course.

2.24 Other Developments

The chapter rounds off with a look at recent developments that have taken place and that apply to one or more programming languages in some cases.

2.24.1 Development Environments

The traditional working practice for program development involves the following steps:–

- edit
- compile
- link
- run
- debug

in a loop. This method has the advantage of working on most hardware and software platforms. The major drawback is learning several ways of doing exactly the same thing as we move from platform to platform and from one language to another. How many editors do you know how to use?

There have also been developments to provide an integrated environment for program development. These environments started out as workbenches providing simple access to the editor, compiler, linker etc through simple key strokes and the mouse. They have grown very sophisticated.

2.24.2 Graphical Development Tools

There has also been the development of a visual interface to programming. Increasingly people want easy to use software that almost by definition has a wndows based interface. Microsoft Visual Basic provides a good example of this. Products like this typically have:–

- a toolbox of components that can be dragged and dropped onto a screen
- a screen or form that the user will see – this comprises the user interface

- a set of properties for each of the components that can be tailored for your own requirements

You typically use the mouse to select the item you want from the toolbox (menu, form etc) drag and dop onto the form and then alter the various associated setting using the propertiy entries on the right hand side. Skeleton code is often generated for you which you then tailor to your own specific requirements.

For some alternatives to using Visual Basic to put a windows based front end to a program have a look at the following url for more details:–

`http://www.kcl.ac.uk/kis/support/cc/fortran/language.html`

2.24.3 Software Components

As the problems that we attempt to solve become more complex and the interfaces we provide become more sophisticated we need better tools to help with program development. One major step forward is in reusable software components. This can be seen as an extensions to the object oriented approach to programming.

Consider building an application that required a spelling checker. The idea is to buy one of the shelf and slot it straight in to our program. This is gradually becoming a reality.

Sun and Microsoft both made developemnts in this area and we will look at each in turn.

2.24.3.1 COM, OLE, ActiveX

First let us define each of the above in turn:–

- COM – Component Object Model
- OLE – Object Linking and Embedding
- ActiveX – Now called Active Technologies

Let is look briefly at each in turn.

ActiveX is a set of technologies that enables software components to interact with one another in a networked environment, regardless of the language in which they were created. ActiveX is build on the Component Object Model.

OLE or OLE Automatation makes it possible for one application to manipulate objects implemented in another application or to expose objects so that they can be manipulated.

A simple example of this would be embedding a spreadsheet of graph from Excel in a Word document. Double clicking on the spreadsheets drops you into Excel. There are some problems with this when working with a document on a number of computer systems with different versions of the applications concerned.

The Component Object Model (COM) is a platform independent, distributed, object orientded, system for creating binary software documents that can interact. COM is the foundation technology for OLE and ActiveX.

2.24.3.2 JavaBeans

JavaBeans is the Java language software component offering. JavaBeans extends the Java language write once run everywhere capability to resuable component development.

There is considerable interest in JavaBeans because of the platform neutrality of the developed code. Sun also provide mechanisms to migrate ActiveX/OLE/COM to JavaBeans.

Sun and Microsoft and battling this one out and there is little love lost between them. Watch this space as they say!

2.25 Coda

There is a lot that John Backus has to be proud of. He achieved a lot with Fortran. Fortran is still the language of first choice for the majority of numeric based problem solving, especially for so called legacy code.There is a lot of very well written code in Fortran, and we find both commercial and public domain numeric libraries available for most platforms.

Wirth has a lot to be proud of too with the Algol family of languages – Algol, Algol W, Pascal, Modula, Modula 2, Oberon, Oberon 2. The sad thing that is that whilst abandoning the previous language enabled successor languages to be well designed and compact, there was a lot of effort required in moving production code from older languages to their successors. Knowledge of one of the more recent languages (Modula 2 or Oberon) in this family is a worthwhile investment.

C and C++ have a considerable amount of code written in them. C++ represents a very major advance over C, correcting some of the program correctness problems that arise in C from array subscript checking, pointer manipulation, type checking and optimisation problems with pointer aliasing and array handling. For many applications they offer quite significant advantages over other languages. Graphical based systems are invariably written in C++. Object oriented programming is often done in C++. The maintenance problems C poses are considerable. The steep learning curve that C++ has poses problems for successive generations of would be C++ programmers.

Look at the Computational Science Education paper for an comparison of C, Fortran 77, Fortran 90 and C++. This is on the King's web server.

Java is a successful language. The widespread use of the internet has made it a very popular choice for web based application programming.

VB is a successful language. Many people need to be able to develop easy to use programs and systems. Windows programs offer the possibility of solving this problem.

No one language is appropriate for solving every problem. Many factors come into play in real life. Learning a variety of languages is a good idea. Learning Fortran 95, C++, Java and Visual Basic provides you with a range of very useful skills for work in the academic, scientific, engineering and commercial worlds.

I've included references to a couple of other languages that you might like to look at, and these are Icon, Snobol, Prolog and SQL. Icon and Snobol are very good string processing languages. SQL is **the** database language, and Prolog is a logic based language.

2.26 Bibliography

Adobe Systems Incorporated, *Postscript Language: Tutorial and Cookbook,* Addison Wesley.

Adobe Systems Incorporated, *Postscript Language: Reference Manual,* Addison Wesley.

Adobe System Incorporated, *Postscript Language: Program Design,* Addison Wesley.

The three books provide a comprehensive coverage of the facilities and capabilities of Postscript.

ACM SIG PLAN, *History of Programming Languages Conference – HOPL-II,* ACM Press.

One of the best sources of information on programming language developments, from an historical perspective. The is coverage of Ada, Algol 68, C, C++, CLU, Concurrent Pascal, Formac, Forth, Icon, Lisp, Pascal, Prolog, Smalltalk and Simulation Languages by the people involved in the original design and or imple-

mentation. Very highly recommended. This is the second in the HOPL series, and the first was edited by Wexelblat. Details are given later.

Adams, Brainerd, Martin, Smith, Wagener, *Fortran 90 Handbook: Complete ANSI/ISO Reference,* McGraw Hill.

A complete coverage of the language. As with the Metcalf and Reid book some of the authors were on the X3J3 committee. Expensive, but very thorough.

Annals of the History of Computing, *Special Issue: Fortran's 25 Anniversary,* ACM publication.

Very interesting comments, some anecdotal, about the early work on Fortran.

Arnold K., Gosling J., *The Java Programming Language*, Addison Wesley.

Written by the people who designed and implemented the language. A definitive source on the language. A bit expensive at just under thirty uk pounds.

Barnes J., *Programming in Ada 95*, Addison Wesley.

A recent update of his previous Ada book. Comprehensive coverage of Ada 95. Not for the beginner.

Birtwistle G.M., Dahl O. J., Myhrhaug B., Nygaard K., *SIMULA BEGIN*, Chartwell-Bratt Ltd.

A number of chapters in the book will be of interest to programmers unfamiliar with some of the ideas involved in a variety of areas including systems and models, simulation, and co-routines. Also has some sound practical advice on problem solving.

Booch G., *Object Oriented Design with Applications*, Benjamin Cummings, 2nd Ed. 1994.

I've not been able to get hold of a copy of this yet. One is on order at Dillons. Don't buy or bother with the first edition, as there are bound to be major advances in this edition due to his experience between editions. Still not available at this time.

Brinch-Hansen P., *The Programming Language Concurrent Pascal*, IEEE Transactions on Software Engineering, June 1975, 199-207.

Looks at the extensions to Pascal necessary to support concurrent processes.

Cannan S., Otten G., *SQL – The Standard Handbook*, McGraw Hill.

Very thorough coverage of the SQL standard, ISO 9075:1992(E).

Chivers I.D., Essential C# Fast, Springer Verlag.

Brief introduction to programming in C#.

Chivers I.D., Essential Visual C++ Fast, Springer Verlag.

Brief introduction to Windows Programming using the Microsoft Foundation Class Library.

Chivers I. D. and Clark M. W., *History and Future of Fortran,* Data Processing, vol. 27 no 1, January/February 1985.

Short article on an early draft of the standard, around version 90.

Chivers I.D., and Sleightholme J., *Introducing Fortran 90*, Springer Verlag.

An introduction to programming using Fortran 90. Aimed at numeric problem solving.

Chivers I.D., and Sleightholme J., *Introducing Fortran 95*, Springer Verlag.

An introduction to programming using Fortran 95, with coverage of ISO TR 15580 and ISO 15581. Aimed at numeric problem solving.

Chivers I.D., and Sleightholme J., *Introducing to programming with Fortran. Springer Verlag.*

An introduction to programming using Fortran with coverage of Fortran 90, Fortran 95, Fortran 2003 and Fortran 77. Aimed at numeric problem solving.

Computational Science Education Project, *Fortran 90 and Computational Science.*

This paper is a comparison of C, Fortran 77, C++ and Fortran 90 using the following five criteria:– numerical robustness, data parallelisation, data abstraction, object oriented programming and functional programming. A copy is available on the college web server. Essential reading if one is involved in programming with one or more of these languages.

Cowell J., *Essential Java: Fast*, Springer

Compact introduction to Java. Insufficient on its own.

Dahl O. J., Dijkstra E. W., Hoare C. A. R., *Structured Programming*, Academic Press, 1972

The seminal book on structured programming.

Date C. J., *A Guide to the SQL Standard*, Addison Wesley.

Date has written extensively on the whole database field, and this book looks at the SQL language itself. As with many of Dates works quite easy to read. Appendix F provides a useful SQL bibliography.

Deitel H.M., and Deital P.J., *Java – How to Program*, Prentice Hall.

A very good introductory Java text. One of the best currently available. Highly recommended.

Dupee B., *A Study of Object Oriented Methods using Fortran 90*., MSc Thesis.

A look at OO methods in F90, rather than full blown OOP.

Flanagan D., *Java in a Nutshell*, O'Reilly and Associates.

One of the Nutshell series. If you want one book on Java then this is the one I'd recommend. Four parts are introduction to Java, programming with the Java api, Java language reference and api quick reference.

Friedman L. W. , *Comparative Programming Languages*, Prentice Hall.

I've got this on order.

Geissman L. B., *Separate Compilation in Modula 2 and the structure of the Modula 2 Compiler on the Personal Computer Lilith*, Dissertation 7286, ETH Zurich

Jacobi C., *Code Generation and the Lilith Architecture*, Dissertation 7195, ETH Zurich

Fascinating background reading concerning Modula 2 and the Lilith architecture.

Goldberg A., and Robson D., *Smalltalk 80: The language and its implementation*, Addison Wesley.

Written by some of the Xerox PARC people who have been involved with the development of Smalltalk. Provides a good introduction (if that is possible with the written word) of the capabilities of Smalltalk.

Goos and Hartmanis (Eds), *The Programming Language Ada - Reference Manual,* Springer Verlag.

The definition of the language.

Gosling J., Yellin F., The Java Team, *The Java API, Volumes I and II,* Addison Wesley.

Volume I looks at the core packages and Volume II looks at the Window Toolkit and Applets. I find the pricing a bit much at just under 40 uk pounds a book.

Griswold R. E., Poage J. F., Polonsky I. P., *The Snobol4 Programming Language,* Prentice-Hall.

The original book on the language. Also provides some short historical material on the language.

Griswold R. E., Griswold M. T., *The Icon Programming Language,* Prentice-Hall.

The definition of the language with a lot of good examples. Also contains information on how to obtain public domain versions of the language for a variety of machines and operating systems.

Hoare C.A.R., *Hints on Programming Language Design,* SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1973.

The first sentence of the introduction sums it up beautifully: *I would like in this paper to present a philosophy of the design and evaluation of programming languages which I have adopted and developed over a number of years, namely that the primary purpose of a programming language is to help the programmer in the practice of his art.*

Jenson K., Wirth N., *Pascal: User Manual and Report,* Springer Verlag.

The original definition of the Pascal language. Understandably dated when one looks at more recent expositions on programming in Pascal.

Kemeny J.G., Kurtz T.E., *Basic Programming,* Wiley.

The original book on Basic by its designers.

Kernighan B. W., Ritchie D. M., *The C Programming Language,* Prentice Hall: Englewood Cliffs, New Jersey.

The original work on the C language, and thus essential for serious work with C.

Kowalski R., *Logic Programming in the Fifth Generation,* The Knowledge Engineering Review, The BCS Specialist Group on Expert Systems.

A short paper providing a good background to Prolog and logic programming, with an extensive bibliography.

Knuth D. E., *The TeXbook,* Addison Wesley.

Knuth writes with an tremendous enthusiasm and perhaps this is understandable as he did design TeX. Has to be read from cover to cover for a full understanding of the capability of TeX.

Lemay L., Perkins, *Teach Yourself Java in 21 Days,* Sams net.

Most gentle of the books I've found. Includes a CD with the Sun JDK.

Lyons J., *Chomsky,* Fontana/Collins, 1982.

A good introduction to the work of Chomsky, with the added benefit that Chomsky himself read and commented on it for Lyons. Very readable.

Marcus C., *Prolog Programming: Applications for Database Systems, Expert Systems and Natural Language Systems,* Addison Wesley.

Coverage of the use of Prolog in the above areas. As with the previous book aimed mainly at programmers, and hence not suitable as an introduction to Prolog as only two chapters are devoted to introducing Prolog.

Metcalf M. and Reid J., *Fortran 90 Explained*, Oxford Science Publications, OUP.

A clear compact coverage of the main features of Fortran 8x. Reid was secretary of the X3J3 committee.

Meyer B., *Object Oriented Software Construction*, Prentice Hall.

I'm trying to get hold of the latest version. The copy I've got is dated 1988. Whilst obviously Eiffel based well worth a read. Also looks at other languages. Still not available at this time.

Mossenbeck H., *Object-Oriented Programming in Oberon-2,* Springer–Verlag.

One of the best introductions to OOP. Uses Oberon-2 as the implementation language. Highly recommended.

Papert S., *Mindstorms - Children, Computers and Powerful Ideas*, Harvester Press

Very personal vision of the uses of computers by children. It challenges many conventional ideas in this area.

Parnas D. L., *On the Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM, 15 (12), 1972.

One of the earliest papers to look at information hiding.

Sammett J., *Programming Languages: History and Fundamentals*, Prentice Hall.

Possibly the most comprehensive introduction to the history of program language development – ends unfortunately before the 1980's.

Reiser M., Wirth N., *Programming in Oberon – Steps Beyond Pascal and Modula,* Addison Wesley.

Good introduction to Oberon. Revealing history of the developments behind Oberon.

Reiser M., *The Oberon System: User Guide and Programmer's Manual,* Addison Wesley.

How to use the Oberon system, rather than the language.

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenson W.,*Object Oriented Modelling and Design*, Prentice Hall.

I like this book a lot. Having been involved in the relational area for over 10 years the book struck an immediate chord.

Seed G., *An Introduction to OOP in C++*, Springer.

Comprehensive introduction to C++ and OOP. Examples drawn from the computer graphics area.

Young S. J., *An Introduction to Ada, 2nd Edition,* Ellis Horwood.

A readable introduction to Ada. Greater clarity than the first edition. Dated in terms of the recent developments with Ada 95.

Wexelblat, *History of Programming Languages, HOPL I*, ACM Monograph Series, Academic Press.

Very thorough coverage of the development of programming languages up to June 1978. Sessions on Fortran, Algol, Lisp, Cobol, APT, Jovial, GPSS, Simula, JOSS, Basic, PL/I, Snobol and APL, with speakers involved in the original languages. Very highly recommended.

Wiener R., *Software Development using Eiffel: There Can Be Life Other than C++*, Prentice Hall.

Well written, and the case studies include an ecological simulation, a game of strategies and investments and simulated annealing. The chapter on object oriented analysis and design is highly recommended.

Wiener R., *Software Development Using Eiffel: There Can be Life After C++*, Prentice Hall.

Very well written. I'd really recommend getting hold of this book if you are going to seriously program in C++ using oo techniques. He teaches both academic and commercial Eiffel and C++ courses. He knows his stuff!

Winder R., Roberts G., *Developing Java software*, Wiley.

Waiting to get a copy.

Wirth N., *An Assessment of the Programming Language Pascal*, IEEE Transactions on Software Engineering, June 1975, 192-198.

Wirth N., Program Development by Stepwise Refinement, Communications of the ACM, April 1971.

Clear and simple exposition of the ideas of stepwise refinement.

Wirth N., *History and Goals of Modula 2*, Byte, August 1984, 145-152.

Straight from the horse's mouth!

Wirth N., *On the Design of Programming Languages*, Proc. IFIP Congress 74, 386-393, North Holland, Amsterdam.

Wirth N., *The Programming Language Pascal*, Acta Informatica 1, 35-63, 1971.

Wirth N., *Modula: a language for modular multi- programming*, Software Practice and Experience, 7, 3-35, 1977.

Wirth N., *Programming in Modula 2*, Springer Verlag.

The original definition of the language. Essential reading for anyone considering programming in Modula 2 on a long term basis.

Wirth N. *Type Extensions*, ACM Trans. on Prog. Languages and Systems, 10, 2 (April 1988), 2004-214

Wirth N. *From Modula 2 to Oberon*, Software – Practice and Experience, 18,7 (July 1988), 661-670

Wirth N., Gutknecht J., *Project Oberon: The Design of an Operating System and Compiler*, Addison Wesley.

Fascinating background to the development of Oberon. Highly recommended for anyone involved in large scale program development, not only in the areas of programming languages and operating systems, but more generally.

2.27 Problems

What programming languages are available on the system you work with?

3
Introduction to Operating Systems

"Where shall I begin your Majesty' he asked.
'Begin at the beginning,' the King said, graveley 'and go
on till you come to the end then stop."

Lewis Carroll, Alice's Adventures in Wonderland.

Aims

The aims of this chapter are to introduce the following:

- To provide a brief history of operating system development.
- To look briefly at some commonly used operating systems
  - DOS and Windows.
  - Unix and X-Windows.
  - Linux and X-Windows
  - VMS and Open VMS.

3 Introduction to Operating Systems

A simple definition of an operating system is the suite of programs that make the hardware usable. Most computer systems provide one. They vary considerably from those available on early microcomputers, like CP/M, to DOS and the various versions of Microsoft Windows on PCs, UNIX and X-Windows and Linux and X-Windows on workstations and supercomputers.

From the designer's point of view they are mainly resource manages. They allow management of the CPU, disks and I/O devices. They have to provide a user interface for computer operators, professional programmers (whether systems or applications), administrators of the system, and finally the casual end user. As can be imagined, these groups have different functional requirements. It is therefore useful to look at the development of operating systems, and see a shift from satisfying the requirements of the professional to satisfying the requirements of the casual end user.

3.1 History of operating systems

3.1.1 The 1940s

Early computer systems had no operating systems in the modern sense of the word. An early commercially available machine was the IBM 604. This machine could undertake some 60 program steps before using punch cards as backing store. The end user had intimate knowledge of the machine and programmed at a very low level.

3.1.2 The 1950s

This era saw a rapid change in the capabilities of operating systems. They were designed to make efficient use of an expensive resource. Jobs were *batched* so that the time between jobs was minimized. The end user was now distanced from the machine. This era saw rapid development in program language design and a notable end to the period was the design of Algol 60.

3.1.3 The 1960s

The next milestone was the introduction of multiprogramming. Probably initially seen as a way of making efficient use of hardware it heralded the idea of time sharing. A time sharing systems is characterized by the conversational nature of the interaction, and the use of a keyboard. This had a tremendous impact on the range of uses that a computer system had and on the program development process.

3.1.4 The 1960s and 1970s

The realization that computer systems could be used in a large range of human activities saw the development of large general purpose systems, and probably the most famous of these was the IBM 360 series. These systems were some of the most complex programming endeavours undertaken, and most projects were late and well over budget. These costly mistakes helped lead to the establishment of software engineering as a discipline.

The contribution of the time sharing system to program development was quickly realized to be considerable. A system that was developed during this period was UNIX – and this operating system has a very sharp set of tools to aid in program development.

3.1.5 The 1970s, 1980s and 1990s

The 70s represented a period of relative stability with newer and more complex versions of existing systems.

During this period the importance of graphical interfaces emerged and with dropping hardware costs graphical interfaces started to dominate.

The Apple Macintosh heralded a new era and became a popular choice for many people. At the same time graphical interfaces were being added to existing major operating systems, with X-Windows and associated higher level systems hiding raw UNIX from many users, and Microsoft Windows on the Intel family of processors.

Linux (a free Unix variant) is popular in the scientific field, and a very good alternative to DOS and Windows on the Intel family of processors.

3.2 Networking

Networking simplistically is a way of connecting two or more computer systems, and networking computer systems is not new one of the first was the SAGE military network, funded by the US DoD in the 1950s.

Networking capability has undergone a massive increase during the computer age. Local networks of two or three systems, through tens of systems in small research groups and organizations are now extremely common place. It is not uncommon now to have in excess of a thousand network connected devices on one local area network.

Wide area networking is also quite common, and most major organizations now have networks spanning a country or even the whole world.

One of the most widely used wide area networks in the academic and scientific world is the Internet, and there are over thirty million systems on the Internet at the time of writing this book.

A number of books on networking are included in the bibliography.

3.3 Problems

1. What type of system do you use, i.e. is it a stand alone microcomputer, terminal, workstation etc?

2. Is it networked, and if so in what way?

3. Is wide area networking available?

4. Is a graphical interface available?

3.4 Bibliography

Brooks F.P., *The Mythical Man Month: Essays on Software Engineering,* Addison Wesley, 1982.

> A very telling coverage of the development of the operating system for the IBM 360 series of systems. A must for any one involved in the longer term in program development.

Chivers I.D., Essential Linux Fast, Springer Verlag.

> Quick introduction to the installation and use of Linux on a pc.

Deitel H.M., *An Introduction to Operating Systems,* Addison Wesley, 1984.

> One of the most accessible books on operating systems with coverage of process management, storage management, processor management, auxiliary storage management, performance, networks and security, with case studies of the major players including UNIX, VMS, CP/M, MVS, VM, DOS and Windows.

Feit S., *TCP/IP, Architecture, Protocols, and Implementation,* McGraw Hill.

> A more technical book than Kroll, well written with a wealth of information for the more inquisitive reader.

Gilly D., Unix in a Nutshell, O'Reilly

Good Unix refererence text.

Kroll E., *The Whole Internet User's Guide and Catalog*, O'Reilly, 1994.

The Internet book. Written with very obvious enthusiasm by Mister Internet himself! Quite dated by todays standards.

Siever E., Linux in a Nutshell, O'Reilly

Good reference text on Linux.

4
Introduction to Using a Computer System

"Maybe one day we will be glad to remember even these things."

Virgil

Aims

The aims of this chapter are to introduce some of the fundamentals of using a computer system including:

Files.

Editors.

Systems access and networking.

4 Introduction to Using a Computer System

There are a number of concepts that underpin your use of any computing system. Sitting at a high resolution colour screen with a myriad of icons this may not be immediately apparent, but developing an appreciation of this will help considerably in the long term, and when inevitably moving from one system to another.

4.1 Files

A file is a collection of information that you refer to by name, e.g. if you were to use a word processor to prepare a letter then the letter would exist independently as a file on that system, generally on a disk. With graphical interfaces there will be a systematic iconic representation of files.

There will be many ways of manipulating files on the operating system that you work on. You will use an editor to make changes to a file. This file might be the source of a program and you could then use a compiler to compile your program. The compiler will generate a number of files, some of interest to you, others for its own use. There will be commands in the operating system to make copies of files, back files up onto a variety of media, etc.

4.2 Editors

All general purpose computer systems have at least one editor so that you can modify programs and data. Screen editors are by far the easiest to use, with changes you make to the file being immediately visible on the screen in front of you.

Some editors will have sophisticated command modes of operation with pattern matching allowing very powerful text processing capabilities. These can automate many common tasks, taking away the manual, repetitive drudgery of screen based editing.

4.3 Single user systems

These are becoming increasingly common, in use both at work and in the home. The IBM PC and compatibles are a very popular choice in the scientific community. They offer ease of use and access to a considerable amount of raw processing power for computer intensive applications.

4.4 Networked systems

It is quite common to interconnect the above to local and wide area networks. This same network would also have file servers, printers, plotters, mail gateways etc. Both authors have PCs with modems at home and have access via the telephone to the Internet.

Workstations are generally networked in an environment like the above, providing very powerful processing capability.

4.5 Multi-user systems

One step above microcomputers and workstations are multi-user systems. The dividing line between microcomputers, individual workstations and multi-user systems is rapidly becoming blurred.

Multi-user systems, especially the larger ones, are very popular as they take away from the casual end user much of the drudgery of the day to day tasks of backing up disks, installing new versions of the software, locating and fixing problems with software that doesn't work quite as it should, etc to a system manager or operator.

Here we find one person with the role of registering new users, backing up the file system, sorting out printer problems, networking problems, etc. This also means that all of the users

of the system do not have to remember rather arcane and sometimes rather magical commands! They can get on with solving their actual problem.

4.6 Other useful things to know

You will soon need to know what files you are working with and there will be commands to do this. There will be a need to get rid of files and there will be commands to achieve this.

There will be ways of getting on-line help, and *help* as a command is (for once!) used by a variety of operating systems. On UNIX systems the rather more unintelligible *man* command is available.

There will be commands to print program listings and data files

With networked and multiuser systems there will be commands to send and receive electronic mail to/from other users. It is easy to send and reply to mail from people across the world, often in hours and even minutes. Table 4.1 has examples of some common operating system commands in DOS, UNIX, Linux and VMS.

Operating System and Command	**DOS**	**UNIX Linux**	**VMS**
What files are there	dir	ls	dir
Get rid of a file	del	rm	del
Copy a file	copy	cp	copy
Display a file on the screen	type	cat	type
Print a file	print	pr	print
Create or make changes to a file	edit	ed vi	edit edit/tpu
Make a sub-directory	mkdir, md	md	create/dir
Change to another directory	chdir, cd	cd	set default

Table 4.1 Common Operating System Commands

4.7 Common Methods of Using Computer Systems to Develop Fortran Programs

The following are some of the ways in which you can use a computer system to develop Fortran programs:

PC running Windows and X-Windowing software to access a remote system with a Fortran compiler installed, GUI interface;

PC running Linux and X-Windows to access a remote system with a Fortran compiler installed, GUI interface;

PC running Windows and telnet or ssh to access a remote system with a Fortran compiler installed, terminal style interface;

PC running Linux and telnet or ssh to access a remote system with a Fortran compiler installed, terminal style interface;;

PC running Windows, local Fortran compiler installed;

PC running Linux, local Fortran compiler installed

Proprietary workstation, local compiler installed;

Proprietary workstation, accessing compiler on remote system;

All will have one thing in common and that is the following cycle is used:

Edit your program

Compile the program

Run the program

Check the answers

Go back and edit the program to correct the errors and repeat until the answers and what you expect!

4.8 King's Specific Information

The following applies to C++ courses given at King's College.

4.8.1 Vista Exceed and ssh accessing the central King's Unix system

Install some screen shots. The pdf can have colour versions.

4.8.2 ssh accessing the King's Unix system

Install some screen shots. The pdf can have colour versions.

4.8.3 School Installed Software - cygwin and g++

Install some screen shots. The pdf can have colour versions.

4.9 Bibliography

The main source here are the manuals and documentation provided by the supplier of whatever system you use. These are increasingly of a very high standard. However they are inevitably written to highlight the positive and downplay the negative aspects of the systems. The next source are the many third party books written and widely available throughout the world. These vary considerably in price from basic introductory coverages to very comprehensive reference works. These are a very good complement to the first. The following url is a very good source of UNIX information.

http://unixhelp.ed.ac.uk/

Chivers I.D., Essential Linux Fast, Springer Verlag.

Quick introduction to the installation and use of Linux on a pc. Bit dated.

Gilly D., *UNIX in a Nutshell*, O'Reilly and Associates, 1992.

A very good quick reference guide. Assumes some familiarity with UNIX. Current edition (at the time of writing this book) was System V Release IV, with Solaris 2.0. Also provides coverage of the various shells, Bourne, Korn and C.

Microsoft, *Windows User's Guide*, Microsoft Press.

Good coverage of Windows and suitable for the beginner and intermediate level user. Sufficient for most users. A massive improvement over earlier versions.

Siever E., Linux in a Nutshell, O'Reilly

Good reference text on Linux.

5
Introduction to Problem Solving

They constructed ladders to reach to the top of the enemy's wall, and they did this by calculating the height of the wall from the number of layers of bricks at a point which was facing in their direction and had not been plastered. The layers were counted by a lot of people at the same time, and though some were likely to get the figure wrong the majority would get it right... Thus, guessing what the thickness of a single brick was, they calculated how long their ladder would have to be.

Thucydides, The Peloponnesian War

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less"
"The question is," said Alice, "whether you can make words mean so many different things."

Lewis Carroll, Through the Looking Glass and What Alice Found There.

Aims

The aims are:

- To examine some of the ideas and concepts involved in problem solving.
- To introduce the concept of an algorithm.
- To introduce two ways of approaching algorithmic problem solving.
- To introduce the ideas involved with systems analysis and design, i.e. to show the need for pencil and paper study before using a computer system.

5 Introduction to Problem Solving

It is informative to consider some of the dictionary definitions of problem:

- A matter difficult of settlement or solution, *Chambers*.
- A question or puzzle propounded for solution, *Chambers*.
- A source of perplexity, *Chambers*.
- Doubtful or difficult question, *Oxford*.
- Proposition in which something has to be done, *Oxford*.
- A question raised for enquiry, consideration, or solution, *Webster's*.
- An intricate unsettled question, *Webster's*.

and a common thread seems to be a question that we would like answered or solved. So one of the first things to consider in problem solving is how to pose the problem. This is often not as easy as is seems. Two of the most common methods are:

- In natural language.
- An artificial language or stylized language.

Both methods have their advantages and disadvantages.

5.1 Natural language

Most people use natural language and are familiar with it, and the two most common forms are the written and spoken word. Consider the following language usage:

- The difference between a three year old child and an adult describing the world.
- The difference between the way an engineer and a physicist would approach the design of a car engine.
- The difference between a manager and worker considering the implications of the introduction of new technology.

Great care must be taken when using natural language to define a problem and a solution. It is possible that people use the same language to mean completely different things, and one must be aware of this when using natural language whilst problem solving.

Natural language can also be ambiguous:

- Old men and women eat cheese.

Are both the men and women old?

5.2 Artificial language

The two most common forms of artificial language are technical terminology and notations. Technical terminology generally includes both the use of new words and alternate use of existing words. Consider some of the concepts that are useful when examining the expansion of gases in both a theoretical and practical fashion:

- Temperature.
- Pressure.
- Mass.
- Isothermal expansion.
- Adiabatic expansion.

Now look at the following:

- A chef using a pressure cooker.
- A garage mechanic working on a car engine.
- A doctor monitoring blood pressure.
- An engineer designing a gas turbine.

Each has a particular problem to solve, and each will approach their problem in their own way; thus they will each use the same terminology in slightly different ways.

5.2.1 Notations

Some examples of notations are:

- Algebra.
- Calculus.
- Logic.

All of the above have been used as notations for describing both problems and their solutions.

5.3 Resumé

We therefore have two ways of describing problems and they both have a learning phase until we achieve sufficient understanding to use them effectively. Having arrived at a satisfactory problem statement we next have to consider how we get the solution. It is here that the power of the algorithmic approach becomes useful.

5.4 Algorithms

An algorithm is a sequence of steps that will solve part or all of a problem. One of the most easily understood examples of an algorithm is a recipe. Most people have done some cooking, if only making toast and boiling an egg.

A recipe is made up of two parts:

- A check list of things you need.
- The sequence or order of steps.

Problems can occur at both stages, e.g. finding out halfway through the recipe that you do not have an ingredient or utensil; finding out that one stage will take an hour when the rest will be ready in ten minutes. Note that certain things can be done in any order – it may not make any difference if you prepare the potatoes before the carrots.

There are two ways of approaching problem solving when using a computer. They both involve *algorithms*, but are very different from one another. They are called *top-down* and *bottom-up*.

5.4.1 Top-down

In a *top down* approach the problem is first specified at a high or general level: prepare a meal. It is then refined until each step in the solution is explicit and in the correct sequence, e.g. peel and slice the onions, then brown in a frying pan before adding the beef. One drawback to this approach is that it is very difficult to teach to beginners because they rarely have any idea of what *primitive* tools they have at their disposal. Another drawback is that they often get the sequencing wrong, e.g. *now place in a moderately hot oven* is frustrating because you may have not lit the oven (sequencing problem) and secondly because you may have no idea how hot *moderately hot* really is. However as more and more problems are tackled top-down becomes one of the most effective methods for programming.

5.4.2 Bottom-up

Bottom-up is the reverse to top-down! As before you start by defining the problem at a high level, e.g. prepare a meal. However, now there is an examination of what tools, etc. you have available to solve the problem. This method lends itself to teaching since a repertoire of tools can be built up and more complicated problems can be tackled. Thinking back to the recipe there is not much point trying to cook a six course meal if the only thing that you can do is boil an egg and open a tin of beans. The bottom-up approach thus has advantages for the beginner. However there may be a problem when no suitable tool is available. One of the authors' colleague and friend learned how to make Bechamel sauce, and was so pleased by his success that every other meal had a course with a Bechamel sauce. Try it on your eggs one morning. Here was a case of specifying a problem *prepare a meal,* and using an inappropriate but plausible tool *Bechamel sauce.*

The effort involved in tackling a realistic problem, introducing the constructs as and when they are needed and solving it is considerable. This approach may not lead to a reasonably comprehensive coverage of the language, or be particularly useful from a teaching point of view. Case studies do have great value, but it helps if you know the elementary rules before you start on them. Imagine learning French by studying Balzac, before you even look at a French grammar. You can learn this way but even when you have finished, you may not be able to speak to a Frenchman and be understood. A good example of the case study approach is given in the book *Software Tools,* by Kernighan and Plauger.

In this book our aim is to gradually introduce more and more tools until you know enough to approach the problem using the top-down method, and also realize from time to time that it will be necessary to develop some new tools.

5.4.3 Stepwise refinement

Both the above techniques can be combined with what is called *step-wise refinement.* The original ideas behind this technique are well expressed in a paper by Wirth entitled *Program Development by Stepwise Refinement,* published in 1971. This means that you start with a global problem statement and break the problem down in stages, into smaller and smaller sub-problems, that become more and more amenable to solution. When you first start programming the problems you can solve are quite simple, but as your experience grows you will find that you can handle more complex problems.

When you think of the way that you solve problems you will probably realize that, unless the problem is so simple that you can answer it straight away some thinking and pencil and paper work is required. An example that some may be familiar with is in practical work in a scientific discipline, where coming unprepared to the situation can be very frustrating and unrewarding. It is therefore appropriate to look at ways of doing analysis and design before using a computer.

5.4.4 Modular programming

As the problems we try solving become more complex we need to look at ways of managing the construction of programs that comprise many parts. Modula 2 was one of the first languages to support this methodology and we will look at modular programming in more depth in the subsequent chapter.

5.4.5 Object oriented programming

There are a class of problems that are best solved by the treatment of the components of these problems as objects. We will look at the concepts involved in object oriented programming and object oriented languages in the next chapter.

5.5 Systems analysis and design

When one starts programming it is generally not apparent that one needs a methodology to follow to become successful as a programmer. This is generally because the problems are reasonably simple, and it is not necessary to make explicit all of the stages one has gone through in arriving at a solution. As the problems become more complex it is necessary to become more rigorous and thorough in one's approach, to keep control in the face of the increasing complexity and to avoid making mistakes. It is then that the benefit of systems analysis and design becomes obvious. Broadly we have the following stages in systems analysis and design:

- Problem definition.
- Feasibility study and fact finding.
- Analysis.
- Initial system design.
- Detailed design.
- Implementation.
- Evaluation.
- Maintenance.

and each problem we address will entail slightly different time spent in each of these stages. Let us look at each stage in more detail.

5.5.1 Problem definition

Here we are interested in defining what the problem really is. We should aim at providing some restriction on both the scope of the problem, and the objectives we set ourselves. We can use the methods mentioned earlier to help out. It is essential that the objectives are:

- Clearly defined.
- When more than one person is involved, understood by all people concerned, and agreed by all people concerned.
- Realistic.

5.5.2 Feasibility study and fact finding

Here we look to see if there is a feasible solution. We would try and estimate the cost of solving the problem and see if the investment was warranted by the benefits, i.e. cost benefit analysis.

5.5.3 Analysis

Here we look at what must be done to solve the problem. Note we are interested in finding what we need to do, but we do not actually do it at this stage.

5.5.4 Design

Once the analysis is complete we know what must be done, and we can proceed to the design. We may find there are several alternatives, and we thus examine alternate ways in which the problem can be solved. It is here that we use the techniques of top-down and bottom-up problem solving, combined with step-wise refinement to generate an algorithm to solve the problem. We are now moving from the logical to the physical side of the solution. This stage ends with a choice between one of several alternatives. Note that there is generally not one ideal solution, but several, each with their own advantages and disadvantages.

5.5.5 Detailed design

Here we move from the general to the specific, The end result of this stage should be a sufficiently tightly defined specification to generate actual program code from.

It is at this stage that it is useful to generate *pseudo-code*. This means writing out in detail the actions we want carried out at each stage of our overall algorithm. We gradually expand each stage (step-wise refinement) until it becomes Fortran – or whatever language we want in fact.

5.5.6 Implementation

It is at this stage that we actually use a computer system to create the program(s) that will solve the problem. It is here that we actually need to know sufficient about a programming language to use it effectively to solve our problems. This is only one stage in the overall process, and mistakes at any of the stages can create severe difficulties.

5.5.7 Evaluation and testing

Here we try to see if the program(s) we have produced actually do what they are supposed to. We need to have data sets that enable us to say with confidence that the program really does work. This may not be an easy task, as quite often we only have numeric methods to solve the problem, which is why we are using the computer to solve the problem – hence we are relying on the computer to provide the proof; i.e. we have to use a computer to determine the veracity of the programs – and as Heller says *Catch 22.*

5.5.8 Maintenance

It is rare that a program is run once and thrown away. This means that there will be an on-going task of maintaining the program, generally to make it work with different versions of the operating system, compiler, and to incorporate new features not included in the original design. It often seems odd when one starts programming that a program will need maintenance as we are reluctant to regard a program in the same way as a mechanical object like a car that will eventually fall apart through use. Thus maintenance means keeping the program working at some tolerable level, with often a high level of investment in manpower and resources. Research in this area has shown that anything up to 80% of the manpower investment in a program can be in maintenance.

5.6 Conclusions

A drawback, inherent in all approaches to programming, and to problem solving in general, is the assumption that a solution is indeed possible. There are problems which are simply insoluble – not only problems like balancing a national budget, weather forecasting for a year, or predicting which radioactive atom will decay, but also problems which are apparently computationally solvable.

Knuth gives the example of a chess problem – determining whether the game is a forced victory for white. Although there is an algorithm to achieve this, it requires an inordinately large amount of time to complete. For practical purposes it is unsolvable.

Other problems can be shown mathematically to be undecidable. The work of Gödel in this area has been of enormous importance, and the bibliography contains a number of references for the more inquisitive and mathematically orientated reader. The Hofstader coverage is the easiest, and least mathematical.

As far as possible we will restrict ourselves to solvable problems, like learning a programming language.

Within the formal world of Computer Science our description of an algorithm would be considered a little lax. For our introductory needs it is sufficient, but a more rigorous ap-

proach is given by Hopcroft and Ullman in *Introduction to Automata Theory, Languages and Computation*, and by Beckman in *Mathematical Foundations of Programming.*

5.7 Problems

1. What is an algorithm?

2. What distinguishes top-down from bottom-up approaches to problem solving? Illustrate your answer with reference to the problem of a car, motor-cycle or bicycle having a flat tire.

5.8 Bibliography

Aho A.V., Hopcroft J.E., Ullman J.D., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1982.

Theoretical coverage of the design and analysis of computer algorithms.

Beckman F.S., *Mathematical Foundations of Programming*, Addison Wesley, 1981

Good clear coverage of the theoretical basis of computing.

Bulloff J.J., Holyoke T.C., Hahn S.W., *Foundations of Mathematics – Symposium Papers Commemorating the 60th Birthday of Kurt Gödel*, Springer Verlag, 1969.

The comment by John von Neumann highlights the importance of Gödel's work, *.. Kurt Gödel's achievement in modern logic is singular and monumental – indeed it is more than a monument, it is a landmark which will remain visible far in space and time. Whether anything comparable to it has occurred in the logic of modern times may be debated. In any case, the conceivable proxima are very, very few. The subject of logic has certainly changed its nature and possibilities with Gödel's achievement.*

Dahl O.J., Dijkstra E.W., Hoare C.A.R., *Structured Programming*, Academic Press, 1972.

This is the seminal book on structured programming.

Davis M., *Computability and Unsolvability*, Dover, 1982.

The book is an introduction to the theory of computability and non-computability – the theory of recursive functions in mathematics. Not for the mathematically faint hearted!

Davis W.S., *Systems Analysis and Design*, Addison Wesley, 1983.

Good introduction to systems analysis and design, with a variety of case studies. Also looks at some of the tools available to the systems analyst.

Fogelin R.J., *Wittgenstein*, Routledge and Kegan Paul, 1980.

The book provides a gentle introduction to the work of the philosopher Wittgenstein, who examined some of the philosophical problems associated with logic and reason.

Gödel K., *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, Oliver and Boyd, 1962.

An English translation of Gödel's original paper by Meltzer, with quite a lengthy introduction by R.B. Braithwaite, then Knightbridge Professor of Moral Philosophy at Cambridge University, England, and classified under philosophy at the library at King's, rather than mathematics.

Hofstadter D.,*The Eternal Golden Braid*, Harvester Press, 1979.

A very readable coverage of paradox and contradiction in art, music and logic, looking at the work of Escher, Bach and Gödel respectively.

Hopcroft J.E., Ullman J.D., *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.

Comprehensive coverage of the theoretical basis of computing.

Kernighan B.W., Plauger P.J., *Software Tools*, Addison Wesley, 1976.

Interesting essays on the program development process, originally using a non-standard variant of Fortran. Also available using Pascal.

Knuth D.E., *The Art of Computer Programming*, Addison Wesley,

Vol 1. *Fundamental Algorithms*, 1974

Vol 2. *Semi-numerical Algorithms*, 1978

Vol 3. *Sorting and Searching*, 1972

Contains interesting insights into many aspects of algorithm design. Good source of specialist algorithms, and Knuth writes with obvious and infectious enthusiasm (and erudition).

Millington D., *Systems Analysis and Design for Computer Applications*, Ellis Horwood, 1981.

Short and readable introduction to systems analysis and design.

Wirth N., *Program Development by Stepwise Refinement*, Communications of the ACM, April 1971, Volume 14, Number 4, pp. 221-227.

Clear and simple exposition of the ideas of stepwise refinement.

6
Introduction to Programming in C++

‘Though this be madness, yet there is method in’t’
Shakespeare.

‘Plenty of practice’ he went on repeating, all the time that Alice was getting him on his feet again. ‘plenty of practice.’
The White Knight, Through the Looking Glass and What Alice Found There, Lewis Carroll.

