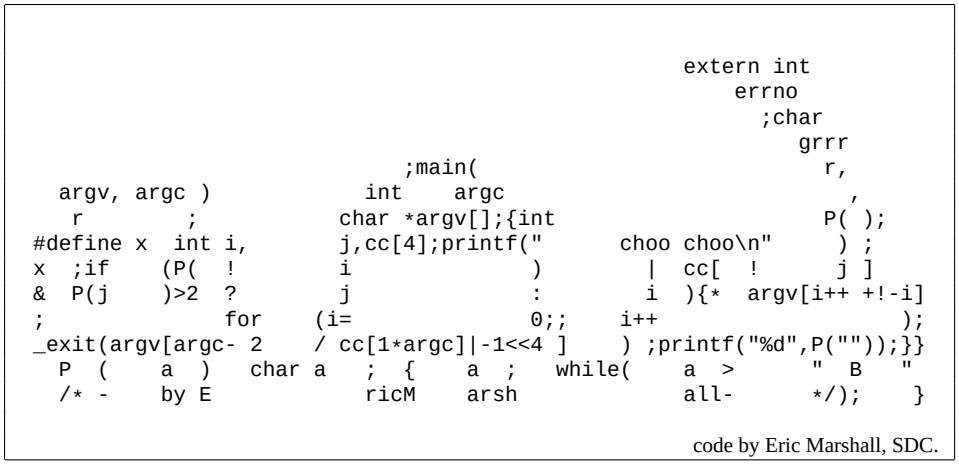
C and C++

A C Norman, Lent Term 1996

Part IB, Part II(G) and Diploma



```
                                                 extern int
                                                     errno
                                                       ;char
                                                          grrr
                          ;main(                            r,
  argv, argc )           int    argc                         ,
   r        ;           char *argv[];{int                    P( );
#define x  int i,       j,cc[4];printf("      choo choo\n"     ) ;
x  ;if    (P(  !        i              )        |  cc[  !      j ]
&  P(j    )>2  ?        j              :        i  ){*  argv[i++ +!-i]
;              for    (i=             0;;    i++                  );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```

code by Eric Marshall, SDC.

1 Introduction: Why These Languages?

These notes cover the languages C and C++. In fact this means that there will be coverage of *three* languages, which represent a historical development that has taken place over about the last twenty years and which is still continuing. The most primitive of these is best described as the "traditional" dialect of C sometimes referred to by the initials "K&R" (which stand for Kernighan and Ritchie who originally designed it). There is still a lot of code written in this dialect around, so all proper Computer Scientists need to be able to make sense of it. There are even some compilers that only support it in use, but if at all possible these should be shunned. The second language will be known here as just plain C but if you need to stress which dialect is involved or make a big point that the K&R one is not what you mean, the qualification ANSI-C can be used. "ANSI" is the name of the formal standardisation authority responsible for defining this version of the language. Their report[7] was published in 1989, and these days if you find a way in which a C compiler does not meet the ANSI specification you should consider it to be broken[1]. Various vendors of C compilers put in language and library extensions to their ANSI C compilers, and a few of the more interesting of these will be mentioned. C++ is a more modern language but is broadly upwards compatible with ANSI C and so (with luck!) existing C code can be compiled with a C++ compiler and linked in with freshly written C++ code when new projects are started. Of course in reality it is a bit harder than that. At present C++ does not have a format standard, although an ANSI committee is hard at work preparing one. An effect of that is that different C++ compilers will not all provide the same facilities, and even when they do there may be subtle differences in the meaning they attribute to some contorted fragments of code. These notes will give some examples of constructions that (at the time of writing) remain delicate.

Why, then, does it make sense to study this family of languages? The following reasons are some of the more important ones:

Management Says So. One compelling reason for studying or using a programming language is that the use of that particular language is Company Policy. In a University environment this translates that there will be examination questions on it at the end of the year!

Legacy Code. Essentially all of the Unix Operating system is written in C, as will be the majority of the important commercial packages you can buy for use on your personal computer. Almost all existing computer companies will have significant bodies of old code that has to be maintained, and a large proportion of this "legacy code" that is not written in COBOL is written in C.

Library Linkage. If you write a new application that is to run under Microsoft Windows, or the Unix-based X-windows system, or the desktop on a locally designed

[1] Of course pretty well every compiler, like every other program you ever come across, will be broken in various ways!

Acorn RISC-PC, or a Macintosh, ..., you need to interface to a large body of library code that helps run the graphical interface for you. In all the above cases (and of course many others) there will be well developed C or C++ interfaces to this library, and very often this will be the *only* language that makes it easy to make full use of the system. As well as user-interface libraries that force you to use C you might come across data-base ones, or numerical ones (eg the NAG library can be called either from Fortran or (yes you guessed) from C).

Practical, Portable, Efficient Code. For a reasonable range of system-building tasks C represents a plausible compromise between practical reality and a Computer Scientist's ideal. The main dreadful features that K&R C had that made it constitutionally insecure and dangerous have been very well addresses by the ANSI dialect, and C++ tends to be even more careful about cross-checking things (although it introduces a whole new collection of delicacies of its very own). When written with care and style C code can be ported to all the machines you will ever want to[2], and again if used with caution it can be used to write code that is performance critical.

Employment Prospects. As already suggested, C and C++ are associated with reasonable scale real commercial projects, and thus with making money. So even if C is not the very latest and most wonderfully forward looking programming language it may be a useful one to be able to claim familiarity with!

Embedded Applications. C and C++ can reasonably be used to write code for dedicated controllers (eg to look after machine tools, network interfaces, industrial plant, car engines, ...) as well as being suited for use on general purpose workstations. This opens up a large extra range of applications.

Closeness to the Machine. With C the statements used in the program language can relate in a clear and direct manner to machine-code constructs. This tends to reduce to opportunity for surprises by way of unexpectedly bad performance, and makes it easier to write code that interacts directly with hardware. It also means that C is a useful language to know when learning about the design of computer instruction sets, as the language user's perspective meets that of the hardware designer.

Fairly Simple Compiler. Following on from the fact that each class of C (or to a slightly lesser extent C++) statement maps neatly onto machine code of a wide range of computers, the language is a good one to keep in mind when first studying compiler construction. The extra complications that have to be handled with some more elaborate languages can be considered later on, while C provides

[2] And unlike some other nominally more machine independent languages, you will find C compilers available on all those machines.

plenty of scope for discussing compiler optimisations (and the limits to what can be done automatically).

Availability of Cheap, Reliable Compilers. For most sorts of Unix workstation and for a variety of personal computers, the Free Software Foundation's "g++" compiler will be available and has the stunning advantage of being free! For IBM-style personal computers there is a highly competitive market in C and C++ compilers. Some of the smaller and cheaper of these look backwards to the days when personal machines were uncomfortably small, and are in various ways limited or full of non-standard space-saving tricks, but the more modern and professional versions are of stunningly high quality — much better than any workstation software development tools I have ever seen. Because of the competition, prices are almost reasonable.

Development Tools. No compiler (be it for C, C++ or any other language) stands totally on its own: associated development tools and subroutine libraries are needed. Because of its widespread use C has collected a good range of these, and C++ inherits most of them. In a typical University Unix environment these tools will still be fairly primitive, but the EMACS editor can be configured to help lay out C code in a systematic manner, and there are Unix tools like "prof" and "pixie" (on some systems) that can help you collect information about which parts of your code are most heavily used. Debuggers such as "gdb" (or "sdb"or "adb") may assist when your programs fail. Resource Control packages can help keep track of all the files in a large project.

On personal systems the development environments have been much more carefully designed, and the debugging tools that I have seen on Microsoft Windows, the Macintosh and on Acorn machines are all streets ahead of the ones made available on most Unix systems. I would single out the debugger that comes with Microsoft's Visual C++ (32-bit edition) running under Windows NT as being almost good enough to justify the rather large amounts of disc space and memory it needs!

Historical Interest. C was developed as a successor to BCPL (which was invented by Martin Richards), and that in turn was at first intended to be a language that would be used to write the compiler for a language CPL. These roots carry the historical thread from C++ and the present back to the mid 1960's. By looking at the ways languages, their compilers and the programs written in them have changed over that time-scale we can get some insight into how Computer Science (as well as Computer Application) has changed, and perhaps that will help us when we try to peer forward and predict future developments.

C and C++ are Fun. Some programming languages are very carefully designed to ensure that only correct code will pass the compiler's scrutiny. They emphasise precision of expression in an utterly humourless and intense way. C and C++

are not like that. They expect the programmer to be competent and careful, but provide some scope for the inclusion of cunning tricks in code, and even jokes. To try to support this claim I am including a collection of somewhat curious sample C programs with these notes...

The above collection of strengths embodied in C++ may suggest that it is the perfect language for all imaginable uses. The fact that in Cambridge we do not teach it as the first and only language presented to students suggests that the department here thinks otherwise. So here are a few possible causes for caution. Even though I have not given as many disadvantages as I listed advantages, some of them should be viewed as pretty serious limitations:

C++ is not Standardised. Until there is a formal standard for C++ (and until, several years after that, the bulk of compilers have caught up with the standard), every C++ compiler will support a slightly different language, and the behaviour or portability of code can not be assured. By falling back to (ANSI) C the benefits of a standard language can be obtained, but at the cost of losing the new features that C++ provides.

C++ is complicated. A comittee of the relevant standards-issuing body is at work on codifying C++. During 1995 it issued its first major document for public review. This will evolve into the official C++ standard, but the final form will probably have changed in several quite significant ways. This first draft of the standard is over 700 pages long, and that is dense description rather than gentle tutorial explanation.

Incomplete compile-time checking. As compared to (for instance) Module-3, C and C++ are insecure languages. In particular uses of casts and union types (which will be described in the lectures) make it possible to write outrageously incorrect code and have the compiler accept it quite placidly. Even when used with care C pointers can too easily escape and allow incorrect code to corrupt almost arbitrary fragments of code or data. Writing safety-critical code in C is probably a really bad idea. Note that K&R C was an almost complete calamity on this front — both ANSI C and C++ are much better, but still they are not the language of choice for missile guidance systems programming, aircraft "fly-by-wire" control, nuclear power station monitoring software or some medical applications.

Cryptic Syntax. Real code has to be re-cycled, modified and maintained. Clear syntax makes it easier for somebody coming across a fragment of long-neglected code to understand it. C can be cryptic in places, while C++ raises this to a high art. In contrast, ADA had as a design goal that code written in it should be easy to read even if that made it more verbose and hence more tedious to write in the first place.

Reliance on Programmer Discipline. The "Spirit of C" is that the programmer knows what to do and will take full responsibility for the code as written. The job of a compiler is to convert this source code into equivalent machine code that will run fast on the relevant target hardware. This works acceptably well with experienced, conservative, well-disciplined programmers who are not working under over-severe time constraints. Or put another way, in reality a lot of C code that is written is ill thought out and shoddy, and the compiler does little even to point this out. In many cases this will lead to much higher whole-life-cycle costs than would the initial use of a much more pedantic and fussy language.

Lack of Modern Features. C lacks objects, and while C++ adds these, it does so in a way that is more static and limited than some other object oriented languages (notably the CLOS component of Common Lisp). Neither C nor C++ provides language-defined facilities to support any form of parallel processing or multi-tasking. The standard libraries for the languages provide somewhat primitive input and output control, and certainly no direct support for persistent data or database searches. Despite the illusion it likes to give, memory management in C++ is (following on from that in C) primitive (in particular it does not have a Garbage Collector). People who want to do large-scale scientific calculations (the sort that involve thousands of millions of floating point operations, where it is sometimes important to save space or time by using single precision floating point arithmetic) find C and C++ seriously deficient. Various of these limitations can be addressed by the provision of extra library functions for C code to call, but that again runs into portability and standardisation problems.

Rational alternatives to the C family of languages could include

Higher Level Languages. Either Modula-3 or ML could provide much better guarantees of the correctness of programs. For example ML has been used in a number of large-scale projects involving formal specification and verification of hardware design, while Modula-3 is designed to encourage (enforce?) good practise in multi-programmer projects.

Other Pragmatic Programming Languages. If the task you need to solve creates and discards data in a sufficiently wild way it may make sense to use a language that provides automatic garbage collection. Equally if your program needs arbitrary precision arithmetic, maybe a language that supports that already will be helpful: Lisp should be considered. For other application areas Prolog (for medium-sized database search, pattern matching and back-tracking) or Snobol (transformations on strings of characters) may be best suited. Some small tasks may of course be most conveniently solved in BASIC[3].

[3]Indeed Microsoft's Visual Basic is clerarly the language of choice for lashing some sorts of one-off Windows interfaces.

Databases and Spreadsheets. Particularly on personal computers various packages exist to provide reasonable programability in a framework well suited to various tasks that commonly arise in an office environment. When a University comes to add up exam marks and sort candidates into rank order the software to do that should be created by adding rules to a database or spreadsheet package rather than by writing a complete program from scratch in C.

Packages. Increasingly it will be the case that tasks can be solved by using a bought-in package that does what is needed: so no fresh programming is needed. Except in artificial cases where the time spent writing a program is not recorded and charged for this will often be the best way to go, even if the package obtained turns out to be slightly limited and awkward to use. Just because you know how to program in C++ there is no need to write your own screen editor — there are more than enough ones out there already.

2 Course Content and Textbooks

This course starts with the assumption that those taking it understand about programming in general (from the earlier courses that involve ML and Modula-3), and that they have taken the local Data Structures and Algorithms course so examples based on that can be used to illustrate aspects of C++ programming. The position taken is that C++ is the main language being described. ANSI C will be treated as if it were just a subset of C++, so most of the discussion of it will be in the form of passing notes that indicate where C++ features are not available in it. The exception to this will be in Chapter 19 which pays particular attention to some of the fine print in the ANSI standard for C.

In a limited length course that is not accompanied by large amounts of mandatory practical work it will not be possible to show you every last detail of C++ and turn you all into expert programmers in it. The objective here is to show you enough of the important aspects of the language that you can go away and start writing code, and to give you enough understanding that you can find your way around the textbooks. I want to stress that in twelve lectures I can not possible turn you into C++ wizards, and so I will not try that. Instead I will introduce the language and try to get across some of the feeling and spirit associated with it. My hope is that by the end of the course you would be able to implement smallish bodies of code (eg implementations of things from the Data Structures and Algorithms course) provided you had one of the textbooks available for reference. The rest only comes with lots of practical exposure over a period of several year.

These lecture notes are intended to accompany the course, and not to replace either the lectures or the textbooks. Although the material covered here will broadly match that in the lectures the detailed order of presentation will certainly differ, as (for instance) points get explained when they arise in an example program that you get shown.

If you go into a large bookshop and find the section devoted to C++ you will find that there are a very large number of textbooks available, and more will come out each month. In the end the selection should depend on your own interests and preferences, and a judgement you may like to make on how much use of C++ you will make after this particular lecture course ends.

Very many textbooks on C++ start with the assumption that the reader is already a seasoned programmer in plain C. Indeed it is often assumed that it is old-fashioned K&R C that is the starting point for a study of C++. Another whole shelf-full of C++ books in the shops relate very specifically to various particular (IBM pc) implementations of C++. These often spend many of their pages explaining how to wave a mouse around and drive the relevant idiosyncratic development toolkit, and how to make write programs in the specific context of Microsoft Windows. Neither of these approaches is really satisfactory for the course given here. At least one other book I considered recommending has been omitted from this list because I consider the typography and layout grimly distracting and ugly. The books suggested here represent perhaps plausible compromises: the ones on C rather than C++ represent more definite recommendations since that section of the market is more stable.

C++ Primer (2nd edition), Stanley Lippman, 1991, Addison Wesley.

This is at least at present my main recommendation. It costs just under £30, and will serve as a reasonable reference manual as well as a textbook to learn C++ from.

The Standard C++ Library, Plauger, 1994.

This has only appeared on the bookshop shelves during the final few months of 1994, but covers the current draft specification that will turn into ANSI's formal recipe for a C++ library. Do not expect all existing implementations of the language to meet this description yet, and consider waiting another year to get edition two or three of this book as the standard gets closer to adoption. Despite these quibbles, this book is something of a landmark, because the other C++ books give decidedly light coverage of library issues.

C++ with Object Oriented Programming, Paul Wang, 1994, ITP.

One reason for me to include this on my list is that I know Paul Wang — and I am quite happy to direct royalties in his direction. He takes the view that teaching should start directly with C++ rather than going through C first, and his book is a little more gentle than Lippman's.

The Annotated C++ Reference Manual, Ellis and Stroustrup, 1990, Addison Wesley.

For reference rather than as a gentle introduction, and hence I suggest that most of you do **not** rush out and buy this. The time to buy definitive reference material relating to C++ will be when the standardisation process is complete and the

language has stabilised. However, borrowing this from the library for a few days may be interesting. The same comment applies to pretty well any other C++ book with Stroustrup as author!

ANSI X3J16 working drafts, 1994-.

At various later stages in their development of a standard for C++ the technical committees involved releases drafts for public comment. The first of these has now been circulated. There are three reasons why I suggest that you do not search for it on the internet and download it: (a) at over 700 pages it would use up your free printing allowance too fast, (b) since this is just the first public-review document the language C++ will change in a number of significant ways between now and the final standard and (c) official standards documents are dense and detailed, and generally **not** useful for learning languages. The information included is also much too much to relate to a twelve lecture course.

ANSI X3.159, American National Standards Institute, 1989.

This is the official standard for C, subject to later statements in clarification or amendment. It is not readily available in bookshops, and probably only really needed by those who are going to try to write their own C compilers or set themselves up as utter expert in all the dark corners of the standard. Find a copy in a library (there should be one in the book-locker) and inspect it for a few minutes. The version published by ANSI has an accompanying "rationale" that explains the thinking behind some of the decisions made by the committee, and this is both readable and enlightening. Most recent reports were that a copy of this document would set you back £180!

C, a Reference Manual, Harbison and Steele, 1987, Prentice Hall.

Probably the most sensible general purpose C book to have on your bookshelf. The main reason it does not now count as quite essential is that this course now views C as a subsidiary language to C++. However most C++ books give only rather superficial coverage of the pre-processor and the C library, so until C++ has been standardised and these aspects of it are well documented in an explicitly C++ context this will remain a very useful reference work to own.

The C Programming Language, Kernighan and Ritchie, Prentice Hall.

You might reasonably believe that the book by the original inventors of a language would be the best volume for you to have on your shelves, and that it would give the clearest and most definitive description of the language. In the case of Snobol, Griswold's original manual still probably counts as a classic, but for C there book by K&R is less precise, less complete and less balanced than several alternatives. This applies both to their original 1978 language definition

and to their updated book documenting ANSI C. I mention the book here to suggest that for all ordinary student use it is not the correct choice (and when I last looked its cost per page seemed exorbitant too).

Obfuscated C and Other Mysteries, Don Libes, 1993, Wiley.

This costs too much at over £40, but gives some insight into the real problems of writing portable and reliable C code: even when it does so by giving examples of the converse. Some of these examples are included in these notes, but without much explanation — if you want the full story on them turn to Don Libes' commentaries. I enjoy this book.

Microsoft Foundation Class Primer, Jim Conger, 1993, Waite Group.

Earlier I dismissed as irrelevant books that concentrated on just one particular brand of computer or version of C++ compiler. With this book I make a (qualified) exception. It is **not** a book to teach you C++, but it does give a respectably clear explanation of a large-scale application of the language, viz the construction of programs that work under Microsoft Windows. I have found its explanations both clear and helpful — so if you find yourself involved in a Windows project sometime this may help you out.

Computer Related Risks, Peter G Neumann, 1995, Addison Wesley

This book is not directly about either C or C++, but it is about the consequences of failure in computer systems. Any time you write part of any important body of code you should read and re-read both this and Fred Brooks' "Mythical Man Month" to remind yourself that programming is not a cost-free abstract activity done just for fun. The Computer Related Risks book is the best collection I know that gives detailed citations of (numerous) cases where programmer error or system-designer oversight led to death or major loss. It is a macabre mixture of the horrifying and hilarious! Strongly recommended.

3 Practical Work with C and C++

It is impossible to get a real feel for any programming language without writing a reasonable number of programs of your own. On the Cambridge Unix systems this may involve the use of "`g++`", the GNU[4] compiler for C++. Equally, on Thor it may be that the approved C++ compiler is one provided by Sun, called `CC`.

[4]The acronym GNU recursively stands for "GNU is Not Unix", and you should read the interesting accompanying copyright notice, which basically says that all GNU software can be copied — and that nobody may do anything to prevent anybody else from freely copying it. If you are concerned about possible future commercial exploitation of any of your code you may like to make a special point of reading the GNU Public Library License. It is reproduced as Appendix A, and further limitations that apply to DOS `g++` executables are listed in Appendix B

To use `g++` you should prepare a source file with a name such as `sample.cc`, and then go