Aims

The aims of the chapter are:–

to look at a simple hello world example;

to look at a simple string example using C style arrays of char;

to look at a simple string example using C++ style string objects;

to look at a simple numeric example;

to introduce some of the formal syntactical rules of C++;

to look at the C++ keywords;

to provide definitions of some technical terms used throughout the rest of the notes;

to look at the C++ memory model;

to look at the C++ character set;

6 An Introduction to C++

In this chapter we will look at some simple program examples. The examples throughout the notes exist in two forms:–

- old c style
- new c++ style

and whilst you should adopt the new C++ style for writing all new programs it is necessary to know about the old style to work with existing programs and libraries.

We will also look at some of the syntax of C++ programs.

6.1 Elements of a programming language

As with ordinary (so-called *natural*) languages, e.g. English, French, Gaelic German, etc., programming languages have rules of syntax, grammar and spelling. The application of the rules of syntax, grammar and spelling in a programming language are more strict. A program has to be unambiguous, since it is a *precise* statement of the actions to be taken. Many everyday activities are rather vaguely defined — *Buy some bread on your way home* — but we are generally sufficiently adaptable to cope with the variations which occur as a result. If, in a program to calculate wages, we had an instruction *Deduct some money* for tax and insurance we could have an awkward problem when the program calculated completely different wages for the same person for the same amount of work every time it was run. One of the implications of the strict syntax of a programming language for the novice is that apparently silly error messages will appear when first writing programs. As with many other *new* subjects you will have to learn some of the jargon to understand these messages.

Programming languages are made up of statements. We will look at the various kinds of statements briefly below.

6.1.1 Data description statements

These are necessary to describe what kinds of data are to be processed. In the wages program for example, there is obviously a difference between peoples names and the amount of money they earn, i.e. these two things are not the same, and it would not make any sense adding your name to your wages. The technical term for this is *data type;* a wage would be of a different data type (a number) to a surname (a sequence of characters).

6.1.2 Control structures

A program can be regarded as a sequence of statements to solve a particular problem, and it is common to find that this sequence needs to be varied in practice. Consider again the wages program. It will need to select between a variety of circumstances (say married or single, paid weekly or monthly etc), and also to repeat the program for everybody employed. So there is the need in a programming language for statements to vary and/or repeat a sequence of statements.

6.1.3 Data processing statements

It is necessary in a programming language to be able to process data. The kind of processing required will depend on the kind or type of data. In the wages program, for example, you will need to distinguish between names and wages. Therefore there must be different kinds of statements to manipulate the different types of data, i.e. *wages* and *names*.

6.1.4 Input and output (I/O) statements

For flexibility, programs are generally written so that the data that they work on exists *outside* the program. In the wages example the details for each person employed would exist in

a *file* somewhere, and there would be a *record* for each person in this file. This means that the program would not have to be modified each time a person left, was ill, etc., although the individual records might be updated. It is easier to modify data than to modify a program, and less likely to produce unexpected results. To be able to vary the action there must be some mechanism in a programming language for getting the data *into* and *out of* the program. This is done using input and output statements, sometimes shortened to I/O statements.

6.2 Hello World – New standard C++ style

The following is the new style.

```
#include <iostream>
using namespace std;
int main()
{
  cout << " Hello World " << endl;
  return(0);
}
```

Let us look at each line in turn.

```
#include <iostream>
```

C++ follows in the footsteps of C where there is the concept of the kernel of the language and an additional set of library routines. The `#include` line is an instruction to the compiler to make available to the following program unit what is defined in `iostream`.. There is the concept of compiler preprocessing in C and C++ programs. The # indicates a preprocessor directive. The <> characters are used to indicate a *standard* language header file, in this case *iostream.*. I/O is not available in the kernel of the language. It is made available by the inclusion of `iostream` in the complete program.

```
using namespace std;
```

The C++ standard has introduced the concept of a namespace. Namespaces provide a way of grouping related items. They are analagous to the functionality provided by modules in other programming languages. The above line makes available the standard namespace, called std. Without this line the above program would have to be rewritten as:–

```
#include <iostream>
int main()
{
  std::cout << " Hello World " << std::endl;
  return(0);
}
```

Here we explicitly qualify *cout* and *endl* to show that they are to be found in the standard namespace.

```
int main()
```

The next line is the start of the program itself. All programs are called *main* in C++. There is also the concept of every program unit being a function. Functions in C++ either return a value (and in this case we are defining main to return an integer value) or not. If we do not want a function to return a value (Pascal procedure or Fortran subroutine) we use *void* rather than a data type in conjunction with the function name to indicate this.

The next thing of interest is the { character which indicates the start of the program.

The next statement (pronounced see out) prints some text to the standard out stream or screen, in this case *Hello World.*

Text is delimited in C++ with " marks.

endl is predefined in C++ to generate an end of line.

The << symbols are classified in C++ as an operator. They are used to separate items in the output stream.

; is the statement separator in C++.

Finally the program terminates with the `return(0)` statement. When calling functions one is used to them returning a value. In the case of the special `main` function, the C++ language returns the value 0 to the operating system level. This is very useful when the overall problem may call for several programs to work together.

The } character signifies the end of the program.

There is also another variant way of writing this program and this is shown below.

```
#include <iostream>
using std::cout ;
using std::endl ;
int main()
{
  cout << " Hello World " << endl;
  return(0);
}
```

In this version we explicitly name the components from the standard namespace that we are interested in, i.e.

```
std::cout
std::endl
```

We will use the first form in most of the examples.

6.3 Hello World – Old C style

The following is a complete old style C++ program.

```
#include <iostream.h>
int main()
{
  cout << " Hello World " << endl;
  return(0);
}
```

Try compiling this program with your compiler. Better compilers will issue warnings whilst compiling this program about the preferred standard way of working. Here is the output from the g++ compiler under Windows XP Professional.

```
$ g++ ch0604.cxx
In file included from /usr/include/c++/3.3.1/back-
ward/iostream.h:31,
                 from ch0604.cxx:1:
/usr/include/c++/3.3.1/backward/backward_warning.h:32:2: warn-
ing: #warning This file includes at least one deprecated or
```

```
antiquated
header. Please consider using one of the 32 headers found in
section 17.4.1.2 of the C++ standard. Examples include

substituting the
 <X> header for the <X.h> header for C++ includes, or
<sstream> instead of the deprecated header <strstream.h>. To
disable

this warn
ing use -Wno-deprecated.


Ian@rowan /cygdrive/d/document/cpp
```

6.4 Simple text i/o using C style arrays of char

The following is a complete, old style, C++ program.

```
#include <iostream>
using namespace std;
int main()
{
char name[20] ;
  cout << " Type in your name \n " ;
  cin >> name ;
  cout << name ;
  return(0);
}
```

We then have a variable declaration. In this case we are defining name to be a character variable and that it can hold up to 20 characters. Note the use of [] in C++ instead of () in Fortran. This helps functions stand out quite clearly from arrays.

Statements within the body of the program (or function) are separated by a semicolon ;.

The next statement `cout` prints some text to the screen. In this case the text `Type in your name` will appear on the screen. The \n within the double quotes generates a new line. This is an alternate way of generating a new line. This was inheritted from the C programming language.

The next statement `cin` (pronounced see in) reads the text that the user types in and assigns it to the character array `name`. A blank is a separator, so the input terminates with the first blank character the user types.

The next `cout` statement echos what the user has typed back to the screen.

Finally the program terminates with the `return(0)` statement. When calling functions one is used to them returning a value. In the case of the special `main` function, the C++ language returns the value 0 to the operating system level. This is very useful when the overall problem may call for several programs to work together.

The } character signifies the end of the program.

When running this example try typing in more than 20 characters. It is quite likely that the program will allow you to type in more than 20 characters and echo them back to you on the screen. This means that an area of computer memory has been overwritten by these ex-

tra characters, and no error has has been reported. This is one feature of C that makes it more unreliable than other languages.

6.5 Variables – name, type and value

The idea of a variable is one that you are likely to have met before, probably in a mathematical context. Consider the following

- circumference = 2 r

This is an equation for the calculation of the circumference of a circle. The following represents a translation of this into a programming language.

- circumference = 2 * pi * radius

There are a number of things to note about the above:

- Each of the *variables* on the right hand side of the equals sign (pi and radius) will have a *value*, which will allow the evaluation of the expression.When the expression is fully evaluated the value is assigned to the variable on the left hand side of the equals sign.
- In mathemetics the multiplication is implied, in most programming languages we have to use the * operator to indicate that we want to multiply 2 by pi by the radius.
- We do not have access to mathematical symbols like in programming languages but have to use variable names based on letters from the Roman alphabet.

The whole line is an example of an *arithmetic assignment statement* in C++.

The following arithmetic assignment statement illustrates clearly the concepts of name and value, and the difference between the = in mathematics and computing:

I = I + 1

In C++ this reads as take the current value of the variable *I* and add one to it, store the new value back into the variable *I*, i.e. *I* takes the value *I+1*. Algebraically,

i = i + 1

does not make any sense.

Variables can be of different types, and Table 6.1 shows some of those available in C++.

Variable Name	Data Type	Value Stored
Temperature	FLOAT	28.55
Number_of_People	INT	100
First_Name	CHAR	J

Table 6.1 Variable, type and value

Note the use of capitalisation and under scores to make the variable names easier to read.

The concept of data type seems a little strange at first, especially as we commonly think of integers and reals as numbers. However, the benefits to be gained from this distinction are considerable. This will become apparent after writing several programs.

Numeric data is normally stored in a format based on the IEEE 754 standard. This is covered in more depth in later chapters. Floats are stored in 32 bits in a mantissa exponent format. Int is stored in a binary format in 32 bits.

6.6 Simple text i/o using C++ style strings

This example uses the new standard conformant style, and uses the new data type – *string*.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string s;
  cout << " Type in your name " ;
  cin >> s ;
  cout << s << endl;
  return(0);
}
```

Note again the omission of the ".h" within the <> brackets.

Note also the lack of an explicit size for the string variable s. A string is a fully dynamic data type. This is a considerable advantage over the old C style array of char. We will look into old and new style strings in greater depth in a later chapter.

6.7 Simple numeric i/o

This example reads in three numbers from the user and sums and averages them and then prints out the sum and average.

```
#include <iostream>
using namespace std;

int main()
{
float a,b,c,sum=0.0, average=0.0 ;
int n=3 ;
  cout << " Type in three numbers \n" ;
  cin >> a >> b >> c ;
  sum = a + b + c ;
  average = sum / n ;
  cout << " Numbers were " ;
  cout << a  << "  " << b << " "  << c <<  " \n" ;
  cout << " Sum is " << sum << " \n" ;
  cout << " Average is " << average << "\n" ;
  return(0) ;
}
```

The first line `#include <iostream>` makes available `cin` and `cout` so that we can interact with the program.

The next line makes available the standard namespace.

The next line is the start of the program.

The { indicates the start of the declaration and executable part of the program.

The next two lines declare the variables we will be using within the program and their types. In this example we will be working with numeric data, and thus we have `a, b, c, sum` and `average` to be of real type, or `float` in C++ terminology. n is of type integer or `int` in C++ terminology.

The semicolon ; is the statement separator.

Note that we also provide initial values for sum, average and n within the declarations. The = symbol is an assignment operator.

`cout << " Type in three numbers \n"` prompts the user for input. Note the use again of \n to provide a newline.

`cin >> a >> b >> c` reads three numbers from the input stream. Spaces are valid separators. Note the use of multiple >> symbols to break up the input stream, not a comma as in other languages.

`sum = a + b + c` adds the values of a, b and c together and assigns the result to sum.

`average = sum / n` calculates the sum divided by n and assigns the result to average.

`cout << " Numbers were "` prints out a text message.

`cout << a  << "  " << b << " "  << c <<  " \n"` echos the numbers back to the user. Note that we use multiple << symbols to break up the output stream, not a comma as in other languages.

`cout << " Sum is " << sum << " \n"` prints out the sum.

`cout << " Average is " << average << "\n"` prints out the average.

`return(0)` ends the program and returns the value 0 to the operating system level.

} is the end of the program.

We have the conventional use of + and / in arithmetic expressions.

6.8 Some C++ Rules and Terminology

Simple programs have the following format:–

```
#include <iostream>
using namespace std;

int main()
{ //   declarations first

   int i;       // i declared as integer
   float f;     // f declared as float
   char x[10]; // x declared as an array of char
   string s;   // s declared as a string

    /*    executable statements go next

    */

    return(0);

}
```

Variable declarations are normally grouped together at the beginning of the program.

Two or three spaces are normally used to indent sections of C++ code.

Case is significant in C++.

There are two ways of providing comments in a C++ program. The first uses the // characters and is restricted to one line. The second uses /* to indicate the start of a comment and */ to indicate the end. The second form can span several lines.

An identifier in C++ is a sequence of one or more characters drawn from the letters, digits and the underscore character (_). Case is significant. They must start with a letter or underscore character. There is no upper bound on the number of characters allowed in an identifier. In practice given a screen width of 80 characters 20 to 30 is a sensible upper bound.

White space characters include blank, horizontal and vertical tab, form feed and new line and comments.

C++ keywords are given below.

6.9 Good Programming Guidelines

Every language has its own style. It is advisable to adopt a style that one is comfortable with that draws on ones experience of other languages, and also is similar to the notational style used by C++ texts. It is inevitable that one will end up working with algorithms and programs already written in C++, and thus one has to be familiar with the conventional C++ style of writing programs. Use indentation and white space to make a program more understandable.

6.10 C++ for C Programmers

The following are some of Stroustrup's guidelines for C programmers moving to C++.

6.10.1 Macros

Use const and enum to define constants; inline to avoid a function call overhead; templates to specify families of functions and types; namespaces to avoid name clashes.

6.10.2 Malloc

Use new instead of malloc and vector instead of realloc.

6.10.3 Pointers

Avoid void*, pointer arithmetic, unions and casts.

6.10.4 Arrays and C style strings

Use the C++ STL string and vector classes.

6.11 C++ Character Set

Section r.2.4 of the standard only mentions the ASCII character set. In English countries it is probably safe to assume the that this character set will be the one available. Figure 2 shows the ASCII character set.

A new standard is becoming increasingly popular where multiple language support is required. This is called UNICODE. This is a sixteen bit character set. C++ offers support in this area via wide characters. There is further coverage of this in the chapter on string and character data types.

EBCDIC is another widely used character set, predominantly on IBM mainframe systems under non UNIX operating systems.

We will look into the whole area of characters, character sets and strings in much greater depth in a later chapter.

6.12 Summary

Don't be put off by the syntax of C++. It doesn't take long to get on top of that syntax. You wouldn't expect natural languages to have identical syntax and semantics, so why expect it from programming languages.

6.13 Key Concepts

The key concepts introduced in this chapter are summarised below.

6.13.1 Basic structure of a C++ program

```
#include <iostream>
using namespace std;
int  main()
{
  declaration 1;
  declaration 2;
  .
  .
    execution statement 1;
    execution statement 2;
    .
    .
    return (0);
```

```
}
```

In English:–

there will normally be one or more #include lines;

the program is always called main;

the body of the program starts after the {;

there will normally be one or more delarations;

the ; is the statement terminator;

there will normally be one or more executable statements;

the program will end with a return(0) typically;

the program ends with the }

6.13.2 Indentation

Indentation is used to highlight the structure of the program and make it more readable by a human being.

6.13.3 Data Types

The following data types were introduced in this chapter:–

- int – integer, i.e. no decimal point;
- float – floating point;
- char– character
- string– sequence of characters;

We will look at these data types in more depth in later chapters.

6.14 Problems

1. Compile and run the examples in this chapter. The easiest thing to do is cut and paste the programs from the notes into an editor. You will invariably make mistakes doing this.. Look at the error messages that the compilers give you. The compilers give error messages from their view point. Make an attempt to understand what this means.

Try the old and new style variants. This will provide some information on how standard conformant your compiler(s) is(are).

2. What happens with the examples with characters outside of the ASCII character set?

What happens when you type in more than 20 characters? Why do you think that is? Is there a compiler switch to trap errors like this?

Does the screen representation of these characters match the printed representation? Why do you think that is?

With the string example how many characters can you type in?

3. With the examples what happens to so called *white space*, i.e spaces, tabs, and carriage returns?

4. With the last example experiment with the number format, i.e. use integers, reals, exponential format. What happens?

Modify the program to work with five numbers.

5. Write a program that will read in your name and address and print them out in reverse order.

How many lines are there in your name and address? What is the maximum number of characters in the longest line in your name and address? What happens at the first blank character of each input line? Are there any characters that can be used to enclose each line of text and hence not stop at the first blank character?

7
Arithmetic and Expressions in C++

Taking Three as the subject to reason about —
  A convenient number to state —
We add Seven, and Ten, and then multiply out
  By One Thousand diminished by Eight.
The result we proceed to divide, as you see,
  By Nine Hundred and Ninety and Two:
Then subtract Seventeen, and the answer must be
  Exactly and perfectly true.

Lewis Carroll, The Hunting of the Snark

Round numbers are always false.

Samuel Johnson.

Aims

The aims of this chapter are to introduce:–

- the numeric data types available in C++;
- the rules for the evaluation of arithmetic expressions;
- the rules that apply in type conversion;
- const attribute;
- to introduce briefly all of the 45 operators in C++;
- to introduce the concept of a sequence point;
- to introduce the twin concepts of lvalue and rvalue;

7 Arithmetic and Expressions in C++

This chapter looks at the fundamental numeric data types in C++ and the rules for expression evaluation. There are a bewildering number of operators at first sight. A thorough understanding of these operators is essential for successful use of C++. Persevere!

7.1 Basic numeric types

C++ supports the two numeric data types we are familiar with from other programming languages, i.e. integer and real. It also supports complex arithmetic through the complex class.

7.2 Integer Numeric Type

The standard requires three types of integers, *short int*, *int* and *long int*. The standard header file <climits> contains details of what is available for an implementation. The standard does not require that they are in fact different from one another.

7.2.1 Variations on a theme - signed, unsigned

For each of the three integer types there exists corresponding unsigned types. The storage requirements are the same as for the corresponding signed type.

They obey the laws of arithmetic modulo 2^n, where n is the number of bits in the implementation, and unsigned arithmetic does not overflow.

Try running the following program on the system you use. What do you think will happen? Also look at <climits> to see what is defined there. An on-line reference is available at the Dinkumware site.

http://www.dinkumware.com/manuals/reader.aspx?lib=cpp

```
#include <iostream>
using namespace std;
int main()
{
  short int a_s = 1;
  int a = 1;
  long int a_l = 1;
  unsigned short int a_s_u = 1;
  unsigned int a_u = 1;
  unsigned long int a_l_u = 1;
    cout << a_s << "   " << a << "   " << a_l << "   ";
    cout << a_s_u << "   " << a_u ;
    cout << "   " << a_l_u <<  endl;
    for (int i=0 ; i < 40 ; ++i)
    {
      a_s = a_s*2;
      a = a *2;
      a_l = a_l*2;
      a_s_u = a_s_u*2;
      a_u = a_u  *2;
      a_l_u = a_l_u*2;
      cout << a_s << "   " << a << "   " << a_l << "   ";
      cout << a_s_u << "   " << a_u ;
      cout << "   " << a_l_u <<  endl;
      cout << endl;
    }
```

```
  return(0);
}
```

7.3 Real Numeric Type

There are three real numeric types, *float*, *double* and *long double*. The standard does not require that they are different from one another. The standard header file <cfloat> contains details of what is available for an implementation.

The standard states *The handling of overflow and divide check in expression evaluation is implementation dependent. Most existing implementations of C++ ignore integer overflows. Treatment of division by zero and all floating point exceptions vary among machines and is usually adjustable by a library function.*

The following program provides an example of the use of each real data type supported in C++.

```
#include <iostream>

using namespace std;

int main()
{
float f = 1.0/7;
double f_d = 1.0/7;
long double f_l = 1.0/7;
  cout.precision(20);
  cout << f << endl;
  cout << 1.0/7 << endl;
  cout << f_d << endl;
  cout << f_l << endl;
  f = 1.0f/7;
  cout << f << endl;
  return(0);
}
```

Do you get any warnings when compiling this example? Here is the compiler warning from the MIcrosoft C++ compiler under .NET.

```
d:\document\cpp\examples\ch07\ch0702.cxx(7) : warning C4305:
'initializing' : truncation from 'double' to 'float'
```

The warning from the Microsoft compiler is about the calculation being carried out in double precision (the numeric constant 1.0 is double precision in C++) is then being assigned to a variable of type float or single precision. Henece there is a loss of information. The output below illustrates this. Compare the first two lines.

Here is the output from the g++ compiler from running the program.

0.1428571492433548

0.14285714285714285

0.14285714285714285

0.14285714285714284921

0.1428571492433548

Here is the output from the Microsoft compiler.

0.1428571492433548

0.14285714285714285

0.14285714285714285

0.14285714285714285

0.1428571492433548

The g++ compiler supports an extended floating point precision. Have a look at <cfloat> We will look at this again in the chapter on IEEE arithmetic.

7.4 Numeric Type Conversion Rules

The following type conversion rules apply in mixed mode expressions:–

- if either operand is of type long double, the other shall be converted to long double
- otherwise, if either operand is double, the other shall be converted to double
- otherwise, if either operand is float, the other shall be converted to float
- otherwise the integral promotions (r4.5) shall be performed on both operands
- then, if either operand is unsigned long the other shall be converted to unsigned long;
- otherwise, if one operand is a long int and the other unsigned int, then if a long int can represent all the values of an unsigned int, the unsigned int shall be converted to a long int; otherwise both operands shall be converted to unsigned long int;
- otherwise, if either operand is long, the other shall be converted to long;
- otherwise, if either operand is unsigned, the other shall be converted to unsigned

7.5 Complex?

This is not provided in the base language. It is available through `<complex>` We will look at this later. Things are slightly problematic in this area at this time as not all compilers support the full range of include files from the 1997 standard.

7.6 const

An integer constant is of the first type that can represent that type.

A real constant is of type double by default. The standard maths library provided with C++ is of type double throughout.

7.7 Range, precision and size of numbers

The range on integer numbers and the the precision and the size of floating point numbers in computing is directly related to the number of bits allocated to their internal representation. Tables 8.2 and 8.3 summarizes this information for the two most common bit sizes in use for integers and reals, 32 bits and 64 bits. The first table looks at integer numbers.

N Bits		Maximum Integer
64	(2**63)–1	9,223,372,036,854,774,807
32	(2**31)–1	2,147,483,647

Table 7.1 Word Size and Integers Numbers

Here is the corresponding table for real numbers.

N Bits	Precision	Smallest real Largest real
64	15-18	~ 0.5E–308
		~ 0.8E+308
32	6-9	~ 0.3E–38
		~ 1.7E38

Table 7.2 Word Size and Real Numbers

Note that access to what the hardware supports is dependent on the operating system and compiler as well.

Precision is not the same as accuracy. In this age of digital time-keeping, it is easy to provide an extremely precise answer to the question *What time is it*? This answer need not be accurate, even though it is reported to tenths (or even hundredths!) of a second. Do not be fooled into believing that an answer reported to ten places of decimals must be accurate to ten places of decimals. The computer can only retain a limited precision. When calculations are performed, this limitation will tend to generate inaccuracies in the result. The estimation of such inaccuracies is the domain of the branch of mathematics known as Numerical Analysis.

To give some idea of the problems, consider an imaginary *decimal* computer, which retains two significant digits in its calculations. For example, 1.2, 12.0, 120.0 and 0.12 are all given to two digit precision. Note therefore that 1234.5 would be represented as 1200.0 in this device. When any arithmetic operation is carried out, the result (including any intermediate calculations) will have two significant digits. Thus:

130 + 12 = 140 (rounding down from 142)

and similarly:

17 / 3 = 5.7 (rounding up from 5.666666...)

and:

16 * 16 = 260

Where there are more involved calculations, the results can become even less attractive. Assume we wish to evaluate:

(16 * 16) / 0.14

We would like an answer in the region of 1828.5718, or, to two significant digits, 1800.0. If we evaluate the terms within the brackets first, the answer is 260/0.14, or 1857.1428; 1900.0 on the two digit machine. Thinking that we could do better, we could re-write the fraction as:

(16 / 0.14) * 16

This gives a result of 1800.0.

Algebra shows that all these evaluations are equivalent if unlimited precision is available.

Care should also be taken when is one is near the numerical limits of the machine. Consider the following:

```
Z = B * C / D
```

where B, C and D are all 10^{30} and we are using 32 bit floating point numbers where the maximum real is approximately 10^{38}. Here the product B * C generates a number of 10^{60} – beyond the limits of the machine. This is called *overflow* as the number is too large. Note that we could avoid this problem by retyping this as:

```
Z = B * (C / D)
```

where the intermediate result would now be $10^{30}/10^{30}$, i.e. 1.

There is an inverse called *underflow* when the number is too small, which is illustrated below:

```
Z = X1 * Y1 * Z1
```

where X1 and Y1 are 10^{-20} and Z1 is 10^{20}. The intermediate result of X1 * Y1 is 10^{-40} – again beyond the limits of the machine. This problem could have been overcome by retyping as:

```
Z = X1 * (Y1 * Z1)
```

This is a particular problem for many scientists and engineers with all machines that use 32-bit arithmetic for integer and real calculations. This is because many physical constants, etc are around the limits of the magnitude (large or small) supported by single precision. This is rarely a problem with machines with hardware support for 64-bit arithmetic.

7.7.1 Real numbers getting too big.

The following program

```
#include <iostream>
using namespace std;

int main()
{
float f = 10.0;
double f_d = 10.0;
long double f_l = 10.0;
  cout.precision(35);
  for (int i=0 ; i < 5000 ; ++i)
  {
    cout << f << " " << f_d << " " << f_l << endl;
    f   = f*10;
    f_d = f_d*10;
```

```
    f_l = f_l*10;
  }
  return(0);
}
```

uses 3 types of reals and loops until they all eventually get too big. Write the output to a file and examine it within the editor, e.g.

a.out > ch0703.txt

or

ch0703.exe > ch0703.txt

or

a.exe > ch0703.txt

depending on whether you are running under Windows or Unix/Linux.

7.7.2 Real numbers getting too small

The following program

```
#include <iostream>
using namespace std;

int main()
{
float f = 10.0;
double f_d = 10.0;
long double f_l = 10.0;
  cout.precision(35);
  for (int i=0 ; i < 5000 ; ++i)
  {
    cout << f << " " << f_d << " " << f_l << endl;
    f   = f/10;
    f_d = f_d/10;
    f_l = f_l/10;
  }
  return(0);
}
```

uses 3 types of reals and loops until eventually they get too small

Some key numeric information for real numbers in C++

The following program uses information from FLOAT.H to provide details of the following

- minimum - smallest number for each real type
- maximum - largest number of each real type
- epsilon - the difference between numbers of each type
- digts - the number of significant digits for each real type

```
#include <iostream>
#include <cfloat>
using namespace std;

int main()
{
```

```
  cout << "    " << FLT_EPSILON << "     " <<  FLT_DIG << "
" <<  FLT_MIN << "     " <<  FLT_MAX << "    " << endl;
  cout << "    " << DBL_EPSILON << "    " <<  DBL_DIG << "
" <<  DBL_MIN << "    " <<  DBL_MAX << "    " << endl;
  cout << "    " << LDBL_EPSILON << "     " << LDBL_DIG << "
" << LDBL_MIN << "    " << LDBL_MAX << "    " << endl;
  return(0);
}
```

Compile and run this program on your computer system and compiler to see what is supported.

Here is the output from running on a system running Windows XP Pro, SP2 and Microsoft Visual Studio .NET 2003.

```
Microsoft Visual C++ .NET    69462-005-6008975-18764

    1.19209e-007     6    1.17549e-038     3.40282e+038
    2.22045e-016    15    2.22507e-308    1.79769e+308
    2.22045e-016     15    2.22507e-308    1.79769e+308
```

Here is the output from running on the same machine with cygwin installed and using g++.

```
$ ./a.exe
    1.19209e-07     6    1.17549e-38     3.40282e+38
    2.22045e-16    15    2.22507e-308    1.79769e+308
    1.0842e-19     18    3.3621e-4932    1.18973e+4932
```

and the cygwin installation makes available the Intel extended 80 bit real.

7.8 Character Data as a form of Integer Data

C++ provides us with character data as a flavour of integer data. This can be very useful when working with 8 bit data, e.g. satellite image data, extended character sets. Characters may be signed or unsigned. The following simple program illustrates the use of this data type to recover all of the printable ASCII characters from a file. The program was developed in response to users with corrupt word-processing files. It provides a quick and dirty way of retrieving some of the user's work.

```
#include <iostream>
using namespace std;
int main()
{
unsigned char c;
unsigned int i=0;
  for (;;)
  {
    c=cin.get();
    if (cin.eof()) break;
    ++i;
    if ( c < 32 || c > 127)
      c = ' ';
    cout << c;
    if ( i > 60 && c == ' ')
    {
```

```
        cout << endl;
        i=0;
      }
   };
   return(0);
}
```

Notice the for loop.

```
for(;;)
```

This is an infinite loop. The loop will actually terminate when end of file is reached, i.e.

```
if (cin.eof()) break
```

We will look into this in more depth in the chapter on io.

The next if test replaces all non ascii characters with blanks. The second breaks the text into 60 character lines.

7.9 Operators and Expression Evaluation

C++ has a large number of operators, 45 all told, and with all of the variants we have 57! A working knowledge of the more commonly used ones is essential for successful use of C++ in numeric programming. You should also be aware of the rest.

7.9.1 Expression Evaluation

The standard states in section r.5 on expressions *Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative. Overloaded operators are never assumed to be associative or commutative. Except where noted, the order of evaluation of operands of individual operators and subexpressions, and the order in which side effects takes place is unspecified. Between the previous and next sequence point a scalar object shall have its value modified at most once by the evaluation of the expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. The requirements of this paragraph shall be met for each allowable ordering of the sub-expressions of a full expression: otherwise the behaviour is undefined.*

Consider the following:–

```
i=v[i++];     //undefined
i=7,i++,i++   //i becomes 9
i=++i+1;     //undefined
i=i+1       //value of i is incremented
```

Care must therefore be taken when writing complex expressions. If your are unsure about the expression evaluation then break it down into smaller subexpressions.

7.9.2 Sequence Points

Because of the nature of C++ (and C) it is necessary to look at the concept of a sequence point. These are places in the code where all side effects of previous computations have to be completed. The following are sequence points:–

- after the complete evaluation of each full-expression;
- when calling a function after the evaluation of all function arguments, and before the execution of anything in the function body;

7.9.3 Lvalue and Rvalue

There is the possibility in C++ to be able to allocate and use variables that don't have names, and we will look at this concept again when we use arrays and look at the way pointers are used with arrays in C++.

What this means is that we can reference an object in memory without having a name for it. The standard states *an object is a region of storage: an lvalue is an expression referring to an object or function.*

The term lvalue originally meant something that could appear on the left hand side of an assignment statement.

7.9.4 Operators, Precedence and Associativity.

The following table summarises the rules concerning precedence and associativity. All operators associate left to right except for those in the third and eighteenth position in the precedence hierarchy, i.e. the unary and assignment operators.

Operator Summary

::	scope resolution	class_name :: member
::	global	:: name
.	member selection	object.member
->	member selection	pointer -> member
[]	subscripting	pointer [expr]
()	function call	expr (expr_list)
()	value construction	type(expr_list)
++	post increment	lvalue ++
—	post decrement	lvalue —
sizeof	size of object	sizeof expr
sizeof	size of type	sizeof (type)
++	pre increment	++ lvalue
—	pre decrement	— lvalue
~	complement	~ expr
!	not	! expr
-	unary minus	- expr
+	unary plus	+ expr
&	address of	& expr
*	dereference	* expr
new	create	new type
delete	destroy	delete pointer
delete[]	destroy array	delete [] pointer
()	cast	(type) expr
.*	member selection	object.* pointer_to_member
->*	member selection	pointer -> * pointer_to_member
*	multiply	expr * expr
/	divide	expr / expt
%	modulo or remainder	expr % expr
+	plus	expr + expr
-	minus	expr - expr
<<	shift left	expr << expr
>>	shift right	expr >> expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equal	expr == expr
!=	not equal	expr != expr

&	bitwise AND	expr & expr
^	bitwise exclusive OR	expr ^ expr
|	bitwise inclusive OR	expr | expr
&&	logical AND	expr && expr
||	logical inclusive OR	expr || expr
?:	conditional expression	expr ? expr : expr
=	conventional assignment	lvalue = expr
*=	multiply and assign	lvalue *= expr
/=	divide and assign	lvalue /= expr
%=	modulo and assign	lvalue %= expr
+=	add and assign	lvalue += expr
-=	subtract and assign	lvalue -= expr
<<=	shift left and assign	lvalue <<= expr
>>=	shift right and assign	lvalue >>= expr
&=	AND and assign	lvalue &= expr
|=	inclusive OR and assign	lvalue |= expr
^=	exclusive OR and assign	lvalue ^= expr
throw	throw exception	throw expr
,	comma	expr , expr

There will be more complete examples of each of the following in later chapters.

7.9.4.1 :: [scope resolution] class_name :: member

This operator allows us to use a name declared two or more times correctly within a name space.

7.9.4.2 :: [global] :: name

This operator allows to use what would otherwise be a hidden global name.

7.9.4.3 . [member selection] object.member

This operator allows us to select a member of a class.

7.9.4.4 -> [member selection] pointer -> member

As above.

7.9.4.5 [] [subscripting] pointer [expr]

The normal array subscripting operator.

7.9.4.6 () [function call] expr (expr_list)

The function call operator.

7.9.4.7 () [value construction] type(expr_list)

Value construction mechanism.

7.9.4.8 ++ [post increment] lvalue ++

Increment after use.

7.9.4.9 — [post decrement] lvalue —

Decrement after use.

7.9.4.10 sizeof [size of object] sizeof expr

Used to determine the memory size of an object.

7.9.4.11 sizeof [size of type] sizeof (type)

Used to determine the memory size of a type.

7.9.4.12 ++ [pre increment] ++ lvalue

Increment before use.

7.9.4.13 — [pre decrement] — lvalue

Decrement before use.

7.9.4.14 ~ [complement] ~ expr

One's complement operator. The operand must be of integral type. Integral promotions are performed.

Also used to identify a destructor.

7.9.4.15 ! [not] ! expr

Logical negation operator.

7.9.4.16 - [unary minus] - expr

As stated.

7.9.4.17 + [unary plus] + expr

As stated.

7.9.4.18 & [address of] & expr

The result of the unary & operator is a pointer to its operand.

7.9.4.19 * [dereference] * expr

The unary * operator means indirection, and the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points.

7.9.4.20 new [create] new type

The new operator attempts to create an object of the type to which it is applied. This type must be an object type, and functions cannot be allocated in this way, though pointers to functions can.

7.9.4.21 delete [destroy] delete pointer

The delete operator destroys an object created by the new operator.

7.9.4.22 delete[] [destroy array] delete [] pointer

This form is used to delete arrays. The expression points to an array. The destructors (if any) for the objects pointed to will be invoked.

7.9.4.23 () [cast] (type) expr

An explicit type conversion can be expressed using either functional notation or the cast notation.

7.9.4.24 .* [member selection] object.* pointer_to_member

The binary operator .* binds its second operand, which must be of type pointer to member of class T to its first operand, which must be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

7.9.4.25 ->* [member selection] pointer -> * pointer_to_member

The binary operator ->* binds its second operand, which must be of type pointer to member of T to its first operand, which must be of type pointer to T or pointer to a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

7.9.4.26 * [multiply] expr * expr

Conventional arithmetic multiplication.

7.9.4.27 / [divide] expr / expt

Conventional arithmetic division.

7.9.4.28 % [modulo or remainder] expr % expr

Remainder.

7.9.4.29 + [plus] expr + expr

Conventional arithmetic addition.

7.9.4.30 - [minus] expr - expr

Conventional arithmetic subtraction.

7.9.4.31 << [shift left] expr << expr

Shift left. The operands must be of integral type and integral promotions are performed.

7.9.4.32 >> [shift right] expr >> expr

Shift right. The operands must be of integral type and integral promotions are performed.

7.9.4.33 < [less than] expr < expr

Conventional relational operator.

7.9.4.34 <= [less than or equal] expr <= expr

Conventional relational operator.

7.9.4.35 > [greater than] expr > expr

Conventional relational operator.

7.9.4.36 >= [greater than or equal] expr >= expr

Conventional relational operator.

7.9.4.37 == [equal] expr == expr

Conventional relational operator.

7.9.4.38 != [not equal] expr != expr

Conventional relational operator.

7.9.4.39 & [bitwise AND] expr & expr

The usual arithmetic conversions are performed: the result is the bitwise and function of the operands. The operator applies only to integral operands.

7.9.4.40 ^ [bitwise exclusive OR] expr ^ expr

The usual arithmetic conversions are performed: the result is the bitwise exclusive or function of the operands. The operator applies only to integral operands.

7.9.4.41 | [bitwise inclusive OR] expr | expr

The usual arithmetic conversions are performed: the result is the bitwise inclusive or function of the operands. The operator applies only to integral operands.

7.9.4.42 && [logical AND] expr && expr

The operands are converted to type bool. The result is true if both operands are true and false otherwise.

Left to right evaluation is guaranteed, and the second operand is not evaluated if the first is false.

All side effects of the first expression except for destruction of temporaries happen before the second expression is evaluated.

7.9.4.43 || [logical inclusive OR] expr || expr

The operands are both converted to type bool. The result is true if either of its operands is true and false otherwise.

Left to right evaluation is guaranteed, and the second operand is not evaluated if the first is true.

All side effects of the first expression except for destruction of temporaries happen before the second expression is evaluated.

7.9.4.44 ?: [conditional expression] expr ? expr : expr

The first expression is converted to bool. It is evaluated and if it is true the result of the conditional expression is the value of the second expression, otherwise that of the third.

All side effects of the first expression except for destruction of temporaries happen before the second or third expression is evaluated.

7.9.4.45 = [conventional assignment] lvalue = expr

Conventional assignment.

7.9.4.46 *= [multiply and assign] lvalue *= expr

Multiply and assign, e.g. a=a*expression

7.9.4.47 /= [divide and assign] lvalue /= expr

Divide and assign, e.g. a=a/expression

7.9.4.48 %= [modulo and assign] lvalue %= expr

Modulo and assign, e.g. a=a%expression

7.9.4.49 += [add and assign] lvalue += expr

Add and assign, e.g. a=a+expression

7.9.4.50 -= [subtract and assign] lvalue -= expr

Subtract and assign, e.g. a=a-expression

7.9.4.51 <<= [shift left and assign] lvalue <<= expr

Shift left and assign

7.9.4.52 >>= [shift right and assign] lvalue >>= expr

Shift right and assign

7.9.4.53 &= [AND and assign] lvalue &= expr

AND and assign

7.9.4.54 |= [inclusive OR and assign] lvalue |= expr

inclusive OR and assign

7.9.4.55 ^= [exclusive OR and assign] lvalue ^= expr

exclusive OR and assign

7.9.4.56 throw [throw exception] throw expr

throw exception

7.9.4.57 , [comma] expr , expr

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. All side effects of the left expression are performed before the evaluation of the right expression. The type and value of the result are the type and value of the right operand; the result is an lvalue if its right operand is.

7.10 Expression Examples

It is not appropriate here to cover each and everyone of the above in great depth. We will introduce examples throughout the notes as we progress.

Note that there is no exponentiation operator.

The following programs and program extracts cover some of the above.

```
#include <iostream>
using namespace std;

int main()
{
int i=0;
int j=0;
  cout << ++i << endl;
  cout << j++ << endl;
  return(0);
}
```

The above example highlights the use of the pre and post increment and decrement operators. Type the above in and run it.

```
#include <iostream>
using namespace std;

int main()
{
int i=9;
int j=2;
int k=-2;
  cout << i/j << endl;
  cout << i/k << endl;
  cout << i%j << endl;
  cout << i%k << endl;
  return(0);
}
```

Type this example in and run it. Work is underway with the new C standard to make division consistent with the Fortran 90 standard, and hence by implication this will apply to C++ also.

Consider the following example:–

```
#include <iostream>
using namespace std;
```

```
int main()
{
int i=0;
int j=0;
int k=0;
int l=0;
  i=i+1;
  j+=1;
  ++k;
  l++;
  cout << i << " " << j << " " << k << " " << l << endl;
  return(0)
}
```

What do you think will be the output of this program? Do we need four ways of achieving the same thing in a programming language?

Consider the following example.

```
#include <iostream>
using namespace std;

int main()
{
float light_minute,distance,elapse ;
int minute,second ;
const float light_year = 9.46*10E12 ;
  light_minute = light_year / ( 365.25 * 24.0 * 60.0 ) ;
  distance = 150.0 * 10E6 ;
  elapse = distance / light_minute ;
  minute = elapse ;
  second = ( elapse - minute ) * 60 ;
  cout << " Light takes " << minute  << " minutes \n" ;
  cout << "               " << second  << " seconds \n" ;
  return (0);
}
```

Do you get any warnings when compiling this program?

Consider the following program.