```
g++ sample.cc
```

After a suitable pause and if your program was syntactically correct, `g++`[5] will leave a file `a.out` with the compiled version of your code in it. Your code is then activated by saying just

```
a.out
```

If you want the executable version of your code called something more interesting that `a.out` just add `-o program_name` to the command line that invokes `g++`.

If you happen to have convenient access to a PC that has a C++ compiler installed on it there is probably extensive on-line documentation to help you get started. If the compiler involved has an integrated development system you should probably use that to create your files, and then compiling and running programs will be a very minor matter of pointing the mouse at some suitable button. But if you do take that route, please note that new releases of most PC compilers are made at 1 year to 18 months intervals, and since C++ has been a rapidly developing language it is almost certain that old compilers will fail to support some of the potentially interesting features[6]. I have used Zortech (now Symantec) C++ and Microsoft's Visual C (the 32-bit edition) and been pleased with both. The Borland compilers are readily available and cheap, and so are also worth considering. If you move on from just learning the language to working on large-scale projects there are other compilers on the market, and (which is perhaps more important) a selection of tools and libraries that may help you control a project or prototype your user interfaces. You can also obtain `g++` for DOS, and a copy of that will be on the Computer Laboratory teaching filespace from which you can take a copy. Note that different releases of this software have different installation rules and different collections of bugs — the version on the Computer Laboratory filespace is reasonably up to date and furthermore is the one that I have used when checking some of the course examples on a PC. With DOS GCC the basic recipe for compiling and running a simple program will be involve ensuring that a an environment variable (`DJGPP`) has been set up to point to one of the files from the distribution kit, and then just

```
gcc sample.cc -lgpl -lm
go32 a.out
```

In this case the driver program `gcc` treats files whose name ends in "`.c`" as containing ordinary C programs, and ones whose name ends "`.cc`" as being in C++. The curious `-lgpl -lm` just links in some extra libraries — if you accidentally forget to request them you will probably see complaints about undefined symbols.

[5] Use `cc` or `gcc` if your program is in C rather than C++.

[6] Templates and Exceptions are the current leading edge, and in MSDOS/Windows products the last year or so has seen a major (and very welcome) move from 16-bit to full 32-bit implementations of languages

Many other computers that you might gain access to will have C++ compilers installed on them, but you should be aware first that the requirements on filenames you use for your source code are not standardised, and that since C++ is still a developing language some implementations will be more bug-free and more up to date than others.

To finish this section I should follow tradition and present a first example of code from the C family. I will use the most primitive version of the language (ie K&R C), and in accordance with long and honoured tradition the entire purpose of the program in figure 1 will be to print the message "hello world!". The version of this program that I have selected to show here was written by Bruce Holloway, is in the Public Domain, and is explained further in [4]. For the moment I will omit a detailed explanation of how the code works (after all it achieves a very simple result, so maybe there will be no problem[7]), but it does illustrate a worthwhile range of C constructs. If you want to try it out please compile using the command `cc` rather than `g++` so that you hand it to a compiler that accepts old-style C.

There are a number of small example programs included in these notes, and it is hoped that you will type in and try (and then modify) the smaller ones of these, and at least try to puzzle through some of what is going on in the larger ones. A selection of machine-readable resources will be available on the Computer Laboratory teaching file-space too.

4 Debugging

A first, and probably best suggestion about debugging C and C++ code is that you should try very hard to arrange not to have to do any. All possible effort should go into ensuring that code is absolutely correct. This is because wild C code can all too easily overwrite memory that contains code or is not related to the data structures mentioned in the erroneous fragment of program. An effect is that a significant number of mistakes show up not as neat local failures but indirectly because of the way they lead to corruption that causes catastrophic failure in unrelated parts of your code.

A second reason to be especially cautious with C and C++ is that the (cheap) compilers that the department provides for your use are not equipped with powerful or convenient debugging tools. For the purposes of this course[8] you should suppose that if a program fails it just stops, without even guaranteeing to finish sending all recent output to the screen.

As a consequence, a proper policy when writing code is to use a somewhat defensive style, so that at well chosen places you apply consistency checks, and at other well chosen places you arrange to be able to make the program report on its progress — so that if it then fails the log of progress reports will allow you to reconstruct what was going on in the run-up to the crash. Section 8 will explain how the extra print statements

[7] Huh?

[8] Unless you have access to one of the more expensive PC compilers that comes with a elaborate debugger that allows you to inspect the values of variables after a program has crashed.

```
/* Program by Bruce Holloway, Digital Research */
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
    case f:
        a=(b=(c=(d=g)<<g)<<g)<<g;
        return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
    case h:
        for(a=f;a<j;++a)
            if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
    case g:
        if(n<h)return(g);
        if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
        else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
        if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
        return(o[b-g]%n+k-h);
    default:
        if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
        for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
    }
}
```

Figure 1: A C program to print “hello world!”.

can be left in your source file even when your code is believed to be finished, ready to be re-activated when the next “last bug” surfaces and needs to be tracked down.

5 Simple Data and Simple Operations

This section explains the most basic parts of C++. It will not be a complete explanation of anything, but is intended to provide enough understanding that modest chunks of code can be read and hence the more elaborate features that come later can be explained. At this stage I will show how to write individual functions, but not include enough information to turn them into complete or useful runnable programs.

The first feature of C++ to explain is how to write comments. This is in the natural expectation that all the (serious) code you ever write will need plenty of these. There are in fact two ways of writing comments in C++. The main rule is that everything from “`//`” until the end of a line will be ignored. In new C++ code you might well use just this style. But (oh joy) C had a different convention (which C++ still honours), whereby a comment starts with “`/*`” and ends with “`*/`”. Missing out the “`*/`” can cause an arbitrarily large segment of your code to be discarded, and the comment delimiters do not nest, so there is possible danger if you write some code and later comment it out as here:

```
    here is some code
/*      Write a comment marker to start commenting code out
    debugging code here, now not wanted
    debug code /* with comment */
    more debug code  // unexpectedly (?) not commented out
    end of unwanted debug code
 */                 // intended to match first '/*'
```

Many programmers will lay block comments out in some stylised format to help keep code in a consistent style. Three useful layouts for the block of comments that will precede each function definition or small block of definitions are:

```
/*
 * Here is a block comment to explain what my code
 * is supposed to do. Valid for C or C++.
 */

/************************************************
 * Nice layout for comment at start of major block? *
 ************************************************/

////////////////////////////////////////////////
//  This style is ONLY valid in C++, not plain C.  //
////////////////////////////////////////////////
```

C++ as a language design supposes that the programmer has good taste and will write code with care and thought. You may like to consider the possible pitfalls of the following uses of "`/*`" comments:

```
/* here is the start of a comment
/* here is continuation of it
/* the extra '/*' has no significance to C!
/* however if I try to put '*/
/* (oops) in my comment I may be in trouble */

/*/ This is comment /*/
this is not in comment, but another '/*/'
even within quote marks switches me back to
within comment /*/ Maybe it is wonderful to
have the same string to start and end comments?
But what /*//*//*/ if you miss one out?
```

Now on to more active aspects of the language. The primitive data types supported are integers, floating point numbers, arrays and pointers. Each of these come in several variants. Variables may be declared using syntax such as:

```
int x, y;      // simple integer variables
double z;      // double precision floating point
int m[10];     // an array of integers
double pi = 3.14159; // initialised declaration
```

The variables so declared can be used in arithmetic expressions with the usual operations applied to them, however the notation used for comparisons (especially the equality test) are worth making a special note of.

Unary `-`: Negate a value, as in `- x`.

`+, -, *, /`: Ordinary arithmetic operations, which can be used on either integer or real values, as in `(x+y)*(p-1)`.

`%`: Used in integer contexts to compute a remainder.

`~, &, |, ^`: Bitwise negation, AND, OR and exclusive OR operations (on integers).

`<<, >>`: Left and right shifts of the bit patters in an integer.

`<, <=, >, >=, ==, !=`: Comparison operators. Note especially that the test for equality is written as `==`, which is not the convention used by other programming languages! In C++ zero is used to represent *false* and non-zero values (with 1 as the standard case) standing for *true*.

`=`: A single `=` sign is used to indicate assignment. So note that `a=0` sets the variable tt a to zero, while `a==0` tests to see if it is already zero.

`+=, -=, ...`: Any binary operator can be combined with `=` to allow assignments such as `a += 2;` which adds two to `a`, or `b &= 1;` which masks `b` with `1`. `++,--`: The use of `++` and `--` is discussed later on.

`!, &&, ||`: These operators are used to operator on truth values, as in composite tests such as `(a>b) && !(p<q)`. Note that in the form `A && B` if `A` evaluates to false (ie zero) then `B` is not evaluated at all, and similarly in `A || B` if `A` is true (nonzero) then `B` is not evaluated.

`*, &, ->, ?`: The use of `*` and `&` as unary operators, `->` as an infix operator and of `?` will be discussed later.

The meanings described above apply when the operators concerned are used in association with integer or floating point values: in C++ the same operators can be re-used for totally different purposes when user-defined data types are introduced.

Floating point values occur in two flavours, and variables of the two sorts are declared using the words `float` and `double`. The first of these is for single-precision floating point, the second for double. There are not many[9] reasons to use single precision arithmetic, and there are some curious pitfalls, so until you have finished this course and read one of the textbooks carefully please use only `double`.

Integers are much more complicated, in that C++ provides many different integer data types. These are:

```
char      signed char     unsigned char
          short int       unsigned short int
          int             unsigned int
          long int        unsigned long int
enumeration types
```

The language allows an implementation to choose the exact precision used by each of these. The intent is that `char` holds a character, and it will generally be an 8-bit data type, ie one byte. `int` is expected to be the integral data type that is most natural for the computer on which you are working. On most current workstations this will be a 32-bit (4-byte) type, but with some compilers on smaller machines it will be 16-bit, while on some newer larger workstations it is 64-bits. `short` may be the same as `int` or it may provide less bits, and `long` may be the same as `int` or it may be longer! Usually (of if the `signed` qualifier is used) integer are taken to have a range that includes both positive and negative numbers: on a typical computer (with

[9]There are two important reasons why you might need to use single precision floating point. You may need to work with data files or structures whose format has already been defined and where that format includes single precision floats, or you may have **very** large arrays of floating point data where saving space by storing just single precision values is vital.

2's complement[10] arithmetic) where `char` is 8-bits this would mean (for instance) that a variable of type `signed char` could hold values in the range -128 to +127. The qualifier `unsigned` changes this interpretation and indicates that values should be interpreted as positive numbers, and so `unsigned char` would cover than range 0 to 255. Note that if you are attempting to write C or C++ code that will port effortlessly across a wide range of machines you need to avoid making hidden assumptions about the exact widths of these integral types.

Integer constants can usually be written in the obvious way, as in 0, 123, 999999 and so on. The compiler will give the constant that has been written one of the types `int`, `unsigned int`, `long int` or `unsigned long int` depending on how large it is, and almost always this will cause things to behave the way you want them to. In a few cases it is desirable (or even essential) to ensure that a numeric constant has a known type. Appending `U` to a number forces the compiler to treat it as unsigned, while suffixes `S` or `L` indicate short or long values. Thus `0L` stands for a zero of type `unsigned int`.

Octal and hexadecimal[11] constants can also be written. Any integer written with a leading zero will be interpreted as being in octal (thus `0377` is a bit-pattern with 8 low-order bits set, and so (on a 2's complement machine) represents the value 255). Hexadecimal constants are introduced by `0x` and use the letters `a` to `f` to stand for digits with weight 10 to 15. For example `0xff` is again 255 (probably), and on a 32-bit machine `0x80000000` is the most significant bit in an `int`.

Enumeration types will be discussed later on.

What about characters? It has been explained that the types `signed char` and `unsigned char` are signed and unsigned integral ones capable to holding one "character". The type `char` (without either `signed` or `unsigned` qualifier) will be identical to one or other of these, but the choice of which is left up to the compiler writer, who is expected to choose whichever will be faster or most natural on the computer involved. Again this is a region where care can be called for when writing portable code. Character constants are written in single quote marks, as in `'A'`. Such constants stand for some numeric code that will be used to represent the specified character. The language makes no guarantees about the encoding used, save that a character constant will yield a number small enough to fit in a `char`. In particular it is not guaranteed that `'0' ... '9'` will be consecutive codes, even though on most implementations they will be. Within character constants the character "\" is very special: it causes the character following it to be grabbed and used to allow the specification of various characters that might otherwise be hard to express[12]:

[10] C++ does not insist that the computer it runs on uses 2's complement arithmetic, but the meaning of bitwise operations, shifts and hexadecimal constants would give trouble otherwise.

[11] Base 16.

[12] This is not quite a complete list of the possibilities

`'\n'`	⇒	Newline character
`'\t'`	⇒	(Horizontal) tab character
`'\b'`	⇒	Backspace
`'\\'`	⇒	A backslash (\)
`'\''`	⇒	A single quote mark
`'\"'`	⇒	A double quote mark
`'\nnn'`	⇒	`nnn` must be up to three octal digits, and this is the character with that code. The most sensible use for this is just `'\0'` to stand for the character code zero.

For almost all purposes you should think of character constants as being of type `int` rather than `char`, and when you declare variables to hold character values you should use `int` variables. This may seem strange at first!

To a first approximation, strings in C++ are just arrays of characters, and string constants can be written ny enclosing the required text in double quotes:

```
"Here is a sample string ending in a newline\n"
```

Now I can introduce the feature that perhaps gives C and C++ their most essential flavour: pointers. If *T* stands for some type, and can be used to declare variables, than *T* `*` can be used to declare variables that can contain pointers to objects of type *T*. The unary `*` operator follows a pointer and retrieves the value pointed to.

Consider the following:

```
   char *s = "test string";  // s points at start of string
   int c0 = *s;              // makes c0 == 't'
   s = s + 1;                // increment pointer
// Note that C++ permits declarations to be written after
// regular executable statements, while C would not, thus
// (for this reason alone) this sample code is valid C++
// but not valid C.    cf C++ allows "for (int i=0; ...)"
   int c1 = *s;              // c1 == 'e'
   int c3 = *(s+3);          // c3 == 's'
   int c4 = s[4];            // c4 == 't'
```

The statements shown are intended to indicate that arithmetic on pointers can be used to step along a vector, and the last line to suggest that the neat array reference syntax `s[4]` is really nothing more than short-hand for `*(s+4)`.

It is very common to want to access in turn each item in a vector, and C++ provides especially convenient notations to support this. The syntax `x++` means return the value of the variable `x` but after this value has been evaluated, increment `x`. Combined with the `*` operator this can be used as `*x++` which follows the pointer `x` and then moves x on to point to the next item to be processed. By analogy `*y--` follows the pointer `y` then decrements `y` thus supporting backwards scans of data. Sometimes it is useful to perform the increment or decrement operation before doing the indirection, and in

such cases the ++ or -- is just written before the variable name. A useful idiom this provides is that of a *stack*. For a stack of characters, growing upwards and with the stack pointer pointing directly at the most recent character pushed, I could write

```
char stack_area[100];   // make space for the stack
char *sp = stack_area;  // points to start of array
*++sp = <value>;        // push value onto stack
*++sp = <value>;        // push another value
// Note I have not implemented any overflow check here.
int res1 = *sp--;       // pop off a value
int res2 = *sp;         // access top value, but leave it
sp--;                   // pop stack as separate operation
```

The ++ and -- operations do not have to be used in association with addresses and indirection, they work on any integral data types as well as on pointers.

The unary operator & can be used to take the address of a variable or other item.

When arithmetic is done on pointers C++ applies rules that try to be helpful — adding one to a pointer increases the address referenced by the size of the object pointed at. The unit of addressing is supposed to be the same as the size of a `char`. In the examples given above the pointers were all to `char`s. If you have a pointer `p` into an array of integers then `++p` is still equivalent to the statement `p = p+1;` but things will be arranges to that p moves on to point at the next integer. On a 32-bit machine if you looked at the bit patterns involved this might appear that `p` has had 4 added not 1! Similarly the expression `p[i]` will still mean just the same as `*(p+i)`, but on a 32-bit system it will compile into code that looks a little more as if you had written `*(p+4*i)` where the multiplication by 4 is to allow for the size of each integer. People who are being silly can exploit the rule that `a[b]` means just the same as `*(a+b)` by deducing that in turn that can be re-written as `*(b+a)` and hence `b[a]`. This can lead to odd-looking code like `3["magic"]` that should normally be avoided.

The macro `NULL` that is defined in various of the standard headers represents a value that will never arise as a "proper" pointer and can be used to mark the ends of linked lists etc.

Conditional statements in C++ are written as

```
if (expression)
    statement1
else
    statement2
```

where the parentheses around the expression to be tested are essential. The `else` clause is optional.

Iteration is expressed either as

```
while (expression)
    statement
```

or as

```
for (initialiser; end-condition; step-on)
    statement
```

where it seems easiest to explain the three components in the header of a `for` loop by giving an example:

```
char *s = "Some String";
char buffer[100];
for (int i=0; *s!=0 && i<100; i++)
    buffer[i] = *s++;
```

This example code copies the string pointed at by `s` into the array `buffer`, stopping either when a zero "character" is found (this is how C++marks the end of a string) or when 100 characters have been copied. Note that in C++ it is possible to declare the `int` variable `i` as shown, while in plain C it would be necessary to declare `i` before the `for` loop.

Within the body of a loop construct `break` can be used to exit from the loop, and `continue` to go on at once to the next iteration.

C++ naturally has a `goto` statement, and labels are set by following their names with a colon. See Figure 2 for a program that illustrates how, in some circumstances, the careful use of `goto` statements can enhance the legibility of C code.

The other interesting control structure featured in C++ is called `switch`, and in one statement it can dispatch to a large number of places, selecting which on the basis of the value of some integer. In straight-forward use the syntax used is illustrated by