```
#include <iostream>
using namespace std;
int main()
{
float Gross_wage, Net_wage, Tax;
float Tax_rate = 0.25;
int   Personal_allowance = 2500;
string Their_Name;
  cout << "Input employees name ? ";
  cin >> Their_Name;
  cout << "Input Gross wage ? ";
  cin >> Gross_wage;
```

```
  Tax = (Gross_wage - Personal_allowance) * Tax_rate;
  Net_wage = Gross_wage - Tax;
  cout << "Employee: " << Their_Name << endl ;
  cout << "Gross Pay: " <<  Gross_wage << endl ;
  cout << "Tax: " << Tax << endl ;
  cout << "Net Pay:" << Net_wage << endl ;
 return(0);
}
```

This is another simple example of arithmetic in C++.

7.11 Summary

As Stroustrup states *C++ is a general purpose programming language; its core application domain being systems programming in the broadest sense. In addition C++ is successfully used in many application areas that are not covered by that label.* This means that there is a lot of functionality within the language and mastery of it will take time.

C++ shares much in common with other programming languages used for numeric programming. However there are a number of areas that are different and you need to be careful here.

One problem lies in expression evaluation. As the standard states this is undefined in many cases. This is to provide the compiler writer with the opportunity to optimise the code. Note however that program correctness is more important than program efficiency. The Fortran 90 standard tackles this issue with the concept of an equivalent mathematical expression.

Another problem comes with unsigned integer arithmetic, where overflow is not regarded as an error, and the calculations can continue.

Another problem is really a quality of implementation issue with regard to floating point error reporting. Hopefully this will improve with the publication of the standard.

7.12 Key Concepts

7.12.1 Numeric Data Types

7.12.1.1 Integer

short int

int

long int

7.12.1.2 Real

float

double

long double

7.12.1.3 Complex – supported via <complex>

7.12.1.4 Character Data as Numeric Data

8 bit

signed -128 to +127

unsigned 0 to 255

7.12.2 Constants – use the const attribute

7.12.3 Operators – 45 but effectively 57 with variants

7.12.4 Expressions – the order of expression evaluation is often undefined

7.12.5 Portable main()

The standard provides only two portable versions of main and these are:

```
int main()
{
}
```

and

```
int main(int argc , char* argv[])
{
}
```

The return type int is required. C++ provides an implicit return(0) at the end of main(). The examples in these notes always provide an explicit return(0).

7.12.6 Sample limits.h

```
#define CHAR_BIT <#if expression >= 8>
#define CHAR_MAX <#if expression >= 127>
#define CHAR_MIN <#if expression <= 0>
#define SCHAR_MAX <#if expression >= 127>
#define SCHAR_MIN <#if expression <= -127>
#define UCHAR_MAX <#if expression >= 255>
#define MB_LEN_MAX <#if expression >= 1>

#define SHRT_MAX <#if expression >= 32,767>
#define SHRT_MIN <#if expression <= -32,767>
#define USHRT_MAX <#if expression >= 65,535>

#define INT_MAX <#if expression >= 32,767>
#define INT_MIN <#if expression <= -32,767>
#define UINT_MAX <#if expression >= 65,535>

#define LONG_MAX <#if expression >= 2,147,483,647>
#define LONG_MIN <#if expression <= -2,147,483,647>
#define ULONG_MAX <#if expression >= 4,294,967,295>
```

7.12.7 Sample float.h

```
#define FLT_RADIX <#if expression >= 2>
#define FLT_ROUNDS <integer rvalue>
#define DBL_DIG <integer rvalue >= 10>
#define DBL_EPSILON <double rvalue <= 10^(-9)>
#define DBL_MANT_DIG <integer rvalue>
#define DBL_MAX <double rvalue >= 10^37>
#define DBL_MAX_10_EXP <integer rvalue >= 37>
#define DBL_MAX_EXP <integer rvalue>
#define DBL_MIN <double rvalue <= 10^(-37)>
#define DBL_MIN_10_EXP <integer rvalue <= -37>
```

```
#define DBL_MIN_EXP <integer rvalue>

#define FLT_DIG <integer rvalue >= 6>
#define FLT_EPSILON <float rvalue <= 10^(-5)>
#define FLT_MANT_DIG <integer rvalue>
#define FLT_MAX <float rvalue >= 10^37>
#define FLT_MAX_10_EXP <integer rvalue >= 37>
#define FLT_MAX_EXP <integer rvalue>
#define FLT_MIN <float rvalue <= 10^(-37)>
#define FLT_MIN_10_EXP <integer rvalue <= -37>
#define FLT_MIN_EXP <integer rvalue>

#define LDBL_DIG <integer rvalue >= 10>
#define LDBL_EPSILON <long double rvalue <= 10^(-9)>
#define LDBL_MANT_DIG <integer rvalue>
#define LDBL_MAX <long double rvalue >= 10^37>
#define LDBL_MAX_10_EXP <integer rvalue >= 37>
#define LDBL_MAX_EXP <integer rvalue>
#define LDBL_MIN <long double rvalue <= 10^(-37)>
#define LDBL_MIN_10_EXP <integer rvalue <= -37>
```

#define LDBL_MIN_EXP <integer rvalue>

7.13 Problems

1. There is no exponentiation operator in C++. You have to use a function. The headers <cmath> and <math.h> provide the commonly used mathematical functions. There is a complete coverage in the chapter on functions. For the purposes of the problems that follow you will need the pow function. The are two forms:

- double pow(double x, double y) – raise x to the power y
- double pow(double x, integer i) – raise x to the power i

You will also need to add statements that control the precision. You have seen examples of this already..

2. Try typing in and running the examples given in this chapter. Remember that you need to gain familiarity with the C++ rules. You need to make mistakes and see what goes wrong.

3. Modify the program that read in your name and address to read in and print out your age, telephone number and sex.

4. Write a program to calculate the period of a pendulum. Use the following formula:–

$t=2* \ *(length/9.81)^2$

Write two versions, one using float, the other using double. The length is in metres and the time is in seconds. Choose a length of 10 metres.

What is a realistic value for ?

Now calculate using Excel and a hand held calculator. Do the answers agree?

5. Write a program that tests the size and precision of numbers on the system that you are using. What is the word size? Experiment with multiplication and division to see what messages you get when numbers become too large (overflow) and too small (underflow).

If you have access to anther compiler and or platform and compiler try the above out on that system.

6. Base conversion.

Consider the following program.

```
#include <iostream>
using namespace std;
int main()
{
float x1=1.0;
float x2=0.1;
float x3=0.01;
float x4=0.001;
float x5=0.0001;
  cout.precision(9);
  cout << x1<< endl;
  cout << x2<< endl;
  cout << x3<< endl;
  cout << x4<< endl;
  cout << x5<< endl;
  return(0);
}
```

Which do you think will provide the same number as originally entered?

7. Simple subtraction. In this chapter we looked at representing floating point numbers in a finite number of bits.

```
#include <iostream>
using namespace std;

int main()
{
float x=1.0002;
float y=1.0001;
float z;
  z=x-y;
  cout.precision(35);
  cout << x << endl;
  cout << y << endl;
  cout << z << endl;
  return(0);
}
```

Run the program and look at the output. Is it what you expected?

8. Expression equivalence. In mathematics the following is true:

$(x^2-y^2) = (x*x-y*y) = (x-y)*(x*y)$

Consider the following program.

```
#include <iostream>
using namespace std;

int main()
{
float x=1.0002;
float y=1.0001;
```

```
float t1,t2,t3;
  t1=pow(x,2)-pow(y,2);
  t2=x*x-y*y;
  t3=(x+y)*(x-y);
  cout.precision(20);
  cout << t1 << endl;
  cout << t2 << endl;
  cout << t3 << endl;
  return(0);
}
```

Solve the problem with pencil and paper, calculator and Excel.

The last three examples show that you must be careful when using a computer to solve problems.

7.14 Bibliography

Some understanding of numerical analysis is essential for successful use of a programming language. As Froberg says '*numerical analysis is a science – computation is an art*.' The following are some of the more accessible books available.

Burden R.L., Douglas Faires J., *Numerical Analysys*, Brooks/Cole, 2001.

The cd has source code in C, Fortran, Maple, Mathematica, Matlab and Pascal. Good modern text.

Froberg C.E., *Introduction to Numerical Analysis*, Addison Wesley, 1969.

The short chapter on numerical computation is well worth a read, and it covers some of the problems of conversion between number bases, and some of the errors that are introduced when we compute numerically. The Samuel Johnson quote owes its inclusion to Froberg!

IEEE, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*, Institute of Electrical and Electronic Engineers Inc.

The formal definition of IEEE 754.

Knuth D., *Seminumerical Algorithms*, Addison Wesley, 1969.

A more thorough and mathematical coverage than Wakerly. The chapter on positional number systems provides a very comprehensive historical coverage of the subject. As Knuth points out the floating point representation for numbers is very old, and is first documented around 1750 B.C. by Babylonian mathematicians. Very interesting and worthwhile reading.

Sun, *Numerical Computation Guide*, SunPro, 1993.

Very good coverage of the numeric formats for IEEE Standard 754 for Binary Floating-Point Arithmetic. All SunPro compiler products support the features of the IEEE 754 standard .

Wakerly J.F., *Microcomputer Architecture and Programming*, Wiley, 1981.

The chapter on number systems and arithmetic is surprisingly easy. There is a coverage of positional number systems, octal and hexadecimal number system conversions, addition and subtraction of non-decimal numbers, representation of negative numbers, two's complement addition and subtraction, one's complement addition and subtraction, binary multiplication, binary division, bcd or binary coded decimal representation and fixed and floating point representations. There is also coverage of

a number of specific hardware platforms, including DEC PDP-11, Motorola 68000, Zilog Z8000, TI 9900, Motorola 6809 and Intel 8086. A little old but quite interesting nevertheless.

8
Strings and other data types

‘Don’t Panic’

Douglas Adams, The Hitch Hiler’s Guide to the Galaxy

Aims

The aims are:–

- to introduce the string data type;
- to look at <string>;
- to introduce the boolean data type;
- to introduce reference variables;
- to introduce enumerated data types;
- to look at a number of other c++ concepts;

8 Strings and other Data Types

This chapter looks at the other data types available in C++ and a number of other concepts present in C++.

8.1 Character Data or Strings

Sequences of characters or strings can be represented in C++ in two ways. The first, old C style, uses an array of characters terminated with a null character. The second treats strings as objects. We will look at examples in both styles.

8.1.1 C Style strings

8.1.1.1 Example 1

Note the values of the lengths of the two character arrays when this program is compiled and executed. The answer of 27 is not one that most people would expect. Note also the last character printed out in the loops.

```
#include <iostream>
using namespace std;
int main()
{
char uppercase[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char lowercase[]="abcdefghijklmnopqrstuvwxyz";
char ch;
int i;
int ic;
int usize=sizeof(uppercase)/sizeof(char);
int lsize=sizeof(lowercase)/sizeof(char);
  cout << " upper case array size is " << usize << endl;
  cout << " lower case array size is " << lsize << endl;
  for (i=0;i < usize-1 ; i++)
  {
    ch=uppercase[i];
    ic=int(ch);
    cout << ch << " = " << dec << ic
              << " = 0" << oct << ic
              << " = 0x" << hex << ic << endl;
  }
  cout << endl;
  for (i=0 ;i < lsize-1 ;i++)
  {
    ch=lowercase[i];
    ic=int(ch);
    cout << ch << " = " << dec << ic
               << " = 0" << oct << ic
               << " = 0x" << hex << ic << endl;
  }
  return(0);
}
```

Note here the use of dec, oct and hex to write out integer values in decimal, octal or hexadecimal format.

We will not look at this way of doing things in more depth until we have covered pointers.

8.1.1.2 Example 2

The following is another example of C style string usage.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
char line[80];
  strcpy(line,"Ian");
  strcat(line," David");
  strcat(line," Chivers");
  cout << line;
  return(0);
}
```

strcpy copies a string from the second argument to the first. Not quite what some people expect!

strcat concatenates the second string to the first, taking care of the \0 terminator.

8.1.1.3 Example 3

The following example prints the first 128 characters. The program is given below.

```
#include <iostream>
using namespace std;
int main()
{
  char c;
  for (int i=0;i<128;++i)
  {
    c=char(i);
    cout << i << " " << c << endl;
  }
  return(0);
}
```

We convert from an integer value to the correspnding character with the statement

```
  c = char(i)
```

What actually prints? Does the program cause a noise to be made?

The following are some of the old C style string functions.

8.1.1.4 strcpy(s1,s2)

Copy string s2 to string s1.

8.1.1.5 strcat(s1,s2)

Join string s2 to string s2.

8.1.1.6 strcmp(s1,s2)

Compare strings s1 and s2.

8.1.1.7 strlen(s)

Return the length of string s.

8.1.1.8 strchr(s,c)

Locate character c in string s.

What we have in the above is a syntax that binds togethor data and an operation. strcpy, strcat, strcmp, strlen and strchr are the operations or actions we want carried out, and the arguments are the data. This is a syntax that you should be familiar with from your experience of functions, procedures or subroutines in Fortran, Pascal etc.

8.1.2 C++ Style strings – <string>

Access to this data type is provided in the <string> include file. Some of the benefits of this data type over C style strings include:–

- more functionality;
- string assignment results in copies – this avoids the problems associated with multiple pointer references to one memory location;
- subscript checking through the use of at() – the out_of_range exception is thrown; we will look into this in more depth in the chapter on arrays, vectors and valarrays.
- comparisons can be done using the conventional relational operators;

The following provides a summary of the new class.

8.1.2.1 assignment

s1=s2

8.1.2.2 character access

s[i]

s.substr(position,length)

8.1.2.3 comparison

s1==s2

s1!=s2

s1<s2

s1>s2

s1<=s2

s1>=s2

8.1.2.4 concatenation

s1+s2

s+=s1

8.1.2.5 constructors

string s

string s("some text")

string s1(s2)

8.1.2.6 i/o

stream << s

stream >> s

getline(stream,string,character)

8.1.2.7 insertion

s.insert(position,string)

8.1.2.8 iterators

s.begin()

s.end()

8.1.2.9 length

s.length()

s.resize(integer,character)

s.empty()

8.1.2.10 removal

s.remove(start,length)

8.1.2.11 replacement

s.replace(start,length,string)

8.1.2.12 resize

s.resize(integer,character)

8.1.2.13 search

s.find(string)

s.find(string,position)

s.find_first_of(string,position)

s.find_first_not_of(string,position)

We will look at examples of string use through the rest of the course material.

Notice also the new object oriented syntac for binding togethor data and an action:–

```
data.action()
```

and contrast this with the other syntax for strings using arrays of char. Things will gradually become more familiar as we progress through the material and examples.

8.1.2.14 Example 3

This is a version of example 2, using the new style syntax.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string s;
  s=" Ian";
  s+=" David";
  s+=" Chivers";
  cout << s << endl;
  return(0);
}
```

Note that the string is fully dynamic and grows to the size required. Note also the better syntax for string manipulation using conventional operators.

8.1.3 Guidelines for use

It should be obvious that the new C++ style strings offer substantial advantages over the old C style strings, and some recommendations about usage include:–

- we can pass and return strings by value and let the system manage memory
- we can use subscripting when we want range checking – covered in the chapter on arrays and vectors;
- we can then catch exceptions – covered in more depth later;
- we can use use iterators when we want speed – covered in the chapter on the standard template library;

Most of the older texts on C++ will use the old C style strings.

We will look at more string examples throughout the notes.

8.2 Boolean or Logical Data

Section r.3.6.1 of the standard introduces boolean as a genuine data type for the first time into C/C++. bool is a unique signed integral type. A bool value may be converted to int by promotion: true converts to 1 and false converts to 0. A numeric or pointer value may be implicitly converted to bool. A zero value becomes false and a non-zero value becomes true. The relational operators yield bool values.

8.3 Reference Variables

A reference variable provides an alias mechanism within C++. This means that two or more variables can refer to the same variable or storage location. The following program illustrates the concept. Note the use of & to denote that the variable is a reference variable. The & character is called the address of operator in C++. We will look into this area in greater depth in the chapter on pointers. Try running the program and looking at the output.

```
#include <iostream>
using namespace std;

int main()
{
int i=10;
int &j=i;
   cout << i << endl;
   cout << j << endl;
   j=20;
   cout << i << endl;
   cout << j << endl;
   return(0);
}
```

The three main uses of reference variables are:–

- as argument types in functions;
- with return values from functions;
- with overloaded operators;

and we will come back to guidelines for the use of reference variables later in the course.

8.4 Enumeration Types

The following program illustrates the use of enumerated types in C++. This is a feature of the Pascal, Modula2 family of languages, but not Fortran. Interestingly this feature has been removed from Oberon.

They provide a very secure program construct when a data type can map onto a subset of integers.

```
#include <iostream>
using namespace std;
int main()
{
enum day {
            monday,
            tuesday,
            wednesday,
            thursday,
            friday
            saturday,
            sunday
          };
day today;
   today=monday;
   cout << today << endl;
   return(0);
}
```

8.5 Type Conversion

In the previous chapter we looked at writing mixed mode arithmetic expressions and covered the type promotions that took place automatically. There is also the need to cast between types, i.e. convert one type to another. This is done in C++ using the syntax:–

- (type) expression

or

- type(expression)

The following program is a complete example.

```
#include <iostream>
using namespace std;
int main()
{
float a;
float b;
int i=0;
   a=(float)i;
   b=float(i);
   cout << i << endl;
   cout << a << endl;
   cout << b << endl;
   return(0);
}
```

8.6 Scope

With all programming languages there is the concept of the scope of a name. Within the examples so far the scope of a name is restricted to the file that we are compiling. We can complicate things in C++ by declaring variables of the same name within a program, e.g.

using a variable within an inner block will hide a variable of the same name within an outer block. The following program illustrates this.

```
#include <iostream>
using namespace std;

int main()
{
float a;
int  i=0;
  a=(float)i;
  cout << i << endl;
  cout << a << endl;
  {
    int i=99;
    cout << i << endl;
  }
  cout << i << endl;
  return(0);
}
```

The { and }define the start and end of a block.

We will look at the concept of scope in much greater depth throughout the notes

8.7 Void

The type VOID behaves syntactically like any other fundamental type. There are some restrictions on its use however. There are no objects of type VOID. The most common use is as the type of a function that does not return a result.

8.8 Memory and C++

C++ provides a number of ways of working with memory and it is necessary to look at the choices on offer.

Firstly we distinguish between static and automatic. A static object is allocated when the program starts executing and exists throughout the lifetime of the program. An automatic object is allocated each time execution reaches that point in the program, and exists until the block that contains it is exited or left.

There is also a method of creating objects on what is called the free store using the new operator.

We will look into this whole area in more depth after the coverage of functions.

8.9 Summary

We will look at character data in much greater detail later in the course.

8.10 Key Concepts

8.10.1 Data Types

8.10.1.1 Sequence of characters as an array of char

char x[10];

8.10.1.2 Seqence of characters as a string

string s;

8.10.1.3 Logical or boolean

bool check;

8.10.1.4 Reference Variable

int i;

int &j=i;

8.10.1.5 Enumerated Types

enum weekday {monday, tuesday, wednesday, thursday, friday};

8.10.1.6 Void

void main()

8.10.1.7 Type Conversion

(type) expression

or

type (expression)

8.10.2 Scope

Range of the validity of the name of a variable.

8.10.3 Memory

static – allocated when the program starts;

automatic – allocated and de-allocated each time execution reaches that point in a program;

free store – allocated and de-allocated use new and dispose, covered later.

8.11 Problems

1. Type in the example one and look at the decimal, octal and hexadecimal output. Is this what you would have expected? Is this consistent with the ASCII character set?

2. Type in and run the second example using strings. Use you own name.

3. Here is some sample output from running example 3.

```
0
1 ?
2 ?
3 ?
4 ?
5 ?
6 ?
7
8
9
10

11 ?
12 ?
13
14 ?
15
16 ?
17 ?
18 ?
```

```
19 ?
20 ¶
21 §
22 ?
23 ?
24 ?
25 ?
26 ?
27 28 ?
29 ?
30 ?
31 ?
32
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41 )
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
49 1
50 2
51 3
52 4
53 5
54 6
55 7
56 8
57 9
58 :
59 ;
60 <
61 =
62 >
63 ?
64 @
65 A
66 B
67 C
68 D
69 E
```

```
70 F
71 G
72 H
73 I
74 J
75 K
76 L
77 M
78 N
79 O
80 P
81 Q
82 R
83 S
84 T
85 U
86 V
87 W
88 X
89 Y
90 Z
91 [
92 \
93 ]
94 ^
95 _
96 `
97 a
98 b
99 c
100 d
101 e
102 f
103 g
104 h
105 i
106 j
107 k
108 l
109 m
110 n
111 o
112 p
113 q
114 r
115 s
116 t
117 u
118 v
119 w
```

```
120 x
121 y
122 z
123 {
124 |
125 }
126 ~
127 ¦
```

Does yours match?

3. Type in and run the reference variable example. Can you think of any use of reference variables at the moment?

4. Type in and run the enumeration example. What will be the output?

9
Arrays, Vectors and valarrays in C++

‘Where shall I begin your Majesty’ he asked.
‘Begin at the beginning,’ the King said, gravely ‘and go on until you come to the end then stop.’

Lewis Carroll, Alice’s Adventures in Wonderland.

Aims

The aims of this chapter are to:–

look at the basic array syntax in C++;

look at the new facilities provided by the vector class;

look at the new facilities provided by the valarray class;

look at the associated control structure, the for loop;

look at array element ordering in C++;

look forward to the use of some of the additional features of C++ that enable us to make array handling more understandable and reliable.

9 Arrays, Vectors and Valarrays in C++

In this chapter we will look three different ways of using arrays in C++. The first is the old C style array mechanism, the second looks at the facilities provided by the vector class, and the third looks at the facilities provided by the valarray class.

You should use the new vector and valarray classes in preference to the old C style array. However you have to know about the old C style arrays for a variety of reasons, including:–

- not all compilers support vectors and valarrays fully;
- you will have to work with code written in the old style;
- you will have to interface to libraries written and compiled using the old C style arrays;
- you will have to work with libraries written in other languages and these will only work with the old C style arrays;
- the vector class cannot be used with the standard mathematical functions in C++.

9.1 Old C Style arrays

9.1.1 One d example – hard coded size

```
#include <iostream>
using namespace std;
int main()
{
float sum=0.0,average=0.0 ;
float rainfall[12] ;
int month ;
  cout << " Type in the rainfall, one per line \n" ;
  for (month=0;month < 12 ; ++month)
    cin >> rainfall[month];
  for (month=0;month < 12 ; ++month)
    cout << month << "   " << rainfall[month] << " \n " ;
  for (month=0;month < 12 ;++month)
    sum = sum + rainfall[month];
  average = sum/12;
  cout << " Average is " << average;
  return(0);
}
```

The first thing of interest is the array declaration.

```
float rainfall[12]
```

This declares the array rainfall to hold 12 elements. The use of the [] means that arrays stand out quite clearly from function references.

The second thing of interest is the array indexing. This goes from 0 through 11. A little odd at first, but don’t panic.

We are used to arrays starting at 1 – naturally, this is the first element. Zero didn’t exist in mathematics for quite some time. However BASIC is a programming language that offers arrays starting at 0 and 1 as an option in most implementations.

Lets look now at the for loop in C++. The general syntax is

```
for  (  inital statement
;
     expression 1     ;
     expression 2     )
       statement
```

The *initial statement* normally sets up an initial value for the loop counter. In this case month.

Expression 1 is the loop control mechanism. In this case we are interested in stopping once we have processed all 12 months.

Expression 2 is normally the loop counter increment mechanism. In this case increment by one.

statement is the statement that will be executed whilst expression 1 is true.

We will look in more detail at the for statement in a later chapter. Here is some data taken from the monthly rainfall figures for London.

3.1

2.0

2.4

2.1

2.2

2.2

1.8

2.2

2.7

2.9

3.1

3.1

The measurements are in inches.

9.1.2 One d array – parameterised size

```
#include <iostream>
using namespace std;

int main()
{
const int number_of_people = 10;
float sum=0.0,average=0.0;
float weights[number_of_people];
int person;
  cout << " Type in their weights, one per line \n";
  for ( person=0 ; person < 10 ; ++person )
  {
    cin >> weights[person];
    sum=sum + weights[person];
  }
  average=sum/number_of_people;
  cout << " sum was      " << sum << " \n ";
```

```
  cout << " average was " << average << " \n " ;
  return(0);
}
```

In this example we use { and } to bracket the statement under the control of the for loop. This enables us to repeatedly execute more than one statement.

9.1.3 Two d Array – latitude and longitude

```
#include <iostream>
using namespace std;

int main()
{
int latitude,longitude;
float height[5][5];
  for ( latitude=0 ; latitude < 5 ; ++latitude)
    for (longitude=0;longitude < 5 ; ++longitude)
      cin >> height[longitude][latitude];
  for (latitude=0;latitude < 5 ; ++latitude )
  {
    for (longitude=0;longitude < 5 ; ++longitude )
      cout << height[longitude][latitude] << " ";
    cout << endl;
  }
  return(0);
}
```

Things are now getting a little away from our original real world way of looking at problems. We now have 0–4 everywhere where the real world says 1–5.

9.1.4 Simple physics – voltage example

Now consider a simple physics example. Here we go from -20 through +20 for the voltage.

```
#include <iostream>
using namespace std;

int main()
{
int voltage;
float current[41];
float resistance;
  for ( voltage=0 ; voltage <= 40 ; ++voltage )
  {
   resistance=voltage+10;
   current[voltage]=voltage/resistance;
   cout  << voltage - 20 << " " << voltage << " "
       << resistance < " " << current[voltage] << "\n";
  }
  return(0);
}
```

The solution to the problem in a programming language is now quite removed from the original problem. -20 through +20 has become 0 through 40. We are now having to do a few mental gymnastics to understand what is going on.

The **real** world	C++
-20	0
-19	1
-18	2
..	..
0	20
..	..
+19	39
+20	40

Given the lack of array subscript checking in many C++ implementations this makes tracking down array indexing errors more difficult than it should be.

9.1.5 Time Zone Example

```
#include <iostream>
using namespace std;

int main()
{
int degree,strip;
float t[361];
float v;
  for (degree=0 ; degree < 346 ; degree = degree+15)
  {
    v = (degree-180)/15.;
    for (strip=0 ; strip < 15 ; ++strip)
    {
        t[degree+strip] = v;
        cout << degree - 180 + strip << " "
        << t[degree+strip] << "\n";
    }
  }
  return(0);
}
```

Things have got worse. The real world has -180 through +180, C++ has 0 through 360. This conceptual gap is one of the reasons why raw C++ poses problems for array based numeric programming.

We will look at ways of improving array handling in C++ using classes. Classes enable us to us array indexing based on the real world, and add error or subscript checking, to make our programs more reliable.

9.2 Array Initialisation

In these examples we look at array initialisation.

9.2.1 One d array initialisation

```
#include <iostream>
using namespace std;
int main()
{
float sum=0.0,average=0.0 ;
float rainfall[12] = { 3.1 , 2.0 , 2.4 , 2.1 , 2.2 , 2.2 ,
1.8 , 2.2 , 2.7 , 2.9 , 3.1 , 3.1 };
int month ;
   for (month=0;month < 12 ;++month)
      sum = sum + rainfall[month];
   average = sum/12;
   cout << " Average is " << average;
   return(0);
}
```

9.2.2 Two d initialisation

```
#include <iostream>
using namespace std;
int main()
{
int a[2][3]=   { { 1,2,3 },
                 { 4,5,6 } };
   for (int row=1;row<3;row++)
   {
      for (int column=1;column<4;column++)
        cout << a[row-1][column-1] << " ";
      cout << endl;
   }
   return(0);
}
```

9.2.3 Whole Array Manipulation

There is no whole array manipulation mechanism for old C style arrays. We can, through the use of classes and operator overloading, provide this functionality for one dimensional arrays, whilst still using the standard C++ array indexing notation, i.e. []. However it is not currently possible in C++ to overload [][], or [][][] etc. We will look into this area in greater detail in later chapters.

9.3 Vectors

The vector class was added to C++ to help overcome some of the deficiencies of raw C style arrays. The most important features of the vector class are:–

- it is an object;
- it is a container – it can hold objects of any type;
- subscript checking is available;
- the size is an integral part of the object;

- unfortunately you can't apply the standard mathematical functions to vectors.
- its size is dynamic;

We will also look at vectors in more depth in a later chapter.

9.3.1 Rainfall example using vectors

```
#include <iostream>
#include <vector>
using namespace std;

int main()

{
float sum=0.0,average=0.0 ;
vector<float> rainfall(12) ;
int month ;

  cout << " Type in the rainfall, one per line \n" ;
  for (month=0;month<rainfall.size();++month)
    cin >> rainfall[month];
  for (month=0;month<rainfall.size();++month)
    cout << month << "   " << rainfall[month] << " \n " ;
  for (month=0;month<rainfall.size();++month)
    sum = sum + rainfall[month];
  average = sum/12;
  cout << " Average is " << average;
  return(0);
}
```

The first thing of interest is:–

```
#include <vector>
```

which makes available the vector container. A container is an object that can hold or contain other objects. We will look at the C++ concept of containers in more depth in later chapters, especially in the chapter on the Standard Template Library.

The next thing of interest is:–

```
vector<float> rainfall(12) ;
```

which declares rainfall to be a vector of size 12, and that it will hold numbers of type float.

The next thing of interest is:–

```
  for (month=0;month<rainfall.size();++month)
```

This is a standard for loop, but now the size of the rainfall vector is an integral part of the object and available through *rainfall.size()* which in this case is 12. If this syntax looks a little strange at first don't worry. What we have is a syntax that binds togethor data and an operation. In this case the data is the rainfall vector and the operation (or function) is size()

9.3.2 Subscript checking

This example uses an alternate syntax that supports subscript checking.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int main()
{
float sum=0.0,average=0.0 ;
vector<float> rainfall(12) ;
int month ;

  cout << " Type in the rainfall, one per line \n" ;
  for (month=0;month<rainfall.size();++month)
    cin >> rainfall.at(month);
  for (month=0;month<rainfall.size();++month)
    cout << month << "  " << rainfall.at(month) << " \n " ;
  for (month=0;month<rainfall.size();++month)
    sum = sum + rainfall.at(month);
  average = sum/12;
  cout << " Average is " << average;
  return(0);
}
```

The syntax is *rainfall.at(month)*. We will look into this syntax later. It will be much clearer after we have covered the chapters on classes.

9.3.3 Subscript checking with try/catch

This example uses the exception handling facilities provided in C++ to trap the error and provide a meaningful error message. We will look into this whole area in a later chapter.

```
#include <iostream>
#include <vector>
using namespace std;

int main()

{
float sum=0.0,average=0.0 ;
vector<float> rainfall(12) ;
int month ;
  try

  {
    cout << " Type in the rainfall, one per line \n" ;
    for (month=0;month<13;++month)
      cin >> rainfall.at(month);
    for (month=0;month<13;++month)
      cout << month << "  " << rainfall.at(month) << " \n "
;
    for (month=0;month<13;++month)
      sum = sum + rainfall.at(month);
    average = sum/12;
    cout << " Average is " << average;
  }

  catch (out_of_range)
```

```
  {
  cout << " index out of range " << month << endl;
  }

  return(0);

}
```

Note the meaningful error message if the index goes out of range.

9.3.4 Whole array assignment

```
#include <iostream>
#include <vector>
using namespace std;

int main()

{
float sum=0.0,average=0.0 ;
vector<float> rainfall(12),t(6) ;
int month ;

  cout << " Type in the rainfall, one per line \n" ;
  for (month=0;month<rainfall.size();++month)
    cin >> rainfall[month];
  for (month=0;month<rainfall.size();++month)
    cout << month << "   " << rainfall[month] << " \n " ;
  for (month=0;month<rainfall.size();++month)
    sum = sum + rainfall[month];
  average = sum/12;
  cout << " Average is " << average<< endl;
  t=rainfall;
  for (month=0;month<t.size();++month)
    cout << t[month] << endl;

  return(0);
}
```

We will be covering more examples of vectors in the chapter on the Standard Template Library. These examples require material that we will be covering in subsequent chapters.

9.4 Valarrays

C++ is used by people within the scientific community for numeric problems. The valarray class has been added to C++ to enable a range of numeric problems to be more easily solved in C++, and provide the high levels of performance often associated with Fortran code suites. We will look much more closely at this class in the chapter on the Standard Template Library.

9.4.1 Rainfall example using valarray

```
#include <iostream>
#include <valarray>
```

```
using namespace std;

int main()

{
float sum=0.0,average=0.0 ;
valarray<float> rainfall(12) ;
int month ;

  cout << " Type in the rainfall, one per line \n" ;
  for (month=0;month<rainfall.size();++month)
    cin >> rainfall[month];
  for (month=0;month<rainfall.size();++month)
    cout << month << "   " << rainfall[month] << " \n " ;
  for (month=0;month<rainfall.size();++month)
    sum = sum + rainfall[month];
  average = sum/12;
  cout << " Average is " << average;
  return(0);
}
```

We will be returning to valarrays in the STL chapter.

9.5 Array Element Ordering in C++

Arrays are ordered by row in C++, unlike Fortran which orders by column. Mixed language programming requires care therefore.

9.6 Summary

There are some problems with C style arrays in C++, and include:–

- lack of array bounds checking and this is a major problem as this is is one of the most common programming errors;
- machine oriented array dimensioning, rather than physical world array dimensioning, e.g. stepping from 0–360 instead of -180–+180;

Some of the problems that occur with arrays in C++ go back to very early decisions made by Kernighan and Ritchie with C. We will look at the technical aspects in a later chapter after we have looked at pointers and functions. We will also look at some of the other problems associated with arrays in raw C++.

The vector class is a major advance for array handling in C++, and the vector class should be used in preference to the old C style array. The three features of interest are:–

- the size of an array is now an integral part of the array object;
- we can achieve array subscript checking using the at() notation;
- we can use try/catch blocks to help in locating errors;

The valarray class should be used when extremely high efficiency is required. The problems that occur with the old C style arrays and pointer aliasing make high optimisation difficult in C++. Certain assumptions are made about valarray usage that permit high levels of optimisation.

9.7 Key Concepts

9.7.1 Array, Vector and Valarray Data Types

	array	vector	valarray
Declaration	float x(10);	vector<float> x(10);	valarray<float> x(10);
Usage	x[i];	x[i]; x.at(i);	x[i];
Size	See note	x.size();	x.size();
Subscript Checking	no	yes	no

9.7.1.1 Note

The following program determines the size of an array in C++.

```
#include <iostream>
using namespace std;

int main()
{
float x[10];
int s;
  s=sizeof(x)/sizeof(float);
  cout <<s << endl;
  return(0);
}
```

9.7.1.2 Associated control structure – for loop

e.g. for (int i=0;i<10;++i)

9.8 Problems

1. Modify example one in this chapter as shown below.

```
  for (month=0;month < 20 ; ++month)
    cout << month << "   " << rainfall[month] << " \n " ;
  for (month=0;month < 20 ;++month)
    sum = sum + rainfall[month];
```

The key issue with the changes is that we are accessing memory outside of that allocated for the array. Do you get any error messages when running the program?

Did the program actually run to completion?

Here is the output from running the program on a pc with cygwin installed. We have entered the rainfall data into a file called ch0901.txt

```
$ a.exe < ch0901.txt
 Type in the rainfall, one per line
0  3.1
 1  2
 2  2.4
 3  2.1
 4  2.2
 5  2.2
 6  1.8
```

```
7   2.2
8   2.7
9   2.9
10   3.1
11   3.1
12   2.62884e-42
13   NaN
14   0
15   0
16   1.49615e+20
17   5.60519e-45
18   3.20874e-39
19   1.48003e+20
 Average is NaN
```

What error options are there during compilation?

2. Write a program that reads in a person's weight and a person's name. Try an array of size 10 in the first instance. Allow 20 characters for the person's name. Print out a table linking a person's name and weight.

3. Modify the program that calculates the total and average of people's weights to additionally read in their heights and calculate the total and average of their height. Use the data given below. This has been taken from a group of first year undergraduates.

```
1.85    85
1.80    76
1.85    85
1.70    90
1.75    69
1.67    83
1.55    64
1.63    57
1.79    65
1.78    76
```

Your body mass index is given by your weight divided by your height squared (all metric measurments). Calculate the BMI for each person.

Grades of obesity according to Garrow as follows:

Grade 0 (desirable) 20–24.9

Grade 1 (overweight) 25–29.9

Grade 2 (obese) 30–40

Grad 3 (morbidly obese) >40

Ideal BMI range,

Men, Range 20.1 – 25 kg/m^2

Women, Range 18.7 – 23.8 kg/m^2

When working on either a Unix system or a PC in a DOS box it is possible to use the following characters to enable you to read data from a file or write output to a file when running your program.

Character	Meaning
<	read from file
>	write to file

On a typical Unix system we could use:

a.out < data.dat > results.txt

to read the data from the file called data.dat and write the output to a file called results.txt.

On a PC in a DOS box the equivalent would be:

program.exe < data.dat > results.txt

This is a quick and dirty way of developing programs that do simple I/O, and we don't have to keep typing in the data and we also have a record of the behaviour of the program.

4. Modify the rainfall program (which assumes that the measurement is in inches) to convert the values to centimetres. One inch equals 2.54 centimetres. Print out the two sets of values as a table.

Hint: Use a second array to hold the metric measurements.

5. In an earlier chapter we used the following formula to calculate the period of a pendulum:

T=2* *sqrt(LENGTH/9.81)

Write a program that uses a loop to make the length go from 1 to 10 metres in 1 metre increments.

Produce a table with 2 columns, the first of lengths and the second of periods.

6. The following is a Fortran temperature conversion program. What do you think the output will be?

```
PROGRAM Convert
IMPLICIT NONE
INTEGER :: Fahrenheit
REAL :: Celsius
!   .
  DO Fahrenheit=-50,250
    Celsius=5/9*(Fahrenheit-32)
    PRINT *, Fahrenheit , Celsius
  ENDDO
!   ...
END PROGRAM Convert
```

Now compile and run the program to verify what you thought. Key temperatures are -40, 32 and 212 Fahrenheit.

Is this what you expected? Try replacing 5/9 with 5./9. and run again. Is this any better?

Now rewrite the program in C++ using 5/9, rather than 5./9. Why do we get different answers?

7. Write a program to convert from Celsius to Fahrenheit. The equation is:

Fahrenheit=32+9/5*Celsius

Use a loop to step the Celsius temperature from -50 to 100.

8. Combine the two temperature conversion programs. Does the program produce anything like the output below?

```
Celsius        T     Fahrenheit

   -45.56     -50    -58.00
   -45.00     -49    -56.20
   -44.44     -48    -54.40
   -43.89     -47    -52.60
   -43.33     -46    -50.80
   -42.78     -45    -49.00
   -42.22     -44    -47.20
   -41.67     -43    -45.40
   -41.11     -42    -43.60
   -40.56     -41    -41.80
   -40.00     -40    -40.00
   -39.44     -39    -38.20
   -38.89     -38    -36.40
   -38.33     -37    -34.60
   -37.78     -36    -32.80
   -37.22     -35    -31.00
   -36.67     -34    -29.20
   -36.11     -33    -27.40
   -35.56     -32    -25.60
   -35.00     -31    -23.80
   -34.44     -30    -22.00
```

This enables the middle column to be treated as either Celsius or Fahrenheit and the corresponding temperature read off easily.

We will look at ways of achieving the above in later chapters.

9. Write a program to either read the following data values into a 3*3 array or assign using a constructor.

```
1     2     3

4     5     6

7     8     9
```

Produce totals for each row and column. The output should be

```
1     2     3     6

4     5     6     15

7     8     9     24

12    15    18
```

There are a variety of ways of setting up the variables to do this.

- One would to use 3 variables for the row sums, say r1,r2,r2, and another 3 variables for the column sums, say c1,c2,c3.
- A second would be to use 2 one dimensional arrays rsum[3], csum[3]

There are a variety of ways of actually summing also.

- Hard code r1 as the sum of three array references for the first row
- using loops for r over the array x for the first row

For small scale problems you can get away with the hard coded approach. It does not scale very well though!

10
Control Structures

Summarising: as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure.
Edsger W. Dijkstra, Structured Programming.

Aims

The aims of this chapter are to introduce:–

selection between various courses of action as part of the problem solution

the concepts and statements in C++ needed to support the above. In particular:–

- logical expressions
- logical operators
- a *block* of statements
- several *blocks* of statements

the *if expression statement*

the i*f expression statement else statement;*

to introduce the *switch* statement with examples

iterative statements:–

- *while expression statement;*
- *do statement while expression;*
- the *for () statement*

the *break, continue* and *goto* statement

10 Control Structures

There are a reasonable range of control structures in C++. We need to review a number of other concepts before looking at them in depth.

10.1 Compound Statement or Block

A compound statement or block of statements is a sequence of statements enclosed in {}. A compound statement is treated as a single item and may appear anywhere that a single statement may occur.

10.2 Expression

An expression is made up of one or more operations. They in turn are a mixture of operands and operators. The evaluation of an expression typically means that one or more operations are carried out and a value is returned.

10.3 Boolean

Remember that within standard C++ boolean exists as a built in type and when we look at casting to an integer 0 is regarded as false and 1 is regarded as true. Non zero values are also regarded as true through backwards compatibility ith C.

10.4 if (expression) statement

Simple if statement. If the expression is true execute the statement that follows. Note that {} have to be used if it is necessary to execute multiple statements.

10.4.1 Example 1

```
if ( i>0 )
  cout << " Now greater than 0 " << endl;
```

If i is greater than 0 the cout statement is executed.

10.4.2 Example 2

```
#include <iostream>
using namespace std;

int main()
{
int i=0;
int j=10;
  cout << " i= " << i << endl;
  cout << " j= " << j << endl;
  if (i=1)
   j=99;
  cout << " i= " << i << endl;
  cout << " j= " << j << endl;
  return(0);
}
```

This is probably not what was intended. Remember that = is an operator and that expressions return values. In this case a boolean value. Thus what we have here is

```
i=1
j=99
```

This is not a C++ syntactic error. You asked for it and you got it.

10.5 if (expression) statement; else statement;

Standard extension to the if statement. Again {} have to be used if it is necessary to execute multiple statements.

10.5.1 Example 1

```
if ( i < 0 )
  cout << " Result not defined for negative values" << endl;
else
  cout << " Calculating for positive i " << endl;
```

One or other of the cout statements will be executed. Note the semi-colons. If you are familiar with Pascal or Ada this will catch you out.

10.5.2 Example 2

```
if ( i < 0 )
  cout << " Entering negative region" << endl;
else if ( i == 0 )
  cout << " crossover reached " << endl;
else
  cout << " Positive region entered " << endl ;
```

Note the semi-colons. Again will catch the Pascal/Modula 2 programmer out.

10.6 switch (expression) statement

This is best illustrated with a simple example.

10.6.1 Example 1

```
#include <iostream>
using namespace std;

int main()
{
int i;
  cout << " Type in an integer value " ;
  cin >> i;
  cout << endl;
  switch (i)
  {case 1 : cout << " one entered " << endl;
    break;
   case 2 : cout << " two entered " << endl;
    break;
   case 3 : cout << " three entered " << endl;
    break;
   default: cout <<
    " number other than 1,2 or 3 entered " << endl;
    break;
  }
// note no terminating semi-colon
  return(0);
}
```

Note the use of the break statement to exit from the switch statement, otherwise execution simply drops through!

Equivalent to the case statement in other languages.

10.7 while (expression) statement

Conventional while statement, i.e. statement may never be executed. This example highlights both the if else statement and the while statement. It is a direct translation of the Fortran 90 example.

10.7.1 Example 1

```
#include <iostream>
using namespace std;

int main()
{
int a[11];
int mark;
int end;
int i;
   cin >> mark;
   cin >> end;
   for (i=0; i < end ; ++i)
    cin >> a[i];
   i=0;
   a[end]=mark;
   while (mark != a[i]) ++i;
   if (i == end )
    cout << " Item not in list " << endl;
   else
    cout << " Item at position " << i << endl;
   return(0);
}
```

We will look at the use of a sentinel in a later example.

10.8 do statement while (expression);

Equivalent to repeat until statement, i.e. the loop is always executed at least once as the test is at the end of the loop. This example is the e^x function taken from the Fortran 90 course.

10.8.1 Example 1

```
#include <iostream>
using namespace std;

int main()
{
float term,x,etox;
int nterm;
const float tol=1.0E-6;
   etox=1.0;
   term=1.0;
   nterm=0;
```

```
   cin >> x;
   cout << endl;
   do
   {nterm+=1;
    term = (x/nterm) * term;
    etox+=term;
   }
   while (term > tol);
   cout << " eotx = " << etox << endl;
   return(0);
}
```

10.9 for (init-statement;expression 1; expression 2) statement

Equivalent to the DO loop in Fortran or FOR loop in the Pascal family of languages. Note that {} must be used when multiple statements need to be executed.

ini-statement may be declaration or an expression. This enables us to introduce for loop control variables at the time we set up the for loop. Some people love the ability to introduce variables in this way, others hate it. It is the way it is.

The following highlight this point–

```
for (int i=0;
```

Declare i and initialise to 0.

```
for ( i=0;
```

i must have been declared prior to this statement.

expression 1 is the loop control mechanism. As long as this is true the for statement will be executed.

expression 2 is evaluated after each loop and is generally used to modify the for loop control variable.

10.9.1 Example 1

This complete program illustrates the above.

```
#include <iostream>
using namespace std;

int main()
{
   float x[10];
   int i;
   cout << " Type in 10 numbers \n";
   for ( i=0 ; i < 10 ; i++)
     cin >> x[i];
   cout << " Numbers were \n ";
   for ( i=0 ; i < 10 ; i++)
     cout <<  x[i] << "  ";
   cout << endl;
   return(0);
}
```

10.9.2 Example 2

```
#include <iostream>
using namespace std;

int main()
{
  for (;;)
    cout << " Hello world \n " ;
    return(0);
}
```

What happens here?

10.10 break, continue, goto statements

The break statement can only occur within a switch or loop (for, do or while). Note however that you can only break from a single loop. If it is necessary to terminate an action within nested loops then the break won't do what we want.

The continue statement only makes sense with a loop. Control passes immediately to the appropriate depending on what type of loop we are in.

The goto will allow us to jump to a labelled statement. As the use of this statement is bad programming practice, we won't cover it at all. We will however look at exception handling in a later chapter. This is something that may not be fully implemented in the compiler you are using.

The following is a complete program that illustrates the use of all three statements. Type the program in and run it to see what happens.

```
#include <iostream>
using namespace std;

int main()
{
int i;
int j[]={0,1,2,3,4,5,6,7,8,9};
  for (i=0;i<10;i++)
  {
   if (j[i]>5) goto end;
   cout << i << " " << j[i] << endl;
   continue;
   end: cout << " j[i] now greater than 5 " << endl;
   break;
  }
  return(0);
}
```

10.11 Summary

C++ has a decent set of control structures. They provide us with most of the functionality we require.

10.12 Key Concepts

10.12.1 logical expressions

An expression that evaluates to true or false.

10.12.2 logical and relational operators

equal – ==

not equal – !=

less than – <

less than or equal – <=

greater than – >

greater than or equal – >=

not – !

and – &&

or – ||

10.12.3 A block of statements – { .;.;.}

10.12.4 Control Statements

10.12.4.1 the if expression statement

10.12.4.2 the if expression statement else statement

10.12.4.3 the switch statement

10.12.4.4 while expression statement

10.12.4.5 do statement while expression

10.12.4.6 the for () statement

10.12.4.7 break statement

10.12.4.8 continue statement

10.12.4.9 goto statement

10.13 Problems

1. Write a program to print out the 12 times table. Output should be roughly of the form

```
1    *    12    =    12
2    *    12    =    24
```

2. Write a program that produces a conversion table from litres to pints and vice versa. One litre is approximately 1 3/4 pints. The output should comprise three columns. The middle column should be an integer and the columns to the left and right should be the corresponding pints and litre values. This enables the middle column to be scanned quickly and the corresponding equivalent in litres or pints read easily.

3. Rewrite the program for the period of a pendulum. The new program should print out the length of the pendulum and period for lengths of the pendulum from 0 to 100 cm in steps of 0.5 cm.

The physical world has many examples where processes require some threshold to be overcome before they begin operation: critical mass in nuclear reactions, a given slope to be exceeded before friction is overcome, and so on. Unfortunately, most of these sorts of calculations become rather complex and not really appropriate here. The following problem tries to restrict the range of calculation, whilst illustrating the possibilities of decision making.

4. If a cubic equation is expressed as

$$z^3 \quad a_2 z^2 \quad a_1 z \quad a_0 \quad 0$$

and we let

$$q \quad a_1 / 3 \quad (a_2 * a_2) / 9$$

and

$$r \quad (a_1 a_2 \quad 3 a_0) / 6 \quad (a_2 \quad a_2 \quad a_2) / 27$$

we can determine the nature of the roots as follows:

$q^3 + r^2 > 0$; one real root and a pair of complex;

$q^3 + r^2 = 0$; all roots real, and at least two equal;

$q^3 + r^2 < 0$; all roots real;

Incorporate this into a suitable program, to determine the nature of the roots of a cubic from suitable input.

5. The form of breaking waves on beaches is a continuum, but for convenience we commonly recognise three major types: surging, plunging and spilling. These may be classified empirically by reference to the wave period, T (seconds), the breaker wave height, H_b (metres), and the beach slope, m. These three variables are combined into a single parameter, B, where

$B = H_b/(gmT^2)$

g is the gravitational constant (981 cm sec^{-2}). If B is less than .003, the breakers are surging; if B is greater than 0.068, they are spilling, and between these values, plunging breakers are observed.

(i) On the east coast of New Zealand, the normal pattern of waves is swell waves, with wave heights of 1 to 2 metres, and wave periods of 10 to 15 seconds. During storms, the wave period is generally shorter, say 6 to 8 seconds, and the wave heights higher, 3 to 5 metres. The beach slope may be taken as about 0.1. What changes occur in breaker characteristics as a storm builds up?

(ii) Similarly, many beaches have a concave profile. The lower beach generally has a very low slope, say less than 1 degree (m=0.018), but towards the high tide mark, the slope increases dramatically, to say 10 degrees or more (m=0.18). What changes in wave type will be observed as the tide comes in?

6. Personal taxation is usually structured in the following way:–

no taxation on the first m_0 units of income;

taxation at t_1% on the next m_1 units;

taxation at t_2% on the next m_2 units;

taxation at t_3% on anything above.

For some reason, this is termed *progressive* taxation. Write a generalised program to determine net income after tax deductions. Write out the gross income, the deductions and the net income. You will have to make some realistic estimates of the tax thresholds m_i and the

taxation levels t_i. You could use this sort of model to find out how sensitive revenue from taxation was in relation to cosmetic changes in thresholds and tax rates.

8. The specific heat capacity of water is 2009 J kg^{-1} K^{-1}; the specific latent heat of fusion (ice/water) is 335 kJ kg^{-1}, and the specific latent heat of vaporization (water/steam) is 2500 kJ kg^{-1}. Assume that the specific heat capacity of ice and steam are identical to that of water. Write a program which will read in two temperatures, and will calculate the energy required to raise (or lower) ice, water or steam at the first temperature, to ice, water or steam at the second. Take the freezing point of water as 273 K, and its boiling point as 373 K. For those happier with Celsius, 0^o C is 273 K, while 100^o c is 373 K. One calorie is 4.1868 J, and for the truly atavistic, 1 BTU is 1055 J (approximately).

10.14 Bibliography

Dahl O. J., Dijkstra E. W., Hoare C. A. R., *Structured Programming*, Academic Press, 1972.

- This is the original text, and a must. The quote at the start of the chapter by Dijkstra summarises beautifully our limitations when programming and the discipline we must have to successfully master programming.

Knuth D. E., *Structured Programming with GOTO Statements*, in Current Trends in Programming Methodology, Volume 1, Prentice Hall.

- The chapter by Knuth provides a very succinct coverage of the arguments for the adoption of structured programming, and dispells many of the myths concerning the use of the GOTO statement. Highly recommended.

11
Pointers

The question naturally arises whether the analogy can be extended to a data structure corresponding to recursive procedures. A value of such a type would be permitted to contain more than one component that belongs to the same type as itself; in the same way that a recursive procedure can call itself recursively from more than one place in its own body.

C.A.R. Hoare, Structured Programming

Aims

The primary aims of the chapter are:–

to look at pointer syntax in C++ using the * operator;

to look at the associated address of operator &;

to look at the use of pointers and arrays;

to look at auto_ptr

11 Pointers

An understanding of pointers is essential for successful use of C++. We will look in this chapter at some of the fundamentals of pointers and their usage in C++.

11.1 Example 1: Basic Pointer Usage

This is a simple example that introduces two of the fundamental operators in C++:–

*	or pointer to
&	or address of

You have to type this program in, compile it and run it to get a feel for what is happening.

```
#include <iostream>

using namespace std;

int main()
{
int i=999;
int* p_i;
  p_i=&i;
  cout << " i " << i << endl;
  cout << " address of i " << &i << endl;
  cout << " value of p_i " << p_i << endl;
  cout << " value of &p_i " << &p_i << endl;
  cout << " value of *p_i " << *p_i << endl;
  *p_i=1010; // alter the value at the address p_i
  cout << i << endl;
  cout << &i << endl;
  cout << p_i << endl;
  cout << *p_i << endl;
  return(0);
}
```

This program will give different results on different compilers even on the same platform. Try it out on more than one platform.

The following is based on output from one of the compilers I use.

Program Variable	Memory Address	Memory Contents
i	0x0012FF7C	999
p_i	0x0012FF78	0x0012FF7C

It is useful here to think about what happens when we compile a program and actually run it. After the compilation and link stage we end up with a file that contains what is called machine code. This is the language that the underlying hardware – be it a pc, Macintosh or UNIX workstation, actually uses. The compiled program is then loaded into memory and executed. Parts of the compiled program are executable code, other parts are data – our variables.

For each variable in our program there will be an area of memory set aside. When you look at the source of the program you think in terms of the variable name. When we work with

the executable program we then move to addresses, rather than variable names. Thus each variable has an associated address in memory. The contents of this address are our actual data values. Don't worry if this all appears baffling things will get clearer eventually!

11.2 Example 2: Arrays and Pointers

This example looks at the use of pointers with arrays. Serious usage of arrays in C++ is not possible without a very good understanding of pointers and pointer arithmetic. This example looks at three ways of achieving the same result with a one dimensional array.

```
#include <iostream>
using namespace std;

int main()
{
float sum=0.0,x[10],*pstart,*pend;
   cout << " Standard array access notation \n";
   for (int i=0;i<10;++i)
   {
    x[i]=i;
    sum += x[i];
    cout << i << " " << x[i] << " " << sum << "\n" ;
   }
   sum=0.0;
   pstart = &x[0];
   pend   = pstart+10;
   cout << " Using &x[0] for start address \n";
   while (pstart < pend)
   {
    sum += *pstart++;
    cout << sum << "\n";
   }
   sum=0.0;
   pstart = x;
   pend   = pstart+10;
   cout << " Using x for start address \n";
   while (pstart < pend)
   {
    sum += *pstart++;
    cout << sum << "\n";
   }
   return(0);
}
```

This highlights quite clearly that if x is an array that x is nothing more in C++ terminology than the base address of the array, and that x is identical in meaning to &x[0]. This was a deliberate design decision in C by Kernighan and Ritchie.

We will look into this whole area in far greater detail when we look at functions and arrays as function arguments in the next chapter.

11.3 Example 3: Pointers and Sentinels

We looked earlier at an example that used a sentinel. In the following example we again use a sentinel. Remember:–

- arrays of char are zero byte terminated;
- zero is classified as false in C++;
- the = is an operator
- Now consider the following program:–

```
#include <iostream>
using namespace std;
int main()
{
char line1[]=" This is a line of text";
char line2[80];
char* p1=line1;
char* p2=line2;
  while ( *p2++ = *p1++);
  cout << line1 << endl;
  cout << line2 << endl;
  return(0);
}
```

We define line1 and line2 to be character arrays.

We define p1 and p2 to be pointers to characters.

The while loop is the major thing of interest in the program. This will copy the contents of the line1 array to the line2 array, ending at the zero byte at the end of the line. Remember that a boolean can be mapped onto integers and 0 is false. So when we hit the 0 byte terminating the string the expression (*p2++ = *p1++) returns 0, and is false.

C++ and C are hated and loved for permitting such terse expressions. Expressions like this are quite common in existing C and C++ code. You need to have a good understanding of operators and expressions in C++. The above represents a very efficient way of copying a data structure, combing the use of sentinels, pointers and C++ expression syntax.

11.4 Example 4: Indiscriminate Pointer Usage

This example shows how one can use the two operators & and * quite indiscriminately in a C++ program. Pointers are restrictive in Fortran 90, and can't be used without a definition of what they point to.

```
#include <iostream>
using namespace std;
int main()
{
int i=1;
double x=1.0;
float  z=1.0;
  cout << " Address of i is " << &i
    << " value of i is   " << i << " \n "
    << " using pointer to access the value of i "
    << *&i << endl;
```

```
   cout << " address of x is " << &x
     << " value is "         << x << " \n"
     << " using pointer to access the value of x "
     << *&x << endl;
   cout << " address of z is " << &z
     << " value is "         << z << " \n"
     < " using pointer to access the value of z "
     << *&z << endl;
   return(0);
}
```

Type this program in, compile and run it to see what happens.

11.5 References

Chapter 5 introduced the concept of a reference. In one sense a reference is a kind of safer pointer. There is no such thing as a null reference. A reference must always refer to something. Another important difference between pointers and references is that pointers may be assigned to different objects. A reference always refers to the object with which it is initialized. We will look into pointers and references in more depth in later chapters.

11.6 auto_ptr

C++ also has a so called smart pointer. This will be looked at in a later chapter.

11.7 Summary

This chapter has only introduced the concept of a pointer in C++. In later chapters we will look at more examples of pointer usage.

11.8 Key Concepts

11.8.1 * – pointer to

11.8.2 & – address of

int i=0;

int *p_i=&i;

11.8.3 arrays and pointers

when x is an array x &x[0]

11.9 Problems

To gain familiarity with the concepts of pointers in C++ try out all of the examples in this chapter.

If you have access to another system then try that system too. You need to get familiar with the address format for the system that you use.

Look at the following example. Will this program compile? What will happen if it does compile and you run it? Verify what you think by typing it in and compiling it. As a crude yardstick the more error messages you get the better the compiler.

```
#include <iostream>
using namespace std;

int main()
{
```

```
int *p1,p2;
char c1,*c2;
   p1 = " Mayhem";
   c2=0;
   p1=0;
   c2=&'a';
   p1=p2;
   c2='\0';
   p1='\0';
   c2=&c1;
   *p1=&p2;
   return(0);
}
```

We will look at more realistic examples later in the course.

12
Functions

I can call spirits from the vasty deep.
Why so can I, or so can any man; but will they come
when you do call for them?
William Shakespeare, King Henry IV, part 1

Aims

The aims of this chapter are:–

to introduce with examples some of the predefined functions available in C++

to introduce the concept of a user defined function;

to look at passing arrays as arguments to functions;

to look at passing functions as arguments to functions;

to look at scope rules in C++ for variables and functions;

to look at the wide range of standard functions provided with C++;

12 Functions

This is the major step forward in the ability to construct larger programs. We have to have a way of breaking problems down into smaller sub-problems. The terminology varies with programming languages, in Fortran we have functions and subroutines, in C++ we have functions. The basic idea is the same.

Where as in Fortran we distinguish between functions and subroutines we do not in C++. Every sub-program unit is a function. They either return a result of a particular type as in Fortran or carry out an action like a subroutine in Fortran. Functions in C++ that carry out actions return a so called void result. Let us look at some examples.

12.1 Predefined Functions

The predefined functions available in C++ are made available in groups and are provided in a standard library.

Use of all library functions requires the inclusion of a header file, which contains function prototypes, constant definitions and macros. We have already seen the use of <iostream> in all of the examples so far.

As C++ is backwards compatible with C we have access to all of the standard C header files.

The following are some of the standard header files:–

ctype.h, float.h, limits.h, math.h, stddef.h, stdio.h, stdlib.h, string.h, time.h

and these are also available in C.

Find out where the header files are on the system you work on and print some of them out. Have a look at math.h, float.h, limits.h and iostream.h at least.

<cmath> and <math.h>

These two headers provide the commonly required maths functions. They are described below:

- double abs(double x)
- double fabs(double x)
- double ceil(double x)
- double floor(double x)
- double sqrt(double x)
- double pow(double x,double y)
- double pow(double x, integer i)
- double cos(double x)
- double sin(double x)
- double tan(double x)
- double acos(double x)
- double asin(double x)
- double atan(double x)
- double atan2(double x,double y)
- double cosh(double x)
- double sinh(double x)

- double tanh(double x)
- double exp(double x)
- doublelog(double x)
- double log10(double x)
- double modf(double x, double* y)
- double frexp(double x,int* i)
- double fmod(double x,double y)
- double ldexp(double x,int i)

There are also float and long double versions.

The following are found in <cstdlib>

- int abs(int i)
- long abs(long l)
- long labs(long l)

Compare this with Fortran 95. Fortran has many more functions and they are generic. They can also take one to seven dimensional arrays of any of the supported numeric kinds types.

12.1.1 Trigonometric function usage

This example just uses the sine function from the maths library.

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
float x;
  cout << " Type in an angle \n";
  cin >> x;
  cout << " Sine of " << x << " is " << sin(x) << " \n ";
  return(0);
}
```

This is identical to function usage in Fortran.

12.1.2 Passing arguments of numeric type to standard maths functions

We can also pass arguments of a variety of numeric types to the standard maths library functions. We will look at how this is achieved later on.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
double z=1.0;
float  x=1.0;
int    i=1;
  cout << " Sine of double is " << sin(z) << endl ;
  cout << " Sine of float is " << sin(x) << endl ;
```

```
   cout << " Sine of int    is " << sin(i) << endl ;
   return(0);
}
```

C++ therefore supports numeric polymorphism, like Fortran.

12.1.3 Functions in <cmath>

These are the functions in <cmath>:–

acos asin atan atan2 cos cosh exp fabs floor fmod

frexp log log10 modf pow sin sinh sqrt tan tanh

and they are only a subset of those available in Fortran 90.

12.2 User Defined Functions

We can define our own functions and this is quite straightforward.

12.2.1 One d array as argument and one function

This simple example looks at a summation function that adds up all of the elements of the array passed to it.

```
#include <iostream>
using namespace std;

int sum(int x[],int n)
{
int t=0;
   for (int i=0;i<n;++i)
    t += x[i];
   return t;
}

int main()
{
int a[10]={1,2,3,4,5,6,7,8,9,10};
int
b[20]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
int nn=10;
int nnn=20;
   cout << sum(a,nn) << "\n";
   cout << sum(b,nnn) << "\n";
   return(0);
}
```

Note the use of the array constructor to provide initial values.

Note also the fact that we have the function sum as a whole, i.e. the interface

```
int sum(int x[],int n)
```

and actual body of executable code are all in one place. This is an effective working technique for many problems.

12.2.2 One function, vector as argument

This example is the same as the previous, but now we are using vectors.

```
#include <iostream>
#include <vector>
using namespace std;

int sum(vector<int> x)
{
   int i;
   int total=0;
   for (i=0;i<x.size();++i)
      total+=x[i];
   return (total);
}

int main()
{
   vector<int> a(10);
   vector<int> b(20);
   int i;
   for (i=0;i<a.size();++i)
      a[i]=i+1;
   for (i=0;i<b.size();++i)
      b[i]=i+1;
   cout << sum(a) << endl;
   cout << sum(b) << endl;
   return(0);
}
```

Note that there is no need to have a second argument as the size of the vector is an integral part of the vector object.

12.2.3 3 functions, one d array as argument

```
#include <iostream>
using namespace std;

void readdata(int a[],int n);

int  sum(int a[],int n);

void printdata(int a[],int n);

void readdata(int a[],int n)
{
   for (int i=0;i<n;++i)
      cin >> a[i];
   return;
}

int  sum(int a[],int n)
{
int s;
   for (int i=0;i<n;++i)
```

```
    s+=a[i];
  return(s);
}

void printdata(int a[],int n
{
  for (int i=0;i<n;++i)
    cout << a[i] << "  ";
  cout << endl;
  return;
}

int main()
{
  int a[10];
  int n=10;
  int s=0;
  readdata(a,n);
  s=sum(a,n);
  printdata(a,n);
  cout << " sum is " << s << endl;
  return(0);
}
```

Note that the array can be any size in this example. It's size is inherited from the calling routine, main.

Note also that we have adopted an alternate coding style in this example. We have the interfaces to the three functions, followed by the actual complete code later. This is an effective coding style with larger programs.

12.2.4 Using 2d arrays as arguments

This example looks at several functions, each of which takes a 2 dimensional array as an argument.

```
#include <iostream>
using namespace std;

void readdata(int a[][5],int n,int m);

int sum(int a[][5],int n,int m);

void printdata(int a[][5],int n,int m);

void readdata(int a[][5],int n,int m)
{
  for (int i=0;i<n;++i)
   for (int j=0;j<m;++j)
    a[i][j]=i+j;
  return;
}
```

```
int  sum(int a[][5],int n,int m)
{
int s=0;
   for (int i=0;i<n;++i)
    for (int j=0;j<m;++j)
     s+= a[i][j];
   return(s);
}

void printdata(int a[][5],int n,int m)
{
   for (int i=0;i<n;++i)
   {
    for (int j=0;j<m;++j)
     cout << a[i][j] << "  ";
    cout << endl;
   }
   cout << endl;
   return;
}

int main()
{
int n=5;
int m=5;
int a[5][5];
int s=0;
   readdata(a,n,m);
   s=sum(a,n,m);
   printdata(a,n,m);
   cout << " sum is " << s << endl;
   return(0);
}
```

All array sizes apart from the first must be known at compile time.

Note also that we define the interfaces for the functions first and then have the implementations later. This aids in readability very considerably.

12.2.5 Passing 2d dynamic arrays as arguments

We can pass a 2 d dynamic array as an argument but we now have to use pointer arithmetic to achieve what we want in the called routine.

```
#include <iostream>
using namespace std;

void readdata(int* a,int n,int m);

int  sum(int* a,int n,int m);

void printdata(int* a,int n,int m);
```

```
void readdata(int* a,int n,int m)
{
   for (int i=0;i<n;++i)
    for (int j=0;j<m;++j)
     cin >> *(a+m*i+j);
   return;
}

int  sum(int* a,int n,int m)
{
int s=0;
   for (int i=0;i<n;++i)
    for (int j=0;j<m;++j)
     s+=*(a+m*i+j);
   return(s);
}

void printdata(int* a,int n,int m)
{
   for (int i=0;i<n;++i)
    for (int j=0;j<m;++j)
     cout << *(a+m*i+j) << "  ";
    cout << endl;
   return;
}

int main()
{
int n;
int m;
   cout << " R and C ";
   cin >> n >> m;
   cout << endl;
   int* a_ptr = new int [n*m] ;
   int s=0;
   readdata(a_ptr,n,m);
   s=sum(a_ptr,n,m);
   printdata(a_ptr,n,m);
   cout << " sum is " << s << endl;
   return(0);
}
```

All we have is an area of memory that will hold m*n integer values.

The expression

```
*(a+m*i+j)
```

replaces

```
a[i][j]
```

as a two d array indexing mechanism. The first expression is very machine oriented and is one of the reasons that array handling in C++ is not straightforward when the number of dimensions is greater than one.

12.2.6 Passing functions as arguments to other functions and procedure variables

We will look at two ways of achieving this in the example below.

```
#include <iostream>
using namespace std;

float f1(int i);

float f2(int i);

float f3(float (*f)(int i), int j);

float f1(int i)
{
return 1.0/i;
}

float f2(int i)
{
return 1.0/(i*i);
}

float f3(float (*f)(int i),int j)
{
float t=0;
   for (int k=1 ; k<j ; ++k)
    t+=(*f)(k);
   return t;
}
int main()
{
float (*t)(int i);
   cout << " Using f1 - simple reciprocal " << endl;
   cout << f3(f1,5) << endl;
   cout << " Using f2 - 1/(i*i) " << endl;
   cout << f3(f2,5) << endl;
   t=f1;
   cout << " using t=f1 " << endl;
   cout << f3(t,5) << endl;
   t=f2;
   cout << " using t=f2 " << endl;
   cout << f3(t,5) << endl;
   return(0);
}
```

The thing of major interest is the way we declare the function f3.

float f3(float (*f)(int i),int j);

This declares the first argument to the function to be a pointer to a function. This looks strange, and quite frankly it is. The syntax is due to the precedence of the operators in C++. Don't worry you do get used to it, it just takes time.

In the first part of the main program we call f3 with f1 and f2 as arguments. In the second part we call f3 with t as the first argument and use the fact that a function name is equivalent to the address of the function.

12.3 Function Arguments

The default parameter passing mechanism in C++ is pass by value, i.e. a copy of the value of each argument is taken and the function works with that.

The default parameter passing mechanism for array arguments in C++ is to pass the base address of the array. No copy is involved.

C++ provides two language features to help out here.

Firstly we can pass by reference. This avoids the copy. However this does mean that the function can now alter the parameters passed by reference.

Secondly we can qualify this by adding const. This means that the function cannot alter the parameters passed as const and by reference. This should generate a compile time error.

Let us look at a simple example of a function that swaps its two arguments and implement this in two ways.

12.3.1 Swapping arguments – passing by address

The first example passes the arguments using pointers. It forces the calling routine to pass the address of the arguments.

```
#include <iostream>
using namespace std;

void swap(int* p_i, int* p_j)
{
int t;
   t=*p_i;
   *p_i=*p_j;
   *p_j=t;
}

int main()
{
int i=1,j=2;
   cout < " i= " < i < " j= " < j < endl;
   swap(&i,&j);
   cout < " i= " < i < " j= " < j < endl;
   return(0);
}
```

12.3.2 Swapping arguments – passing by reference

This example passes by reference. The calling routine just passes the arguments through.

```
#include <iostream>
using namespace std;
```

```
void swap(int &i, int &j)
{
int t;
   t=i;
   i=j;
   j=t;
}

int main()
{
int i=1,j=2;
   cout < " i= " < i < " j= " < j < endl;
   swap(i,j);
   cout < " i= " < i < " j= " < j < endl;
   return(0);
}
```

If you need to alter the arguments in a function then pass by reference.

12.3.3 Mayhem

This example looks at the care that needs to be taken when mixing the various parameter passing methods.

```
#include <iostream>
using namespace std;

void f(int& a,int* b,int c)
{
cout << a << " " << b << " " << c << endl;
   a=1;
cout << a << " " << b << " " << c << endl;
   *b=2;
cout << a << " " << b << " " << c << endl;
   c=3;
cout << a << " " << b << " " << c << endl;
}

int main()
{
   int i=0;
   f(i,&i,i);
   cout << i << endl;
   return (0);
}
```

You must run this program to see what is actually happening. What is the value of i after calling the function f?

12.4 C++ Standard Functions

The C++ standard defines a number of libraries, and these include:–

- language support library

cstddef, limits, climits, cfloat, cstdlib, new, typeinfo, exception, csdtarg, csetjmp, ctime, csignal, cstdlib;

- diagnostics library

  stdexcept, cassert, cerrno;

- general utilities library

  utility, functional, memory, ctime;

- strings library

  string, cctype, cwctype, cstring, cwchar, cstdlib;

- localisation library

  locale, clocale;

- containers library

  bitset, deque, list, queue, stack, vector, map, set;

- iterators library

  iterator;

- algorithms library

  algorithm, cstdlib;

- numerics library

  complex, valarray, numeric, cmath, cstdlib;

- iolibrary

  iosfwd, iostream, ios, streambuf, istream, ostream, iomanip, sstream, cstdlib, fstream, cstdio, cwchar;

and you should have a look to see what has already been provided in the language before writing your own functions. The only reliable source of information I've found here is the standard.

12.5 Summary

We have only touched on some of what is possible in C++ in this chapter. We will look in later chapters at extending our knowledge after we have looked at classes.

12.6 Key Concepts

12.6.1 Predefined functions

There are a large number of predefined functions available in C++. It is good sense to develop some familiarity with them to avoid unnecessary work.

12.6.2 User Defined functions

It is possible to define your own functions. There are two main programming styles for doing this.

- interface and code body togethor
- interface and code body separate

the second tends to be used when solving larger problems.

12.6.3 Basic syntax

```
type name(parameters)
{
executable code
}
```

12.6.4 Parameter Passing

There are a number of parameter passing methods in C++. They are listed below.

12.6.4.1 Pass by value – copy made

This is the default for non-array arguments. This means that the original data in the calling routine can't be altered. There can be quite a performance overhead when passing large objects.

12.6.4.2 Array as parameter – base address used

The default for array arguments. The data in the called routine can be modified.

12.6.4.3 Pass by reference

Now can alter the data in the called routine, but the copy is avoided.

12.6.4.4 Pass by const reference

Now stop the called routine altering the data.

12.7 Problems

1. Write a program that prints out a table of values for sines, cosines and tangents from 0 degrees to 90 degrees in 1 degree intervals. There are a few catches here.

2. Print out math.h to get more information about the standard maths functions available in C++. Are they identical to those in Fortran or Pascal?

3. Modify the example that passes two d dynamic arrays (example 6) to work with three d arrays. Take care to ensure that you get the expression for calculating the position in the three d array correct. How long did it take you? Would you like to repeat this for 4, 5 and 6 d arrays?

4. Try some of the examples out. They are useful examples and can be used as sections of real programs.

13
Classes:
User Defined Types

Russell's theory of types leads to certain complexities in the foundations of mathematics,... Its interesting features for our purposes are that types are used to prevent certain erroneous expressions from being used in logical and mathematical formulae; and that a check against violation of type constraints can be made purely by scanning the text, without any knowledge of the value which a particular symbol might happen to have.
C. A. R. Hoare, Structured Programming.

It is said that Lisp programmers know that memory management is so important that it cannot be left to the users and C programmers know that memory management is so important that is cannot be left to the system
anon

Aims

The aim of this chapter is to introduce the concepts and ideas involved in using the facilities offered in C++ for the construction and use of user defined types or classes:–

concrete data types;

abstract data types;

constructors and destructors;

the C++ object model;

13 Classes – User Defined Data Types

User defined data types are an essential part of general purpose programming languages. Early languages provided this functionality via concrete data types, later languages by abstract data types.

A concrete data type is one where the user has to have an intimate knowledge of how the data type is implemented. Pascal is an early language to offer concrete data types. The date of the *Pascal Manual and Report* is 1975.

An abstract data type hides the implementation details from the user. Functionality is provided by procedures that manipulate the data. Modula 2 is a language that offered this and the first edition of *Programming in Modula 2* is 1982.

These are not new ideas!

Both have their strong points. If high efficiency is required then concrete data types are the method of choice. If data hiding is required then abstract data types are the method of choice.

Classes provide a way of achieving both. We will look in the first instance at the implementation of a range of user defined types using a concrete data structure approach. We will then look at implementing the same data types using an abstract approach.

13.1 Concrete Data Types

Some of these examples are taken from the Fortran 95 material.

13.1.1 Dates

The examples we have covered far have looked at the intrinsic types available in C++. In this chapter we look at the way we can add additional new types to a program, and extend the range of problems we can solve.

There are two stages in the process of creating and using our own data types, we must first define that type and second create variables of that type. Consider the following example.

```
#include <iostream>
using namespace std;

class date
{
   public:
   int day,month,year;
};

int main()
{
date d;
   d.day=1;
   d.month=1;
   d.year=1996;
   cout << d.day << "/";
   cout << d.month << "/";
   cout << d.year << endl;
   return(0);
}
```

New data types are defined in C++ using the class statement. In the above example we define a class of type date. This class has three component parts, day, month and year, all of type integer. Everything within a class is hidden or private by default. To make the components visible outside we use the public statement.

In the main program we then define a variable d to be of type date. The following statements assign values to the components of d, and then print out their values.

This is very similar to Fortran 90 and Pascal in syntax.

13.1.2 Addresses

This example looks at an address data type. We then create an array of addresses.

```
#include <iostream>
using namespace std;
class address
{
  public :
  char name[41];
  char street[81];
  char district[81];
  char city[41];
  char postcode[9];
};

int main()
{
address a[2];
  for (int  i=0 ; i<2 ; ++i)
  {
   cin >> a[i].name;
   cin >> a[i].street;
   cin >> a[i].district;
   cin >> a[i].city;
   cin >> a[i].postcode;
  }
  for (      i=0 ; i<2 ; ++i)
  {
   cout << a[i].name << endl;
   cout << a[i].street << endl ;
   cout << a[i].district << endl ;
   cout << a[i].city << endl ;
   cout << a[i].postcode << endl ;
  }
  return(0);
}
```

We here define the address data type to have components of name, street, district, city and postcode. Each of these is an array of characters, of varying lengths. We add one to the array length to allow for the string terminator in C++.

In the main program we then create a variable which is an array of addresses.

Again this is similar to the Fortran 90 and Pascal syntax.

13.1.3 Nested Data Types

In this example we nest data types. Here we have the two data types address and date of birth. We then use these within the personal data type.

```
#include <iostream>
using namespace std;

class address
{
  public :
  char street[81];
  char district[81];
  char city[41];
  char postcode[9];
};

class date_of_birth
{
  public:
  int day,month,year;
};

class personal
{
  public:
  char first_name[21];
  char surname[41];
  date_of_birth dob;
  address      add;
};

int main()
{
personal p_details;
  cin >> p_details.first_name ;
  cin >> p_details.surname;
  cin >> p_details.dob.day;
  cin >> p_details.dob.month;
  cin >> p_details.dob.year;
  cin >> p_details.add.street;
  cin >> p_details.add.district;
  cin >> p_details.add.city;
  cin >> p_details.add.postcode;
  cout << p_details.first_name << endl;
  cout << p_details.surname << endl;
  cout << p_details.dob.day << endl;
  cout << p_details.dob.month << endl;
  cout << p_details.dob.year << endl;
  cout << p_details.add.street << endl;
  cout << p_details.add.district << endl;
```

```
   cout << p_details.add.city << endl;
   cout << p_details.add.postcode << endl;
   return(0);
}
```

The syntax is very similar to Pascal and Fortran 90.

13.1.4 Concrete Time class

The following example is a concrete time class. It handles both 12 hour and 24 hour clocks.

```
#include <iostream>
using namespace std;

class Time
{
public :
   int hour, minute , second;
};

void print24(Time t)
{
   cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
   << ( t.minute < 10 ? "0" : "" ) << t.minute << ":"
   << ( t.second < 10 ? "0" : "" ) << t.second;
}

void print12(Time t)
{
cout << ( ( t.hour == 0 || t.hour == 12 ) ? 12 : t.hour %
12 )
<< ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
<< ":" << ( t.second < 10 ? "0" : "" ) << t.second
<< ( t.hour < 12 ? " AM" : " PM" );
}

int main()
{
    Time t;

    t.hour = 22;
    t.minute = 30;
    t.second = 0;

    cout << " Match of the day is at ";
    print24(t);
    cout << " in a 24 hour clock or "<< endl ;
    cout << "   " ;
    print12(t);
    cout << " in a 12 hour clock" << endl;
    return 0;
}
```

One of the problems at the end concerns converting this class to an abstract data type.

13.1.5 Reading user input using a singly linked list

This program will eat up all input and terminate with a control z or end of file. Run the program. What do you notice about the white space characters, i.e. spaces and end of lines?

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

struct link {
  char c;
  struct link *next;
};

int main()
{
  char c;
  struct link *first = NULL; /*  Start of list   */
  struct link *current; /*  End of list     */
  struct link *ptr; /*  Temporary       */

/* Loop reading characters until the end of file */
/* character is reached (EOF)                    */
/* Note that one cannot use eof() to check for   */
/* EOF until one has actually tried to read it   */
/* - unlike Pascal.                               */

  while (cin >> c ) { /*  Loop for all characters  */
   ptr = new (link);
    if (ptr == NULL) {
   cout << " Insufficient memory\n";
      exit(1);
    }

/*  Add new value and clear subsequent link  */

    ptr->c = c;
    ptr->next = NULL;

/*  Update pointers to add new value at end of list. */
/* The "if" statement could be omitted by making the */
/* first character entered a "special" case - as in  */
/* Example 10.1.4 - but generality is a good idea!  */

    if (first == NULL) /* If this is the 1st character */
      first = current = ptr;
    else /*  Otherwise... */
      current->next = ptr;
    current = ptr;
  }
```

```
/*  Now print out the list  */

  ptr = first;
  while (ptr != NULL) {
   cout << ptr->c;
    ptr = ptr->next;
  }
  cout << endl;
  return 0;
}
```

NULL is predefined within C++ and is used with pointers to terminate lists etc. Let us look at this program in some depth to convince ourselves that it will work as we want.

13.1.6 Reading user input using C++ style strings

The previous program read in all user input until and end of file was entered. It then echoed the input back to the user. What do you think the following program does?

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string s;
  char c;
  while ( cin.get(c) ) s+=c;
  cout << s << endl;
  return(0);
}
```

Which is the easier to understand?

13.2 Abstract Data Types

We look here at the same problems but now implement using an abstract data type approach. We are now looking at data and methods applied to that data. Access to the objects is restricted to a set of functions. These functions are called class member functions in C++ terminology.

13.2.1 Dates

```
#include <iostream>
using namespace std;

class date
{
  int day,month,year;
  public:
  date(int,int,int);
  void print_date();
};
```

```
date::date(int d=0,int m=0,int y=0)
{
   day=d;month=m;year=y;
}

void date::print_date()
{
   cout << day << "/" << month << "/" << year << endl;
}

int main()
{
date d(1,1,1996);
   d.print_date();
   d=date(11,2,1952);
   d.print_date();
   return(0);
}
```

Let us look quite carefully here at the differences.

Firstly within the class the three data components are no longer visible externally. Everything within a class is private or hidden by default.

We have two publicly visible components of this class, date (a so called constructor) and print_date. This means if we need to do anything with the date class that we have to use these two access mechanisms. We are now looking at binding together data and methods.

The class declaration, in this case, only has definitions. The implementation of the date constructor and print_date function occur later. This is a common C++ style where we do not want to clutter up the class definition with unnecessary baggage.

The next thing we have are the actual implementations for the date constructor and print_date function.

A constructor is the C++ way of providing a data type with initial values. It has the same name as the data type or class. In this case date. To identify that the date procedure is part of the date class definition we preface the constructor name with the class name, i.e. use date::date.

If we don't supply any parameters when we define a variable to be of type date then initial values are provided of 0,0 and 0 for day, month and year respectively.

If we do supply three parameters then these values will set the values for day, month and year respectively. Note that day, month and year have no visibility outside the date class.

The next procedure print_date has no type associated with it and is therefore declared to be a function of type void.

It takes no arguments. It will then print out the values of day, month and year.

Within the main program we define the variable d to be of type date, and use the constructor to provide an initial value.

The next statement quite clearly highlights the way in which we are binding together data and methods. The d.print_date statement invokes the print_date function with date values taken from the variable d.

The next statement assigns a new value to d.

The last statement prints out these new values.

This is quite an improvement over the earlier concrete data structure example. The main body of the program is greatly simplified, and it is much clearer as to what is actually happening. We have hidden behind the constructor and print_date function all of the messy implementation details of the date data type.

If we wanted to check for valid dates then we could do so within the constructor. Some possibilities include:–

- firstly checking days to be in the range 1-31 and months in the range 1-12;
- secondly checking day and month combinations;
- thirdly adding leap year checking;

Each of these would increase the security of any program that used this data type. This is set as a problem at the end of the chapter. There is also a coverage of the Gregorian calender.

13.2.2 Addresses

This example is a conversion of the earlier example of addresses to use abstract data types.

```
#include <iostream>
#include <string>
using namespace std;
class address
{
  char name[41];
  char street[81];
  char district[81];
  char city[41];
  char postcode[9];
  public :
  address (char name[],
       char street[],
       char district[],
       char city[],
       char postcode[]);
  void set_address();
  void print_address();
};

address::address(char n[]=" ",
            char s[]=" ",
            char d[]=" ",
            char c[]=" ",
            char p[]=" ")
{
  strcpy(name,n);
  strcpy(street,s);
  strcpy(district,d);
  strcpy(city,c);
  strcpy(postcode,p);
}
```

```
void address::set_address()
{
   cout << " name       " ; cin >>  name;
   cout << " street     " ; cin >>  street;
   cout << " district   " ; cin >>  district;
   cout << " city       " ; cin >>  city;
   cout << " postcode   " ; cin >>  postcode;
}

void address::print_address()
{
   cout << name      << endl;
   cout << street    << endl;
   cout << district << endl;
   cout << city      << endl;
   cout << postcode << endl;
}


int main()
{
address a[2];
a[0]=address(
"Ian Chivers","21 Winterwell Road","Brixton","London","SW2
5JB");
a[1]=address();
   for (int  i=0 ; i<2 ; ++i)
    a[i].print_address();
   for (      i=0 ; i<2 ; ++i)
    a[i].set_address();
   for (      i=0 ; i<2 ; ++i)
    a[i].print_address() ;
   return(0);
}
```

In this example we have a constructor and two functions, set_address and print_address.

The constructor will initialise variables of type address to be blank if no arguments are supplied to the constructor.

If arguments are supplied then we have to use the strcpy function in <string> to set the corresponding internal variables.

The set_address function prompts the use for values for the components of the address data type.

The print_address function prints out the values of the address variable.

Within the main program we define a to be an array of type address.

We then initialise a[0].

We then use the fact that if we don’t supply arguments to the constructor we set each component value to blank.

We then have a simple for loop that invokes the print_address function with the array a. Note we have a binding of data to a method.

We then use the set_address function to prompt the use for new values.

We then use the print_address function to print out these new values.

Again the main body of the program is greatly simplified.

13.3 Constructors and Destructors

It is necessary to look at what happens with classes when we use a class, i.e. what happens when objects are created and destroyed. There is the concept of a constructor and a destructor in C++ to handle this.

An object can be created/destroyed:–

- automatically each time a section of code is entered and/or exited;
- as a static object when the program starts and destroyed when the program ends;
- as a free store object using new/delete;
- as a member object – created as a member of another class or as an array element;

An object can also be constructed using an explicit constructor in an expression or as a temporary (see the standard r12.2 for a more complete coverage of this). In both of these cases the object is an automatic object.

13.3.1 Constructor/Destructor Example 1

The following examples are all based on the old C style string class. I've added an extra component to the class to enable us to see what is going on. The key point in the following examples are the problems that can occur when the class involves memory allocation and deallocation.

```
#include <iostream>
#include <string>

using namespace std;

class str
{
public :
char* p;
int size;

str(int sz)
{
  p=new char[size=sz] ;
  cout << " Constructor &p " << &p << " p = " << p  <<
endl;
}

~str()
{
  cout << " Destructor  &p " << &p << " p = " << p  <<
endl;
```

```
   delete[] p;
}

};

void f()
{
   str s1(10);
   strcpy(s1.p,"abcdefghi");
   str s2(20);
   strcpy(s2.p,"abcdefghijklmnopqrs");
   cout << " Address of s1 " << &s1 << endl;
   cout << " Address of s2 " << &s2 << endl;
   cout << " s1.p " << s1.p << endl;
   cout << " s2.p " << s2.p << endl;
   s1=s2;
   cout << endl << " After assignment " << endl << endl;
   cout << " s1.p " << s1.p << endl;
   cout << " s2.p " << s2.p << endl;
   cout << " Address of s1 " << &s1 << endl;
   cout << " Address of s2 " << &s2 << endl;
   return;
}

int main()
{
   int i;
   cout << " Program starts \n";
   f();
   cout << " Function ended - returned to main. Type an inte-
ger to continue \n";
   cin >> i;
   return(0);
}
```

In this example we construct two string objects. We then assign values to the string component of this class. We then make the objects the same by the use of the assignment statement. As function f() terminates the destructor is called twice. The snag is that we deallocate the same object twice. Not what was intended.