```
    char *m;
    int i = some_random_function();
    switch (i)
    {
default: m = "Unknown Number"; break;
case 2:  m = "The only even prime"; break;
case 10: m = "What I get when I count my fingers"; break;
case 1729:
         m = "smallest sum of 2 cubes in 2 different ways";
         break;
case 'A':m = "character code for 'A' on this computer";
         break;
case 0:  m = "don't be silly"; break;
    }
```

The `switch` construct finds many uses for dispatching on the basis of integer code values (eg in an emulator for some real computer one might well `switch` on the opcode field from the next instruction to be simulated) or character values.

C++ function definitions are written as in the following example

```
/* Program by Spencer Hines, Online Computer Systems */
#include <stdio.h>
#include <malloc.h>
main(togo,toog)
int togo;
char *toog[];
{char *ogto,   tgoo[80];FILE  *ogot;  int    oogt=0, ootg,  otog=79,
ottg=1;if (    togo==  ottg)   goto   gogo;  goto    goog;  ggot:
if (   fgets(  tgoo,   otog,   ogot)) goto   gtgo;   goto   gott;
gtot:  exit(); ogtg: ++oogt;   goto   ogoo;  togg:   if (   ootg > 0)
goto   oggt;   goto    ggot;   ogog:  if (  !ogot)   goto   gogo;
goto   ggto;   gtto:   printf( "%d    goto   \'s\n", oogt); goto
gtot;  oggt:   if (   !memcmp( ogto, "goto", 4))     goto   otgg;
goto   gooo;   gogo:   exit(   ottg); tggo:  ootg=   strlen(tgoo);
goto   tgog;   oogo: --ootg;   goto   togg;  gooo: ++ogto;  goto
oogo;  gott:   fclose( ogot);  goto   gtto;  otgg:   ogto=  ogto +3;
goto   ogtg;   tgog:   ootg-=4;goto   togg;  gtgo:   ogto=  tgoo;
goto   tggo;   ogoo:   ootg-=3;goto   gooo;  goog:   ogot=  fopen(
toog[  ottg],  "r");   goto    ogog;  ggto:  ogto=   tgoo;  goto
ggot;}
```

Figure 2: A C program to illustrate the use of “`goto`”.

```
int fib(int n)
{
    if (n < 2) return 1;
    else return fib(n-1) + fib(n-2);
}
```

where the header line defines the type of both arguments and result from the function, and `return` is used to exit from the function with a result. As a special case a function can be defined to return the type `void`, which indicates that no value at all will be returned. In the definitions of such functions `return` is used without a following expression. In old K&R C the above example would have to be written slightly differently:

```
int fib(n)
    int n;
{
    if (n < 2) return 1;
    else return fib(n-1) + fib(n-2);
}
```

with the type of the argument specified below the header line rather than as part of it. This old style needs to be recognised so that you can make sense of existing code, but should not be used when new code is being written.

It should be observed that C++ assignments and function calls can be used as statements, terminated by a semicolon. A null statement can be made by writing a semicolon after nothing at all. Collections of small statemnents can be grouped by wrapping them up in a pair of braces (`{ s1 ; s2 ; s3 }`). Similarly if an expression rather than a statement is wanted, several expressions can be combined so that they get evaluated one after the other (and only the last value preserved) by concatenating them with a comma (“`,`”) as connective. For instance `*++x` might alternatively have been expressed as `(x+=1, *x)`.

6 Library Functions that Everybody Needs

C++ can be used as a language for programming raw hardware where the purpose of the code is to activate various input and output hardware. If I/O ports happen to be memory mapped (as is the case with some styles of microprocessor) the usual C++ operators can make it possible to use them. For instance on an IBM PC there may be video memory at addresses around 0xc0000, so with at least some compiler and configuration of a machine code[13] such as

```
for (char *p = 0xc0000; p<0xc1000; p++) *p = ~*p;
```

[13]The use of integer constants with pointer variables as shown here is pretty dubious, and at a minimum a proper piece of C++ code will need to contain extra decorations (called *casts*) to reassure the compiler that the programmer really intended that.

might have some effect on what was displayed.

More normally all input and output for C++ will be done by calling library functions and using and types, operators and variables defined in system-provided header files. To use these facilities it is necessary to know which system files declare the functions you want to use. The facilities listed in this section represent a tiny selection of some of the ones most needed while getting started: browse a textbook to find out what else is available.

For C the ANSI standard provides a clear definition of a core of functions that can be relied upon. For C++ the situation is much less stable. At present it seems reasonable to expect that the ANSI C functions will always be available, and that C++ adds a new set of capabilities known as `iostream`. But as the C++ standardisation process continues things are very likely to change.

All real C++ systems will provide a significant collection of extra library functions that can be called. Under Unix, for instance, there will be all of the Unix system calls, and probably the entire interface to X-Windows. On an IBM PC there will be functions that give access to low-level MSDOS features, plus all of the Microsoft Windows interface. Similarly on other systems. This course views such extra libraries as to a large extent outside its scope, but serious C++ users will eventually need to come to terms with at least one (and probably several) of them.

For C++ simple input and output can be arranged using the `iostream` library. This provides pre-declared variables `cin` and `cout` that are used to refer to the default input and output streams (often connected to the keyboard and screen of your computer), and operations that can be performed on these variables to cause text to be read or printed. Really simple functionality is supported by the use of operators `<<` and `>>`[14] applied to `cin` and `cout`. The following program illustrates this — together with a few other new features of C++.

```
#include <iostream.h>

int main()    // not really long enough to need comments?
{   int a, b;
    cout << "Please type in two numbers\n";
    cin >> a; // read in integer a, then b.
    cin >> b;
    cout << "The sum is " << (a+b) << endl;
    return 0; // exit back to Unix or DOS or whatever
}
```

The line starting `#include` directs C++ to process the standard header file named there. Without that header file the variables `cin` and `cout` and all operations on them would be undefined.

The program defines a function called `main`. A convention with C++ is that a program is started by calling a function with that name, so every complete program

[14]Recall that on integer types these called for shifts left and right of bit-patterns.

is expected to define `main`. The `main` function returns an integer code that may be available to the command-line decoder that launched the program. A value of zero usually indicated success[15]. The uses of `>>` and `<<` are influenced not only by the presence of `cin` or `cout` as one operand, but by the type of the other operand, so for instance printing uses a reasonable default format with integers printed in decimal and strings displayed in the obvious way. The printing the predefined object `endl` causes a newline to appear. The usage `cout << endl` is slightly different from `cout << "\n"` in that in addition to putting out a newline character it forces any pending output to appear on the user's screen or in the relevant output file. This effect becomes important if the printing was to collect a record of what went on prior to a collapse — if `"\n"` were used in place on `endl` output can remain buffered and may be lost when the code fails.

The example also shows that uses of `<<` for output can be chained together. The reason this is possible is that the expression

```
cout << a << b << c;
```

will parse as

```
((cout << a) << b) << c;
```

and it is arranged that `(cout << x)` just returns `cout` after printing `x`.

On occasions it is more useful to read and print things one character at a time. The `iostream` way of achieving this is illustrated in:

```
#include <iostream.h>

int main()
{   char c;
    while (cin.get(c))
        cout.put(c);
    return 0;
}
```

where the "`.`" operator applied to `cin` and `cout` selects an associated operation, and `get` and tt put are single-character input and output functions. `cin.get` returns a true value until it reaches the end of the input stream.

If instead of having a variable that is directly of a class type you have one that contains a pointer to an object then the selector "`->`" should be used instead of "`.`". In general `a->b` is equivalent to `(*a).b`.

As one might expect there are extra facilities for setting input and output into octal or hexadecimal modes, controlling the widths of values processed and directing data to places other than the standard streams — but the facilities listed thus far are sufficient for now so far as C++ is concerned!

[15] See `EXIT_SUCCESS` and `EXIT_FAILURE` in your C manual.

Until C++ has totally displaced C[16] it is necessary to know at least the bare bones of C-style input-output too. This centres around a package known as `stdio` and the standard streams are known as `stdin` and `stdout`. Character-at-a-time input is done using `getchar()` which normally returns a character, but which hands back the special value `EOF`[17] at end of file. Correspondingly `putchar` is a function of one argument that sends one character to `stdout`.

```
#include <stdio.h>

int main()
{   int c;   // use integer variable here so that
             // it can hold the non-character EOF
             // when needed.
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

More elaborate printing is done using `printf`, which has as its first argument a *format string*. This is printed, except where it contains `%` characters. These cause one of the subsequent arguments to `printf` to be printed in some specified format. The latter "`d`" causes `printf` to print a (decimal) integer and "`s`" a string: there are very many other options.

```
#include <stdio.h>

int main()
{   int a = 100;
    char *s = "string";
    printf("An integer %d and a string %s\n", i, s);
    return 0;
}
```

With both C and C++ schemes there are full sets of facilities to open and close files and control format, and to do transfers on blocks of characters as well as single characters[18]. There will also be schemes that allow variants on the usual reading and printing operations to work on arrays of characters held in memory rather than on text kept in files or accessed directly from the keyboard.

It is not going to be a good idea to try to mix C and C++ style input and output in the same program!

---

[16] Perhaps it never will.

[17] `EOF` often has the value -1, but it is defined in the standard header and you should nor rely on its exact value.

[18] At least on some systems it can be **much** more efficient to read and write large files in blocks of several thousand characters at a time

As well as supporting input and output, standard libraries look after memory allocation and string manipulation. Memory allocation is another area where C++ differs from C. The C++ code

```
{   int vector[10];
    for (int i=0; i<10; i++) vector[i] = i;
    ...
}
```

declares a vector of length 10, initialises it and presumably uses it within the block (delimited by `{}`) shown. With this form the vector only exists for as long as your program is executing code within the block, and the size of the array must be a constant. These limitations can be relaxed but at a cost (the required memory may not be available) by allocating the vector dynamically. The operator `new` allocates memory, and `free` can be used to release it. Inappropriate use of `free` can be the source of amazingly obscure and hard-to-track bugs.

```
int n;         // size of array
cin >> n;      // read in size as variable value
int *vector = new int[n];      // Try to allocate...
if (vector == 0) goto failure; // ... Oh Dear?
for (int i=0; i<n; i++) vector[i] = i;
...
delete vector; // when vector is finished with
```

In C instead of using operators `new` and `free` there are library functions `malloc` and `free` that serve the same purpose.

Typical string operations available in the `string` library are:

`strcpy(char *dest, char *src):` Copies characters from the source string to the destination.

`strcmp(char *a, char *b):` returns an integer that is negative, zero or positive depending on whether string `a` comes before, is equal to or comes after `b` in lexical order.

`strlen(char *s):` Returns the length of the given string. Note that a string such as `"abcd"` has length 4, but in memory it will be stored with an extra byte at its end — this byte contains zero and is used to mark the end of the string.

Spot the odd behaviour of

```
strlen("A string\0with a zero character in it")
```

which returns 8, treating the string as ending at the `'\0'` character.

After `#include <ctype.h>` a range of character classification functions become available. `isalpha`, `islower`, `isupper`, `isdigit` and `isspace` detect

letters, lower and upper case letters, digits and white-space. `toupper` and `tolower` force the case of letters.

This sampling of library facilities should at least give a start to your programming, but checking reference manuals to find complete lists of functions and options will eventually be necessary.

7 Header Files and Separate Compilation

Although C++ is a perfectly good language for writing small programs, it can (and very often has been) used for very substantial projects. These will involve keeping source code in a collection of files rather than in just one file, and compiling each of these files separately. The natural problem that arises is that of ensuring that code in all the separate source files is kept consistent: for instance that if a function is defined in one file then calls to it from another pass it the correct number and types of argument. Languages such as Modula-3 have rigidly designed in mechanisms for solving this problem, while in C and C++ more is left up to the programmer.

The main method used to keep things in step is the use of *header files*. These will be generally be files that contain just declarations that specify the types of variables and functions. They can also contain the definitions of user-defined data types or pretty well anything else. The C++ directive `#include` (which should be written at the start of a line) in a main program file causes the compiler to scan the header file there, thus bringing the declarations into effect. It is then able to confirm that both definitions and uses of functions and variables are compatible with the declarations. Note of course that **local** variables should not be mentioned in header files — it is just ones defined at the top level in your file that can even possibly be visible from another file.

`#include` statements may name one of the system-provided header files by writing the header identification in angle brackets, as in the examples shown so far, or can specify an arbitrary user-provided file by writing its name in double quotes.