The delete operator destroys an object created using the new operator. There are two forms of the delete operator, and these are:–

- delete cast-expression

and

- delete [] cast-expression

and the second form is used to delete arrays. The effect of deleting an array with the first form is undefined and the effect of deleting an individual object with the second form is undefined. Incorrect usage of delete will result in memory loss. We will look into this later.

13.3.2 Constructor/Destructor Example 2

This example corrects the previous problem by overloading the assignment operator to achieve the desired effect. Note that there are valid reasons for the bitwise direct copy in the previous example, e.g. with complex numbers.

```
#include <iostream>
#include <string>

using namespace std;

class str
{
public :
char* p;
int size;

str(int sz)
{
  p=new char[size=sz] ;
  cout << " Constructor &p " << &p << " p = " << p  <<
endl;
}

~str()
{
  cout << " Destructor  &p " << &p << " p = " << p  <<
endl;
  delete[] p;
}

str& operator=(const str&);

};

str& str::operator=(const str& a)
{
  cout << " In the assignment operator \n";
  if (this != &a)
  {
    delete[] p;
    p=new char[size=a.size];
    strcpy(p,a.p);
  }
  return *this;
}


void f()
{
  str s1(10);
```

```
  strcpy(s1.p,"abcdefghi");
  str s2(20);
  strcpy(s2.p,"abcdefghijklmnopqrs");
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  s1=s2;
  cout << endl << " After assignment " << endl << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  return;
}

int main()
{
  int i;
  cout << " Program starts \n";
  f();
  cout << " Function ended - returned to main. Type an inte-
ger to continue \n";
  cin >> i;
  return(0);
}
```

Note in the function that overloads the assignment operator the use of `this` – the pointer to the current object.

Note also the check to make sure assigning something to itself.

Note finally the use of strcpy to to the assignment.

13.3.3 Constructor/Destructor Example 3

This example introduces another subtle error. This one arise because of the difference between assignment and initialisation.

```
#include <iostream>
#include <string>

using namespace std;

class str
{
public :
char* p;
int size;

str(int sz)
{
  p=new char[size=sz] ;
```

```
  cout << " Constructor &p " << &p << " p = " << p  <<
endl;
}

~str()
{
  cout << " Destructor  &p " << &p << " p = " << p  <<
endl;
  delete[] p;
}

str& operator=(const str&);

};

str& str::operator=(const str& a)
{
  cout << " In the assignment operator \n";
  if (this != &a)
  {
    delete[] p;
    p=new char[size=a.size];
    strcpy(p,a.p);
  }
  return *this;
}


void f()
{
  str s1(10);
  strcpy(s1.p,"abcdefghi");
  str s2(20);
  strcpy(s2.p,"abcdefghijklmnopqrs");
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  s1=s2;
  cout << endl << " After assignment " << endl << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  str s3=s2;
  cout << endl << " After initialisation " << endl << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  cout << " s3.p " << s3.p << endl;
  cout << " Address of s1 " << &s1 << endl;
```

```
  cout << " Address of s2 " << &s2 << endl;
  cout << " Address of s3 " << &s3 << endl;
  return;
}

int main()
{
  int i;
  cout << " Program starts \n";
  f();
  cout << " Function ended - returned to main. Type an inte-
ger to continue \n";
  cin >> i;
  return(0);
}
```

The problem in this example occurs with

```
string s3=s2
```

In this example the user defined assignment operator is NOT applied to an uninitialised object. This is reasonable as the pointer would contain a random value.

13.3.4 Constructor/Destructor Example 4

This example corrects the previous problem, by the provision of what is called a copy constructor.

```
#include <iostream>
#include <string>

using namespace std;

class str
{
public :
char* p;
int size;

str(int sz)
{
  p=new char[size=sz] ;
  cout << " Constructor &p " << &p << " p = " << p  <<
endl;
}

~str()
{
  cout << " Destructor  &p " << &p << " p = " << p  <<
endl;
  delete[] p;
}
```

```
str& operator=(const str&);

str(const str&);

};

str& str::operator=(const str& a)
{
  cout << " In the assignment operator \n";
  if (this != &a)
  {
    delete[] p;
    p=new char[size=a.size];
    strcpy(p,a.p);
  }
  return *this;
}

str::str(const str& c)
{
  cout << " copy constructor \n";
  p=new char[size=c.size];
  strcpy(p,c.p);
}

void f()
{
  str s1(10);
  strcpy(s1.p,"abcdefghi");
  str s2(20);
  strcpy(s2.p,"abcdefghijklmnopqrs");
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  s1=s2;
  cout << endl << " After assignment " << endl << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  str s3=s2;
  cout << endl << " After initialisation " << endl << endl;
  cout << " s1.p " << s1.p << endl;
  cout << " s2.p " << s2.p << endl;
  cout << " s3.p " << s3.p << endl;
  cout << " Address of s1 " << &s1 << endl;
  cout << " Address of s2 " << &s2 << endl;
  cout << " Address of s3 " << &s3 << endl;
  return;
```

```
}

int main()
{
  int i;
  cout << " Program starts \n";
  f();
  cout << " Function ended - returned to main. Type an inte-
ger to continue \n";
  cin >> i;
  return(0);
}
```

The copy constructor header is

```
string::string(const string& c)
```

and the provision of a copy constructor overcomes the problem in the previous example.

13.3.5 Constructor/Destructor Example 5

The following example highlights the sequence of events when calling functions with objects as arguments. The example looks at a variety of different ways of doing this to illustrate some of the flexibility (and complexity!) of C++.

```
#include <iostream>
using namespace std;

class c
{
   private :
      int i;
   public  :

      c(int ii)
      {
         cout << " ctor ";
         i=ii;
      }

      c(const c& c)
      {
         cout << " copy ctor ";
         i=c.i;
      }

      c& c::operator=(const c& c)
      {       {
         cout << " = ";
         i=c.i;
         return *this;
      }

      ~c()
```

```
        {
            cout << " dtor ";
        }

        void seti(int ii) { i=ii;}

        void print();

        c add1(const c& c);

        c add2(const c& c1,const c& c2);

};

void c::print()
{
    cout << i << " ";
}

c c::add1(const c& c1)
{
   return c(i+c1.i);
}

c c::add2(const c& c1,const c& c2)
{
   i=c1.i+c2.i;
   return *this;
}

c f1(const c& c)
{
    cout << " f1 " ;
    return c;
}

c f2(c c)
{
    cout << " f2 " ;
    return c;
}

int main()
{
    c c1(1);
    c c2(2);
    c c3(3);cout << endl;
    c1.print();
    c2.print();
    c3.print();cout << endl;
```

```
    c2=c1;
    cout << endl;
    c1.print();
    c2.print();
    c3.print();cout << endl;
    c3=f1(c2);
    cout << endl;
    c1.print();
    c2.print();
    c3.print();cout << endl;
    c1.seti(10);c2.seti(20);c3.seti(30);
    c1.print();
    c2.print();
    c3.print();cout << endl;
    c3=f2(c2);
    cout << endl;
    c1.print();
    c2.print();
    c3.print();cout << endl;
    c1=c2.add1(c3);
    cout << endl;
    c1.print();
    c2.print();
    c1.seti(100);c2.seti(200);c3.seti(300);
    c1.add2(c2,c3);
    cout << endl;
    c1.print();
    c2.print();
    c3.print();cout << endl;
    return (0);
}
```

You must run this example to see what is actually happening as we invoke the constructors and call a variety of functions.

13.3.6 Constructor/Destructor Example 6

Now consider the following. Can you predict what the output of the program will be?

```
#include <iostream>
#include <string>
using namespace std;

class String
{
private:
  char *str;
public:
  String()
  {
  cout << " In null constructor for String \n";
  str=new char[30];
```

```
  str[0]='\0';
  }
  String(char *s)
  {
  cout << " In one parameter constructor for String\n";
  strcpy(str=new char[strlen(s) +1],s);
  }
  ~String()
  {
  cout << " In destructor for class String\n";
  delete str;
  }
  // other methods go here
};

class A
{
private:
  String a1;
public :
  A() { cout << " In null constructor fpr class A\n";}
  A(char *s) : a1(s)
  {
  cout << " In 1 parameter constructor for class A\n";
  }
};

class B
{
private:
  A b1;
public :
  B() { cout << " In null constructor fpr class B\n";}
  B(char *s) : b1(s)
  {
  cout << " In 1 parameter constructor for class B\n";
  }
};

class E
{
private :
  String e1;
  String e2;
public :
  E() { cout << " In null constructor for class E\n";}
  E(char *str1,char *str2): e1(str1),e2(str2)
  {
  cout << " In 2 parameter constructor for class E\n";}
};
```

```
class C : public B
{
private :
  String c1;
public :
  C() { cout << " In null constructor for class C\n"; }
  C(char *str1,char *str2) : B(str1), c1(str2)
  {
  cout << " In 2 parameter constructor for class C\n";
  }
};

class D : public A
{
private :
  C d1;
  E d2;
public :
  D() { cout << " In null constructor for class D\n";}
  D(char *str1,char *str2,char *str3,char *str4,char *str5)
  : A(str1),d1(str2,str3),d2(str4,str5)
  { cout << " In 5 parameter constructor for class D\n";}
};

int main()
{
  D obj1;
  D obj2("one","two","three","four","five");
  return(0);
}
```

Run this example to see what happens.

13.4 Sphere class

In this example we look at a sphere class and the emphasis is on how to pass objects and the related syntax.

```
#include <iostream>
#include <math>
using namespace std;

class sphere
{
public:
  sphere(double x,double y,double z,double r);
  sphere(const sphere& s);
  void get_centre(double &x,double &y,double &r);
  void set_centre(double,double,double);
  void get_radius(double &r);
  void set_radius(double);
```

```
   void print_sphere();
   double volume(void);
   void translate_with_copy_sphere(sphere s, double x, double
y, double z);
   void translate_by_reference_sphere(sphere &s,double x,dou-
ble    y,double z);
   void translate_by_pointer_sphere(sphere *p_s,double x,dou-
ble    y,double z);
private:
   double x_c,y_c,z_c,radius;
};

sphere::sphere(double x=0,double y=0,double z=0,double r=0)
{
   x_c=x;
   y_c=y;
   z_c=z;
   radius=r;
   cout << " Inside constructor " << endl;
}

sphere::sphere(const sphere& s)
{
   x_c=s.x_c;
   y_c=s.y_c;
   z_c=s.z_c;
   radius=s.radius;
   cout << " Inside copy constructor " << endl;
}

void sphere::get_centre(double &x,double &y,double &z)
{  x=x_c;
   y=y_c;
   z=z_c;
}

void sphere::set_centre(double x,double y,double z)
{
   x_c=x;
   y_c=y;
   z_c=z;
}

void sphere::get_radius(double &r)
{
   r=radius;
}

void sphere::set_radius(double r)
{
```

```
   radius=r;
}

void sphere::print_sphere()
{
   cout << " x = " << x_c << endl;
   cout << " y = " << y_c << endl;
   cout << " z = " << z_c << endl;
   cout << " radius = " << radius << endl;
}

double sphere::volume(void)
{
   return 4.0 / 3.0 * 3.1419265358 * pow(radius,3);

}

void sphere::translate_with_copy_sphere(sphere s, double x,
double y, double z)
{
   s.x_c=x_c+x;
   s.y_c=y_c+y;
   s.z_c=z_c+z;
   s.radius=radius;
}

void sphere::translate_by_reference_sphere(sphere &s,double
x,double y,double z)
{
   s.x_c=x_c+x;
   s.y_c=y_c+y;
   s.z_c=z_c+z;
   s.radius=radius;
}

void sphere::translate_by_pointer_sphere(sphere *pt_s,double
x,double y,double z)
{
   pt_s->x_c=x_c+x;
   pt_s->y_c=y_c+y;
   pt_s->z_c=z_c+z;
   pt_s->radius=radius;
}

int main()
{
   double x1=10,y1=10,z1=10,r1=10;
   double x2=20,y2=20,z2=20,r2=20;
   sphere s1(x1,y1,z1,r1);
   sphere s2(x2,y2,z2,r2);
```

```
   s1.print_sphere();
   s2.print_sphere();
   s1.translate_with_copy_sphere(s2,9,9,9);
   s1.print_sphere();
   s2.print_sphere();
   s1.translate_by_reference_sphere(s2,99,99,99);
   s1.print_sphere();
   s2.print_sphere();
   s1.translate_by_pointer_sphere(&s2,999,999,999);
   s1.print_sphere();
   s2.print_sphere();
   return 0;
}
```

The first thing we have are two constructors. We provide both a four argument constructor and a copy constructor.

The second thing we have are four member functions that provide read/write access to the x,y,z co-ordinates and radius of a sphere object.

The next member function provides a way of printing information about the objects.

The next member function calculates the volume of a sphere object.

Finally we have three member functions that provide ways of moving a sphere to a new position on the basis of the current position of another sphere object plus an additional x,y,z displacement, and these are called:–

- translate_with_copy_sphere
- translate_by_reference_sphere
- translate_by_pointer_sphere

We will look at each in turn.

13.4.1 translate_with_copy_sphere

This is the default parameter passing mechanism in C and C++. It is the intention that s2 is the object to be moved. This will not happen as a copy will be made. The copy constructor will be called.

13.4.2 translate_by_reference_sphere

The other way of passing a parameter is by reference. Now we pass the object as a parameter on the member function.

```
translate_by_reference_sphere(sphere &s,
double x,double y,double z)
```

Note the & – pass by reference.

This means that the procedure directly modifies the contents of the objects passed through and there is no intermediate copy.

Note also the syntax difference in the main program, i.e.

```
s1.translate_by_reference_sphere(s2,99,99,99)
```

It is s2 that is being moved to a new position.

13.4.3 translate_by_pointer_sphere

In this example we pass the parameter as a pointer to a sphere object.

```
translate_by_pointer_sphere(sphere *p_s,
double x,double y,double z);
```

Note the difference in the way we call this function.

```
s1.translate_by_pointer_sphere(&s2,999,999,999);
```

Now we are passing the address of the object.

Note the new syntax for accessing the component parts of the sphere object, i.e.

pt_s->x_c=x_c+x

where -> is regarded as a single lexical token in C++. Strictly speaking we could have written the following

(*pt_s).x_c=x_c+x

and this would have achieved the same effect. Why do we have the brackets round *pt_s? I guess that the -> was introduced to overcome this awkward syntax.

13.4.4 Constructor/Destructor Recommendations

So for a class of any real complexity it is recommended that the following minimal set of constructors and destructors be defined:–

```
class A {
  A();                          //simple constructor
  A(const A&);                  //copy constructor
  A& operator=(const A&)        //assignment
  ~A();                         //destructor };
```

It also pays to be very careful with parameter passing. Using const and reference variables can help in avoiding the creation of temporary copies of objects.

13.4.5 Memory Allocation and Deallocation – Leakage and Garbage Collection

There are typically three kinds of memory:–

- static memory for global variables;
- stack based memory for procedures and local variables;
- heap memory for objects referred to by pointers;

It is the third that normally causes problems, i.e. memory is incorrectly deallocated and memory gradually leaks away. There are two schools of thought here. One believes that the underlying system should take care of it – so called garbage collection behind the scenes. The other places the burden fairly and squarely on the user.

Oberon 2 does not leak memory. It uses garbage collection, and there is no dispose or deallocate statement in the language.

C++ puts the onus on the user – *caveat emptor*.

13.5 The C++ object Model – taken from the standard

Section 1.6 of the standards states *The constructs in a C++ program create, refer to, access and manipulate objects. An object is a region of storage and except for it fields occupies one or more contiguous bytes of storage. An object is created by a definition, by a new expression, or by the implementation when needed. The properties of an object are determined when the object is created. An object can have a name. An object has a storage duration which influences its lifetime. An object has a type. The term object type refers to the type with which the object is created. The object's type determines the number of bytes that the object occupies and the interpretation of its content. Some objects are polymorphic; the*

implementation generates information carried out in each object such that it makes it possible to determine that object's type during program execution. For other objects, the meaning of the values found therein is determined by the type of the expressions used to access them.

This obviously means that when we look at polymorphism there will be a run time overhead to determine the correct action during program execution, what method to invoke.

13.6 Summary

Classes provide us with a way of implementing user defined data types. We can do this in one of two ways. First using a concrete data structure approach where we need to have a good working knowledge of the internals of that data type, and secondly using abstract data types where the internals are hidden from use and access is via procedures or member functions.

Please note that we have not covered all aspects of class usage in this chapter. We will be providing more information in later chapters.

13.7 Key Concepts

13.7.1 Concrete data types

13.7.1.1 Data public

13.7.1.2 Functions public

13.7.2 Abstract data types

13.7.2.1 Data Private

13.7.2.2 Functions public

13.7.3 Constructors and Destructors

13.7.3.1 Simple constructor

13.7.3.2 Copy constructor

13.7.3.3 Overload assignment operator

13.7.3.4 Destructor

```
class A
{
  A();                        // simple constructor
  A(const A&)                 // copy constructor
  A& operator=(const A&);     //assignment
  ~A();                       //destructor
}
```

13.7.4 Basic Class Syntax

```
class name
{
  public: ...
;
  private: ...
;
};
```

13.8 Problems

1. The first program in this chapter looks at a concrete date data type. The type is made up of three integer components called day, month and year. As the program stands there is nothing to stop the assignment of illegal date values to these three components.

Modify the program to check for valid dates.

- Days are in the range 1-31, depending on the month
- Months are in the range 1-12
- Valid years depend on what calender you are using. We will use the Gregorian. More information about this calender is given in the bibliography.

What about leap years?

Because of the way that this example has been written there is nothing we can do to stop the assignment of invlaid values. Integers can take on a very wide range of valid values! The Pascal and Modula 2 languages have the concept of a number of data structuring facilities that would help out with the above example. The languages have ranges and sub ranges. Invalid values are then trapped (some of them at least) by a simple compiler switch.

2. The date example also exists as an abstract data type. Modify this program to check for valid dates. You have done most of the donkey work in the previous example.

Add a set date routine. Try this program out with three kinds of parameters:

- setdate(int d, int m, int y)
- setdate(int &d, int &m, int &y)
- setdate(const int &d, const int &m, const int &y)

I would create three setdate routines, one for each of the above types of parameters, e.,g. adate, bdate, cdate.

Modify the constructor to print out a message when it is invoked. In the first example a copy is made. In the second example the actual variable is passed. This could be modified. In the third example the variable is defined as a constant – you are not allowed to alter it within the setdate routine.

3. The third example in this chapter looks at nested data types. One of these is date_of_birth. Modify this program to check for a valid date of birth.

4. The sphere class in this chapter provides examples of some of the ways of passing parameters in C++. Modify the two constructors to output a text message showing when the constructor has been called. Run the program and examine the output.

5. Convert the concrete time class to an abstract time class. Consider the following issues:

- How many constructors should you use? Look at the issue of 0, 1, 2 and 3 arguments to a constructor. What default values would you choose?
- You should have a public function to set the time.
- Where do you check for a valid time? Should you distinguish between valid 12 hour times and valid 24 hour times?
- Should this be a separate function that can be called by both the set time routine and the one or more constructors?

The SQL standard actually has data types for both date and time.

6. Rewrite the linked list example as an abstract data type using classes.

13.9 Bibliography

To gain some idea about user defined data types there is a list of books below that look at this whole area. Note that most of these are not written from the perspective of C++, but from the concepts of data structuring and algorithms. Pascal is probably the dominant language used in teaching about data structuring and algorithms. Modula 2 sources are also widely available. This is because these are two of the major languages used in introductory computer science degrees. Ada and C++ are increasingly taught in CS departments, and books looking at algorithms and data structures in these languages will become more widely available.

Many numerical algorithm sources on the Internet are Fortran based.

Chivers I.D., *A Practical Introduction to Standard Pascal*, Ellis Horwood.

- A simple introduction to ISO/ANSI Pascal. Some of the data structures and algorithms used in this chapter are implemented in Pascal.

Chivers I.D., Sleightholme J., *Introducing Fortran 95, Springer Verlag.*

- As this course is aimed at people converting from Fortran 95 (as well as Pascal and other languages) examples used in that text are recast in C++ in this course.

Deitel H.M., Deitel P.J., *C++ How to Program*, Prentice Hall.

- The chapters on classes and data abstraction are worth looking at.

Ladd Scott Robert., *C++ Components and Algorithms*, M&T Books.

- Old style examples, but still worth a look at.

Schneider G.M., Bruell S.C., *Advanced Programming and Problem Solving with Pascal*, Wiley.

- The book is aimed at computer science students and follows the curriculum guidelines laid down in Communications of the ACM, August 1985, Course CS2. The book is very good for the complete beginner as the examples are very clearly laid out and well explained. There is a coverage of data structures, abstract data types and their implementation, algorithms for sorting and searching, the principles of software development as they relate to the specification, design, implementation and verification of programs in an orderly and disciplined fashion – their words.

Sedgewick R., *Algorithms in C++*, Addison Wesley.

- A C++ version of his original book *Algorithms*, 1983. This used Pascal.

Gregorian Calender

The calender used by the world today; a revised version of the Julian Calender, introduced by the Pope Gregory XIII in 1582.

In the Gregorian calendar, the tropical year is approximated as 365 97/400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar. The approximation 365 97/400 is achieved by having 97 leap years every 400 years. These are calculated as follows : Every year divisible by 4 is a leap year. However, every year divisible by 100 is not a leap year. However, every year divisible by 400 is a leap year after all.

The change-over from the Julian to the Gregorian calendar. The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October, and from then on the reformed calendar should be

used. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek orthodox countries didn't change until the start of this century.

It should be noted that the Gregorian Calendar is useless for astronomy because it has a ten-day hiatus in it. For the purpose of calculating positions backward in time, astronomers use the Julian Date Calendar. I found the following site very useful when researching this area.

http://www.geocities.com/CapeCanaveral/Lab/7671/index.html

The following list contains the dates for changes in a number of countries.

Albania: December 1912

Austria: Different regions on different dates

5 Oct 1583 was followed by 16 Oct 1583

14 Dec 1583 was followed by 25 Dec 1583

Belgium: Different authorities say 14 Dec 1582 was followed by 25 Dec 1582 and 21 Dec 1582 was followed by 1 Jan 1583

Bulgaria: Different authorities say Sometime in 1912 and Sometime in 1915

18 Mar 1916 was followed by 1 Apr 1916

China: Different authorities say 18 Dec 1911 was followed by 1 Jan 1912 and 18 Dec 1928 was followed by 1 Jan 1929

Czechoslovakia (i.e. Bohemia and Moravia): 6 Jan 1584 was followed by 17 Jan 1584

Denmark (including Norway): 18 Feb 1700 was followed by 1 Mar 1700

Egypt: 1875

Estonia: January 1918

Finland: Then part of Sweden. (Note, however, that Finland later became part of Russia, which then still used the Julian calendar. The Gregorian calendar remained official in Finland, but some use of the Julian calendar was made.)

France: 9 Dec 1582 was followed by 20 Dec 1582

Germany: Different states on different dates: Catholic states on various dates in 1583-1585 and Prussia: 22 Aug 1610 was followed by 2 Sep 1610 and Protestant states: 18 Feb 1700 was followed by 1 Mar 1700

Great Britain and Dominions (including what is now the USA): 2 Sep 1752 was followed by 14 Sep 1752

Greece: 9 Mar 1924 was followed by 23 Mar 1924

Hungary: 21 Oct 1587 was followed by 1 Nov 1587

Italy: 4 Oct 1582 was followed by 15 Oct 1582

Japan: Different authorities say: 19 Dec 1872 was followed by 1 Jan 1873 and 18 Dec 1918 was followed by 1 Jan 1919

Latvia: During German occupation 1915 to 1918

Lithuania: 1915

Luxembourg: 14 Dec 1582 was followed by 25 Dec 1582

Netherlands: Brabant, Flanders, Holland, Artois, Hennegau: 14 Dec 1582 was followed by 25 Dec 1582, Geldern, Friesland, Zeuthen, Groningen, Overysel: 30 Nov 1700 was followed by 12 Dec 1700

Norway: Then part of Denmark.

Poland: 4 Oct 1582 was followed by 15 Oct 1582

Portugal: 4 Oct 1582 was followed by 15 Oct 1582

Romania: 31 Mar 1919 was followed by 14 Apr 1919

Russia: 31 Jan 1918 was followed by 14 Feb 1918

Spain: 4 Oct 1582 was followed by 15 Oct 1582

Sweden (including Finland): 17 Feb 1753 was followed by 1 Mar 1753 (see note below)

Switzerland: Catholic cantons: 1583 or 1584. Zurich, Bern, Basel, Schafhausen, Neuchatel, Geneva: 31 Dec 1700 was followed by 12 Jan 1701. St Gallen: 1724

Turkey: 18 Dec 1926 was followed by 1 Jan 1927

USA: See Great Britain, of which it was then a colony.

Yugoslavia:14 January 1919 was followed by 28 January 1919 but parts of the country had changed over earlier.

Sweden has a curious history. Sweden decided to make a gradual change from the Julian to the Gregorian calendar. By dropping every leap year from 1700 through 1740 the eleven superfluous days would be omitted and from 1 Mar 1740 they would be in sync with the Gregorian calendar. (But in the meantime they would be in sync with nobody!) So 1700 (which should have been a leap year in the Julian calendar) was not a leap year in Sweden. However, by mistake 1704 and 1708 became leap years. This left Sweden out of synchronisation with both the Julian and the Gregorian world, so they decided to go ‘back’ to the Julian calendar. In order to do this, they inserted an extra day in 1712, making that year a double leap year! So in 1712, February had 30 days in Sweden. Later, in 1753, Sweden changed to the Gregorian calendar by dropping 11 days like everyone else.

14
Templates

A good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher.
Bertrand Russell.

Aims

The aims of the chapter are to look at the template feature provided in C++ with two examples:–

simple minimum;

quicksort;

14 Templates

Templates provide the ability to do generic programming, i.e. using a type as a parameter. This means the ability to write an algorithm that can be used by data of a variety of types, e.g. a minimum function, swap function or sort function that works with data of any type. Other terminology used includes polymorphic functions. In this chapter we will look at defining and using our own templates. In a latter chapter we will look at the standard template library (STL) and what that has to offer.

14.1 Simple minimum

This example looks for the minimum of the elements of an array.

```
template <class Type>
Type minimum(Type array[], int size)
{
   Type minimum_value=array[0];
   for (int i=1 ; i < size; ++i)
    if (array[i] < minimum_value)
      minimum_value = array[i];
   return minimum_value;
}

#include <iostream>
using namespace std;

int main () {
   int ia[]={10,1,9,2,8,3,7,4,6,5};
   double da[]={10.0,1.0,9.0,2.0,8.0,3.0,7.0,4.0,6.0,5.0};
   int isize = sizeof(ia) /sizeof(int);
   int ii=minimum(ia,isize);
   cout << " minimum for integer array is " << ii << endl;
   int dsize = sizeof(da) /sizeof(double);
   double di=minimum(da,dsize);
   cout << " minimum for real array is " << di << endl;
   return(0);
}
```

The first thing to look at are the two lines at the start of this example, i.e.

```
   template <class Type>
   Type minimum(Type array[], int size)
```

These two lines set up the function *minimum* to be a generic or polymorphic function. The function minimum returns a value of type *Type*, and takes two arguments, the first an array of type *Type*, the second an integer defining the size of the array.

This minimum function will work for any type where the < operator is defined. We will look at operator overloading in a later chapter.

The function then takes the first element as the smallest and sequentially scans the rest of the array comparing each element in turn with the smallest. If the new element is smaller they are swapped over and the scan continues.

The main program shows how the function is invoked with an array of integers, and then an array of reals.

14.2 Hoare's Quicksort

In this example we set up and use two generic functions using the template facility. The first swaps two elements in an array. The second is an implementation of Hoare's Quicksort.

```
template <class Type>
void swap(Type array[],int i, int j)
{
  Type tmp=array[i];
  array[i]=array[j];
  array[j]=tmp;
}

template <class Type>
void quicksort( Type array[], int l, int r)
{
int i=l;
int j=r;
Type v=array[int((l+r)/2)];
for (;;)
{
  while (array[i] < v) i=i+1;
  while (v < array[j]) j=j-1;
  if (i<=j) { swap(array,i,j); i=i+1 ; j=j-1; }
  if (i>j) goto ended ;
}
ended: ;
if (l<j) quicksort(array,l,j);
if (i<r) quicksort(array,i,r);
}

template <class Type>
void print(Type array[],int size)
{
  cout << " [ " ;
  for (int ix=0;ix<size; ++ix)
   cout << array[ix] << "  ";
  cout << "] \n";
}

#include <iostream>
using namespace std;

int main()
{
double da[]={1.9,8.2,3.7,6.4,5.5,1.8,9.2,3.6,7.4,5.5};
int    ia[]={1,10,2,9,3,8,4,7,6,5};
int size=sizeof(da)/sizeof(double);

  cout << " Quicksort of double array is \n";
```

```
  quicksort(da,0,size-1);
  print(da,size);

  size=sizeof(ia)/sizeof(int);
  cout << " Quicksort of integer array is \n";
  quicksort(ia,0,size-1);
  print(ia,size);
return(0);
}
```

The syntax for the definition of the generic functions is as in the first example. The body of the swap function is straightforward.

It is not appropriate here to go into a full coverage discussion of Quicksort. There are a number of sources in the bibliography that look into the algorithm in some depth. This algorithm is based on Wirth's, and is a translation of the Modula 2 example in *Algorithms and Data Structures*. It behaves quite respectably in a wide range of cases. There are a number of improvements that can be made and will give a 20–30% improvement in running time.

Compare this with the generic integer and real quicksort example from the Fortran 90 course.

14.3 Sorting data files

This example looks at sorting an external data file. It uses the earlier quicksort and adds a few refinements. Firstly the example shows how to read and write to files. Secondly it has code added to time the different parts of the program. We first need a file with the data to be sorted.

14.3.1 Generating the randon numbers

The following program generates the random numbers.

```
#include <fstream>
#include <string>
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <cstdio>

using namespace std;

int main()
{
  time_t local_time;
  struct tm *today;

  int x;
  int i;
  int n=1000000;

  char file_name[] ="random.txt";

  time(&local_time);
```

```
  printf("%s",ctime(&local_time));

  ofstream output(file_name);
  for (i=1;i<=n;++i)
  {
    x=rand();
    output << x << endl;
  }

  time(&local_time);
  printf("%s",ctime(&local_time));

  return(0);
}
```

14.3.2 Sorting the data

This is the program that reads the data into an array, sorts the data and then writes the sorted data back out to a file.

```
#include <fstream>
#include <string>
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <cstdio>

using namespace std;

template <class Type>
void swap(Type array[],int i, int j)
{
  Type tmp=array[i];
  array[i]=array[j];
  array[j]=tmp;
}

template <class Type>
void quicksort( Type array[], int l, int r)
{
int i=l;
int j=r;
Type v=array[int((l+r)/2)];
for (;;)
{
  while (array[i] < v) i=i+1;
  while (v < array[j]) j=j-1;
  if (i<=j) { swap(array,i,j); i=i+1 ; j=j-1; }
  if (i>j) goto ended ;
}
ended: ;
if (l<j) quicksort(array,l,j);
```

```
if (i<r) quicksort(array,i,r);
}

template <class Type>
void print(Type array[],int size)
{
  char file_name[] ="sorted.txt";
  ofstream output(file_name);
  for (int i=0;i<size;++i)
  {
    output << array[i] << endl;
  }
}

int main()
{

  time_t local_time;
  struct tm *today;

  int size=1000000;

  cout << " Before allocate " << endl;
  int x[size];
  cout << " After  allocate " << endl;

  char file_name[] ="random.txt";

  time(&local_time);
  cout << " Initial " << endl;
  printf("%s",ctime(&local_time));

  ifstream input(file_name);
  for (int i=0;i<size;++i)
  {
    input >> x[i] ;
  }

  time(&local_time);
  cout << " read " << endl;
  printf("%s",ctime(&local_time));

  cout << " Quicksort of integer array is \n";
  quicksort(x,0,size-1);

  time(&local_time);
  cout << " Sort " << endl;
  printf("%s",ctime(&local_time));

  print(x,size);
```

```
   time(&local_time);
   cout << " print " << endl;
   printf("%s",ctime(&local_time));

   return(0);
}
```

You may have problems with arrays with over a million elements. If that is the case use the following code instead of the array allocation.

```
   int* x;
//  int x[size];
   x=new int[size];
   if (x==NULL)
   {
      cout << " Allocation failed " << endl;
      exit(1);
   }
```

Here is the output from one system I use at home.

```
$ g++ ch1440.cxx

Ian@rowan /cygdrive/d/document/cpp/examples
$ a.exe
 Before allocate
 After  allocate
 Initial
Tue Jul 12 15:25:42 2005
 read
Tue Jul 12 15:27:08 2005
 Quicksort of integer array is
 Sort
Tue Jul 12 15:27:17 2005
 print
Tue Jul 12 15:30:12 2005
```

As can be seen most of the time is spent in the i/o, not the sorting.

14.4 Other Issues

The name of a template obeys the usual scope and access control rules.

A member template shall not be virtual.

A destructor shall not be a template.

A local class shall not have a template member.

A template shall not have C linkage.

14.5 Summary

Most of the older texts don’t contain any coverage of templates. Stroustrup has a good coverage of how to use templates. Budd has a good coverage of how to use the STL. There is also pdf and postscript files on the web server with more information about the STL.

Templates should be seen as a complement to the object oriented programming paradigm. Look at the problem that you have to solve carefully. Chose the most appropriate way of solving it. We look in this in more depth in later chapters.

14.6 Key Concepts

The template mechanism provides a very powerful programming method. It allows us to write an algorithm once and have it work in a ploymorphic manner.

14.6.1 Basic Syntax Example

```
template <class Type>
Type minimum(Type array[], int size)
{
   Type minimum_value=array[0];
   for (int i=1 ; i < size; ++i)
    if (array[i] < minimum_value)
      minimum_value = array[i];
   return minimum_value;
}
```

14.7 Problems

1. Try both of these examples with the your compiler. Do they work? Try using the other numeric data types available in C++.

2. Try using complex with the quicksort template example. You will need to look at the way in which your compiler actually offers support for the complex data type. Remember not all compilers fully support the standard.

3.

15
Operator and Function Overloading

‘When I use a word,’ Humpty Dumpty said, in a rather scornful tone, ‘it means just what I choose it to mean - neither more nor less’
‘The question is,’ said Alice, ‘whether you can make words mean so many different things.’

Lewis Carroll, Through the Looking Glass and What Alice found there.

Aims

The aims are to introduce some of the rules and restrictions in C++ for successful use of operator and function overloading, in particular:–

overloading unary operators;
overloading binary operators;
commutative operators;
overloading [] for one dimensional arrays;
overloading () for two and higher dimensioned arrays;
overloading << and >> for class i/o;
smart pointers;

15 Operator and Function Overloading

In this chapter we will look at operator and function overloading. Overloading is the ability to attach more than one meaning to an operator or name.

An obvious example is in the previous chapter where using templates we saw how we could write one piece of code and have it work with several data types. In these examples the code would work providing the operations involved in the code segments were defined for the data types we were working with:–

- in the minimum example we obviously had to have a meaning to the comparison operator.
- in the sort example we again had to have a meaning attached to the comparison operator for each data type we wanted sorted.

For the built in types this is the case. For our data types we have to define our own meaning, i.e. write code that can be invoked whenever the comparison operator is used. This is called operator overloading.

We have also seen examples of function overloading using classes, where we have the same name and multiple instances of constructors for a class.

Note that not all oo languages support operator overloading. There is some disagreement within the oo world over the necessity of operator overloading within an oo language. The chapter in Meyer on genericity versus inheritance is well worth a read.

For the basic rules I've found Stroustrup to be the best source.

15.1 Operator Overloading.

The basic idea is that for each operator we are interested in overloading we have a corresponding function that will be called to carry out the desired action.

The following operators cannot be overloaded:–

.

.*

::

?:

all of the others can be. See the earlier chapter for a complete list and coverage of the operators available in C++. The preprocessing symbols # and ## cannot be overloaded either.

There are the following restrictions:–

- it is not possible to change the precedence, grouping, or number of operands of operators;
- an operator function shall either be a non-static member function or a non-member function and have at least one parameter whose type is a class, a reference to a class, an enumeration or a reference to an enumeration;
- we cannot introduce new operators, e.g. ^ or ** for exponentiation
- because of historical accident the following operators: = *assignment*, & *address of* and , *sequencing* have predefined meanings when applied to class objects.

15.1.1 Unary and Binary Operators

The first thing we need to know are the rules that govern what we can and cannot do. Let us consider unary and binary in turn.

A unary operator can be defined in two ways, firstly as a member function taking no arguments or as a global function taking one argument.

A binary operator may defined in one of two ways, firstly as a member function with one argument or secondly as a global function taking two arguments.

A global function is usually a friend.

It is worthwhile noting that there is quite a remarkable syntactic difference between the above two approaches, and this is shown below:–

Expression	As member function	As non-member function
@a	(a).operator@()	operator@(a)
a@b	(a).operator@(b)	operator@(a,b)
a=b	(a).operator=(b)	
a[b]	(a).operator[](b)	
a->	(a).operator->()	
a@	(a).operator@(0)	operator@(a,0)

This is taken from table 8 in the draft standard.

An operator function that is intended to be able to take a base type as the first argument cannot be a member function, and this some implications when we look at commutivity. It means that we must use non-member functions with binary operators. This is illustrated in the example below.

Let us look at some simple examples now.

15.1.2 Operator Overloading 1: `+ , - , =`

This example is based on the implementation of a complex arithmetic class in C++. Whilst there is a complex class in the standard it is very illuminating to look at the work involved in solving a numeric problem in C++.

```
#include <iostream.h>

class complex

{

friend complex operator+(const complex &a,const complex &b);
friend complex operator-(const complex &a,const complex &b);
friend double  real(const complex &a);
friend double  imag(const complex &a);

private:

  double re,im;

public:

  complex(double r,double i=0) { re=r;im=i;}
  complex &operator=(const complex &a);
  complex &operator=(double &a);
```

```
};

  inline double real(const complex &a)
  {
    return a.re;
  }

  inline double imag(const complex &a)
  {
   return a.im;
  }

  inline complex operator+(const complex &a , const complex
&b)
  {
   return complex(a.re + b.re , a.im + b.im);
  }

  inline complex operator-(const complex &a , const complex
&b)
  {
    return complex(a.re - b.re , a.im - b.im);
  }

  inline complex &complex::operator=(const complex &a)
  {
    re=a.re;
    im=a.im;
    return *this;
  }

  inline complex &complex::operator=(double &a)
  {
    re=a;
    im=0.0;
    return *this;
  }

  void print(const complex &a)
  {
    cout << real(a) << "  + i * " << imag(a) << endl;
  }

int main()
{
complex x(0,0);
complex y(1,1);
complex z(2,2);
```

```
  print(x);
  print(y);
  print(z);
  print(x+y);
  print(z-x);
  print(z+10);
  print(10+z);
  print(z-10);
  print(10-z);
  return(0);
}
```

A number of things to note:–

- the binary addition operator is declared as a friend; if we had declared the operator to be a member function then the program would not compile as addition as defined in this way is not commutative. The `print(10+z)` would fail.
- the binary subtraction operator is declared as a friend; if we had declared the operator to be a member function then the program would not compile as subtraction as defined in this way is not commutative. The `print(10-z)` would fail.
- the real function is declared as a friend; this is a syntactic issue.
- the imag function is declared as a friend; this is a syntactic issue.

Try the following program out. It replaces all friend functions with member functions to highlight the points raised above.

```
#include <iostream.h>

class complex
{
private:
  double re,im;

public:
  complex(double r,double i=0) { re=r;im=i;}
  complex &operator=(const complex &a);
  complex &operator=(double &a);
  double real(void) const;
  double imag(void) const ;
  complex operator+(const complex &b);
  complex operator-(const complex &b);
};

inline double complex::real(void) const
{
  return re;
}

inline double complex::imag(void) const
{
  return im;
```

```
}

inline complex complex::operator+(const complex &b)
{
  return complex(re + b.re , im + b.im);
}

inline complex complex::operator-(const complex &b)
{
  return complex(re - b.re , im - b.im);
}

inline complex &complex::operator=(const complex &a)
{
  re=a.re;
  im=a.im;
  return *this;
}

inline complex &complex::operator=(double &a)
{
  re=a;
  im=0.0;
  return *this;
}

void print(const complex &a)
{
  cout << a.real() << " +i *"<<a.imag() << endl;
}

int main()
{
  complex x(1,1);
  complex y(3,3);
  complex z(5,5);
  print(x);
  print(y);
  print(z);
  print(x+y);
  print(z-x);
  print(z+10);
  print(10+z); // comment out this line to compile
  print(z-10);
  print(10-z); // comment out this line to compile
  return(0);
}
```

15.1.3 Operator Overloading 2: []

This is a simple example to illustrate overloading the array subscripting mechanism of C++. As a bonus we add array checking.

```
#include <iostream.h>
#include <stdlib.h>

class array
{
public:
  double &operator[](int i);
private:
  double a[5];
};

double &array::operator[](int i)
{
  if (i<1 || i>5 )
    {
    cout << " Error - index out of range \n";
    exit(EXIT_FAILURE);
    }
  return a[i-1];
}

int main()
{
array x;
int i;
  for ( i=1;i<=5;++i)
   x[i]=i*i;

  for ( i=1;i<=5;++i)
       cout << x[i] << " ";

  x[11]=1.1;

  return(0);

}
```

15.1.4 Operator Overloading 3: ()

This simple example highlights the implementation of a two dimensional array, with the use of () rather than [][] to achieve the array indexing. Array bounds checking is also included.

```
#include <iostream.h>
#include <stdlib.h>

class array
{
public:
```

```
  double &operator()(int i,int j);
private:
  double a[5][5];
};

double &array::operator()(int i,int j)
{
  if (i<1 || j<1 || i>5 || j>5)
    {
    cout << " Error - index out of range \n";
    exit(EXIT_FAILURE);
    }
  return a[i-1][j-1];
}

int main()
{
array x;
int i,j;
  for ( i=1;i<=5;++i)
   for ( j=1;j<=5;++j)
   x(i,j)=i*j;

  for ( i=1;i<=5;++i)
   for ( j=1;j<=5;++j)
      cout << x(i,j) << " ";

  x(11,6)=1.1;

  return(0);

}
```

15.1.5 Operator Overloading 4: << and >>

It would be nice if we could use the C++ conventional operators for i/o. This can be done by overloading the << and >> operators. The following example illustrates this.

```
#include <iostream.h>
#include <stdlib.h>

class array
{
public:
  double &operator()(int i,int j);

private:
  double a[5][5];
};

double &array::operator()(int i,int j)
{
```

```
  if (i<1 || j<1 || i>5 || j>5)
   {
   cout << " Error - index out of range \n";
   exit(EXIT_FAILURE);
   }
  return a[i-1][j-1];
}

ostream&   operator<<(ostream& s , array x)
{
int i,j;
  for ( i=1;i<=5;++i)
   for ( j=1;j<=5;++j)
   cout << x(i,j) << " ";
  return s;
}


int main()
{
array x;
int i,j;
  for ( i=1;i<=5;++i)
   for ( j=1;j<=5;++j)
   x(i,j)=i*j;
  cout << x;
  return(0);
}
```

15.1.6 Guidelines for Operator Overloading

There is the potential with operator overloading to write code that is incomprehensible. Operator overloading should be restricted to cases where its use is intuitive. This is obviously the case with many mathematical problems.

The function call mechanism should be used when the problem area doesn't lend itself easily to intelligible operator overloading.

If you do hit problems with operator overloading then I recommend three sources of information:–

- the chapter in Stroustrup on operator overloading. This combined with the reference section at the back provide a lot of useful information. Quite a few gotchas in this area;
- chapter 13 of the standard. Remember most compilers aren't standard conformant and many books can't possibly be bang up to date with the latest state of affairs. When C++ is standardised then some of the dust should settle down.
- chapter 10 of Seed's book.

Also try compiling with another compiler. I find this very useful.

15.2 Function Overloading

You have already seen some examples of function overloading with the use of constructors. In this chapter we will look in a bit more depth at the whole idea.

Certain function declarations can't be overloaded. Firstly if two functions differ only in the return type then this is an error. Secondly member functions can't be overloaded if one of them is static. The following example from the standard illustrates this.

```
class x
{
  static void f();
  void f() ; // ill formed
  void f() const ; // ill formed
  void f() const volatile ; // ill formed
  void g();
  void g() const; //ok, no static g
  void g() const volatile ; //ok no static g
}
```

15.2.1 Overload Resolution

The standard states *Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of candidate functions that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the types of the parameters of the candidate function, how well (for non-static member functions) the object matches the implied object parameter and certain other properties of the candidate function.*

Quite a mouthful!

At the simplest level we have:–

- the signatures of the functions;
- the return types of the functions;

Matching is achieved in four ways, in the following order of precedence:–

- an exact match
- a match through promotion
- a match through standard conversion
- a match through user defined conversion;

and there are examples of each given below.

15.2.2 Exact Matching

Argument matching can distinguish between constant and non-constant pointer and reference arguments.

Consider the following date class with several constructors.

```
#include <iostream.h>
class date
{
public:
  date(int,int,int);
  date(int,int);
```

```
   date(int);
   date();
private:
   int day,month,year;
   int get_day();
   int get_month();
   int get_year();
};

int date::get_day()
{
return 0;
}
int date::get_month()
{
return 0;
}
int date::get_year()
{
return 0;
}

date::date(int d,int m,int y)
{
   cout << " in 3 parameter constructor " << endl;
   day=d;month=m;year=y;
}

date::date(int d,int m)
{
   cout << " in 2 parameter constructor " << endl;
   day=d;month=m;year=get_year();
}
date::date(int d)
{
   cout << " in 1 parameter constructor " << endl;
   day=d;month=get_month();year=get_year();
}
date::date()
{
   cout << " in 0 parameter constructor " << endl;
   day=get_day();month=get_month();year=get_year();
}
int main()
{
   date d1(25,12,1996);
   date d2(25,12);
   date d3(25);
   date d4;
   return 0;
```

```
}
```

An examination of this program shows that we have an exact match between the variable definitions in the main program and the constructors.

15.2.3 Matching through promotion

If there is no exact match promotions are applied in the following order:–

- an argument of type char, unsigned char or short is promoted to type int. An argument of type unsigned short is promoted to type int.if the machine size of an int is larger than that of a short, otherwise it is promoted to type unsigned int;
- an argument of type float is promoted to type double;
- an argument of an enumeration is promoted to type int;

After these promotions have taken place exact matching is attempted with the newly promoted types.

15.2.4 Matching through Standard Conversion

The following standard conversions are applied:–

- any numeric type will match a formal argument of any other numeric type including unsigned;
- enumeration types will match a formal argument of numeric type;
- zero will match both a formal argument of a pointer type and a formal argument of a numeric type;
- a pointer of any type will match a formal argument of void*;

Now consider the following example.

```
#include <iostream.h>

class date
{
public:
   date(int,int,int);
   date(int,int);
   date(int);
   date();
   void print_date();
private:
   int day,month,year;
   int get_day();
   int get_month();
   int get_year();
};

int date::get_day()
{
return 0;
}
int date::get_month()
{
return 0;
```

```
}
int date::get_year()
{
return 0;
}

date::date(int d,int m,int y)
{
  cout << " in 3 parameter constructor " << endl;
  day=d;month=m;year=y;
}

date::date(int d,int m)
{
  cout << " in 2 parameter constructor " << endl;
  day=d;month=m;year=get_year();
}
date::date(int d)
{
  cout << " in 1 parameter constructor " << endl;
  day=d;month=get_month();year=get_year();
}
date::date()
{
  cout << " in 0 parameter constructor " << endl;
  day=get_day();month=get_month();year=get_year();
}

void date::print_date()
{
  cout << day << "/" << month << "/" << year << endl;
}
int main()
{
  date d1('a',12.9,1996);
  date d2('b',12);
  date d3('c');
  date d4;
  d1.print_date();
  d2.print_date();
  d3.print_date();
  d4.print_date();
  return 0;
}
```

Think about this or a moment! Now run it to verify for yourself what is happening.

15.2.5 Matching through User Defined Conversion

It is of course possible for you to write your own functions to achieve type conversion.

15.2.6 Guidelines for Function Overloading

The key issue is differences in the function signatures. All that is necessary is for the compiler to be able to determine which function you want to call.

When designing a class it is good practice to define a number of constructors, to cover common usage. Look back at the date class to see what is meant here.

15.3 Key Concepts

Overloading is the ability to extend the meaning of a function or operator.

15.3.1 Function Overloading

15.3.1.1 Functions distinguished by signature

15.3.2 Operator Overloading

15.3.2.1 Unary operators

15.3.2.2 Binary operators

15.3.2.3 Commutivity

An operator that is intended to be able to take a base type as the first argument cannot be a member function.

15.4 Problems

0. Try the examples out in this chapter.

1. Using the example that overloaded [] rewrite the monthly rainfall example below to use a range checked array.

```
#include <iostream.h>
int main()
{
  float sum=0.0,average=0.0 ;
  float rainfall[12] ;
  int month ;
  cout << " Type in the rainfall values" ;
  for (month=0;month < 12 ; ++month)
    cin >> rainfall[month];
  for (month=0;month < 12 ; ++month)
    cout << month << " " << rainfall[month] << " \n " ;
  for (month=0;month < 12 ;++month)
    sum = sum + rainfall[month];
  average = sum/12;
  cout << " Average is " << average;
  return(0);
}
```

2. Modify problem 1 to overload << and >> so that you can read and write a whole array in one statement.

3. Overload the equality (==) and inequality (!=) to test if two arrays are the same or different. Use the previous problem as the basis for this problem. You will need three rainfall arrays. Use the following data to test the program.

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

0 1 2 3 4 5 6 7 8 9 10 11

Use the redirection symbol to make the program read from a data file.

16
Virtual Functions

For Madmen Only

Hermann Hesse, Steppenwolf.

Aims

The aims of this chapter are to introduce some of the ideas involved in the use of virtual functions. In particular:–

virtual functions;

pure virtual functions;

abstract data types;

single inheritance;

dynamic binding;

16 Virtual Functions and Abstract Data Types

Two concepts that are of major importance in oo programming are inheritance and dynamic binding. We have looked at classes and C++ supports inheritance through class derivation. We now need to look at dynamic binding.

The first example looks simply at this issue. It illustrates a number of C++ concepts including:–

- virtual functions, and a function should be declared virtual if:–
  - a class expects to be an object of derivation;
  - the implementation of the function is type dependent;
- pure virtual functions, and a function should be a pure function if we are interested in creating an abstract data type;
- abstract data types;
- single inheritance;
- dynamic binding.

Let us look at this now with an example.

16.1 Virtual Function Example 1

Consider the date class from chapter 10. The date format in this example is European. If we were in US then this would not work. Let's use this therefore as an example.

```
#include <iostream>
using namespace std;

class date
{
  protected:
  int day,month,year;
  public:
  date(int=0,int=0,int=0) ;
  virtual void print_date() = 0;
};

date::date(int d,int m,int y)
{
  day=d;month=m;year=y;
}

class world_date : public date
{
  public:
  world_date(int=0,int=0,int=0);
  void print_date();
};

world_date::world_date(int d,int m,int y)
{
```

```
   day=d;month=m;year=y;
}

void world_date::print_date()
{
   cout << day << "/" << month << "/" << year << endl;
}

class us_date : public date
{
   public:
   us_date(int=0,int=0,int=0);
   void print_date();
};

us_date::us_date(int m,int d,int y)
{
   day=d;month=m;year=y;
}

void us_date::print_date()
{
   cout << month << "/" << day << "/" << year << endl;
}

int main()
{
world_date d1;
world_date d2(1,1,1996);
us_date d3;
us_date d4(1,1,1996);
   d1.print_date();
   d2.print_date();
   d1=world_date(1,3,1956);
   d1.print_date();
   d3.print_date();
   d4.print_date();
   d3=us_date(1,3,1956);
   d3.print_date();
   return(0);
}
```

The first thing we have a the declaration of a base date class. This class has three private variables, day, month and year.

It has two public functions, the date constructor and a print_date function. As a means of documenting what we are doing it is useful to follow the convention of providing the arguments for the various functions of the class within the declaration. The whole of the interface can be examined in one place, without having to scan the rest of the code looking at the function bodies.

Note that the print_date function has been declared virtual. This means that any class that inherits it MUST provide a function that will enable printing that variation of the date.

Note also that the function print_date is made equal to zero. This means that it is a pure virtual function, and therefore that date is an **abstract class.** A class with one or more pure virtual functions is an abstract class, and no objects of that class can be created.

An abstract class means that we can provide an interface without providing any details of the implementation.

The next thing we have is a constructor for this base class.

We now have a derived class world_date. This is based on the base class date. Note that we have to define our private variables day, month and year. Private variables cannot be inherited from a base class. This is a fundamental premise of information hiding.

Note also that we have both a constructor and a print_date function. Constructors **cannot** be inherited from the base class. This defines the interface of the world_date class. We then have the implementation of both the constructor and the print_date function.

We then repeat the above for the us_date class. Again it is derived from the base date class, and has a constructor and print_date function.

Note that the constructor swaps over day and month to provide the us style of date. Note also that the print_date function provides a similar facility for printing in us date style.

Finally we have the main program.

We set up for date variables, d1 and d2 of type world_date and d3 and d4 of type us_date.

We then have the executable part of the code. The thing of interest here is the way in which we can write date.print_date and it doesn't matter of what type date is. We have dynamic binding and the correct print_date function is invoked at the time the program runs.

16.2 Virtual Function Example 2

This example looks at a more complex example. It also uses some of the alternate ways of doing things in C++ using constructors. It also uses the ubiquitous example from the graphics world of a shape.

```
#include <iostream.h>

class shape
{
public:
   shape(int x,int y) : x_(x) , y_(y)  {}
   virtual ~shape()                    {}
   virtual void draw()                 =0;
   virtual void move()                 =0;
protected:
   int x_,y_;
   void operator=(const shape& s) { x_=s.x_ ; y_=s.y_ ; }
   shape(const shape& s) : x_(s.x_) , y_(s.y_) {}
};

class square : public shape
{
public:
```

```
   square(int x,int y,int length)
    : shape(x,y) , length_(length) {}
   void draw() { cout << " inside draw square " << endl;}
   void move()
   {
      x_=10;
      y_=10;
      cout << " inside move square " << endl;
   }
protected:
   int length_;
};

class rectangle : public shape
{
public:
   rectangle(int x,int y,int length,int height)
    : shape(x,y) , length_(length) , height_(height) {}
   void draw() { cout << " inside draw rectangle " << endl;}
   void move()
   {
    x_=10;
    y_=10;
    cout << " inside move rectangle " << endl;
   }
protected:
   int length_;
   int height_;
};

void move_and_repaint(shape& s)
{
   s.move();
   s.draw();
}

int main()
{
square s(50,50,10);
rectangle r(100,100,10,20);
   move_and_repaint(s);
   move_and_repaint(r);
   return (0);
}
```

The first thing to note here is the new form of the constructor. The standard states (section 12.6.2) *in the definition of a constructor for a class, initialisers for direct and virtual class subobjects and nonstatic data members can be specified by a* ***ctor-initialiser****, which has the form*

ctor: : mem-initialiser-list

mem-initialiser-list: mem-initialiser

mem-initialiser , mem-initialiser

mem-initialiser: mem-initialiser-id (expression-list$_{opt}$)

mem-initialiser-id: ::$_{opt}$ nested-name-specifier$_{opt}$ class-name

identifier

What we have therefore is the use of the addition of

```
: x_(x) , y_(y) {}
```

to the constructor to enable us to initialise the x_ and y_ components of the class. Note the use of the underscore character appended to the variable names. This is a common working practice in C++. It helps to get around name conflicts.

Note also in this case the lack of a ; after the constructor and destructor.

Note also that whilst we can't make a constructor virtual we can make a destructor virtual.

We overload the assignment operator in the conventional way.

Note that we also have a copy constructor. Remember that a copy constructor for a class x is of the form x(x&).

Note here also the use of protected, rather than private. A member of a class can be private, protected or public and

- if it is private it can only be used by member functions and friends of the class in which it is declared;
- if it is protected it can only be used by member functions and friends of the class in which it is declared plus by member functions and friends of classes derived fom this class;
- if it is public its name can be used by any function;

Thus protected is of considerable use when we are looking at inheritance.

So what we have is an abstract data type shape.

We then have two derived classes, squares and rectangles.

Note that in these classes we use a form of the constructor that calls the constructor for the base class.

Both draw and move are implemented in a simple way just to illustrate what is actually happening. In reality with a genuine graphics application there would be a lot more code!

We essentially just move by 10 units in the x and y directions with move.

We then have the function move_and_repaint(). This takes an argument of type shape. It then applies move and draw to the object.

The clever thing now is that we can add at a later date further classes derived from shape and still use the move_and_repaint function *without changing the source and recompiling*. Dynamic binding takes care at run time of the problem of what move and draw functions to apply to the shape object. Compare this with the traditional programming approach using languages like Pascal, C, Modula 2 and Fortran 90, where we can build software where new code can call old code. What we can't do is to get old code to call new code, without (typically) modifying the source and recompiling. This is time consuming and error prone.

16.3 Summary

Virtual functions and pure virtual functions provide us with the last of the tools that we need in C++ to write programs in an object oriented way. Inheritance is provided via classes and dynamic binding is provided by virtual functions.

16.4 Key Concepts

16.4.1 Virtual function

A function should be declared virtual if

- a class is expected to be an object of derivation
- the implementation is type dependent

16.4.2 Pure Virtual Function

A function should be declared a pure virtual function if we are interested in creating an abstract data type.

16.4.3 Private

Can only be used by member functions and friends

16.4.4 Protected

Can only be used by member functions, friends and friends of classes dervived from it.

16.4.5 Public

Can be used by anyone.

16.5 Problems

1. The following is a simple program that uses some of the date functions provided in <time.h>

```
#include <iostream.h>
#include <time.h>
#include <stdio.h>
int main()
{
time_t local_time;
struct tm *today;
int day=0,month=0,year=0;
  time(&local_time);
  printf("%s",ctime(&local_time));
  today = localtime(&local_time);
  day   = today->tm_mday;
  month = today->tm_mon;
  year  = (today->tm_year)+1900;
  cout << day << " " << month << " " << year << endl;
  return(0);
}
```

The variable local_time is of type time_t. time_t is a long integer and is used to represents time values in time.

The variable today is a pointer to the tm structure. The tm structure is used by asctime, gmtime, localtime, mktime, and strftime to store and retrieve time information.The fields of the structure type tm store the following values, each of which is an int:

- tm_sec - Seconds after minute (0 – 59)
- tm_min - Minutes after hour (0 – 59)
- tm_hour - Hours after midnight (0 – 23)
- tm_mday - Day of month (1 – 31)
- tm_mon - Month (0 – 11; January = 0)
- tm_year - Year (current year minus 1900)
- tm_wday - Day of week (0 – 6; Sunday = 0)
- tm_yday - Day of year (0 – 365; January 1 = 0)
- tm_isdst - Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status of daylight saving time is unknown. The C run-time library assumes the United States's rules for implementing the calculation of Daylight Saving Time (DST).

The time function gets the system time. Its prototype is

- time_t time(time_t *timer)

It returns a value of type time_t and takes one argument which is a pointer to a variable of type time_t.

The localtime function converts a time value and corrects for the local time zone. Its prototype is:

- struct tm *localtime(const time_t *timer);

It takes one argument which is a pointer to a constant variable of type time_t. It returns a pointer to a structure tm. If the value in timer represents a date before midnight, January 1, 1970, localtime returns NULL. I have no idea why!

Components of a structure are accessed by the -> operator. Have a look at the earlier chapter on expressions for more information.

Run the program and have a look at the output. We will be using this example to modify our date class to make it more useful.

2. The date example in this chapter is quite simple. The following is a more realistic example. It adds a number of features that make the class more useful. It is still not complete.

```
#include <iostream>
#include <ctime>
using namespace std;

class date
{
  public:
  date(int = 0,int = 0,int = 0);
  virtual void print_date() = 0;
  protected:
  int day,month,year;
};

  static const int ndays[12]={31,28,31,30,31,30,
                              31,31,30,31,30,31};
```

```
  static const char *monthname[12]={"January",
                                    "February",
                                    "March",
                                    "April",
                                    "May",
                                    "June",
                                    "July",
                                    "August",
                                    "September",
                                    "October",
                                    "November",
                                    "December"};

  time_t local_time;
  struct tm *tday;

  void default_date(int &d,int &m,int &y)
  {
  if ( (d==0) || (m==0) || (y==0) )
  {
    time(&local_time);
    tday = localtime(&local_time);
    d    = tday->tm_mday;
    m = tday->tm_mon+1;
    y   = tday->tm_year+1900;
  }
    return;
  }


date::date(int d,int m,int y)
{
  default_date(d,m,y);
  day=d;month=m;year=y;
}

class world_date : public date
{
public:
  world_date(int =0,int =0,int =0);
  void print_date();
};

world_date::world_date(int d,int m,int y)
{
  default_date(d,m,y);
  day=d;month=m;year=y;
}

void world_date::print_date()
```

```
{
  cout << day << "/" << monthname[month-1] << "/" << year <<
endl;
}

class us_date : public date
{
public:
  us_date(int =0,int =0,int =0);
  void print_date();
};

us_date::us_date(int m,int d,int y)
{
  default_date(d,m,y);
  day=d;month=m;year=y;
}

void us_date::print_date()
{
  cout << monthname[month-1] << "/" << day << "/" << year <<
endl;
}

int main()
{
world_date d1;
world_date d2(1,1,1996);
us_date d3;
us_date d4(1,1,1996);
  d1.print_date();
  d2.print_date();
  d1=world_date(1,3,1956);
  d1.print_date();
  d3.print_date();
  d4.print_date();
  d3=us_date(3,1,1956);
  d3.print_date();
  return(0);
}
```

The variable ndays is an array that holds the number of valid days in a month. We have a slight problem with leap years though.

The variable monthname holds actual strings for the months rather than integers. Note that the structure tm returns month values in the range 0-11, rather than the real world of 1-12.

local_time is a variable of type time_t and is used in the default_date function.

The variable tday is a pointer to a structure tm. It is used in the default_date function.

The default_date function is called by the constructors and if any of day, month or year are zero the date is set to today's date.

Access to the components of the structure are made through the -> operator.

The arguments to the default_date function are passed be reference. We need to pass back the modified values.

The value of the month is incremented by one as moths are in the range 0-11, rather than the real world of 1-12.

The value for the year is incremented by 1900, due to the way that the function works.

Within the two routines that print the date we decrement the index by one as the index to the onthname array goes from 0-11, not 1-12.

Run the program and have a look at the output.

Now modify the program to add cout statements showing entering and exiting the following:

the date constructor

the world_date constructor

the us_date constructor

the default_date function

Print out the values of the arguments to these functions as well.

We will look at this in more detail in the lecture practicals.

3. In the previous example the variables

- ndays
- monthname
- local_time
- tday

and the function

- default_date

are not part of the date class, but are available at a file level. This is not particularly desirable. The following example corrects this.

```
#include <iostream>
#include <ctime>
using namespace std;

class date
{
  public:
  date(int = 0,int = 0,int = 0);
  virtual void print_date() = 0;
  protected:
  int day,month,year;

  public:
  static const int ndays[];

  static const char *monthname[];

  private:
```

```
  time_t local_time;
  struct tm *tday;

  public:
  void default_date(int &d,int &m,int &y)
  {
  if ( (d==0) || (m==0) || (y==0) )
  {
    time(&local_time);
    tday = localtime(&local_time);
    d    = tday->tm_mday;
    m = tday->tm_mon+1;
    y  = tday->tm_year+1900;
  }
    return;
  }
};

const int date::ndays[]={31,28,31,30,31,30,31,31,30,31,30,31};

const char *date::monthname[12]={"January",
               "February",
               "March",
               "April",
               "May",
               "June",
               "July",
               "August",
               "September",
               "October",
               "November",
               "December"};

date::date(int d,int m,int y)
{
  default_date(d,m,y);
  day=d;month=m;year=y;
}

class world_date : public date
{
public:
  world_date(int =0,int =0,int =0);
  void print_date();
};

world_date::world_date(int d,int m,int y)
{
  default_date(d,m,y);
  day=d;month=m;year=y;
```

```
}

void world_date::print_date()
{
  cout << day << "/" << monthname[month-1] << "/" << year <<
endl;
}

class us_date : public date
{
public:
  us_date(int =0,int =0,int =0);
  void print_date();
};

us_date::us_date(int m,int d,int y)
{
  default_date(d,m,y);
  day=d;month=m;year=y;
}

void us_date::print_date()
{
  cout << monthname[month-1] << "/" << day << "/" << year <<
endl;
}

int main()
{
world_date d1;
world_date d2(1,1,1996);
us_date d3;
us_date d4(1,1,1996);
  d1.print_date();
  d2.print_date();
  d1=world_date(1,3,1956);
  d1.print_date();
  d3.print_date();
  d4.print_date();
  d3=us_date(3,1,1956);
  d3.print_date();
  return(0);
}
```

Where can we replace public with protected in the above example?

4. This example is still not complete. We need to ensure that only valid dates are allowed. This will mean checking for leap years, and when we have done this testing for valid day and month combinations. The following is a boolean function that checks for leap years.

```
bool date::leapyear(int y)
{
  if ( y % 400 ==0) || ( y % 100 !=0 && y % 4 ==0))
    return true;
  else
    return false;
}
```

Once you know if it is a leap year what are you going to do about the number of days in February?

You can index into the ndays array with the month to pick up a valid value for the number of days in a month.

What kind of error message are you going to provide? Are you going to terminate the program?

17
Putting it all togethor

A man should keep his brain attic stacked with all the furniture he is likely to use, and the rest he can put away in the lumber room of his library, where he can get at it if he wants.

Sir Arthur Conan Doyle, Five Orange Pips.

Aims

The aim of this chapter is to look at a complete example that puts togethor some of what has been covered so far.

17 Putting it all togethor

This chapter looks at a complete example in C++ that should help pull togethor some of the material we've already covered.

17.1 Raw arrays in C++

Raw arrays in C++ are relatively primitive. This is because of backwards compatability with C and the decision made by Kernighan and Ritchie to make the following equivalent when X is an array

- X &X[0]

thus an array is little more than a pointer and a chunk of memory.

Some of the things we might look at including in an array class are

- subscript checking
- array assignment
- array copying
- array i/o
- dynamic array sizing
- arrays whose size are an integral part of the object
- default initial values

A numeric example has been chose again.

17.2 Integer example

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <assert.h>

using namespace std;

class array
{

friend ostream &operator<<( ostream &, const array &);

friend istream &operator>>( istream &, array &);

public:

  array(int =10);

  array(const array &x);

  ~array() {delete [] ptr;}

  const array &operator=(const array &x);

  const int &operator[](int index) const ;
```

```
  int &operator[](int index);

  int getsize() const;

  bool operator==(const array &) const;

  bool operator!=(const array &rhs) const
    { return ! ( *this == rhs) ; }

  int sum();


protected:

  int size;

  int *ptr;


};

array::array(int n)
{
  size = ( n> 0 ? n : 10);
  ptr  = new int[size];
  assert( ptr != NULL );
  for (int i=0;i<size;i++)
    ptr[i]=0;
}

array::array(const array &initial) : size(initial.size)
{
  ptr = new int[size];
  assert(ptr != NULL);
  for (int i=0;i<size;i++)
    ptr[i]=initial.ptr[i];
}

int array::getsize() const { return size; }

const array &array::operator=(const array &rhs)
{
  if ( &rhs != this)
  {
    if(size != rhs.size)
    {
      delete [] ptr;
      size=rhs.size;
      ptr = new int[size];
```

```
      assert(ptr != NULL);
    }
  for (int i=0;i<size;i++)
    ptr[i]=rhs[i];
  }
  return *this;
}

const int &array::operator[](int n) const
{
  assert( (n>=0) && (n<size) );
  return ptr[n];
}

int &array::operator[](int n)
{
  assert( (n>=0) && (n<size) );
  return ptr[n];
}

istream &operator>>(istream &input, array&a)
{
  for (int i=0;i<a.size;i++)
    input >> a.ptr[i];
  return input;
}

ostream & operator<<(ostream &output,const array &a)
{
  int i;
  for (i=0;i<a.size;i++)
  {
    output << setw(6) << a.ptr[i];
    if ( (i+1) % 8 ==0 )
      output << endl;
  }
  if ( (i%8) != 0 )
    output << endl;
  return output;
}


int array::sum()
{
  int result = ptr[0];
  for (int i = 1;i <size ;i++)
    result += ptr[i];
  return result;
}
```

```
int main()
{
   const int n=20;
   int nn=5;
   array x(n);
   cin >> x;
   array y=x;
   cout << x;
   cout << y;
   array z(nn);
   cout << z;
   cout  << x.sum() << endl;
   return (0);
}
```

17.2.1 Notes

We will go throuugh the code in sections and have a look at each section in turn.

```
friend ostream &operator<<( ostream &, const array &);
friend istream &operator>>( istream &, array &);
```

If these were member functions the syntax we would have to use would be:

- array << cout;

and

- array >> cin;

so we have declared them as friends, so we can use the normal right hand side way of writing them that we have used so far. So when we have

- cin >> x;

the following call is generated:

- operator >>(cin,x);

The process is identical for output.

```
array(int =10);
array(const array &x); // copy constructor
~array() {delete [] ptr;}
const array &operator=(const array &x);
```

This class does dynamic memory allocation so we must provide a constructor, copy constructor, destructor and an overloaded assignment operator to avoid memory problems.

The copy constructor must avoid simply copying the pointer to the object. This would lead to memory deallocation problems when the second object is dealloacted as the memory has already been freed. The second object would not point to a valid address.

We pass by reference to avoid a copy being made and qualify this this with const to indicate that the array is not to be modified. If we did use call by value the call would also result in infinite recursion, as a copy of the array would be made by calling the copy constructor, and so on ad infinitum.

Copy constructors are called under the following situations

- call by value
- when returning an array by value from a function
- when initialising an array to be a copy of another
- when defing and initialising an array using another array

We also provide a default size. In this case 10.

```
const int &operator[](int index) const ;
int &operator[](int index);
```

We need two versions of the overloaded array subscript operators. One for ordinary arrays, the second for const arrays.

```
int getsize() const;
bool operator==(const array &) const;
bool operator!=(const array &rhs) const
     { return ! ( *this == rhs) ; }
int sum();
```

The size of the array is a constant.

We overload == and != to allow us to easily compare arrays. The != uses the == to reduce the amount of code that needs to be written, and therefore minimise the number of places where errors might occur.

Sum is a function that calculates the sum of the array.

```
protected:
   int size;
   int *ptr;
```

These are the hidden variables for the class.

```
array::array(int n)
{
   size = ( n> 0 ? n : 10);
   ptr  = new int[size];
   assert( ptr != NULL );
   for (int i=0;i<size;i++)
     ptr[i]=0;
}
```

The first statement says if n is greater than 0 then size = n, else size=10.

The second attempts the allocation.

The assert tests to see if the allocation has succeeded. If not the program terminates.

The loop initialises the array to 0.

```
array::array(const array &initial) : size(initial.size)
{
   ptr = new int[size];
   assert(ptr != NULL);
   for (int i=0;i<size;i++)
     ptr[i]=initial.ptr[i];
}
```

This creates an array as a copy of another.

```
const array &array::operator=(const array &rhs)
{
```

```
  if ( &rhs != this)
  {
    if(size != rhs.size)
    {
      delete [] ptr;
      size=rhs.size;
      ptr = new int[size];
      assert(ptr != NULL);
    }
  for (int i=0;i<size;i++)
    ptr[i]=rhs[i];
  }
  return *this;
}
```

The first thing to do is check that we have not written a=a.

The second is check the sizes. If they are not the same deallocate the old array and reallocate and then initialise.

Finally return a pointer to the current object to allow a=b=c.

Pass by reference to avoid the copy and make const to disallow modifying the array.

```
const int &array::operator[](int n) const
{
  assert( (n>=0) && (n<size) );
  return ptr[n];
}
```

Check the index to ensure it is in range.

```
int &array::operator[](int n)
{
  assert( (n>=0) && (n<size) );
  return ptr[n];
}
```

As above.

```
istream &operator>>(istream &input, array&a)
{
  for (int i=0;i<a.size;i++)
    input >> a.ptr[i];
  return input;
}
```

The code that actually reads in the data.

```
ostream & operator<<(ostream &output,const array &a)
{
  int i;
  for (i=0;i<a.size;i++)
  {
    output << setw(6) << a.ptr[i];
    if ( (i+1) % 8 ==0 )
      output << endl;
  }
  if ( (i%8) != 0 )
    output << endl;
```

```
  return output;
}
```

The code that actually does the output. I've included some output formatting.

```
int array::sum()
{
  int result = ptr[0];
  for (int i = 1;i <size ;i++)
    result += ptr[i];
  return result;
}
```

The function that actually calculates the sum.

17.3 Float example

The following is a reworking of the above to work with floats. Use diff to compare the two source files to see what changes had to be made.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <assert.h>

using namespace std;

class array
{

friend ostream &operator<<( ostream &, const array &);

friend istream &operator>>( istream &, array &);

public:

  array(int =10);

  array(const array &x);

  ~array() {delete [] ptr;}

  const array &operator=(const array &x);

  const float &operator[](int index) const ;

  float &operator[](int index);

  int getsize() const;

  bool operator==(const array &) const;

  bool operator!=(const array &rhs) const
    { return ! ( *this == rhs) ; }
```

```
   float sum();


protected:

   int size;

   float *ptr;


};

array::array(int n)
{
   size = ( n> 0 ? n : 10);
   ptr  = new float[size];
   assert( ptr != NULL );
   for (int i=0;i<size;i++)
     ptr[i]=0;
}

array::array(const array &initial) : size(initial.size)
{
   ptr = new float[size];
   assert(ptr != NULL);
   for (int i=0;i<size;i++)
     ptr[i]=initial.ptr[i];
}

int array::getsize() const { return size; }

const array &array::operator=(const array &rhs)
{
   if ( &rhs != this)
   {
     if(size != rhs.size)
     {
       delete [] ptr;
       size=rhs.size;
       ptr = new float[size];
       assert(ptr != NULL);
     }
   for (int i=0;i<size;i++)
     ptr[i]=rhs[i];
   }
   return *this;
}

const float &array::operator[](int n) const
```

```
{
  assert( (n>=0) && (n<size) );
  return ptr[n];
}

float &array::operator[](int n)
{
  assert( (n>=0) && (n<size) );
  return ptr[n];
}

istream &operator>>(istream &input, array&a)
{
  for (int i=0;i<a.size;i++)
    input >> a.ptr[i];
  return input;
}

ostream & operator<<(ostream &output,const array &a)
{
  int i;
  for (i=0;i<a.size;i++)
  {
    output << setw(6) << a.ptr[i];
    if ( (i+1) % 8 ==0 )
      output << endl;
  }
  if ( (i%8) != 0 )
    output << endl;
  return output;
}


float array::sum()
{
  float result = ptr[0];
  for (int i = 1;i <size ;i++)
    result += ptr[i];
  return result;
}



int main()
{
  const int n=6;
  int nn=2;
  array x(n);
  cin >> x;
  array y=x;
```

```
  cout << x;
  cout << y;
  array z(nn);
  cout << z;
  cout  << x.sum() << endl;
  x[0]=100.0;
  return (0);
}
```

This is therefore a problem that would benefit from being rewritten using templates.

17.4 Summary

This chapter has looked at using material that we have covered in examples that are teaching based, i.e. they have been simple to highlight the language features we are interested in learning about. There is obviously a fair bit of work involved in using these concepts in a more realistic example. The error messages generated when doing this and their resolution are part of the learning curve associated with C++, or any programming language in reality.

17.5 Where do I go next?

The following are the some of the sources I've had to use to find out why things don't work

Stroustrup B., *The C++ Programming Language*, Addison Wesley.

Bit tough.

Deitel H.M., Deitel P.J., *C++ How to Program*, Prentice Hall.

Has lots of good examples, hints and tips about C++ programming.

Seed G., *An Introduction to OO programming in C++ with Applications in Computer Graphics*, Springer Verlag.

Shows it age now but still a very good coverage of the language. A new version is in the pipeline.

Meyers S.D., *Effective C++*, Addison Wesley.

How to use the language and what to avoid. If you don't know what the following is about you need to read the book.

Here are the table of contents from this book.

Shifting from C to C++

Item 1: Prefer const and inline to #define.

Item 2: Prefer <iostream> to <stdio.h>.

Item 3: Prefer new and delete to malloc and free. TOC, P9

Item 4: Prefer C++-style comments.

Memory Management

Item 5: Use the same form in corresponding uses of new and delete.

Item 6: Use delete on pointer members in destructors. EC++ TOC, P13

Item 7: Be prepared for out-of-memory conditions. TOC, P14

Item 8: Adhere to convention when writing operator new and operator delete.

Item 9: Avoid hiding the "normal" form of new. TOC, P16

Item 10: Write operator delete if you write operator new.

Constructors, Destructors, and Assignment Operators P18

Item 11: Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.

Item 12: Prefer initialization to assignment in constructors.

Item 13: List members in an initialization list in the order in which they are declared.

Item 14: Make sure base classes have virtual destructors.

Item 15: Have operator= return a reference to *this. EC++ TOC, P23

Item 16: Assign to all data members in operator=. TOC, P24

Item 17: Check for assignment to self in operator=. EC++ TOC, P25

Classes and Functions: Design and Declaration

Item 18: Strive for class interfaces that are complete and minimal.

Item 19: Differentiate among member functions, non-member functions, and friend functions.

Item 20: Avoid data members in the public interface. EC++ TOC, P29

Item 21: Use const whenever possible.

Item 22: Prefer pass-by-reference to pass-by-value. EC++ TOC, P31

Item 23: Don’t try to return a reference when you must return an object.

Item 24: Choose carefully between function overloading and parameter defaulting.

Item 25: Avoid overloading on a pointer and a numerical type.

Item 26: Guard against potential ambiguity. P35

Item 27: Explicitly disallow use of implicitly generated member functions you don’t want.

Item 28: Partition the global namespace.

Classes and Functions: Implementation

Item 29: Avoid returning "handles" to internal data. EC++ TOC, P39

Item 30: Avoid member functions that return non-const pointers or references to members less accessible than themselves.

Item 31: Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.

Item 32: Postpone variable definitions as long as possible.

Item 33: Use inlining judiciously.

Item 34: Minimize compilation dependencies between files.

Inheritance and Object-Oriented Design

Item 35: Make sure public inheritance models "isa." EC++ TOC, P46

Item 36: Differentiate between inheritance of interface and inheritance of implementation.

Item 37: Never redefine an inherited nonvirtual function.

Item 38: Never redefine an inherited default parameter value.

Item 39: Avoid casts down the inheritance hierarchy. EC++ TOC, P50

Item 40: Model "has-a" or "is-implemented-in-terms-of" through layering.

Item 41: Differentiate between inheritance and templates.

Item 42: Use private inheritance judiciously. P53

Item 43: Use multiple inheritance judiciously. TOC, P54

Item 44: Say what you mean; understand what you’re saying.

Miscellany

Item 45: Know what functions C++ silently writes and calls.

Item 46: Prefer compile-time and link-time errors to runtime errors.

Item 47: Ensure that non-local static objects are initialized before they're used.

Item 48: Pay attention to compiler warnings. P60

Item 49: Familiarize yourself with the standard library. EC++ TOC, P61

Item 50: Improve your understanding of C++.

Meyers S.D., *More Effective C++,* Addison Wesley.

As the first book.

Here are the table of contents from this book.

Basics

Item 1: Distinguish between pointers and references TOC, P6

Item 2: Prefer C++-style casts

Item 3: Never treat arrays polymorphically P8

Item 4: Avoid gratuitous default constructors P9

Operators

Item 5: Be wary of user-defined conversion functions TOC, P11

Item 6: Distinguish between prefix and postfix forms of increment and decrement operators

Item 7: Never overload &&, ||, or ,

Item 8: Understand the different meanings of new and delete

Exceptions

Item 9: Use destructors to prevent resource leaks TOC, P16

Item 10: Prevent resource leaks in constructors P17

Item 11: Prevent exceptions from leaving destructors TOC, P18

Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function

Item 13: Catch exceptions by reference

Item 14: Use exception specifications judiciously TOC, P21

Item 15: Understand the costs of exception handling TOC, P22

Efficiency

Item 16: Remember the 80-20 rule

Item 17: Consider using lazy evaluation

Item 18: Amortize the cost of expected computations TOC, P26

Item 19: Understand the origin of temporary objects TOC, P27

Item 20: Facilitate the return value optimization TOC, P28

Item 21: Overload to avoid implicit type conversions TOC, P29

Item 22: Consider using op= instead of stand-alone op TOC, P30

Item 23: Consider alternative libraries

Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Techniques

Item 25: Virtualizing constructors and non-member functions MEC++ TOC, P34

Item 26: Limiting the number of objects of a class TOC, P35

Item 27: Requiring or prohibiting heap-based objects TOC, P36

Item 28: Smart pointers

Item 29: Reference counting

Item 30: Proxy classes

Item 31: Making functions virtual with respect to more than one object

Miscellany

Item 32: Program in the future tense

Item 33: Make non-leaf classes abstract

Item 34: Understand how to combine C++ and C in the same program

Item 35: Familiarize yourself with the language standard

17.6 Problems

1. Run both examples.

2. Go back and rewrite one of the earlier array examples to use the above.

3. Modify one of the above to provide array indexing from 1-n, rather than 0-(n-1). If things don't work out quite as you expect insert statements in each function to actually show what is happening as the program runs. You need to look at the main program and match up each statement with the output from the functions as they are called.

4. For the really ambitous use one of the above as the basis for a template version.

18
Files and I/O – Streams

Common sense is the best distributed commodity in the world, for every man is convinced that he is well supplied with it.

Descartes.

Aims

The aims of this chapter are to look at the features in C++ that support files and I/O. In particular:–

setting the width of the output field;

setting the precision for numeric output;

justification within a field;

reading from files;

writing to files;

picking up file names from command line arguments;

18 Files and i/o – Streams

In this chapter we will look at the the facilities in C++ for input and output. This is provided through a class library. The I/O stream library was an early part of the standard library. Some text books will have program examples using the older style. We will try in this chapter to follow the emerging standards. Some of changes that occured include

- I/O became internationalised
- The string classes for character i/o using arrays of char were replaced. Theses have been maintained for backwards compatability.
- Exception handling was added
- The i/o stream classes were made templates to support different character representations
- All symbols of the i/o stream libary are now part of the std namespace.

18.1 A Stream as an Object

A stream can be regarded as an object with certain properties.

18.2 Stream Classses

Two of the key stream classes are

- istream – based om the template class basic_istream

and

- ostream – based on the basic template class basic_ostream

Some of the key objects are:

- cin – standard input or the keyboard
- cout – standard output, or the screen
- cerr – standard error message channel
- clog – standard logging channel

The above obviously has been influenced by Unix.

18.3 Key header files

These are

- <iosfwd>
- <streambuf>
- <istream>
- <ostream>
- <iostream>
- <iomanip>

18.4 Buffers

The actual reading and writing out of characters is buffered in C++. You may need to flush the output buffer to actually see where you are within your program when debugging.

18.5 Numeric i/o: width and precision

This example looks at the defaults for numeric i/o and then sets both the width and precision to achieve more conrol over the output.

```
#include <iostream.h>

int main()
{
int       i=123456789;
float     f=1.23456789;
double    d=1.23456789;
  cout << i << "   " << f << "   " << d << endl;
  cout.width(10);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(15);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(10);
  cout.precision(10);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(15);
  cout.precision(15);
  cout << i << "   " << f << "   " << d << endl;
  return(0);
}
```

18.6 Numeric i/o: setting justification

You also have control over the justification within a field. This can be left, right or internal. Internal left adjusts the sign and right adjusts the number.

```
#include <iostream.h>

int main()
{
int       i=123456789;
float     f=1.23456789;
double    d=1.23456789;
  cout.setf(ios::left,ios::adjustfield);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(10);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(15);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(10);
  cout.precision(10);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(15);
  cout.precision(15);
  cout << i << "   " << f << "   " << d << endl;
  return(0);
}
```

18.7 Numeric i/o: scientific notation

We can also output in scientific notation.

```
#include <iostream.h>

int main()
{
int       i=123456789;
float     f=1.23456789;
double    d=1.23456789;
  cout.setf(ios::scientific,ios::floatfield);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(10);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(15);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(10);
  cout.precision(10);
  cout << i << "   " << f << "   " << d << endl;
  cout.width(15);
  cout.precision(15);
  cout << i << "   " << f << "   " << d << endl;
  return(0);
}
```

18.8 Numeric i/o: alternate number bases, octal and hexadecimal

We can choose between three number bases

- 10 – decimal
- 8 – octal
- 16 – hexadecimal

```
#include <iostream.h>

int main()
{
int       i=1024;
  cout << i  << endl;
  cout << oct << i << endl;
  cout << hex << i << endl;;
  return(0);
}
```

18.9 File i/o: picking up the names from the command line

There are a variety of ways of doing simple i/o to and from files. We will cover a couple of them.

```
#include <fstream.h>

void error(char* s1,char* s2="")
{
```

```
  cerr<< s1 << " " << s2 << endl;
  exit(1);
}

int main(int argc,char* argv[])
{
  if (argc != 3) error(" Wrong number of arguments");

  ifstream in(argv[1]);
  if (!in) error(" cannot open input file ",argv[1]);

  ofstream out(argv[2]);
  if (!out) error(" cannot open output file",argv[2]);

  char c;
  while ( in.get(c) ) out.put(c);

  return (0);
}
```

18.10 File i/o: hard coded file names in constructors

```
#include <fstream.h>

int main()
{

  ifstream in("i.dat");

  ofstream out("o.dat");

  char c;
  while ( in.get(c) ) out.put(c);

  return (0);
}
```

18.11 File i/o: strings passed as arguments to constructors

```
#include <fstream.h>
#include <string.hxx>

int main()
{
  String i="i.dat";
  ifstream in(i);

  String o="o.dat";
  ofstream out(o);

  char c;
```

```
  while ( in.get(c) ) out.put(c);

  return (0);
}
```

18.12 Using the operating system.

Another way is to use the operating system redirection symbols, and the following would work under DOS, Unix and Linux.

```
a.out < readfrom.txt > writeto.txt
```

would read standard input from file readfrom.txt and redirect standard output to writeto.txt.