Declarations of variables in C++ for inclusion in header files look very much like the ones we have seen already, save that initialisations of the variables are not permitted. Declarations of functions look just like the header line from the function definition, but terminated with a semicolon. In each case you should usually put the word `extern` in front of the declaration. This keyword tells C++ that the value being declared will be referenced from several different files so the compiler must ensure that its name is made globally available in compiled (object) files. Figure 3[19] shows all the files you might want for a somewhat minimal multi-file project. It is worth setting up your code in this way while your files are still small, so that by the time they get large you are thoroughly used to the mechanisms involved. Unless **all** the things that you define in one file and use in another are declared in header files the C++ compiler will have no early warning if your usage is inconsistent. But still many C++ systems

[19] Note that the multiple blanks shown in the `Makefile` are tab characters: Unix is fussy about this.

```
-----  File proj.h  -----
// Sample header file
extern long int count;  // declare variable
extern long int ack(long int m, long int n);

----- File proj1.cc -----
#include "proj.h"

long int ack(long int m, long int n)
{   // well known function to waste time!
    count++;    // count number of times called
    if (m == 0) return n+1;
    else if (n == 0) return ack(m-1, 1);
    else return ack(m-1, ack(m, n-1));
}

----- File proj2.cc -----
#include <iostream.h>  // standard header
#include "proj.h"      // private header

long int count = 0;    // define variable

int main()
{
    cout << ack(4,2) << endl;
    return 0;
}

----- File Makefile -----
# Small Unix "Makefile" for 2-source-file program

proj:           proj1.o proj2.o
        g++ -o proj proj1.o proj2.o

proj1.o:        proj1.c proj.h
        g++ -c proj1.c

proj2.o:        proj2.c proj.h
        g++ -c proj2.c

# end of Makefile
```

Figure 3: A C++ program in several files.

(unlike C) provide a measure of protection[20] that can result in an "undefined function" error message when you try try to link together the object files from inconsistent sources. This is typically achieved by arranging that when you define a function such as `ack` in Figure 3 the name C++ uses internally for the function includes the name `ack` that you used, but adds extra characters that encode information about the number and type of arguments expected. This process is known as "name mangling". So if you see diagnostics that refer to long long names that are built from the names you used plus a load of gibberish that is probably what is happening.

There are extra complications here if you want to mix C and C++ code (the usual reason you need to do this is when there is some existing library of C code to be linked to). If the declarations in the header file are written simply they all refer to functions compiled using the C++ conventions. An extra annotation

```
extern "C" {
extern void somefunction(int arg1, char *arg2);
}
```

is needed to direct a C++ compiler to expect or generate object code using conventions compatible with C. With C linkage you should not expect as good cross-file type-checking as you get in regular C++ mode, and funtion names should not be overloaded.

Good C and C++ compilers should provide an option that will print a warning message if you define anything that has not been declared earlier. This idea of such a warning is that you discipline yourself to keep declarations just in header files and definitions in source files, and the compiler message can allow you to check that everything that could possibly need to be in the header file is in fact there.

C++ does nothing to prevent you from putting function definitions or indeed absolutely anything into header files. It works as if before the compiler started looking at your program in any detail it scanned it and textually replaced each line starting with `#include` with the contents of the named file. In very nearly all circumstances you should `only` put declarations in header files and **never** use `#include` to pull in chunks of random source code.

8 Basic Features of the Pre-Processor

The `#include` facility was described as having the effect of transforming the user's source code before the compiler proper got to look at it. With C++ there are a number of other operations that are implemented as if they are textual rearrangements of the source done very early in the compilation process. In early C compilers these transformations were even performed by a separate program, known as the C pre-processor. Nowadays the pre-processor is usually implemented as part of the full compiler, but its behaviour is still kept somewhat separate. Preprocessor directives live on lines of their own, and start with the # character. Apart from `#include` the most important

[20] Referred to as "type-safe linkage.

ones relate to conditional compilation. This is a facility that makes it possible to have one source file where some of the details in it depend on (say) the computer it will be run on. The C++ compiler for each different computer will establish, for the benefit of the pre-processor, some predefined symbol that can then be tested so that the desired fragment of code can be included. For instance `gpp` under MSDOS defines (at pre-processor time) a flag called `__MSDOS__`, so if you want code that produces a customised banner you might write your source file as

```
#include <iostream.h>

int main()
{
    cout <<
#ifdef __MSDOS__
            "The compiler though \"MSDOS\"\n";
#else
            "Possibly not MSDOS?\n";
#endif
    return 0;
}
```

and only one of the strings will be passed to the `<<` operator and printed. Observe that C++ conditional compilation is a transformation that is utterly unaware of the boundaries between C++ statements and other syntax rules, because it is done as simple adjustment to the text in the file.

Apart from symbols that are pre-defined by the C++ compiler that you use it is possible to parameterise your code on that basis of other symbols, and then cause them to be defined (or not) by giving command-line options to the compiler. For example if the above example was stored in a file `"Am-I-DOS.cc"` on Unix it could still compiled using the command

```
g++ -D__DOS__ Am-I-DOS.cc
```

and the `-D` on the command line causes the specified symbol to become defined. One use I make of this capability is with a file compression utility I have. The source code is in one file, and contains both the code for the compression case and the decompression (which I want to end up as separate programs). The two algorithms share quite a lot of code, and I use `#ifdef EXTRACT` ... `#else` ... `#endif` to separate those (smaller) chunks of code that make the whole program into either a compressor or an extractor. The entry in the `Makefile` I use with this file contains essentially the two lines:

```
g++ compression.cc -o squash
g++ -DEXTRACT compression.cc -o unsquash
```

which uses the same master source file but builds two quite distinct executables.

The remaining pre-processor features I will describe are **much** less important in C++ than they were in plain C. However I have not yet described for you the C++

constructions that render them (almost) obsolete. So what I will do here is document whet the pre-processor can do, then indicate how the addition of simple extra keywords to ways of using C++ that you already know can achieve similar effects in what is often a safer way.

The symbols tested with `#ifdef` are referred to as *macros*, and except after the word `#ifdef` any use of them will get expanded to some replacement text. The symbols `__MSDOS__` and `EXTRACT` seen so far did not have useful values to expect to — all that was of interest was whether they could be considered to be "defined". However a pre-processor directive named `#define` makes it possible to set up macros that do have useful expansions. These macros can either be simple words (that can expand into arbitrary text), or can take arguments. Here are some examples

```
#define MAXIMUM_LINE_LENGTH 128        // A parameter
char line_buffer[MAXIMUM_LINE_LENGTH]; // use it.

#define FIRST_CHAR line_buffer[0]
#define LAST_CHAR  line_buffer[MAXIMUM_LINE_LENGTH-1]

#define number_is_even_(n) \
    (((n) % 2) == 0)  // macros may extend over several
                      // lines by use of a trailing "\".
```

It is often considered sensible to use some clear lexical convention so that anybody reading your code can tell when a name you use is actually a macro. C++ does not enforce this (of course!). The convention used in the example is to spell simple macros in upper case, and to have "_" as the final character of those macros that accept arguments. The use of macros can very greatly clarify programs, see for instance Figure 4 which may help you with your Morse Code. Most sensible large C programs will use macros (probably defined in header files) to establish useful parameters such as the size of buffers that are to be allocated, and to introduce macros with parameters that provide clean abstract access to data structures. Consider the merits of

```
#define RADIX_FOR_NUMERIC_INPUT  10
#define RADIX_FOR_NUMERIC_OUTPUT 10
#define ASCII_CODE_FOR_LINEFEED  10
#define NUMBER_OF_COMMANDMENTS   10
char *commandment[NUMBER_OF_COMMANDMENTS];
```

and the way that it could lead to code that would be much easier to maintain than a corresponding version that just had the literal text "`10`" scattered through it.

In C++ macros are **much** less important than they are in C. This is because most of the benefits of a simple macro can be achieved by putting the word `const` in an initialised declaration, and in most cases macros with arguments can be replaced by ordinary C++ functions that have been decorated with the `inline` directive:

```
const double VAT_rate = (17.5/100.0);
```

```
/* Program by Jim Hague, University of Kent, Canterbury */
#define DIT     (
#define DAH     )
#define __DAH   ++
#define DITDAH  *
#define DAHDIT  for
#define DIT_DAH malloc
#define DAH_DIT gets
#define _DAHDIT char
_DAHDIT _DAH_[]=
      "ETIANMSURWDKGOHVFaLaPJBXCYZQb54a3d2f16g7c8a90l?e'b.s;i,d:"
;main                   DIT                     DAH{_DAHDIT
DITDAH                  _DIT,DITDAH             DAH_,DITDAH DIT_,
DITDAH                  _DIT_,DITDAH            DIT_DAH DIT
DAH,DITDAH              DAH_DIT DIT             DAH;DAHDIT
DIT _DIT=DIT_DAH        DIT 81                  DAH,DIT_=_DIT
__DAH;_DIT==DAH_DIT     DIT _DIT                DAH;__DIT
DIT'\n'DAH DAH          DAHDIT DIT              DAH_=_DIT;DITDAH
DAH_;__DIT              DIT                     DITDAH
_DIT_?_DAH DIT          DITDAH                  DIT_ DAH:'?'DAH,__DIT
DIT' 'DAH,DAH_ __DAH    DAH DAHDIT              DIT
DITDAH                  DIT_=2,_DIT_=_DAH_;     DITDAH _DIT_&&DIT
DITDAH _DIT_!=DIT       DITDAH DAH_>='a'?       DITDAH
DAH_&223:DITDAH         DAH_ DAH DAH;           DIT
DITDAH                  DIT_ DAH __DAH,_DIT_    __DAH DAH
DITDAH DIT_+=           DIT DITDAH _DIT_>='a'?  DITDAH _DIT_-'a':0
DAH;}_DAH DIT DIT_      DAH{                    __DIT DIT
DIT_>3?_DAH             DIT                 DIT_>>1 DAH:'\0'DAH;return
DIT_&1?'-':'.';}__DIT   DIT                     DIT_ DAH _DAHDIT
DIT_;{DIT void DAH write DIT                    1,&DIT_,1 DAH;}
```

Figure 4: Use of `#define` to improve code style.

```
inline int abs(int x)
{
    if (x < 0) return -1; else return x;
}
```

The `const` decoration tells the compiler that the value set up must never and will never change, while the `inline` annotation advises the compiler that it may well be worth merging the body of the given function into any place where the function is called from, thereby avoiding any function-call overhead. In general for C++ code these ways of expressing things are much preferred over the use of macros, since they ensure that the syntactic structure of code is preserved and they allow C++ to keep do better type-checking and error reporting than do macros. They also avoid some of the funny things than can occur with macros and side-effects and bracketing, as in

```
#define square(x) (x*x)
int x = 1;
cout << square(x++) << endl; // (x++*x++) so x gets
                             // incremented twice!
cout << square(x+x) << endl; // (x+x*x+x) which is
                             // NOT (x+x) squared!
```

For cases where `inline` functions do not provide enough generality (for instance the `square` macro above could have been tried on either real or integer arguments, while a regular C++ function could not have coped with both cases all at once) there are things called “templates”, which will be mentioned later on.

A reasonable person would expect that a formal definition and complete understanding of the pre-processor would not raise any big problems. Unexpectedly it does, especially with regard to the treatment of macros that expand to other macros and pairs of adjacent symbols that expand to text that seems to mean something special. These notes will not discuss what happens in such cases — they just warn that people who want to make truly elaborate use of the pre-processor or who intend to play tricks with it need to read the standards documents very carefully indeed.

```
// The following examples are intended to raise
// doubts in your mind about just how the preprocessor
// will work.
#define startcomment     /* some comment text
#define endcomment       */
#define divide_operator  /
a = b /divide_operator  divide_operator* ??? endcomment;
#define mytest(x)  (x == 0 || mytest(x))
if (mytest(0)) cout << "zero"; // valid ??
#define int #define
int x = 3; // wow, what does this now mean, if anything?
```

9 A more-or-less sensible example

Figure 5 gives a first fairly realistic application of C++. It computes checksums of files which might be useful if you want to verify that a file has been transferred safely from one machine to another — checksum at either end and see if the numbers match. It introduces a few C++ features that have not been mentioned before but which, in context, should not cause great difficulty:

Arguments for `main`**:** `argc` will give a count of how many words were written on the command-line when the program was called, and `argv` is an array of strings, with each string being one of these words. `argv[0]` will be the first item on the line, ie the name of the program being run. Hence `argv[1]` is the first word after that.

`ifstream`**:** The line starting "`ifstream`" is the declaration of a variables called `infile` of type `ifstream`. This type is defined in the standard C++ header files, and when a variable of that types is declared some extra arguments can be given (as shown). The effect is that `infile` ends up as a C++ stream (supporting the same operations as does `cin`) attached to the given file.