18.13 I/O and the STL

The following is a list of the i/o related facilities provided by the STL.

iosfwd	forward declaration of i/o facilities
iostream	standard iostream objects and operations
ios	iostream bases
streambuf	stream buffers
istream	input stream template
ostream	output stream template
iomanip	manipulators
sstream	streams to and from strings
cstdlib	character type functions
fstream	streams to and from files
cstdio	printf style i/o
cwchar	printf style i/o of wide characters

18.14 Manipulators

There is also a class of manipulators called

- <iomanip>

and you have access to the following

dec	decimal
oct	octal
hex	hexadecimal
endl	end line, or new line on output
ends	inserts NULL in a string

flush	flush output
ws	skip leading white space
setbase(int i)	set base
setfill(int c)	set fill character to c
setw(int w)	set width
setprecison(int p)	set precision

18.15 Low level input

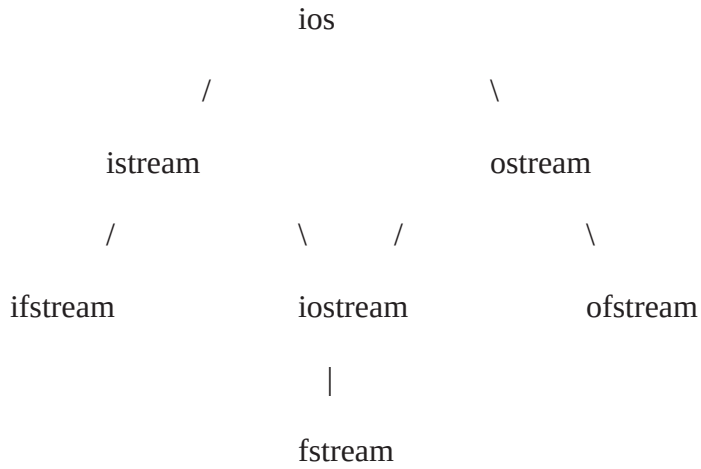
The following are provided to provide low level character access.

- get(s,n) – read up to n-1 characters into s and exclude newline or end of line, returns an istream
- get(s,n,t) – as above but now exclude t and end of file, returns istream.
- getline(s,n) – read up to n-1 characters and assign to s, including newline or end of file, returns istream.
- getline(s,n,t) – as above but now include t or end of file, returns istream.
- read(s,n) – read n characters, reads until end of file, returns istaream.
- readsome(s,n) – read up to n and stop at end of file, returns number read.

You then have control over how you interpret the character data read.

18.16 Some of the i/o stream hierarchy

The following is a part of the i/o stream hierarchy.



iostream is therefore an example of multiple inheritance as it inherits from both istream and ostream.

18.17 Summary

The examples here should be fairly self explanatory. They have been chosen to provide examples of the most commonly asked questions. If what you need isn't covered in these examples then try the books in the bibliography.

18.18 Problems

1.

18.19 Bibliography

I've found the following useful. The books are most recent first.

Josuttis N.M., *The C++ Standard Library*, Addison Wesley.

Deitel H.M., Deitel P.J., *C++ How to Program*, Prentice Hall.

Strosutrup B., *The C++ Programming Language*, Addison Wesley.

Seed G., *Introduction to C++ and OO Programming*, Springer Verlag.

19
Errors and Exception Handling

Don’t interrupt me while I’m interrupting.

Winston Churchill.

Aims

The aims of this chapter are look at errors and exceptions and the features in C++ that support exception handling.

19 Errors and Exception Handling

Let us look at some common errors and solutions:–

- invalid input: ask the user to retype the data values;
- file not found: ask the user to retype the file name;
- premature end of file: issue warning and terminate;
- numeric overflow: terminate the program with a warning;
- numeric underflow: terminate the program with a warning;
- array out of bounds: terminate the program with an error message;

and you will all be familiar with one or more of these common errors.

A common problem is that what is really quite simple code can be lost within the error handing code. Let us look at examples in a couple of programming languages of reading user input.

The following are coding examples that read all input from a user until they type CTRL Z – end of file. The first is in Pascal, the second in Fortran 90, the third in C++. The examples also highlight the quite different ways that languages have in their handling of end of line and end of file.

Fortran 90 has the concept of implicit gotos via the END= and ERR= options and also the concept of IOSTAT returning a value to let you know something has gone wrong.

Pascal and Modula 2 have the concept of EOLN and EOF.

19.1 Linked List – Pascal

```
PROGRAM LinkedList(INPUT,OUTPUT);
TYPE Link = @ Node
   Node = RECORD
   C : CHAR;
   Next : Link;
   END;
VAR Root     : Link;
    Current  : Link;
BEGIN
   NEW(Root);
   READ(Root@.C);
   Current:=Root;
   WHILE NOT EOF DO
   BEGIN
   NEW(Current@.Next);
   Current:=Current@.Next);
   READ(Current@.C)
   END;
   Current@.Next:=NIL;
   Current:=Root;
   WHILE Current  NIL DO
   BEGIN
    WRITE(Current@.C);
    Current:=Current@.Next
   END;
```

```
END.
```

19.2 Linked List – Fortran 90

```
PROGRAM C20_01
!
! Simple linked list
!
TYPE Link
   CHARACTER :: C
   TYPE (Link) , POINTER    :: Next
END TYPE Link
TYPE (Link) , POINTER :: Root , Current
INTEGER :: IO_Stat_Number=0
   ALLOCATE(Root)
   READ (UNIT=*,FMT=10,ADVANCE='NO',&
   IOSTAT=IO_Stat_Number) Root%C
   10 FORMAT(A1)
   print *,io_stat_number
   IF (IO_Stat_Number == -1) THEN
    NULLIFY(Root%Next)
   ELSE
    ALLOCATE(Root%Next)
   ENDIF
   print *, 'At start of input DO WHILE'
   Current=Root
   DO WHILE (ASSOCIATED(Current%Next))
    Current=Current%Next
    READ (UNIT=*,FMT=10,ADVANCE='NO',&
    IOSTAT=IO_Stat_Number) Current%C
    print *,io_stat_number
    IF (IO_Stat_Number == -1) THEN
     NULLIFY(Current%Next)
    ELSE
     ALLOCATE(Current%Next)
    ENDIF
   END DO
   print *, 'At end of input DO WHILE'
   print *, 'At start of output DO WHILE'
   Current=Root
   DO WHILE (ASSOCIATED(Current%Next))
    PRINT * , Current%C
    Current=Current%Next
   END DO
   print *, 'At end of output DO WHILE'
END PROGRAM C20_01
```

19.3 Linked List – C++, old C syntax

```
#include <iostream.h>
#include <stdio.h>
```

```
#include <stdlib.h>
struct link
{
  char c;
  struct link *next;
};
int main()
{
  char c;
  struct link *first = NULL;   /*  Start of list   */
  struct link *current;        /*  End of list     */
  struct link *ptr;            /*  Temporary       */

/* Loop reading characters entil the End Of File (EOF) */
/* Note that one cannot use eof() to check for EOF until*/
/* one has actually tried to read it - unlike Pascal.*/
  while (cin >> c ) /*  Loop for all characters  */
  {
   ptr = new (link);
   if (ptr == NULL)
   {
    cout << " Insufficient memory\n";
    exit(1);
   }

/*  Add new value and clear subsequent link  */
   ptr->c = c;
   ptr->next = NULL;

/*
Update pointers to add new value at end of list.  The "if"
statement could be omitted by making the first character en-
tered a "special" case - as in Example 10.1.4 - but general-
ity is a good idea!
*/
   if (first == NULL) /*  If this is the first character
*/
    first = current = ptr;
   else                          /*  Otherwise. */
    current->next = ptr;
   current = ptr;
  }

/*  Now print out the list  */

  ptr = first;
  while (ptr != NULL)
  {
   cout << ptr-c;
   ptr = ptr->next;
```

```
   }
   cout << endl;
   return 0;
}
```

19.4 Discussion

It is apparent that the essential code is hidden to greater or lessor extent in the requirements to handle special conditions and the features provided by each language. The Pascal code is probably the cleanest.

The facilities for handling errors and managing exceptional conditions has gradually improved in programming languages. The concept of raising an exception and then passing control to an error handler is one that is now seen in a number of programming languages including C++, Ada 95 and Java.

Let us now look at using the exception handling features of C++ to come to our help here.

19.5 Example 1 – Basic Syntax

```
#include <iostream.h>

int main()
{
   int i=0;
   int j=0;
   cout << " Type in a number " ;
   cin  >> i;

   try
   {
      if ( i<0 )
         throw j;
      else
         cout << " Number was " << i << endl;
   }

   catch (int j)
   {
      cout << " Exception raised " << endl;
   }

   return (0);
}
```

19.6 Example 2 – Exception raised in a function

```
#include <iostream.h>

int f(int i)
{
```

```
    throw i;
}

int main()
{
    int a=0;
    int b=1;
    int c;

    try
    {
        c=f(a);
        c=f(b);
    }

    catch (int i)
    {
        cout << " Exception raised and handled " << endl;
    }

    return(0);

}
```

19.7 Example 3 – Function with Exception Specification

The following example uses 3 functions to illustrate the variety of ways that we can use exception handling in C++.

```
#include <iostream.h>

int f1(int i) throw(int , double)
{
  if (i<0)
    throw i;
  if (i>0)
    throw (double)i;
  return i;
}

int f2(int i) throw()
{
  if (i<0)
    throw i;
  if (i>0)
    throw (double)i;
  return i;
}

int f3(int i)
```

```
{
   if (i<0)
     throw i;
   if (i>0)
     throw (double)i;
   return i;
}

int main()
{
   try
   {
    f1(-10);
    f1(10);
    f2(-10);
    f2(10);
    f3(-10);
    f3(10);
   }

   catch (int i)
    {
    cout << " integer exception raised and handled " <<
endl;
    }

    catch (double d)
    {
    cout << " double exception raised and handled " << endl;
   }

    return(0);

}
```

The general form of the function header is:–

```
return_type function_name (argument list) throw (type list)
```

You will need to comment out the various function calls to verify what is happening here. You should see the following:–

f1(-10) works
f1(10) works
f2(-10) fails
f2(10) fails
f3(-10) works
f3(10) works

There are extensions to this program set as a problem at the end of this chapter.

19.8 Example 5 – Exceptions and constructors and destructors.

It makes sense to look at throwing exceptions if there is the possibility of a constructor failing. If any memory allocation is involved then the success or failure of the memory allocation should be determined and the appropriate action taken.

19.9 Key Concepts

19.9.1 Basic Syntax

```
try
{
...
}
catch ()
```

19.10 Hierarchy of Standard Exceptions

All exceptions thrown from the language and library are derived from the base exception class. The hierarchy is given below.

	|1[BAD_ALLOC]		|4.1[DOMAIN_ERROR]
	|2[BAD_CAST]		|4.2[INVALID_ARGUMENT]
	|3[BAD_TYPEID]		|4.3[LENGTH_ERROR]
[EXCEPTION] -	|4[LOGIC_ERROR]	-	|4.4[OUT_OF_RANGE]
	|5[IOS_BASE::FAILURE]		|6.1[RANGE_ERROR]
	|6[RUNTIME_ERROR]	-	|6.2[OVERFLOW_ERROR]
	|7[BAD_EXCEPTION]		|6.3[UNDERFLOW_ERROR]

19.11 Problems

1. Modify the third example to throw a variable of type char from each functions. What happens?

2. Locate the except.h header file and have a look at what is supported in the version of C++ that you use.

Look for terminate(), set_terminate(), unexpected() and set_unexpected.

Look aso for the xmsg and xalloc classes.

19.12 Bibliography

20
The Standard Template Library

The good teacher is a guide who helps others dispense with his services.

R. S. Peters, Ethics and Education.

Aims

The aims of this chapter are to provide a brief coverage of the facilities provided by the Standard Template Library.

20 The Standard Template Library

This chapter looks at the standard template library. This library represents a major component of C++. The concept of generic programming has emerged as offering considerable advantages when solving certain kinds of problems.

The STL is the outcome of many years of work by a number of people including Alex Stepanov and Meng Lee at Hewlett Packard, and David Musser at Rensselaer Polytechnic Institute. Stroustrup provides more information about people involved in specific parts of the STL.

In reality some problems are well suited to a traditional procedural approach, others are well suited to an oo approach and others well suited to a generic approach. Pick the right tool for the job!

There aren't many examples in this chapter at this time. I will be adding them as the year progresses.

20.1 Library Organisation

We are not going to cover the whole of the library in great depth. Our primary aim is to give a quick overview and make you aware of what is offered by the library. Don't reinvent the wheel!

- Containers – vector, list, queue, stack, deque, map, set, bitset
- Iterators – iterator
- Algorithms – algorithm
- Strings – string
- Numerics – complex, valarray, numeric

20.1.1 Some basic terminology

Container – an object that you can put something in. In C++ a class that holds objects of a particular type.

Iterator – a mechanism that enables us to access and process all of the objects in a container.

Think of a telephone list. We can use the list container in the C++ STL to hold the names and telephone numbers. We can then use the iterators to traverse the list and identify the one we want.

20.2 Containers

A range of containers are provided in the STL. They are sufficient to allow us to solve quite a wide range of problems.

20.2.1 vector

Key Features include:–

- group of items of the same type
- selected or indexed by an integer key
    - hence constant access time – O(1)
- higher level than arrays
    - subscript checking possible
    - size an integral part of the object

    - can be resized up or down – very inefficient compared to raw arrays if the size varies widely
    - this carries a time penalty, can be O(n) in the worst case
- unsequenced

20.2.2 list

Key features include:–

- group of items of the same type
- accessed by traversal – O(n)
- efficient data structure when the size varies widely
- has both a head and a tail

Operations include:–

- erase(a) – remove element at a
- erase(first,last) – remove elements first to last
- clear() – remove all elements
- insert(a,b) – add b before a
- insert(a,n,b) – add n copies of b before a
- insert(a,first,last) – add elements first through last before p

20.2.3 queue

Key features include:–

- first in first out data structure
- insert at back
- remove from front

Operations include:–

- push_back()
- pop_back()
- push_front()
- pop_front()

20.2.4 stack

Key features include:–

- last in first out data structure
- insert at front
- remove from front

Operations include:–

- push_back()
- pop_back()
- push_front()
- pop_front()

20.2.5 deque

Key features include:–

- insert and remove from eithor end

20.2.6 map, multimap

Key features include:–

- key and value data structure
- selected by key, but key isn't restricted to be an integer

A simple example from every day life is a dictionary, where we index by words and the associated values are the meanings.

One variant.

20.2.7 set, bitset, multiset

Key features of set include:–

- group of items of the same type
- each item is unique
- unordered
- insertion, removal and testing can be done in O(log n) time

Two variations are provided, bit set and multiset.

20.3 Iterators

One of the most commonly wanted operations with a container is to process all of the elements. The STL solves this problem by the provision of iteraters.

- begin() – Point to first element
- end() – Point to one past last element
- rbegin() – Points to first element of reverse sequence
- rend – Points to one past last element of reverse sequence

20.4 Miscellaneous Operations

The STL can be looked at in a variety of ways. Any classification scheme is in some sense arbitrary.

- size()
- empty()
- max_size()
- capacity()
- reserve()
- resize()
- max_size()
- swap()
- get_allocator()
- ==
- !=

- <

20.5 Constructors

There are a small number of constructors.

- container()
- container(n)
- container(n,x)
- container(first,last)
- container(x)
- ~container()

20.6 Algorithms

The STL provides algorithms to solve many commonly ocurring problems. Don't reinvent the wheel!

20.6.1 Non modifying sequence operations

As the name says they don't modify the sequence.

- for_each()
- find()
- find_if()
- find_first_of()
- adjacent_find()
- count()
- count_if()
- mismatch()
- equal()
- search()
- find_end()
- search_n()

20.6.2 Modifying sequence operations

Sequence is modified. Some thought needs to be taken here regarding perfomance. The cost of some of these operations may be quite large.

- transform()
- copy()
- copy_backward()
- swap()
- iter_swap()
- swap_ranges()
- replace()
- replace_if()

- replace_copy()
- replace_copy_if()
- fill()
- fill_n()
- generate()
- generate_n()
- remove()
- remove_if()
- remove_copy()
- remove_copy_if()
- unique()
- unique_copy()
- reverse()
- reverse_copy()
- rotate()
- rotae_copy()
- random_shuffle()

20.6.3 Sorted sequences

Don't forget that sorting is also provided elsewhere.

- sort()
- stable_sort()
- partial_sort()
- partial_sort_copy()
- nth_element()
- lower_bound()
- upper_bound()
- equal_range()
- binary_search()
- merge()
- inplace_merge()
- partition()
- stable_partition()

20.6.4 Set algorithms

Sets provide very elegant ways of solving certain problems.

- includes()
- set_union()
- set_intersection()

- set_difference()
- set_symmetric_difference()

20.6.5 Heap operations

There are a small number of heap operations.

- make_heap()
- push_heap()
- pop_heap()
- sort_heap()

20.6.6 Minimum and maximum

Five functions provided.

- min()
- max()
- min_element()
- max_element()
- lexicograhical_compare()

20.6.7 Permutations

Two functions.

- next_permutation()
- prev_permutation()

20.7 Strings

The major headache here is the provision of two header files:–

- <string.h>

and

- <string>

The first provides access to the old C style string as an array of char. The second provides the C++ fully blown string as object style.

You should use the new style whenever possible. Many of the programming problems that occur with the old style strings just can't happen with the new style string. Why waste time getting bugs out of a program when you can avoid them altogethor? Refer to the earlier chapter for more information.

20.8 Complete list of C++ template classes

This was taken from table 12 of the standard.

allocator	codecvt	messages	queue
auto_ptr	codecvt_byname	messages_byname	raw_storage_iterator
back_insert_iterator	collate	moneypunct	reverse_bidirectional_iterator
basic_filebuf	collate_byname	moneypunct_byname	reverse_iterator
basic_ifstream	complex, ctype	money_get	set
basic_ios	ctype_byname	money_put	slice_array
basic_isstringstream	deque	multimap	stack
basic_ofstream	front_insert_iterator	multiset	time_get
basic_ostringstream	gslice_array	numeric_limits	time_get_byname
basic_streambuf	indirect_array	numpunct	unary_negate
basic_string	insert_iterator	num_get	valarray
basic_stringbuf	istreambuf_iterator	num_put	vector
basic_negate	istream_iterator	ostream_iterator	
binder1st	list	pointer_to_binary_function	
binder2nd	map	pointer_to_unary_function	
bitset	mask_array	priority_queue	

Complete Template Class List

20.9 Useful Sources

http://www.josuttis.com/libbook/

ftp://ftp.cs.orst.edu/pub/budd/stl/ReadMe.html

20.10 Example 1 - Map

The first example is a map. In the example below we use a first name and an address.

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main()
{

typedef map<string,string> NameAddress;

NameAddress NamesAddresses;

NamesAddresses["Ian"] = "21 Winterwell Road, Brixton, London,
SW2 5JB";
NamesAddresses["Joan"] = "21 Winterwell Road, Brixton, Lon-
don, SW2 5JB";
NamesAddresses["Martin"] = "21 Winterwell Road, Brixton, Lon-
don, SW2 5JB";
```

```
NamesAddresses["Jenny"] = "21 Winterwell Road, Brixton, Lon-
don, SW2 5JB";
NamesAddresses["Thea"] = "7 Winterwell Road, Brixton, Lon-
don";
NamesAddresses["Jack"] = "116 Branksome Road, Brixton, Lon-
don";

NameAddress::iterator i;

  for (i=NamesAddresses.begin();i!=NamesAddresses.end();++i)
  {
    cout << i->first ;
    cout << " lives at \t" ;
    cout << i->second ;
    cout << endl;
  }

  return(0);

}
```

20.11 Example 2 - Sets

The second uses a set. In mathematics sets are unordered and have no duplicates.

```
#include <iostream>
#include <fstream>
#include <set>
#include <string>

using namespace std;

int main()
{
  typedef std::set<string> words;

  words dictionary;

  string word;

  char file_name[]="words";

  ifstream input_file(file_name);

  const int buffer_size=80;
  char       buffer[buffer_size];

  words::iterator j;

  for (int i=1;i<93398;++i)
  {
    input_file.getline(buffer,buffer_size-1);
```

```
    word=buffer;
    dictionary.insert(word);
  }

  cout << "Type in a word " ;
  cin >> word ;

  j = dictionary.find(word);

  if ( j!= dictionary.end() )
    cout << " Word " << word << " in dictionary " << endl;
  else
  cout << " word not found " << endl;

  return(0);

}
```

In C++ the set is ordered.

20.12 Summary

The STL provides very good solutions to a wide range of frequently occuring problems. Use it.. Don't reinvent the wheel.

I will be putting examples up on the College web server, as the course progresses.

20.13 Problems

1. Run the map example. What do you notice about the output?

2. Run the set example. What happens if you type in a word with upper case characters? How long before the prompt is printed?

3. Modify the map example to read the name and address data from a file.

4. Chapter 11 had a quicksort program writtten as a template. This sorted arrays. The <stdlib.h> header provides details of qsort. The function prototype is

```
void qsort(     void *array,

                size_t n,

                size_t size,

                int (*compare)(const void *, const void *))
```

The qsort function sorts an array array[0] .. array[n-1] of objects of size size. The comparison function must return a negative value if the first argument is less than the second, zero if equal and a positive value if the first is greater than the second.

The STL has functionality too. A random data file is on the College web server at

http://www.kcl.ac.uk/kis/support/cit//fortran/random.dat

Use the template example from chapter 11 and obtain details of timing of this version.

Secondly use qsort from <stdlib.h> to sort the same data file and obtain timing information.

Thirdly use the sorting functionality provided by the STL.

Compare the three programs and the timing information.

21
Arithmetic and IEEE 754

The purpose of computation is insight not numbers.

I’m still looking for the source of this one.

Aims

The aims of this chapter are look at arithmetic and IEEE 754 and what support there is in C++ in this area.

21 Arithmetic and IEEE 754

Most modern processors support IEEE arithmetic to some extent. We will look briefly at what C++ offers in this area.

21.1 IEEE Arithmetic

The following is based on Chapter 25, ISO TR 15580, taken from Chivers and Sleightholme, Introducing Fortran 95.

The literature contains details of the IEEE 754 standard and the bibliography contains details of a number of printed and on-line sources.

21.2 History

When we use programming languages to do arithmetic two major concerns are the ability to develop reliable **and** portable numerical software. Arithmetic is done in hardware and there are number of things to consider:

- The range of hardware available both now and in the past.
- The evolution of hardware.

There has been a very considerable change in arithmetic units since the first computers. The following is a list of hardware and computing systems that the authors have some used or have heard of. It is not exhaustive or definitive. It reflects the authors' age and experience.

- CDC
- Cray
- IBM
- ICL
- Fujitsu
- DEC
- Compaq
- Gateway
- Sun
- Silicon Graphics
- Hewlett Packard
- Data General
- Honeywell
- Elliot
- Mostek
- National Semiconductors
- Intel
- Zilog
- Motorola
- Signetics
- Amdahl
- Texas Instruments

- Cyrix

Some of the operating systems include:

- NOS
- NOS/BE
- Kronos
- Unix
- VMS
- Dos
- Windows 3.x
- Windows 95
- Windows 98
- Windows NT
- Windows 2000
- MVS
- VM
- CP/M
- Macintosh
- OS/2

and again the list is not exhaustive or definitive. The intention is to provide with some idea of wide range of hardware, computer manufacturers and operating systems that have existed in the last 50 years.

To help with the anarchy that existed in this area Doctor Robert Stewart (acting on behalf of the IEEE) convened a meeting which led to the birth of IEEE 754.

The first draft was prepared by William Kahan, Jerome Coonen and Harold Stone, called the KCS draft and eventually adopted as IEEE 754. A fascinating account of the development of this standard can be found in An Interview with the Old Man of Floating Point, and the bibliography provides a web address of this interview. Kahan went on to get the ACM Turing Award in 1989 for his work in this area

This has become a de facto standard amongst arithmetic units in modern hardware. Note that it is not possible to precisely describe the answers a program will give and the authors of the standard knew this. This goal is virtually impossible to achieve when one considers floating point arithmetic. Reasons for this include:

- The conversions of numbers between decimal and binary formats.
- The use of elementary library functions.
- Results of calculations may be in hardware inaccessible to the programmer.
- Intermediate results in subexpressions or arguments to procedures.

The bibliography contains details of a paper that addresses this issue in much greater depth Ä Differences Among IEEE 754 Implementations.

Fortran is one of a small number of languages to provide access to IEEE arithmetic, and it achieves this via TR1880 which will become an integral part of Fortran 2000. The C stan-

dard (C9X) addresses this issue and Java offers limited IEEE arithmetic support. More information can be found in the references at the end of the chapter.

21.3 IEEE 754 Specifications

The standard specifies a number of things including:

- Single precision floating point format.
- Double precision floating point format.
- Two classes of extended floating point formats.
- Accuracy requirements on the following floating point operations:
    - Add.
    - Subtract.
    - Multiply.
    - Divide.
    - Square root.
    - Remainder.
    - Round numbers in floating point format to integer values.Convert between different floating point formats.
    - Convert between floating point and integer format.
    - Compare.
- Base conversion, i.e. when converting between decimal and binary floating point formats and vice versa.
- Exception handling for:
    - Divide by zero.
    - Overflow.
    - Underflow.
    - Invalid operation.
    - Inexact.
- Rounding directions.
- Rounding precisions.

and we will look briefly at each of these requirements.

21.3.1 Single precision floating point format

This is a 32-bit quantity made up of a sign bit, 8-bit biased exponent and 23-bit mantissa. The standard also specifies that certain of the bit patterns are set aside and do not represent normal numbers. This means that valid numbers are in the range 3.40282347E+38 to 1.17549435E-38 and the precision is between 6 and 9 digits depending on the numbers.

The special bit patterns provide the following:

- +0
- -0
- subnormal numbers in the range 1.17549421E-38 to 1.40129846E-45
- + infinity

- - infinity
- quiet NaN (Not a Number)
- signalling NaN

One of the first systems that the authors worked with that had special bits patterns set aside were the CDC 6000 range of computers that had negative indefinite and infinity. The ideas are not new therefore, as this was in the late 1970s.The support of positive and negative zero means that certain problems can be handled correctly including:

- The evaluation of the log function which has a discontinuity at zero.
- The equation $\sqrt{\frac{1}{z}} \quad \frac{1}{\sqrt{z}}$ can be solved when z = -1.

See also the Kahan paper Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit for more details.

Subnormals, which permit gradual underflow, fill the gap between 0 and the smallest normal number.

Simply stated underflow occurs when the result of an arithmetic operation is so small that it is subject to a larger than normal rounding error when stored. The existence of subnormals means that greater precision is available with these small numbers than normal numbers. The key features of gradual underflow are:

- When underflow does occur there should never be a loss of accuracy any greater than from ordinary roundoff.
- The operations of addition, subtraction comparison and remainder are always exact.
- Algorithms written to take advantage of subnormal numbers have smaller error bounds than other systems.
- If x and y are within a factor of 2 then x-y is error free, which is used in a number of algorithms that increase the precision at critical regions.

The combination of positive and negative zero and subnormal numbers means that when x and y are small and x-y has been flushed to zero the evaluation of:

- $\frac{1}{(x \quad y)}$

can be flagged and located.

Certain arithmetic operations cause problems including:

- 0 *
- 0 / 0
- $\sqrt{x}$ when x < 0

and the support for NaN handles these cases.

The support for positive and negative infinity allows the handling of:

- x / 0 when x is non-zero and of either sign

and the outcome of this means that we write our programs to take the appropriate action. In some cases this would mean recalculating using another approach.

For more information see the references in the bibliography.

21.3.2 Double precision floating point format

This is a 64 bit quantity made up of a sign bit, 11-bit biased exponent and 52-bit mantissa. As with single precision the standard specifies that certain of the bit patterns are set aside and do not represent normal numbers. This means we have valid numbers in the range 1.7976931348623157E308 to 2.2250738585072014E-308 and precision between 15 and 17 digits depending on the numbers.

As with single precision there are bit patterns set aside for the same special conditions.

Note that this does not mean that the hardware has to handle the manipulation of this 64-bit quantity in an identical fashion. The Sparc and Intel family handle the above as two 32-bit quantities but the order of the 2 component parts is reversed Ä so called big endian and little endian.

21.3.3 Two classes of extended floating point formats

These formats are not mandatory. A number of variants of double extended exist including:

- Sun Ä 4 32-bit words, one sign bit, 15-bit biased exponent and 112-bit mantissa, numbers in the range 3.362E-4932 to 1.189E4932, 33 to 36 digits of significance.
- Intel Ä 10 bytes Ä one sign bit, 15=bit biased exponent, 63-bit mantissa, numbers in the range 3.362E-4932 to 1.189E4932, 18-21 digits of significance.
- PowerPC Ä as Sun.

21.3.3.1 Accuracy requirements

Remainder and compare must be exact. The rest should return the exact result if possible. If not, there are well defined rounding rules to apply.

21.3.3.2 Base conversion - i.e. when converting between decimal and binary floating point formats and vice versa

These results should be exact if possible; if not the results must differ by tolerances that depend on rounding mode.

21.3.3.3 Exception handling

It must be possible to signal to the user the occurrence of the following conditions or exceptions:

- Divide by zero.
- Overflow.
- Underflow.
- Invalid operation.
- Inexact.

The ability to detect the above is a big step forward in our ability to write robust and portable code. These operations do occur in calculations and it is essential to have user programmer control over what action to take.

21.3.3.4 Rounding directions

Four rounding directions are available:

- nearest Ä the default
- down
- up

- chop.

Access to directed rounding can be used to implement interval arithmetic for example.

21.3.3.5 Rounding precisions

The only mandatory part here is that machines that perform computations in extended mode let the programmer control the precision via a control word. This means that if software is being developed on machines that support extended modes they can be switched to a mode that would enable the software to run on a system that didn't support extended mode. This area looks like a can of worms. Look at the Kahan paper for more information – *Lecture Notes on the Status of IEEE 754.*

21.4 Numerics

As Stroustrup says neither C or C++ were designed with numeric computation in mind. However for a variety of reasons both langauges are used in this area. The new standard provides facilities that make the language better at solving numeric problems. Some of the key header files that you need to know about are:–

- <math.h>
- <cmath> – contains function prototypes for the math library functions, replaces <math.h>
- <limits.h>
- <climits> – contains the integral size limits of the system, replaces <limits.h>
- <float.h>
- <cfloat> – contains the floating point size limits of the system, replaces <float.h>
- <stdlib.h>
- <cstdlib> – contains function prototypes for conversion of text to numbers and vice versa, memory allocation, random numbers amongst others, replaces <stdlib.h>
- <numeric> – contains accumulate, inner_product, partial_sum, adjacent_difference
- <numeric_limits>

We will look briefly at these header files and what functionality is provided.

21.5 numeric_limits

Arithmetic is hardware dependent as you should be aware now from the earlier material in this chapter, and from the examples and problems you actually written and tried out yourself. To accomodate this the C++ standard provides the numeric_limit header file. It is more type safe than the older ways of doing things and also allows templates to be used which help make the code more portable. The following provides details of what has been provided.

```
static const bool has_denorm = false;
static const bool has_denorm_loss = false;
static const bool has_infinity = false;
static const bool has_quiet_NaN = false;
static const bool has_signaling_NaN = false;
```

```
static const bool is_bounded = false;
static const bool is_exact = false;
static const bool is_iec559 = false;
static const bool is_integer = false;
static const bool is_modulo = false;
static const bool is_signed = false;
static const bool is_specialized = false;
static const bool tinyness_before = false;
static const bool traps = false;
static const float_round_style round_style = round_toward_zero;
static const int digits = 0;
static const int digits10 = 0;
static const int max_exponent = 0;
static const int max_exponent10 = 0;
static const int min_exponent = 0;
static const int min_exponent10 = 0;
static const int radix = 0;
static T denorm_min() throw();
static T epsilon() throw();
static T infinity() throw();
static T max() throw();
static T min() throw();
static T quiet_NaN() throw();
static T round_error() throw();
static T signaling_NaN() throw();
};
```

An example at the end of this chapter illustrates the use of some of the above. The program will be run on several processors including

- Cyrix 133
- Pentium P166
- Pentium II P350
- Sun UltraSparc

and the output compared. These files will be put on the College web server in due course. If you have access to additional hardware then please try it out.

21.5.1 complex

Complex is now formally a part of C++.

21.5.2 valarray

The valarray template class was added to provide the high performance levels associated with Fortran in C++. Compiler writers are allowed to try all manor of optimisation techniques. They are assumed to be alias free, which is a major advance over old C style arrays.

The concept of a slice is important in matrix manipulation. A slice is every nth element of a vector. It is provided for valarrays. Masks are also supported.

- valarray
- slice_array
- gslice_array
- mask_array
- indirect_array

It is not possible to go into this area in any real depth. If you have a background in numerical methods and some familiarity with Lapack, Blas and the NAG and IMSL libraries then I would recommend getting hold of Stroustrup and seeing what C++ can offer.

21.5.3 numeric

Some very useful general algorithms are provided:–

- accumlate()
- inner_product()
- partial_sum()
- adjacent_difference()

amongst others.

21.6 Problems

1. The following example is taken from the samples provided within Microsoft Developer Studio 6. Sample output follows.

Try this program out on another system and compare the output. Did you have to make any changes?

```
//////////////////////////////////////////////////////////////////////
// Compile options needed: /GX
//
// <filename> :  Numeric.cpp
//
// Variables and Functions:
//
//     has_denorm
//     has_denorm_loss
//     has_infinity
//     has_quiet_NaN
//     has_signaling_NaN
//     is_bounded
//     is_exact
//     is_iec559
//     is_integer
//     is_modulo
//     is_signed
//     is_specialized
//     tinyness_before
//     traps
//     round_style
//     digits
//     digits10
//     max_exponent
//     max_exponent10
//     min_exponent
```

```
//     min_exponent10
//     radix;
//     denorm_min()
//     epsilon()
//     infinity()
//     max()
//     min()
//     quiet_ NaN()
//     round_error()
//     signaling_NaN()
////////////////////////////////////////////////////////////////////
/* Compile options needed: /GX*/#include <iostream>#include <limits>
using namespace std;void main() {
     cout << " 1 The minimum value for char is " <<
          (int)numeric_limits<char>::min() << endl;
     cout << " 2 The minimum value for int is " <<
          numeric_limits<int>::min() << endl;
     cout << " 3 The maximum value for char is " <<
          (int)numeric_limits<char>::max() << endl;
     cout << " 4 The maximum value for int is " <<
          numeric_limits<int>::max() << endl;
     cout << " 5 The number of bits to represent a char is " <<
          numeric_limits<char>::digits << endl;
     cout << " 6 The number of bits to represent an int is " <<
          numeric_limits<int>::digits << endl;
     cout <<" 7 The number of digits representable in base 10 for float
is"
           << numeric_limits<float>::digits10 << endl;
     cout << " 8 Is a char signed?                " <<
          numeric_limits<char>::is_signed << endl;
     cout << " 9 Is an unsigned integer signed? " <<
          numeric_limits<unsigned int>::is_signed << endl;
     cout << "10 Is a integer an integer? " <<
          numeric_limits<int>::is_integer << endl;
     cout << "11 Is a float an integer?    " <<
          numeric_limits<float>::is_integer << endl;
     cout << "12 Is a integer exact? " <<
          numeric_limits<int>::is_exact << endl;
     cout << "13 Is a float  exact?  " <<
          numeric_limits<float>::is_exact << endl;
     cout << "14 The radix for float is              "  <<
          numeric_limits<float>::radix << endl;
     cout << "15 The epsilon for float is           " <<
          numeric_limits<float>::epsilon() << endl;
     cout << "16 The round error for float is       " <<
          numeric_limits<float>::round_error() << endl;
     cout << "17 The minimum exponent for float is " <<
          numeric_limits<float>::min_exponent << endl;
     cout << "18 The minimum exponent in base 10   " <<
          numeric_limits<float>::min_exponent10 << endl;
     cout << "19 The maximum exponent is            " <<
          numeric_limits<float>::max_exponent << endl;
     cout << "20 The maximum exponent in base 10   " <<
          numeric_limits<float>::max_exponent10 << endl;
     cout << "21 Can float represent positive infinity?  " <<
          numeric_limits<float>::has_infinity << endl;
     cout << "22 Can double represent positive infinity? " <<
          numeric_limits<double>::has_infinity << endl;
     cout << "23 Can int represent positive infinity? " <<
```

```
          numeric_limits<int>::has_infinity << endl;
     cout << "24 Can float represent a NaN?            " <<
          numeric_limits<float>::has_quiet_NaN << endl;
     cout << "25 Can float represent a signaling NaN? " <<
          numeric_limits<float>::has_signaling_NaN << endl;
     cout << "26 Does float allow denormalized values?   " <<
          numeric_limits<float>::has_denorm << endl;
     cout << "27 Does float detect denormalization loss? " <<
          numeric_limits<float>::has_denorm_loss << endl;
     cout << "28 Representation of positive infinity for float " <<
          numeric_limits<float>::infinity() << endl;
     cout << "29 Representation of quiet NaN for float          " <<
          numeric_limits<float>::quiet_NaN() << endl;
     cout << "30 Minimum denormalized number for float          " <<
          numeric_limits<float>::denorm_min() << endl;
     cout << "31 Minimum positive denormalized value for float " <<
          numeric_limits<float>::denorm_min() << endl;
     cout << "32 Does float adhere to IEC 559 standard?  " <<
          numeric_limits<float>::is_iec559 << endl;
     cout << "33 Is float bounded? " <<
          numeric_limits<float>::is_bounded << endl;
     cout << "34 Is float modulo?  " <<
          numeric_limits<float>::is_modulo << endl;
     cout << "35 Is int modulo?    " <<
          numeric_limits<float>::is_modulo << endl;
     cout << "36 Is trapping implemented for float?    " <<
          numeric_limits<float>::traps << endl;
     cout << "37 Is tinyness detected before rounding? " <<
          numeric_limits<float>::tinyness_before << endl;
     cout << "38 What is the rounding style for float? " <<
          (int)numeric_limits<float>::round_style << endl;
     cout << "39 What is the rounding style for int? " <<
          (int)numeric_limits<int>::round_style << endl;
     cout << "40 How does a float represent a signaling NaN? " <<
          numeric_limits<float>::signaling_NaN() << endl;
     cout << "41 Is int specialized? " <<
          numeric_limits<float>::is_specialized << endl;}
```

Program output is shown below:–

```
 1 The minimum value for char is -128
 2 The minimum value for int is  -2147483648
 3 The maximum value for char is 127
 4 The maximum value for int is  2147483647
 5 The number of bits to represent a char is 7
 6 The number of bits to represent an int is 31
 7 The number of digits representable in base 10 for float
is 6
 8 Is a char signed?                1
 9 Is an unsigned integer signed? 0
10 Is an integer an integer? 1
11 Is a float an integer?   0
12 Is an integer exact? 1
13 Is a float exact?  0
14 The radix for float is              2
15 The epsilon for float is            1.19209e-007
16 The round error for float is       0.5
```