`infile.fail()`**:** This just checks that the file that was wanted could indeed be opened successfully.

`cerr`**:** Very similar to `cout`, `cerr` is (by convention) used as a place to send error messages.

Use of `&` **in function header:** Normally arguments to C++ functions are passed by value, so that whatever happens inside the function it does not alter the value written where the function is called. With a `&` used as shown the function should be called with an updatable object (typically a variable) as its argument, and side-effects are possible.

`hex`**:** Sending the predefined value `hex` to an output stream directs it to display subsequent integers in hexadecimal. There are of course plenty of other format-control directives available in the `iostream` library.

Limited portability: The code shown performs arithmetic and bitwise operations on the codes for characters read from a file. A result will be that the checksum computed will be sensitive to the character code used on your computer. The code also requires that the type `unsigned long int` should be capable of handling values as big as `0x7fffffff` (ie in realistic terms 32 bits). In these two respects at least the code is not guaranteed portable, although the use of `long int` rather than just `int` improves its chances. Would the code behave identically on machines with 32 and 64 bit `unsigned long int`s? If not can you fix it?

```
// Utility to compute a checksum for a file.  A C Norman, 1994

//   Usage:
//     checksum <filename>
//   prints a checksum of the file contents

#include <iostream.h>
#include <fstream.h>

// This is my favourite hash function at present.
// It cycles a 31-bit shift register with maximum period.
void update_hash(unsigned long int &hash, int ch)
{   // WARNING this code expects long ints to be 32 bits
    unsigned long int hashtemp = (hash << 7);
    hash = ((hash >> 25) ^     // remember ^ is
            (hashtemp >> 1) ^  //   exclusive OR
            (hashtemp >> 4) ^
            ch) & 0x7fffffff;  // mask to 31 bits
}

int main(int argc, char *argv[])
{   // expect use to be "checksum filename", ie 2 words
    if (argc != 2)   // argc, argv give command-line args.
    {   cerr << "This utility requires one argument" << endl;
        return 1;
    }
    ifstream infile(argv[1], ios::in); // open input file
    if (infile.fail())                 // did it exist?
    {   cerr << "Unable to open the file " << argv[1] << endl;
        return 1;
    }
    unsigned long int hash = 1;
    int ch;
    while ((ch = infile.get()) != EOF)
        update_hash(hash, ch);         // compute checksum
    cout << argv[1] << ": " << hex << hash << endl;
    return 0;
}

// End of checksum.cc
```

Figure 5: A C++ program to checksum files.

Perhaps the main lesson from this example is than as you start to write real examples of C++ code there will be a lot of fine details of library calls and minor language features that it will be necessary to come to grips with. This lecture course and especially these notes will not even attempt to provide full coverage, but will instead try to give enough ideas that you will be equipped to dive into reference material.

It also shows that some of the sorts of program that C++ is naturally used for can become non-portable remarkable easily — in most programming tasks the issue is one of striking a balance between writing concise and natural code and allowing for improbable oddities in the machine on which the code is to be run. For the last five years it has been important but sometimes hard to ensure that code would run on both 16 and 32-bit machines. For the next few years life will be yet harder in that the legacy of 16-bit systems will remain with us, while the migration from 32 to 64-bits for main-stream workstations will continue.

10 Pointers, Structures, Unions and Classes

So far the features of C++ discussed will only allow you to write rather simple programs, using arrays and a few pointers. the next set of language constructs to be covered involve the creation of new user-defined data types. Since this course is concentrating on C++ and viewing C as (roughly) a subset and (certainly) a historical predecessor, and given that courses on Modula 3 have already introduced the ideas of objects, classes and inheritance, it seems reasonable to jump in at the deep end.

In C++ a *class* will define a data type that has a number of component fields, and a number of operations that may be performed upon the data. The components of a class may either be *public* in which case any arbitrary piece of code can access them, or *private* in which case they can only be used within functions that are themselves members of the given class. One of the main reasons for introducing classes is so that control can be exerted over the visibility of code and data: a class should only make public components that give it a clean and easily documentable interface, and all internal implementation details should be kept protected by making them private.

An example of a class that has been seen already is the type `ifstream`, which has public members `fail`, `get` and for the operator `>>` (and for a collection of other things), but which can prevent the ordinary user from abusing its internal state by making all that private.

Classes will often define a public member function that is to be called when an instance of the class is created, and can define another one that will be called (automatically) to tidy up when an object is discarded.

My first example of a class will not be complete and may not be especially exciting, but should show how things are expressed. It provides for the support of complex numbers. But just to be awkward even though the interface that it supports views numbers in a Cartesian representation $x + iy$ the values are stored in the structure in polar form ($re^{i\theta}$). The declaration of a class can be kept separate from the definition

of the functions that form part of it, so here is the definition:

```
#include <math.h>  // For sin(), cos().

class Complex
{
public:
    Complex(double x = 1.0, double y = 0.0);
    double RealPart();
    double ImagPart();
private:
    double r;
    double theta;
};
```

The public function called `Complex` has the same name as the class being defined, and is thus marked out as being used when creating instances of the class. A further new C++ feature that is shown here is the provision of initialised declarations in a function header. This gives default values for the arguments. A function that has default values specified in this way can be called with fewer than the full number of arguments and the compiler will fill in the blanks. This can be done for any C++ function at all, not just ones used to construct instances of class objects, but it is perhaps particularly useful in this case.

The `RealPart` and `ImagPart` public member functions will just extract information, while the private variables `r` and `theta` will store it. This example happens to have all its public members functions and all its private ones data, but this is not necessary and will in general not be the case: functions and data can each be declared in either part of a class description.

Next the member functions need to be defined. I will also include a minimal main program that creates a single complex number and then extracts its real and imaginary parts. Syntax using `::` is used to show when names being used are members of some class. Within the body of a member function the compiler knows that names are liable to refer to class members, and so it is not necessary to write `Complex::r` and `Complex::theta` in the following definitions, although it would not be incorrect (but it would be ugly, so should be avoided unless there was some very special reason for stressing that some particular reference was to a member variable). The main program illustrates two additional things. When a variable is declared and its type is a class the class constructor will be called, and it may need arguments. These are written as shown in the declaration. Member functions from a class can generally only be called if you have an object of the class type somewhere[21], and the functions are referred to by using a dot ("`.`") selector on that object. Note that this is the same use of a dot that arose when the `get` member function of an `ifstream` was used.

```
Complex::Complex(double x, double y)
```

[21] This is because the member functions will need to access data from the object concerned.

```
    {
        r = sqrt(x*x + y*y);
    // The next line is inadequate if x <= 0.
        theta = atan(y/x);
    }

    double Complex::RealPart()
    {
        return r*cos(theta);
    }

    double Complex::ImagPart()
    {
        return r*sin(theta);
    }

    int main()
    {
#ifdef ONE_WAY  // Show two valid alternatives here
        Complex z(2.0, 1.0);  // Create z as a complex
#else
        Complex z; // use default args for constructor
        z.Complex(2.0, 1.0); // now fill in true values
#endif
        cout << z.RealPart() << " + "
             << z.ImagPart() << "*i" << endl;
        return 0;
    }
```

On important restricted case of classes is when the class defined contains only data items (and not any member functions) and when all the data is public[22]. This case is just the definition of a simple data structure, and can be achieved using the keyword `struct` instead of `class`. Programs in C can only use `struct`: `class` is one of the ways in which C++ has extended the language. With C++ and `class` the class-name becomes usable as the name of a new type, eg `Complex` in the example. With C this is not so and the most convenient practise is to use a construct `typedef` to give a convenient new name for the type that is represented as a `struct`.

```
    /* This code is valid in C as well as C++ */

    typedef struct tree_tag
    {       /* To be used for binary trees of integers */
        int value;                    /* value in node */
        struct tree_tag *left;  /* pointer to sub-tree */
```

[22]If any were private it would be pretty useless, since only member functions can refer to private items.

```
    struct tree_tag *right; /* pointer to sub-tree */
} tree;
```

Here the word used after `struct` is a *structure tag*, and the new type can always be referred to by using the word `struct` followed by this tag. Inside the definition of the structure where it is necessary to declare pointers to the newly defined structure this notation is used. However the `typedef` introduces a new type-name (in this case just `tree`) that can be used in all subsequent declarations. If the word `union` is used instead of `struct` all the members of the union overlap in memory, so it may be possible to write data to the object via one path and read it out using another!

```
typedef union floating_cheat
{   double d;
    char i[8];
} floating_cheat;

int one_byte_from(double v)
{   floating_cheat w;
    w.d = v;
    return w.i[3];  /* Wow! */
}
```

As an aside, `typedef` has good uses quite separate from those associated with structures, and can be used to provide new names for existing types, as in

```
#ifdef SIXTEEN_BIT
typedef long int int32;
#else
#ifdef SIXTY_FOUR_BIT
typedef short int int32;
#else
typedef int int32;
#endif
#endif
```

which might well be used in a header file for a program which needed to use an integral data type that it could rely on being 32-bits wide. By arranging to predefine one of the macros as necessary the type `int32` could refer to whichever built in data type was most suitable.

Back to C++ classes! Once one class has been defined others can be created as derivatives. Derived classes carry forward the properties of their parent, but can add new data fields and either add or alter the definitions of member functions. The header to use when declaring a derived class looks something like:

```
class variant_on_Complex : public Complex
{   ...
```

where the "`:`" is followed by a description of what is to be inherited. Constructor functions are not inherited, so a derived class can always be expected to define one, but otherwise it only needs to define things that are new.

When you have a base class and a derived class any object that is created as a member of the derived class can also be treated as belonging to the base class. Thus it is perfectly possible (and indeed very common) to declare a variable whose type indicates that it contains a pointer to an object in the base class and use it to point at all sorts of derived objects. Normally if your code then performs operations on the things that are pointed at in this way the functions called will be those associated with the base class even if the actual object you are referring to is in some class that attempted to replace everything. If, however, you declare a function in the base class to be `virtual` then C++ takes extra care[23] so that calls to that member dispatch on the basis of the exact position of the object involved in the class hierarchy. In cases where you will build data structures that contain pointers to many closely related sorts of objects it can make very good sense to organise all the variants of your data as sub-classes of some generic class, and define many of the functions in this parent class as virtual.

As an example of how classes and class libraries can both help make what would otherwise be lengthy and complicated code almost manageable, and also how it has a big effect on how code is structured, see Figure 6 which is the complete C++ source for a program that opens and displays a window under Microsoft Windows. The code can be built and tested with the MFC libraries if you have Microsoft Visual C++, and using the 32-bit edition of same the commands that compile the code are

```
cl -c mini.cpp
link -subsystem:windows -entry:WinMainCRTStartup \
     mini.obj nafxcw.lib kernel32.lib gdi32.lib  \
     advapi32.lib shell32.lib comdlg32.lib
```

The development of code like this into complete and interesting applications is covered in [1]. You are not expected to follow all the details of Figure WinMini now, but should be able to appreciate that it is a fairly short body of code (and for many window systems even minimal programs are often painfully long), and that it is based around deriving new classes from existing library ones and over-riding some of the definitions of member functions. In this case two classes are adjusted — one is the basis for applications (ie programs), while the other supports windows on the screen. To a reasonable approximation adding extra functionality to the code is "just" a matter of over-riding more members of the two classes with application specific code, and causing several windows to appear involves little more than declaring several variables of type `CamWind!`.

[23] Ie it imposes a little extra overhead on your code.

```
// A minimal program to use Microsoft Windows
// using the Microsoft Foundation Classes C++ library.
// This is essentially a standard demo pgm for MFC

#include <afxwin.h>   // MFC header file

////////////////////////////////////////////////

class CamWind : public CFrameWnd
{   // derive CamWind from MFC'c CFrameWnd
public:
    CamWind();  // constructor function needed
};

CamWind::CamWind()
{   // when constructing a window call library
    // function to do all the hard work
    Create(NULL, "Demo");
}

////////////////////////////////////////////////

class CamApp : public CWinApp
{   // derive CamApp from MFC's CWinApp
public:  // and override a critical member function
    BOOL InitInstance();
};

BOOL CamApp::InitInstance()
{   // This is called when your code starts up
    // It create and displays a window.
    m_pMainWnd = new CamWind();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    return TRUE;
}

////////////////////////////////////////////////
// declaring a variable of type CamApp will cause
// its constructor function to be called, and this
// rather than "main" is used to start things off!
CamApp xxx;

////////////////////// END ////////////////////////
```

Figure 6: The start of a MS-Windows project

11 When to use Classes

Perhaps I should first review some of the differences between C++ classes and the `struct`s that ordinary C provides. With structures about all you are doing is declaring that several items will be collected together and treated as a single data object. The `struct` definition is little more than a template for how the data should be laid out in memory — although you ought not to rely on the C compiler actually places individual members in a structure where you would have at first expected. C++ classes do a lot more for you. They can have some of their members private and others public, so giving controlled access to their internal state. As well as containing data they can declare functions to work on that data and hence (via the access control arrangements) provide a major way of organising code. New classes can be derived from existing ones, so the complete set of classes in a program will normally fall into a number of families. This can be a very great help when a number of related data types need to be implemented. Finally the C++ class structure can fit in with operator overloading, so at least some that user-defined operations on the class can be invoked using symbols such as `+` and `-`.