```
17 The minimum exponent for float is -125
18 The minimum exponent in base 10   -37
19 The maximum exponent is             128
20 The maximum exponent in base 10   38
21 Can float represent positive infinity?  1
22 Can double represent positive infinity? 1
23 Can int represent positive infinity? 0
24 Can float represent a NaN?             1
25 Can float represent a signaling NaN? 1
26 Does float allow denormalized values?   1
27 Does float detect denormalization loss? 1
28 Representation of positive infinity for float 1.#INF
29 Representation of quiet NaN for float          -1.#IND
30 Minimum denormalized number for float
1.4013e-045
31 Minimum positive denormalized value for float 1.4013e-045
32 Does float adhere to IEC 559 standard?  1
33 Is float bounded? 1
34 Is float modulo?  0
35 Is int modulo?    0
36 Is trapping implemented for float?    1
37 Is tinyness detected before rounding? 1
38 What is the rounding style for float? 1
39 What is the rounding style for int? 0
40 How does a float represent a signaling NaN? -1.#INF
41 Is int specialized? 1
```

21.7 Bibliography

Hauser J.R., *Handling Floating Point Exceptions in Numeric Programs,* ACM Transaction on Programming Languages and Systems, Vol. 18, No. 2, March 1996, Pages 139–174.

- The paper looks at a number of techniques for handling floating point exceptions in numeric code. One of the conclusions is for better structured support for floating point exception handling in new programming languages, or of course later standards of existing languages.

IEEE, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985,* Institute of Electrical and Electronic Engineers Inc.

- The formal definition of IEEE 754.

Knuth D., *Seminumerical Algorithms,* Addison Wesley.

- There is a coverage of floating point arithmetic, multiple precision arithmetic, radix conversion and rational arithmetic.

Sun, *Numerical Computation Guide,* SunPro.

- Very good coverage of the numeric formats for IEEE Standard 754 for Binary Floating-Point Arithmetic. All SunPro compiler products support the features of the IEEE 754 standard.

21.7.1 Web based sources

http://validgh.com/goldberg/addendum.html

- *Differences Among IEEE 754 Implementations*. The material in this paper will eventually be included in the Sun Numerical Computation Guide as an addendum to Appendix D, David Goldberg's *What Every Computer Scientist Should Know about Floating Point Arithmetic*.

http://docs.sun.com/

- Follow the links to the *Floating Point and Common Tools AnswerBook*. The *Numerical Computation Guide* can be browsed on-line or downloaded as a pdf file. The last time we checked it was about 260 pages. Good source of information if you have Sun equipment.

http://www.validgh.com/

- This web site contains technical and business information relating to the validgh professional consulting practice of David G. Hough. Contains links to the Goldberg paper and the above addendum by Doug Priest.

http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html

- Brief coverage of IEEE arithmetic with pointers to further sources. There is also a coverage of the storage layout and ranges of floating point numbers. Computer Science 341 is an introduction to the design of a computer's hardware, particularly the CPU and memory systems.

http://www.nag.co.uk/nagware/NP/TR.html

- NAG provide coverage of TR 15580 and TR 15581. The first is the support Fortran has for IEEE arithmetic.

http://www.cs.berkeley.edu/~wkahan/

- Willam Kahan home page.

http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

- *An Interview with the Old Man of Floating Point*. Reminiscences elicited from William Kahan by Charles Severance, which appears in an issue of IEEE Computer - March 1998 (not confirmed).

http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

- Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic. Well worth a read.

http://www.stewart.cs.sdsu.edu/cs575/labs/l3floatpt.html

- *CS 575 Supercomputing – Lab 3: Floating Point Arithmetic*. CS 575 is an interdisciplinary course to introduce students in the sciences and engineering to advanced computing techniques using the supercomputers at the San Diego Supercomputer Center (SDSC).

http://www.mathcom.com/nafaq/index.html

- *FAQ: Numerical Analysis and Associated Fields Resource Guide*. A summary of Internet resources for a number of fields related to numerical analysis.

http://www.math.psu.edu/dna/disasters/ariadne.html

- *The Explosion of the Ariane 5*: A 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a

16-bit signed integer. The number was larger than 32,768, the largest integer storeable in a 16-bit signed integer, and thus the conversion failed.

21.7.2 Hardware sources

Osbourne A., Kane G., *4-bit and 8-bit Microprocessor Handbook,* Osbourne/McGraw Hill.

- Good source of information on 4 and 8 bit microprocessors.

Osbourne A., Kane G., *16 Bit Microprocessor Handbook,* Osbourne/McGraw Hill.

- Ditto 16 bit microprocessors.

Intel, *386 DX Microprocessor Hardware Reference Manual*, Intel.

- The first Intel offering with 32 bit addressing.

Intel, *80386 System Software Writer's Guide*, Intel.

- Developer's Guide to the above.

http://www.intel.com/

- Intel's home page.

http://developer.intel.com/design/pentiumiii/

- Details of the Pentium III processor.

http://www.cyrix.com/

- Cyrix home page.

Bhandarkar D.P., *Alpha Implementations and Architecture: Complete Reference and Guide,* Digital Press

- Looks at some of the trade offs and design philosophy behind the alpha chip. The author worked with VAX, MicroVAX and VAX vectors as well as the Prism. Also looks at the GEM compiler technology that DEC/Compaq use.

http://www.digital.com/alphaserver/workstations/

- Home page for the Compaq/DEC Alpha systems.

http://www.sgi.com/

- Silicon Graphics home page.

http://www.sun.com/

- Sun home page.

http://www.ibm.com/

- IBM home page.

21.7.3 Operating Systems

Deitel H.M., *An Introduction to Operating Systems*, Addison Wesley, 1990.

- The revised first edition includes case studies of Unix, VMS, CP/M, MVS and VM. The second edition adds OS/2 and the Macintosh operating systems. There is a coverage of hardware, software, firmware, process management, process concepts, asynchronous concurrent processes, concurrent programming, deadlock and indefinite postponement, storage management, real storage, virtual storage, processor management, distributed computing, disk performance optimization, file and database systems, performance, coprocessors, risc, data flow, analytic modelling, networks, security and concludes with case studies of the above operating systems. The book is well written and an easy read.

21.7.4 Java and IEEE 754

http://www.cs.berkeley.edu/~darcy/Borneo/

- *Borneo Language Homepage*: Borneo is a dialect of the Java language designed to have true support for the IEEE 754 floating point standard. The status of arithmetic in Java is fluid. At the time of writing this book Sun had withdrawn from the formal language standardization process. Sun have a publication at their web site that addresses changes to the Java language specification for JDK Release 1.2 floating point arithmetic. Their home Java page is:

http://www.java.sun.com/

21.7.5 C and IEEE 754

http://wwwold.dkuug.dk/JTC1/SC22/WG14/

- The official home of JTC1/SC22/WG14 - C. The C programming language standard ISO/IEC 9899 was adopted by ISO in 1990. ANSI then replaced their first standard X3.159 by the ANSI/ISO 9899 standard identical to ISO/IEC 9899:1990.

http://www.c9x.org/

- Another source of information regarding C9X. There is a draft of the standard available and Annex F, G and H contain details of the changes concerning arithmetic.

22
Mixed Language Programming

Overview

The aim of this chapter is to look at how to go about mixed language programming.

22 Mixed Language Programming

It is possible to call procedures written in one language from another. To do this successfully one must be aware of the following:

- variable naming conventions
- stack usage
- argument passing

and we will look at each of these in turn. This is an area that is platform specific so we will look at the pc platform and the following compilers:

- Microsoft Visual C++
- Microsoft Visual Basic
- Dec/Compaq Fortran 90/95

under Windows 95, 98 and NT

22.1 Dynamic Link Libraries – Dlls.

The key concept in mixed language programming work is a dll. We compile our routines and incorporate them into dll's. The other language then calls the routine within the dll. The routine within the dll does the work and control passses back to the calling routine.

22.2 Language Equivalents

The following summarises the differences between Fortran, C, C++, Visual Basic.

Language	Call with return value	Call with no return value
Fortran	FUNCTION	SUBROUTINE
C and Visual C++	function	(void) function
Visual Basic	Function	Sub

22.3 Data Types

The following summarises the equivalent data types of Fortran, C, C++, Visual Basic and assembler.

Fortran	C & C++	Visual Basic
INTEGER(1)	char	—-
INTEGER(2)	short	Integer
INTEGER(4)	int, long	Long
REAL(4)	float	Single
REAL(8)	double	Double
CHARACTER(1)	unsigned char	
CHARACTER*(*)	See Handling Character Strings	
COMPLEX(4)	struct complex4 {	

```
    float real, imag;

    };
```

COMPLEX(8) as above

All LOGICAL types Use integer types for C, MASM, and Visual Basic.

Fortran and Visual Basic do not support the C and C++ concept of unsigned integer.

22.4 Arrays

Fortran, Visual Basic, MASM, C and C++ vary in the way that array elements are indexed. Visual Basic stores arrays and character strings as descriptors: data structures that contain array size and location.

Fortran allows arrays to be passed as array elements, as array subsections, or as whole arrays referenced by array name.

To pass an array from Visual Basic to Fortran, pass the first element of the array. By default, Visual Basic passes variables by reference, so passing the first element of the array will give Fortran the starting location of the array, just as Fortran expects. Visual Basic indexes the first array element as 0 by default, while Fortran by default indexes it as 1.

Fortran and C arrays differ in two ways, firstly the value of the lower array bound is different. By default, Fortran indexes the first element of an array as 1. C and Visual C++ index it as 0. Fortran subscripts should therefore be one higher. (Fortran also provides the option of specifying another integer lower bound.)

Secondly with arrays of more than one dimension, Fortran varies the left-most index the fastest, while C varies the right-most index the fastest. These are sometimes called column-major order and row-major order, respectively.

The examples show how to pass arrays between:

- VB and Fortran
- VB and C++.

We will be adding mixing C++ and Fortran.

22.5 Character Strings

We are not going to cover this at this time.

22.6 User-Defined Types

Fortran supports user-defined types (data structures similar to C structures). User-defined types can be passed in modules and common blocks just as other data types, but the other language must know the type's structure.

22.7 Visual Basic calling Fortran

These examples have been written using a project within Visual Basic and a project within Visual Fortran.

22.7.1 Function example

The source code is shown below. Note the use of compiler directives to define the routine name and make it visible out side of the dll.

22.7.1.1 Fortran Source

```
real function cube(x)
real , intent(in) :: x
!
!dec$attributes dllexport :: cube
!dec$attributes alias : 'cube' :: cube
!
  cube = x*x*x
end function cube
```

22.7.1.2 Visual Basic Form

This is the whole Form1.frm file.

```
VERSION 5.00
Begin VB.Form Form1
    Caption           =    "Form1"
    ClientHeight      =    3192
    ClientLeft        =    60
    ClientTop         =    348
    ClientWidth       =    4680
    LinkTopic         =    "Form1"
    ScaleHeight       =    3192
    ScaleWidth        =    4680
    StartUpPosition =    3   'Windows Default
    Begin VB.TextBox Text1
        Height            =    975
        Left              =    2400
        TabIndex          =    1
        Text              =    "Text1"
        Top               =    240
        Width             =    1455
    End
    Begin VB.CommandButton Command1
        Caption           =    "Call Fortran 90 subroutine"
        Height            =    975
        Left              =    240
        TabIndex          =    0
        Top               =    240
        Width             =    1815
    End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Dim x As Single
Dim y As Single
```

```
  x = 200#
  y = cube(x)
Text1.Text = Str$(y)
End Sub
```

22.7.1.3 Visual Basic Module

This is the text file Module1.bas. The declare function statement must all be on one line.

```
Attribute VB_Name = "Module1"
Declare Function cube Lib "c:\document\f90\mixed01\de-
bug\mixed01.dll" (x As Single) As Single
```

22.7.2 Subroutine example

This example swaps the arguments.

22.7.2.1 Fortran Source

```
subroutine swap(x,y)
real , intent(inout) :: x
real , intent(inout) :: y
real :: t
!
!dec$attributes dllexport :: swap
!dec$attributes alias : 'swap' :: swap
!
  t=x
  x=y
  y=t
end subroutine swap
```

22.7.2.2 Visual Basic Form

```
VERSION 5.00
Begin VB.Form Form1
   Caption           =    "Form1"
   ClientHeight      =    3192
   ClientLeft        =    60
   ClientTop         =    348
   ClientWidth       =    4680
   LinkTopic         =    "Form1"
   ScaleHeight       =    3192
   ScaleWidth        =    4680
   StartUpPosition =    3   'Windows Default
   Begin VB.TextBox Text1
      Height            =    1215
      Left              =    2520
      TabIndex          =    1
      Text              =    "Text1"
      Top               =    360
      Width             =    1815
   End
   Begin VB.CommandButton Command1
      Caption           =    "simple scalar subroutine call"
      Height            =    1215
```

```
        Left               =    360
        TabIndex          =    0
        Top                =    360
        Width              =    1935
    End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Dim x As Single
Dim y As Single
x = 10#
y = 999#
Call swap(x, y)
Text1.Text = Str$(y)
End Sub
```

22.7.2.3 Visual Basic Module

The declare statement is all on one line.

```
Attribute VB_Name = "Module1"

Declare Sub swap Lib "c:\document\f90\mixed02\de-
bug\mixed02.dll" (x As Single, y As Single)
```

22.7.3 Subroutine example – passing 1 d array

22.7.3.1 Fortran Source.

The arguments are

- x - the real array
- i - the size of the array, integer, scalar
- r - the result, real and scalar

```
subroutine sum(x,i,r)
real    , intent(in)  :: x(i)
integer , intent(in)  :: i
real    , intent(out) :: r
!
!dec$attributes dllexport :: sum
!dec$attributes alias : ‘sum’ :: sum
!
integer :: j
  r=0.0
  do j=1,i
    r=r+x(j)
  end do
end subroutine sum
```

22.7.3.2 Visual Basic Form

```
VERSION 5.00
Begin VB.Form Form1
    Caption           =    "Form1"
    ClientHeight      =    3192
    ClientLeft        =    60
    ClientTop         =    348
    ClientWidth       =    4680
    LinkTopic         =    "Form1"
    ScaleHeight       =    3192
    ScaleWidth        =    4680
    StartUpPosition =    3   ‘Windows Default
    Begin VB.TextBox Text1
        Height            =    1215
        Left              =    2520
        TabIndex          =    1
        Text              =    "Text1"
        Top               =    360
        Width             =    1815
    End
    Begin VB.CommandButton Command1
        Caption           =    "passing 1 d array as argument"
        Height            =    1215
        Left              =    360
        TabIndex          =    0
        Top               =    360
        Width             =    1935
    End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Dim x(1 To 10) As Single
Dim r As Single
Dim n As Integer
Dim i As Integer
n = 10
For i = 1 To 10
  x(i) = i
Next i
Call sum(x(1), n, r)
Text1.Text = Str$(r)
End Sub
```

Note that we call the subroutine sum with the arguments

x(1) - the first element of the array

```
n - the size of the array, integer
r - the result, real and scalar
```

22.7.3.3 Visual Basic Module

This is the module1.bas file. The declare statement is all one line. Note that the first argument is declared as a scalar and not an array.

```
Attribute VB_Name = "Module1"
Declare Sub sum Lib "c:\document\f90\mixed03\de-
bug\mixed03.dll" (x As Single, n As Integer, r As Single)
```

22.7.4 File i/o

This example looks at file i/o. We call a Fortran function that opens a file, reads data from it, does some calculations and then writes some data out to a second file.

22.7.4.1 Fortran Source

```
subroutine io(x,i,r)
real    , intent(inout)  :: x(i)
integer , intent(in)  :: i
real    , intent(out) :: r
!
!dec$attributes dllexport :: io
!dec$attributes alias : 'io' :: io
!
integer :: j
  open (unit=1,file='c:\document\mixed\io\before.dat')
  open (unit=2,file='c:\document\mixed\io\after.dat')
  r=0.0
  do j=1,i
    read (1,10) x(j)
    10 format(f7.2)
    r=r+x(j)
    x(j)=x(j)*x(j)
    write(2,20) x(j)
    20 format(1x,f9.2)
  end do
  close(1)
  close (2)
end subroutine io
```

22.7.4.2 Visual Basic Form

```
VERSION 5.00
Begin VB.Form Form1
   Caption          =   "Form1"
   ClientHeight     =   2496
   ClientLeft       =   48
   ClientTop        =   336
   ClientWidth      =   3744
   LinkTopic        =   "Form1"
   ScaleHeight      =   2496
   ScaleWidth       =   3744
```

```
    StartUpPosition =   3  'Windows Default
    Begin VB.TextBox Text1
        Height            =   732
        Left              =   360
        TabIndex          =   1
        Text              =   "Text1"
        Top               =   1440
        Width             =   1332
    End
    Begin VB.CommandButton Command1
        Caption           =   "VB calling subroutine with i/o"
        Height            =   732
        Left              =   360
        TabIndex          =   0
        Top               =   360
        Width             =   1332
    End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Dim x(1 To 10) As Single
Dim r As Single
Dim n As Integer
Dim i As Integer
n = 10
For i = 1 To 10
x(i) = i
Next i
Call io(x(1), n, r)
Text1.Text = Str$(r)
End Sub
```

22.7.4.3 Visual Basic Module

Note that we make the array available to the Fortran subroutine by passing the first element and define it to be a scalar. The declare statement is all on one line.

```
Attribute VB_Name = "Module1"
Declare Sub io Lib "c:\document\mixed\io\debug\io.dll" (x As
Single, i As Integer, r As Single)
```

22.8 Visual Basic calling C++

We will repeat the above, replacing the routines written in Fortran by routines written in C++.

22.8.1 Function example

22.8.1.1 Visual basic – Module

```
Attribute VB_Name = "Module1"
Declare Function cube Lib
"e:\export\home\sttp1553\document\cpp\mixed\cube\debug\cube.dll"
Alias "_cube@4" (ByRef x As Single) As Single
```

Some of the things to note include:–

- The Declare statement has to be on one line. It has been broken here for printing purposes.
- You have to provide details of where the dll is. In this case I explicitly define where the dll is.
- The alias is essential. Note that through name mangling there is a leading underscore and that we have an @ symbol and a number at the end of the routine name. The number is four times the number of arguments.

Take great care to ensure that you don't mistype anything here!

22.8.1.2 Visual Basic – Form

```
VERSION 5.00
Begin VB.Form Form1
   Caption          =   "Form1"
   ClientHeight     =   2556
   ClientLeft       =   48
   ClientTop        =   276
   ClientWidth      =   3744
   LinkTopic        =   "Form1"
   ScaleHeight      =   2556
   ScaleWidth       =   3744
   StartUpPosition =   3  'Windows Default
   Begin VB.TextBox Text1
      Height           =   972
      Left             =   2040
      TabIndex         =   1
      Text             =   "Text1"
      Top              =   360
      Width            =   1212
   End
   Begin VB.CommandButton Command1
      Caption          =   "Command1"
      Height           =   972
      Left             =   360
      TabIndex         =   0
      Top              =   360
      Width            =   1332
   End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
```

```
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Dim x As Single
Dim y As Single
x = 10#
y = cube(x)
Text1.Text = y
End Sub
```

22.8.1.3 C++

```
extern "C"

{

   _declspec(dllexport)

   float

    _stdcall

   cube(float& x)
   {
      return x*x*x;
   }

}
```

Things to note include:–

- extern "C" – required to define the way names are decorated.
- _declspec(dllexport) – make the routines visible outside the dll
- _stdcall – parameter passing method

You must explicitly define all of these.

22.8.2 Void function example

22.8.2.1 Visual Basic – Module

```
Attribute VB_Name = "Module1"
Declare Sub swap Lib
"e:\export\home\sttp1553\document\cpp\mixed\swap\de-
bug\swap.dll"
Alias "_swap@8" (ByRef x As Single, ByRef y As Single)
```

22.8.2.2 Visual Basic – Form

```
VERSION 5.00
Begin VB.Form Form1
   Caption          =   "Form1"
   ClientHeight     =   2556
   ClientLeft       =   48
   ClientTop        =   276
```

```
    ClientWidth      =    3744
    LinkTopic        =    "Form1"
    ScaleHeight      =    2556
    ScaleWidth       =    3744
    StartUpPosition =    3  ‘Windows Default
    Begin VB.TextBox Text1
        Height            =    972
        Left              =    1800
        TabIndex          =    1
        Text              =    "Text1"
        Top               =    360
        Width             =    1452
    End
    Begin VB.CommandButton Command1
        Caption           =    "Command1"
        Height            =    972
        Left              =    360
        TabIndex          =    0
        Top               =    360
        Width             =    1092
    End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Private Sub Command1_Click()
Dim x As Single
Dim y As Single
x = 10#
y = 20#
Text1.Text = x
Call swap(x, y)
Text1.Text = y
End Sub
```

22.8.2.3 C++

```
extern "C"

{

  _declspec(dllexport)

  void

  _stdcall

  swap(float& x,float& y)
  {
```

```
    float t;
    t=x;
    x=y;
    y=t;
    return;
  }

}
```

22.8.3 Void function example – passing a 1 d array

22.8.3.1 Visual Basic – Module

```
Attribute VB_Name = "Module1"
Declare Sub sum Lib
"e:\export\home\sttp1553\document\cpp\mixed\sum\debug\sum.dll"
Alias "_sum@12" (x As Single, y As Single, i As Integer)
```

22.8.3.2 Visual Basic – Form

```
VERSION 5.00
Begin VB.Form Form1
   Caption          =   "Form1"
   ClientHeight     =   2556
   ClientLeft       =   48
   ClientTop        =   276
   ClientWidth      =   3744
   LinkTopic        =   "Form1"
   ScaleHeight      =   2556
   ScaleWidth       =   3744
   StartUpPosition =   3  ‘Windows Default
   Begin VB.TextBox Text1
      Height            =   852
      Left              =   2280
      TabIndex          =   1
      Text              =   "Text1"
      Top               =   360
      Width             =   1212
   End
   Begin VB.CommandButton Command1
      Caption           =   "Press to call C++ dll"
      Height            =   852
      Left              =   480
      TabIndex          =   0
      Top               =   360
      Width             =   1452
   End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
```

```
Private Sub Command1_Click()
Dim x(10) As Single
Dim y As Single
Dim i As Integer
y = 0#
For i = 0 To 9
  x(i) = i
Next i
i = 10
Call sum(x(0), y, i)
Text1.Text = y
End Sub
```

22.8.3.3 C++

```
extern "C"

{

  _declspec(dllexport)

  void

  _stdcall

  sum(float& x,float& y,int& j)
  {
  float total=0.0;
  int i;
  float* t=&x;

    for(i=0;i<j;++i)
    {
      total = total + *t;
      t++;
    }

    y=total;
    return;
  }
}
```

22.8.4 Passing data back

The previous program example passes data into a C++ function. The next example shows how one can pass an array back from a C++ function. The construction of a complete VB and C++ example is left as an exercise.

```
#include <iostream>

void assign(int *a)
{
int i;
```

```
   for (i=0;i<5;++i)
   {
      *a=i;
      a++;
   }
   return;
}

int main()
{
int x[5];
int i;
   for (i=0;i<5;++i)
   x[i]=i*i;
   assign(x);
   for (i=0;i<5;++i)
   cout << x[i] << endl;
   return(0);
}
```

22.9 Bibliography

The best source of information is the on-line documentation. You need to access this from Visual Fortran On-Line Documentation, not from Developer Studio. The document can be printed and is about 40 pages.

Microsoft also provide some information within Developer Studio.

22.10 Problems

1. Run these examples.

23
Using a class library

One step beyond

Madness

Aims

The chapter looks at the Microsoft Foundation Class Library.

23 Using a class library

These chapter looks at using a large class library – the Microsoft Foundation Class library. The examples obviously run under Windows. We will look at three complete examples.

The source can be found at:

http://www.kcl.ac.uk/kis/support/cit//fortran/mfc/examples/

23.1 Example 1 – Base MFC Classes

The complete program

```
#include <afxwin.h>
class CMainWindow : public CFrameWnd
{
public :
   CMainWindow();
   DECLARE_MESSAGE_MAP()
};

CMainWindow::CMainWindow()
{
   Create(NULL , " Minimal MFC program or application");
}

class CMyApplication : public CWinApp
{
public :
   BOOL InitInstance();
};

BOOL CMyApplication::InitInstance()
{
   m_pMainWnd = new CMainWindow;
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();

   return TRUE;
}

BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
END_MESSAGE_MAP()

CMyApplication Application;
```

Let us now look at each line of the program in more depth.

#include <afxwin.h>

The MFC Library can be divided into two major sections:

The MFC classes

and

macros and globals.

If a function or variable is not a member of a class, it is a global function or variable. This header file makes available all of the necessary globals and macros. These can be used and shared between all MFC classes. This avoids unnecessary duplication throughout the MFC hierarchy.

class CMainWindow : public CFrameWnd

The CFrameWnd class provides the functionality of a windows single document interface (SDI) overlapped or pop-up frame window and member functions for managing the window. To create a window for your application you:

Derive a class from CFrameWnd.

Add member variables to the derived class to store data specific to your application.

Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

Construct a frame window.

In this example we do it directly using Create.

CFrameWnd is three levels into the MFC hierarchy.

CObject is the MFC root. The tree is shown below.

Cobject -> CCmdTarget -> Cwnd -> CFrameWnd

If you want more information on the complete hierarchy then use the help menu and contents to bring up the MSDN on-line documentation. Look under Reference beneath the C++ documentation.

public :

Standard C++ declaration.

CMainWindow();

Constructor prototype. Body given later.

DECLARE_MESSAGE_MAP()

Each CCmdTarget-derived class in your program must provide a message map to handle messages. You must include a DECLARE_MESSAGE_MAP macro at the end of your class declaration. Then in the source file that defines the member functions for the class, use the BEGIN_MESSAGE_MAP macro, macro entries for each of your message-handler functions, and the END_MESSAGE_MAP macro. In this simple example we don’t handle any messages.

CMainWindow::CMainWindow()

This is the actual constructor.

Create(NULL , " Minimal MFC program or application");

Call to create and initialise the windows frame window associated with the CFrameWnd object. This makes the window visible.

The first argument, if NULL, uses the predefined default CFrameWnd attributes.

The second argument points to a null-terminated character string that represents the window name. Used as text for the title bar.

There are a number of additional arguments. We accept the defaults in this simple example. We will look at other parameters in later examples. More information can be found in the on-line MSDN documentation.

class CMyApplication : public CWinApp

The main application class in MFC encapsulates the initialisation, running, and termination of an application for windows. An application built on the framework must have one (and only one) object of a class derived from CWinApp – in this case CMyApplication. This object is constructed before windows are created. (CWinApp is derived from CWinThread, which represents the main thread of execution for your application, which might have one or more threads.)

public :

Standard C++ declaration.

BOOL InitInstance();

Windows allows several copies of the same program to run at the same time. Application initialisation is conceptually divided into two sections: one-time application initialisation that is done the first time the program runs, and instance initialisation that runs each time a copy of the program runs, including the first time. The framework's implementation of WinMain calls this function. We will come back to WinMain later.

We override InitInstance to construct the main window object and set the CWinThread::m_pMainWnd data member to point to that window.

BOOL CMyApplication::InitInstance()

See above.

m_pMainWnd = new CMainWindow;

Use this data member to store a pointer to your thread's main window object. The MFC library will automatically terminate your thread when the window referred to by m_pMainWnd is closed. If this thread is the primary thread for an application, the application will also be terminated. If this data member is NULL, the main window for the application's CWinApp object will be used to determine when to terminate the thread. m_pMainWnd is a public variable of type CWnd*. Typically, you set this member variable when you override InitInstance.

The CWinThread data member m_pMainWnd points to the new window.

m_pMainWnd->ShowWindow(m_nCmdShow);

Sets the visibility state of the window. ShowWindow must be called only once per application for the main window with CWinApp::m_nCmdShow.

m_pMainWnd->UpdateWindow();

Updates the client area by sending a WM_PAINT message if the update region is not empty. The UpdateWindow member function sends a WM_PAINT message directly, by-passing the application queue. We will return to the WM_PAINT message and concept of an application queue later.

return TRUE;

It has worked!

BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)

In the source file that defines the member functions for your class, start the message map with the BEGIN_MESSAGE_MAP macro, then add macro entries for each of your message-handler functions, and complete the message map with the END_MESSAGE_MAP macro. There are no other entries in this example.

END_MESSAGE_MAP()

See above.

CMyApplication Application;

The CWinApp class is the base class from which you derive a windows application object. The class hierarchy is given below. CObject is the MFC root.

Cobject -> CCmdTarget -> CWinThread -> CWinApp

An application object provides member functions for initialising your application (and each instance of it) and for running the application. Each application that uses the Microsoft Foundation classes can only contain one object derived from CWinApp. This object is constructed when other C++ global objects are constructed and is already available when windows calls the WinMain function, which is supplied by the MFC library.

23.2 Example 2 – Toolbars

A number of files are now required.

23.2.1 Menu ID file

```
#define ID_MA_FILE_NEW        10
#define ID_MA_FILE_OPEN         11
#define ID_MA_FILE_SAVE         12
#define ID_MA_FILE_SAVE_AS      13
#define ID_MA_FILE_PAGE_SETUP   14
#define ID_MA_FILE_PRINT      15
#define ID_MA_APP_EXIT        16

#define ID_MA_EDIT_UNDO         20
#define ID_MA_EDIT_CUT        21
#define ID_MA_EDIT_COPY         22
#define ID_MA_EDIT_PASTE      23
#define ID_MA_EDIT_DELETE       24

#define ID_MA_VIEW_NORMAL       30
#define ID_MA_VIEW_ZOOM         31

#define ID_MA_APP_HELP        40
#define ID_MA_APP_ABOUT         41

#define ID_TOOLBAR            50

#define ID_TOOLBAR_BMP        60
```

23.2.2 Resource file

```
#include "resource.h"
#include "menuids.h"

DEFAULTMENU MENU DISCARDABLE
BEGIN
     POPUP "&File"
     BEGIN
          MENUITEM "&New\tCntl+N",ID_MA_FILE_NEW
          MENUITEM "&Open\tCntl+O",ID_MA_FILE_OPEN
          MENUITEM "&Save\tCntl+S",ID_MA_FILE_SAVE
          MENUITEM "Save &as",ID_MA_FILE_SAVE_AS
```

```
          MENUITEM SEPARATOR
          MENUITEM "Page Set&up",ID_MA_FILE_PAGE_SETUP
          MENUITEM "&Print\tCntl+P",ID_MA_FILE_PRINT
          MENUITEM SEPARATOR
          MENUITEM "&Exit",ID_MA_APP_EXIT
     END
     POPUP "&Edit"
     BEGIN
          MENUITEM "&Undo\tCntl+Z",ID_MA_EDIT_UNDO
          MENUITEM "Cu&t\tCntl+X",ID_MA_EDIT_CUT
          MENUITEM "&Copy\tCntl+C",ID_MA_EDIT_COPY
          MENUITEM "&Paste\tCntl+V",ID_MA_EDIT_PASTE
          MENUITEM "&Delete",ID_MA_EDIT_DELETE
     END
     POPUP "&View"
     BEGIN
          MENUITEM "&Normal",ID_MA_VIEW_NORMAL
          MENUITEM "&Zoom",ID_MA_VIEW_ZOOM
     END
     POPUP "&Help"
     BEGIN
          MENUITEM "&Help",ID_MA_APP_HELP
          MENUITEM "&About",ID_MA_APP_ABOUT
     END
END


DEFAULTMENU ACCELERATORS MOVEABLE PURE
BEGIN
     "N",ID_MA_FILE_NEW,VIRTKEY, CONTROL
     "O",ID_MA_FILE_OPEN,VIRTKEY, CONTROL
     "S",ID_MA_FILE_SAVE,VIRTKEY, CONTROL
     "P",ID_MA_FILE_PRINT,VIRTKEY, CONTROL
     "Z",ID_MA_EDIT_UNDO,VIRTKEY, CONTROL
     "X",ID_MA_EDIT_CUT,VIRTKEY, CONTROL
     "C",ID_MA_EDIT_COPY,VIRTKEY, CONTROL
     "V",ID_MA_EDIT_PASTE,VIRTKEY, CONTROL
END

DEFAULTMENU TOOLBAR DISCARDABLE  24,23
BEGIN
     BUTTON       ID_MA_FILE_NEW
     BUTTON       ID_MA_FILE_OPEN
     BUTTON       ID_MA_FILE_SAVE
     SEPARATOR
     BUTTON       ID_MA_EDIT_CUT
     BUTTON       ID_MA_EDIT_COPY
     BUTTON       ID_MA_EDIT_PASTE
     SEPARATOR
     BUTTON       ID_MA_FILE_PRINT
```

```
    SEPARATOR
    BUTTON      ID_MA_APP_ABOUT
END

DEFAULTMENU BITMAP DISCARDABLE "defaultm.bmp"
```

23.2.3 C++ source file

```
#include <afxwin.h>
#include <afxcmn.h>
#include <afxext.h>
#include "menuids.h"

class CMainWindow : public CFrameWnd
{

public :
  CMainWindow();

  afx_msg void OnNew();
  afx_msg void OnOpen();
  afx_msg void OnSave();
  afx_msg void OnSaveas();
  afx_msg void OnPageSetup();
  afx_msg void OnPrint();
  afx_msg void OnExit();

  afx_msg void OnUndo();
  afx_msg void OnCut();
  afx_msg void OnCopy();
  afx_msg void OnPaste();
  afx_msg void OnDelete();

  afx_msg void OnNormal();
  afx_msg void OnZoom();

  afx_msg void OnHelp();
  afx_msg void OnAbout();

protected:

  CToolBar    MyToolBar;

  DECLARE_MESSAGE_MAP()
};

CMainWindow::CMainWindow()
{
  Create(NULL ,
             " Toolbar program " ,
             WS_OVERLAPPEDWINDOW ,
```

```
                rectDefault ,
                NULL ,
                "DEFAULTMENU");
   LoadAccelTable("DEFAULTMENU");

   MyToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISI-
BLE | CBRS_TOP
     | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC);

   MyToolBar.LoadToolBar("DEFAULTMENU");
}


class CMyApplication : public CWinApp
{
public :
   BOOL InitInstance();
};

BOOL CMyApplication::InitInstance()
{
   m_pMainWnd = new CMainWindow;
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();

   return TRUE;
}

BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)

     ON_COMMAND( ID_MA_FILE_NEW,OnNew )
     ON_COMMAND( ID_MA_FILE_OPEN,OnOpen)
     ON_COMMAND( ID_MA_FILE_SAVE,OnSave)
     ON_COMMAND( ID_MA_FILE_SAVE_AS,OnSaveas)
     ON_COMMAND( ID_MA_FILE_PAGE_SETUP,OnPageSetup)
     ON_COMMAND( ID_MA_FILE_PRINT,OnPrint)
     ON_COMMAND( ID_MA_APP_EXIT,OnExit)

     ON_COMMAND(   ID_MA_EDIT_UNDO,OnUndo)
     ON_COMMAND(   ID_MA_EDIT_CUT,OnCut)
     ON_COMMAND(   ID_MA_EDIT_COPY,OnCopy)
     ON_COMMAND(   ID_MA_EDIT_PASTE,OnPaste)
     ON_COMMAND(   ID_MA_EDIT_DELETE,OnDelete)

     ON_COMMAND(   ID_MA_VIEW_NORMAL,OnNormal)
     ON_COMMAND(ID_MA_VIEW_ZOOM,OnZoom)

     ON_COMMAND(ID_MA_APP_HELP,OnHelp)
     ON_COMMAND(ID_MA_APP_ABOUT,OnAbout)
```

```
END_MESSAGE_MAP()

afx_msg void CMainWindow::OnNew()
{ MessageBox("New","New");}

afx_msg void CMainWindow::OnOpen()
{ MessageBox("Open","Open");}

afx_msg void CMainWindow::OnSave()
{ MessageBox("Save","Save");}

afx_msg void CMainWindow::OnSaveas()
{ MessageBox("Saveas","Saveas");}

afx_msg void CMainWindow::OnPageSetup()
{ MessageBox("PageSetup","PageSetup");}

afx_msg void CMainWindow::OnPrint()
{ MessageBox("Print","Print");}

afx_msg void CMainWindow::OnExit()
{ MessageBox("Exit","Exit");}

afx_msg void CMainWindow::OnUndo()
{ MessageBox("Undo","Undo");}

afx_msg void CMainWindow::OnCut()
{ MessageBox("Cut","Cut");}

afx_msg void CMainWindow::OnCopy()
{ MessageBox("Copy","Copy");}

afx_msg void CMainWindow::OnPaste()
{ MessageBox("Paste","Paste");}

afx_msg void CMainWindow::OnDelete()
{ MessageBox("Delete","Delete");}

afx_msg void CMainWindow::OnNormal()
{ MessageBox("Normal","Normal");}

afx_msg void CMainWindow::OnZoom()
{ MessageBox("Zoom","Zoom");}

afx_msg void CMainWindow::OnHelp()
{ MessageBox("Help","Help");}

afx_msg void CMainWindow::OnAbout()
{ MessageBox("About","About");}
```

```
CMyApplication Application;
```

23.3 Example 3 – Simple graphics

23.3.1 C++ source file

```
#include <afxwin.h>
#include <math.h>

int mx;
int my;

class CMainWindow : public CFrameWnd
{
public :

   CDC         my_memDC;
   CBitmap    my_bitmap;
   CBrush     my_brush;
   CFont       my_font;

   CPen        my_redpen;
   CPen        my_bluepen;
   CPen        my_greenpen;
   CPen        my_blackpen;

   CMainWindow();

   afx_msg void OnPaint();

   void DrawGraph();

   DECLARE_MESSAGE_MAP()
};

CMainWindow::CMainWindow()
{
   Create(
   NULL ,
" Simple graphics output using virtual device context"
   );

   CClientDC dc(this);

   mx=GetSystemMetrics(SM_CXSCREEN);
   my=GetSystemMetrics(SM_CYSCREEN);

   my_memDC.CreateCompatibleDC(&dc);
```

```
   my_bitmap.CreateCompatibleBitmap(&dc,mx,my);
   my_memDC.SelectObject(&my_bitmap);

   my_brush.CreateStockObject(WHITE_BRUSH);
   my_memDC.SelectObject(&my_brush);

   my_memDC.PatBlt(0,0,mx,my,PATCOPY);

   my_redpen.CreatePen(   PS_SOLID,2,RGB(255,   0,   0));
   my_bluepen.CreatePen( PS_SOLID,2,RGB(   0,   0,255));
   my_greenpen.CreatePen(PS_SOLID,2,RGB(   0,255,   0));
   my_blackpen.CreatePen(PS_SOLID,2,BLACK_PEN);

}

class CMyApplication : public CWinApp
{
public :
BOOL InitInstance();
};

BOOL CMyApplication::InitInstance()
{

   m_pMainWnd = new CMainWindow;
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();

   return TRUE;
}

BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
   ON_WM_PAINT()
END_MESSAGE_MAP()


afx_msg void CMainWindow::OnPaint()
{
   DrawGraph();

   CPaintDC dc(this);

       dc.SetMapMode(MM_ISOTROPIC);
       dc.SetWindowExt(720,720);
       dc.SetViewportExt(500,500);
       dc.SetViewportOrg(50,50);


   dc.BitBlt(0,0,mx,my,&my_memDC,0,0,SRCCOPY);
```

```
   return;
}

void CMainWindow::DrawGraph()
{

   int    angled;
   double angler;
   double y;
   int sine=180;
   int cosine=360;
   int tangent=180;
   const double pi=4*atan(1.0);

   my_memDC.SelectObject(&my_redpen);

   for (angled=0;angled<361;++angled)
   {
    my_memDC.MoveTo(angled,sine);
    angler = angled*2*pi/360;
    y       = sin(angler);
    sine    = 180 + int(180*y);
    my_memDC.LineTo(angled,sine);
   }

   my_memDC.SelectObject(&my_bluepen);

   for (angled=0;angled<361;++angled)
   {
    my_memDC.MoveTo(angled,cosine);
    angler = angled*2*pi/360;
    y       = cos(angler);
    cosine = 180 + int(180*y);
    my_memDC.LineTo(angled,cosine);
   }

   my_memDC.SelectObject(&my_greenpen);

   for (angled=0;angled<361;++angled)
   {
    my_memDC.MoveTo(angled,tangent);
    angler   = angled*2*pi/360;
    y        = tan(angler);
    tangent = 180 + int(180*y);
    my_memDC.LineTo(angled,tangent);
   }

   my_memDC.SelectObject(&my_blackpen);

   my_memDC.MoveTo(0,180);
```

```
   my_memDC.LineTo(360,180);

}

CMyApplication Application;
```

23.4 Summary

Compiled versions will be made available on the College web server. The source of all of the examples from the book are too. Url is

http://www.kcl.ac.uk/kis/support/cit//fortran/mfc/examples/

23.5 Bibliography

Chivers I.D., *Essential Visual C++ 6.0 Fast*, Springer Verlag.

- Quick introduction to the MFC class library and simple Windows programming.

24
Miscellanea

‘The Time has come,’ the Walrus said,
  ‘To talk of many things:
of shoes–and ships–and sealing wax–
  of cabbages–and kings–
And why the sea is boiling hot–
  And whether pigs have wings.’

Lewis Carroll, Through the Looking Glass and What Alice Found There.

Aims

The chapter rounds off with coverage of a number of topics.

24 Miscellanea

This chapter ties up a number of loose ends.

24.1 Development and Evolution

C++ is evolving. Since the publication of the standard it has emerged that there are a number of areas that need clarification and correction. This is understandable given the ambitous nature of the development of C++. C++ is not alone here and other programming languages have been subject to revision.

The key home page for up to date information in this area can be found at:

- http://www.open-std.org/jtc1/sc22/wg21/

which is the home page for the C++ standard. Two key documents are (as of the time of writing these notes)

- The C++ Standard Core Issues List – Revision 43
  http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html
- The C++ Standard Library Issues List – Revision 44
  http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html

These documents are 110 pages and 79 pages respectively at the moment.

24.2 The Approach

The first aim has been first to look at converting from an existing procedural language. The examples have been taken from a common set of problems that people come across.

The second aim has been to look at the features of C++ that support object oriented programming. The examples are small and highlight language usage in as simple a way as possible.

Finally there has been a coverage of the template facility in C++. This is a very powerfull facility, and it means that you can take advantage of a wealth of pre written code, and you don't have to roll your own.

If you are going to use C++ serious in so called production programs then you need to go one step further. You need to develop a style of programming that is defensive. You need to be able to have confidence that the results you get are correct and how to locate errors when things go wrong.

Hopefully you now have enough knowledge to progress on your own. I've found the following a good source of further information

- Deitel and Deitel - Good coverage, more aimed at someone with a computing background.
- Seed – good wide coverage of C++
- Sroustrup – bit tough, but takes you beyond the simple coverage of the Seed book.
- Josuttis – excellent coverage of the STL

If you have got Visual Studio then I would install the documentation that comes with it.

24.3 C++ Standard Definitions

The following terms are defined in the standard.

24.3.0.1 argument

An expression in the comma separated list bounded by the parentheses in a function call expression, a sequence of pre-processing tokens in the comma separated list bounded by the parentheses in a function like macro invocation, the operand of throw, or an expression in the comma separated list bounded by the angle brackets in a template instantiation. Also know as an *actual argument* or *actual parameter*

24.3.0.2 dynamic type

The dynamic type of an expression is determined by its current value and can change during the execution of a program. If a pointer whose static type is pointer to class B is pointing to an object of class D, derived from B, the dynamic type of the pointer is pointer to D. References are treated similarly.

24.3.0.3 implementation defined behaviour

Behaviour, for a correct program construct and correct data, that depends on the implementation and that each implementation shall document. The range of possible behaviours is delineated by the standard.

24.3.0.4 implementation limits

Restrictions imposed upon programs by the implementation.

24.3.0.5 multibyte character

A sequence of one or more bytes representing a member of the extended character set of either the source or execution environment. The extended character set is a superset of the basic character set.

24.3.0.6 parameter

An object or reference declared as part of a function declaration of definition in the catch clause of an exception handler that requires a value on entry to the function or handler, an identifier from the comma separated list bounded by the parentheses immediately following the macro name in a function like macro definition, or a *template parameter*. A function can be said to *take arguments* or to *have parameters*. Parameters are also known as *formal arguments* or *formal parameters*.

24.3.0.7 signature

The signature of the function is the information about that function that participates in over load resolution: the types of its parameters and, if the function is a non-static member of a class, the CV qualifiers (if any) on the function itself and whether the function is a direct member of its class or inherited from a base class. The signature of a template function specialisation includes the types of its template arguments.

24.3.0.8 static type

the static type of an expression is the type resulting from analysis of the program without consideration of execution semantics. It depends only on the form of the program and does not change.

24.3.0.9 undefined behaviour

Behaviour, such as might arise upon use of an erroneous program construct or of erroneous data, for which the standard imposes no requirements. Permissable undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without a diagnostic message) to terminating a translation or execution (with the issue of a diagnostic message). Note that many erroneous program constructs do not engender undefined behaviour, they are required to be diagnosed.

24.3.1 C++ Implementation Options: Free vs Hosted

Section 1.7.4 of the standard states *Two kinds of implementation are defined: hosted and freestanding. For a hosted implementation, this standard defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation defined set of libraries that includes certain language support libraries.*

This is obviously of importance with embedded systems.

24.3.2 The C++ Memory Model, taken from the standard

Section 1.5 of the standard states *The fundamental storage unit in the C++ memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set and is composed of a contiguous sequence of bits, the number of which is implementation defined. The least significant bit is the low order bit; the most significant bit is the high order bit. The memory accessible C++ program is one or more contiguous sequence of bytes. Each byte has a unique address.*