One extreme use of C++ classes would correspond to when a C programmer might have used a `union` so that one field in a data structure could have held many different sorts of item (use of `void *` pointers could probably have achieved similar effects). A base class could have been built without the relevant field, and a whole host of derived classes created, one for each different type to be supported. There is a sense in which this is systematic and clean, but sometimes it amounts to using a sledge-hammer to crack a nut, and if then virtual functions are used to support operations in all the derived classes both efficiency and clarity might suffer. I believe that for data representation the best class hierarchies will be quite small ones.

A quite different style of using C++ classes will expect that when a major class has been defined there will only ever be one variable of that type declared. In that case the data members of the class might almost have the same status as ordinary `static` variables, except that their visibility is controlled better, and the relationship between public and private member functions is strongly remeniscent of the difference between exported and non-exported functions in a language that allows one to organise code into modules. This use of the C++ class structure to provide global control over code and data visibility through an entire program is clearly good.

A further (perhaps more specialised) use for classes arises because whenever an object is to be discarded (including the common case where it has been declared as a local variable, but is now just going out of scope) a user-defined destructor function can be called. In some cases it can be invaluable to be able to register a function that will be called just before the current one exits, and classes provide this ability.

Some programming tasks have been associated for a long time with the development of object oriented programming and hence with the exploitation of class structured. Programming for windowed user interfaces and coding up certain sorts of simulation package are the most clear cut cases. If you work in one of these areas then

the class mechanisms in C++ need to be your first concern. In many other application areas it is worth considering the introduction of a few code-centred classes as a way of structuring your entire application into modules, but over-use of deeply nested derived classes for small and simple data structures will prove an unnecessary distraction and may obscure performance issues in your code.

Since this is only a short course on C++ the coverage of the class mechanism will stop here. There should be some further example programs illustrating it available by the time the lecture course is given, but these are not included in this document. If you feel thoroughly happy with all aspects of C++ except the class structure, inheritance and object oriented programming arrangements it may help you to check the library or bookshops for texts intended to teach C++ to people who are already competent C programmers.

12 Overloading, Templates

In a C++ world one could imagine wanting to have a function that would apply some process to its argument more with a whole range of types of argument supported. An obvious set of simple examples come from functions that take numeric arguments — eg one that squares a value — where the regular type-discipline of strongly types languages seems to get in the way. C++ provides four or five ways of coping with this need! The fact that there are all these different mechanisms should alert you to the fact that each will have a different set of strengths and limitations.

The first way of relaxing type constraints can be useful for very small functions, and is just to use a macro.

```
#define square(x) ((x)*(x))
```

The problem with macros is that they become ugly and inconvenient to use for anything other than very short code sequences, and there can be unexpected and unwanted effects if the actual arguments are expressions rather than just simple variables or literals. Syntax and type checking with macros is weak, but they are available with both C and C++.

The next scheme is C++ specific. In C++ it is valid to define two or more functions with the same name, provided their arguments differ in type. Doing this is known as *overloading* the name.

```
inline int square(int x) { return x*x; }
inline double square(int x) { return x*x; }
```

I used the `inline` keyword here only because the functions in this example are very short so flagging them for in-line expansion will probably help efficiency. Overloading is amazingly convenient if used with taste. But it can lead to the need to write multiple copies of essentially the same code, and if one function name is overloaded with multiple meanings that are not closely enough related it can become confusing.

It has already been explained that member functions in classes (and especially virtual functions) provide the means for one function name to have a meaning dependent on the class of an object that is being worked with. This is an excellent arrangement when it fits in with a class hierarchy that you already wanted, but wrapping both integers and doubles up in classes just so you could have a single name `square` for some operation on them would be overkill.

The C and C++ type "`void *`" denotes a pointer that can point to any sort of thing, and use of these generic pointers can make it possible to have a single function that can process (at arm's length) arbitrary sorts of data. The C library function `qsort` that sorts arrays of arbitrary items illustrates what can be done. It comes uncomfortably close to abandoning all hope of having the compiler do a comprehensive job of type-checking your code. To a large extent `void *` should be a**void**ed in C++ since it defeats too much of the type checking.

Finally there are *templates*. These are a relatively new inclusion in C++ so the support for them in compilers may not always be complete, but they represent a balance between the convenience of macros and the security of overloaded functions.

```
template <class Type>
    Type square(Type x)
    {   return x*x;
    }
```

The example has the definition of a squaring function written just once, with the type of its argument and result lest as a parameter that is introduced by the `template` construct. When this definition has been introduced, C++ will generate a type-specific version of the code suitable for every use of the function that it sees. Thus the effect at run-time will be similar to the case where all the various type-specific versions of the code where written by hand, but the source code is kept clean and tidy.

Lippman[5] gives a plausible example where a template function would be very natural in use in a general purpose sorting algorithm which he then demonstrates in use sorting vectors of double and then vectors of ints.

As well as template functions, C++ supports class definitions that are parameterised with regard to the types of component entities. It would be natural to use these in the implementation of a class for lists or queues, in that it would allow C++ to keep track of the types of items stored in the data structures while maintaining important flexibility.

Overall the template mechanism represents a level above regular C++ code within which types can be represented by type variables. You should look back to the way that ML supports polymorphism and type variables and compare with the relatively awkward way that C++ provides a subset of the ML capabilities.

13 User-defined operators

Just as C++ allows the user to introduce multiple definitions associated with a single function name, it makes it possible to have multiple meanings associated with operators such as + and *. It does not allow the introduction of new spellings for operators, or adjustment of the syntax associated with them. Thus the symbols available to the user for use as new operators is[24]

+	-	*	/	%	!	&	|
~	^	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	*=
<<=	>>=	[ ]	( )	->	->*	new	delete

It also only permits new meanings of operators to be defined when class types are present. Note that very early on we saw the overloading of the << and >> operators to support write and read operations on members of the stream classes. It can now be explained that what happened there was not some very special language provision just used by the library, but an example of a generally available option.

The operators +, -, * and & may be overloaded in either unary or binary form. But again note that you may not redefine the meaning these operators have when applied to numeric operands, only when they have at least one class object to work on. The syntax used to redefine an operator is trivially simple — to overload + say you just define (in the ordinary way) a function with name `operator+`.

The `[]` operator allows you to overload the idea of subscripting, and it will very often be desirable that overloaded uses of subscript expressions can appear on the left hand side of an assignment statement. This can be supported by declaring `operator[]` functions to return a reference type using the `& type qualifier`. Here is an example for a rather limited and hence not very useful array class:

```
#include <iostream.h>

template <class T>  // parameterised class
class Array
{
public:
    T &operator[](int n)
    {   return data[n];
    }
private:
    T data[10];     // hidden from user
};

int main()
```

[24] Note that a few of these have not been described in these notes.

```
{   Array<int> v;    // an array of integers
    Array<double> w; // an array of doubles
    for (int i=0; i<10; i++)
    {   v[i] = i;    // calls overloaded []
        w[i] = 1.0/(double)(i+1);
    }
    for (i=0; i<10; i++)  // check results
        cout << v[i] << " " << w[i] << endl;
    return 0;
}
```

The example gives another use of templates to make the array class able to support arrays of things or arbitrary type T, and shows that it is possible to include the definition of (short) member functions within the description of a class. The "`&`" in the line defining `operator[]` arranges that the operator passes back its result by reference. The code shown could be extended so that the "Array" class did something more interesting, while preserving exactly the same interface. For instance it could use a hash table rather than a simple vector to store the data, or do bound checking on array accesses, or maintain statistics about which of its elements was used most frequently.

The `()` overloadable operator can be used to make objects in a class appear to be callable as functions!

As with all aspects of C++ there are a few delicate areas surrounding the use of overloaded operators — for instance unlike ordinary functions they may not have default arguments. But for the full details you are referred to textbooks and reference manuals.

14 Exception Handling in C and C++

C supports recovery from exceptional conditions via a collection of library functions. The function `signal` takes two arguments, one being used to indicate what condition should be trapped, and the other being a function that is to called if the given condition arises. It will typically be possible to get `signal` to set up user-defined handlers for floating point exceptions, the use of bad memory addresses, the interrupt provoked when a DOS or Unix user types "`^C`" to interrupt a process, and various others. In a few cases the handler function may be able to repair the damage and allow the computation to continue, but there tend to be rather strict rules about just what library functions may be called from a handler and how it may behave.

One thing that a handler set up using `signal` is generally entitled to do is to call a function `longjmp`. This takes an argument that is of type `jmp_buf`, exits from multiple levels of nested function calls until it reaches one that used the function `setjmp` to fill in the `jmp_buf` involved. In fact `longjmp` may be called from arbitrary C code not just via `signal`, and is very useful for unwinding the stack when some calculation fails. A small example may help illustrate how the two function are used together.

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf bbb;

void recurse(int n);
{
    if (n == 0) longjmp(bbb, 1);
    printf("%d\n", n);
    recurse(n-1);
}

int main()
{   if (setjmp(bbb))
    {    printf("longjmp caught\n");
         return 0;
    }
    printf("setjmp returned 0 first time\n");
    recurse(10);
}
```

The magic is that when `main` calls `setjmp` in the usual way it just returns zero (ie false) and the main program can follow through with its calculation. When in due course `longjmp` is called it makes `setjmp` return (again!) but this time with a non-zero result.

C++ introduces a new and rather more civilised way of achieving similar effects[25]. Instead of using `setjmp` one writes the keyword `try` followed by a block of commands (enclosed in). This can be followed by a number of `catch` clauses that indicate exactly which exceptions can be processed at this level. Corresponding to `longjmp` there is an operator `throw`. It can be given an argument of any type, and passes this value out until a `catch` that is prepared to accept that type is found. It will sometimes be useful to declare a new class just so that that class can be the type associated with some particular exceptional condition.

```
class myerror
{   // class used just to report error
}

void recurse(int n)
{   if (n == 0) throw myerror();
    cout << n << endl;
    recurse(n-1);
}
```

[25]The exception handling parts of C++ are even newer and less stable than the template facility, so do not be too upset if the compiler you use does not support them yet. In particular the mid 1994 version of g++ does not.

```
int main()
{   try  // keyword to introduce protected block
    {   cout << "starting ... ";
        recurse();
        cout << "finishing\n";
    }
    catch(myerror x) // definition of a "catch"
                     // function for type myerror
    {   cout << "Error handler activated";
    }
    return 0;
}
```

15 Casts and other ways of Cheating

Most of what has been described of C++ so far will have given the impression that it is a strongly typed language where (at least if everything is carefully pre-declared in header files) the compiler can fully check everything. This can be almost the case in C++, but there are important loopholes available. Although most code will not make heavy use of them, almost all large C++ programs will use them somewhere.

Any type description (eg either a simple predefined type name such as `int` or a more complex that refers to functions, pointers or structures) can be enclosed in parentheses[26] and used as a prefix operator. This operator converts its operand to the named type. In fact one of the examples already given contained a cast that converted an integer into a double:

```
w[i] = 1.0/(double)(i+1);
```

In cases such as the conversions between integral and real numeric types casts will cause a change of representation to occur. Casts between integers of different widths (eg between `char` and `unsigned long int`) can result in zero- or sign-extension or range reduction. If you assign a value to a variable of a different type or call a function with an actual argument that differs in type from the required one then C++ will sometimes insert a cast for you to convert the type. I will **not** document the exact rules associated with this here, but rather recommend that you either make things match types exactly or put in explicit casts of your own, thereby avoiding any possibility of confusion[27].

Casts applied to pointer types are much more delicate. There is a gap between what can be guaranteed by a language standard as applicable on all possible compilers and

[26] In C++ the parentheses are optional, while in C they are needed.

[27] One case that is so common and generally harmless that I will mention as not usually being worth an explicit cast is assigning integer values into character arrays (ie strings) where the integer variable is known to hold a proper character value. This works “as expected”.

computers and what will "usually" happen on "typical" machines. About the most that can be guaranteed about pointer casts is that any pointer to an object (but not a pointer to a function) may be cast to the type "`void *`". The resulting generic pointer can then be passed to a function or stored in a `void *` variable. If subsequently cast back to its original type before use nothing will be lost. The use for this apparently pointless combination of casts is that it allows functions with `void *` arguments to accept pointers to arbitrary sorts of objects, despite the usual rigours of the type checker. It also makes it possible to have structures that have one field that contains a tag value, and another a field of type `void *` that in fact contains a pointer the type of which is dynamically recorded (by the user) in the tag field.

It is normally not very proper to cast between integral and pointer types, but C defines that casting from the integer zero to a pointer type will give a `NULL` pointer. Note that this does **not** necessarily mean that the machine-level representation of `NULL` is a bit-pattern of all zero bits, but it does mean (by virtue of the fact that C applies helpful implicit casts in various places) that loops such as

```
typedef struct linked_list_of_integers
{   int value;
    struct linked_list_of_integers *next;
} list;
int length(list *x)
{   int l;
    for (l=0; x!=0; x=x->next) l++;
    return l;
}
```

are valid, as well as the end-test `x!=NULL` or even just `!x`.

If you know enough details about the machine you are using and you do not need to write portable code it may be interesting to case a `char *` pointer to `long int *` and thus become able to manipulate memory four or perhaps even eight bytes at a time. Pointer casts provide the C++ programmer with very direct control over how they can access data. But note again that they will almost always limit code portability.

Another way of cheating on types uses a variant on the `struct` definition. If the keyword `union` is used instead of `struct` a new type is declared with a number of components, but now these components all overlap in memory. The clean use for this is where you need a field in a data structure that at different times contains objects of different types. The more devious (and non-guaranteed) use is when you write to one field and read back from another, thereby gaining direct access to something that depends on machine-level representations of data. The following code uses this idea to access the bit-patterns in memory associated with a floating point number. It would be possible to extend it to unpack the sign, exponent and mantissa fields, and such unpicking may be a critical part of the implementation of some of the low level floating point library functions.

```
#include <stdio.h> /* Do this on in C */
/* The union "cheat" puns between a double
```

```
   and an array of two integers */
typedef struct cheat
{   double d;  /* I expect 64 bits here */
    int i[2];  /* I hope int=32 bits */
} cheat;

int main()
{   cheat x;
    x.d = 1.0/7.0; /* some floating point value */
    printf("%x %x\n",    /* look at bit-pattern */
        x.i[0], x.i[1]);
    return 0;
}
```

Observe as before that "`.`" is used to select out components from a structure — here to select which variant in the union is to be used.

A final place where C and C++ abandon strict type-checking is to allow for functions with variable numbers and types of argument. The example that has been used so far in these notes is the C formatted output function `printf`. Look up `<stdarg.h>`, `va_arg` and "`...`" in a C manual for details of how to write functions of your own that have similar capabilities, and observe that the types of arguments passed and retrieved are not subject to type checking. To illustrate this, consider the call

```
/* The following line of code is not good! */
printf("%x %x\n", 3.1415926);
```

which passes a double precision value but then tries to display it as a pair of integers (in hex). On **some** computers the hex values displayed will be the representation of the floating point number, while on others one may get just garbage — possibly values that are not even consistent from one run of the program to the next.

The techniques mentioned in this section can be critical in making some programs easy to write or in making them run fast, but equally they can lead to opaque code with unexpected portability problems. If you read the full ANSI C standard it appears to have a number of unreasonable restrictions about what you can rely on, and demand almost neurotic caution in code that is to qualify as strictly conforming to the standard — but each of its strictures is based on knowledge of some awkward computer or compiler where attempts to cheat have unexpected consequences. So by all means use casts, but use them with care.

16 Writing Robust, Portable Code

Perhaps the first feature of good programming style will be that code is well documented and easy to read. That way the almost inevitable changes that will have to be made to it will feel less painful. Figure 8 shows one approach to making C code read-

```
char*lie;
        double time, me= !0XFACE,
        not; int rested,   get, out;
        main(ly, die) char ly, **die ;{
            signed char lotte,

dear; (char)lotte--;
        for(get= !me;; not){
        1 -  out & out ;lie;{
        char lotte, my= dear,
        **let= !!me *!not+ ++die;
            (char*)(lie=
"The gloves are OFF this time, I detest you, snot\n\0sed GEEK!");
        do {not= *lie++ & 0xF00L* !me;
        #define love (char*)lie -
        love 1s *!(not= atoi(let
        [get -me?
            (char)lotte-

(char)lotte: my- *love -

        'I'  -  *love -  'U' -
        'I'  -  (long)  - 4 - 'U' ])- !!

        (time  =out=  'a'));} while( my - dear
        && 'I'-1l  -get-  'a'); break;}}
            (char)*lie++;

(char)*lie++, (char)*lie++; hell:0, (char)*lie;
        get *out* (short)ly   -0-'R'-  get- 'a'^rested;
        do {auto*eroticism,
        that; puts(*( out
            - 'c'

-('P'-'S') +die+ -2 ));}while(!"you're at it");

for (*((char*)&lotte)^=
        (char)lotte; (love ly) [(char)++lotte+
        !!0xBABE];){ if ('I' -lie[ 2 +(char)lotte]){ 'I'-1l ***die; }
        else{ if ('I' * get *out* ('I'-1l **die[ 2 ])) *((char*)&lotte)-=
        '4' - ('I'-1l); not; for(get=!
```

Figure 7: Continued overleaf. Code by Merlyn Leroy of DigiBoard.

```
get; !out; (char)*lie  &  0xD0- !not) return!!
        (char)lotte;}
(char)lotte;
        do{ not* putchar(lie [out
        *!not* !!me +(char)lotte]);
        not; for(;!'a';);}while(
            love (char*)lie);{

register this; switch( (char)lie
        [(char)lotte] -1s *!out) {
        char*les, get= 0xFF, my; case' ':
        *((char*)&lotte) += 15; !not +(char)*lie*'s';
        this +1s+ not; default: 0xF +(char*)lie;}}}
        get - !out;
        if (not--)
        goto hell;
            exit( (char)lotte);}
```

Figure 8: An example of readable C code?

able —- note the careful choice of names for functions and variables and the thoughtful use of layout. This code also illustrates a further collection of C (and hence in general C++ keywords and functions that you may wish to look up in the reference manuals.

Both [2] and [4] contain much good advice about C programming style and explanations of various of the ways in which code can unexpectedly prove to be non-portable. Rather than recite here all the particular points they make I will just catalogue the ones that I consider most critical:

1. Design your program before you write it: Especially if you are going to make extensive use of class hierarchies it is not a smart idea to start writing a program by typing random fragments of code into an editor. Just how formal a design phase you need will depend on the expected eventual size of the code to be written.

2. Be disciplined: C and C++ place ultimate responsibility for almost everything on the programmer. They make it possible to write clear and reliable code, but they also make it possible to construct abominable muddles. As examples of language design they have been guided by the desire to support the competent programmer solving a serious problem, rather than the need to protect a novice who only has to write a page or so of code.

3. Use the security features your compiler does have: Modern compilers can gen-

erally be instructed to give you warning messages about constructs in your code they find questionable. Enable this option, and pay attention to the warnings – even if at first you find some of them look fussy they are probably there because they mark potential hiding places for bugs. Write your code so you get no warnings. With old style C this sort of checking was done with a separate program called `lint`.

4. Declare things in header files: when they need to be global, or otherwise use the `static` directive or C++ classes to keep both variables and data consistent and as localised as possible. When designing classes avoid making more things `public` than you need to.

5. Code in a defensive style: Avoid unnecessary application of puns and low cunning. Help the person who next reads your code by putting in blocks of comments that explain methods and global intent as well as find details. Put in occasional checks for self-consistency, perhaps using the `assert` library operation. Assume that your code will at some stage need to be ported to a very different computer with a compiler that comes from a different vendor.

6. Know your language: Browse the most definitive reference manual you can find for your language and its libraries so that you know exactly what it does, and (even more important) so you know how and where you can look things up when you need to.

7. Get somebody else to read your code: And listen if they say that they find it hard to understand!

8. Avoid known danger areas and pitfalls: See Section 21.

9. Avoid premature optimisation: It is very nearly always much better to write a program that is correct and then worry about speeding it up than it is to write a program that is supposed to be fast and then try to remove the bugs. Furthermore if you try to optimise your code early on you may waste effort on parts of the code that are in fact unimportant, or obscure later opportunities for more radical but better speed-ups.

Of course the above ideas are relevant to almost all languages. But with C and C++ they are particularly important since the languages are often used to implement quite large and complicated packages, compilers are readily available on many different computers, so people frequently try to port code, and thoughtless use of the languages can easily lead to cryptic or fragile code.

17 Writing Fast Code

On of the main claims to fame for C (and hence C++) is that it can be used to write code that goes fast. It is common to claim "As fast as hand-written machine-code", although in reality that will only be true in limited circumstances. What does help the efficiency of C code is that most constructs in the language are really very low level and correspond quite directly to things that a typical computer can do in one or two instructions. This means that the programmer can usually keep track of just how much real work is being called for by a fragment of code, and can express things so as to minimise it. The most pervasive way in which this happens is probably through the fact that C++ provides smooth and natural support for pointers and pointer arithmetic, and makes it convenient to write code the uses them — and most computers have internal registers that can hold pointers and manipulate them directly. In contrast, most other languages make much heavier use of arrays, and subscripting into an array is often (at the machine level) more expensive than just following a pointer. Many computers (from Digital Equipment's VAX through to Acorn's ARM) provide machine code addressing that give direct support for the C idiom `*x++` that accesses an item and then steps the pointer on ready to grab the next.

Especially on modern "RISC" processors it is important to keep the correct values in the machine's internal registers. When you declare a C variable[28] that is to hold an integer, floating point value or pointer you can qualify the declaration with the word `register`. This suggests to the compiler that it keep the variable in a machine register. Depending on your computer and compiler it may be that most of your variables will be kept in registers anyway, or it may be that there are not really enough registers to go around so even variables declared as `register` are kept in memory (on a stack). But on occasions the careful use of `register` declarations can have a major effect on code quality.

If you know enough about the computer you are working on it may be possible to implement many operations on blocks of characters by using word operations, thus moving several bytes in each operation and making full use of the word-size of your machine. This can have significant pay-off when you are implementing either a string processing or a graphics library.

Use of the bitwise `&` and `|` operations together with shifts can allow the implementation of mildly clever algorithms. Figure 9 shows three versions of functions that count the number of bits that are set in an integer, supposing that integers are 32 bits long, and illustrates some of these points. Note that in `version3` the use of an `unsigned int` argument ensures (on a 32-bit machine) that `a>>24` is in the range 0 to 255 so a mask operation with `0xff` is avoided.

Generally if you want to produce the best and fastest possible code the steps are first to produce a version of the code that works correctly all the time, then to see if the algorithm involved can be improved, and only at the end move on to local fine-tuning

[28] This will only apply to local declarations, not to `extern` or `static` ones.

```
int version1(register int a)
{   // naive method, use register vars
    register int total=0;
    for (register int i=0; i<32; i++)
       if (a & (1 << i)) total++;
    return total;
}

int version2(int a)
{   // (a & -a) = least sig bit of a
    // if you have binary arithmetic
    int total = 0;
    while (a != 0)
        a -= (a & -a), total++;
    return total;
}

static unsigned char cc[256] =
{ 0, 1, 1, 2, 1, ... };  // initialised array

int version3(unsigned int a)
{   // make cc a table counting bits in a byte
    return cc[a & 0xff] + cc[(a>>8) & 0xff] +
      cc[(x >> 26) & 0xff) + cc[a>>24];
}
```

Figure 9: Counting the bits in a 32-bit word.

of the code. On many systems there will be profiling tools that can help you find out which paths through your code are exercised most and hence deserve most careful attention. Directing your compiler to display the assembly code that it generates may allow you to spot ways in which the code could be improved. Remarkably often it is possible to juggle with your C or C++ source code until the compiler generates the optimal code that you wanted. Note very well that ultimate optimisation and ultimate portability are often not compatible, so at a minimum you should collect speed-critical code in one localised part of your program, and preserve the original correct and clear versions of the code (hidden in comments, maybe) together with the final highly tuned version, ready for when you need to re-optimise for a different computer.

18 Internationalisation of Code

A small program written for use just by its author will use the character set natural for the computer on which it is developed, and will naturally display all its messages in a readable form. However it is increasingly the case that commercial code has to work in an international market, where both character sets and the language in which messages are displayed must be adaptable. One approach to the character set problem is shown in Figure 10 which, if you run it, can be seen to be code to print the value of e, the base of natural logarithms. Two other (and perhaps more productive) approaches are commonly used. The first addresses the character set problem. Just as C has ordinary integers and long integers it can have ordinary characters and strings and "wide" ones. The idea is that ordinary characters are used in circumstances where a simple 8-bit characters set is adequate, while wide characters (typically 16 bits each) can be used top cope with the extended sets of symbols needed for (particularly) far eastern markets. Wide character and string literals are written by prefixing the usual sort of notation with "`L`" as in `L"This is a wide string"`. The issue of how interesting characters might be written inside such a string is dependent on the compiler used. There are standard library functions for extracting characters from wide strings, copying them and generally working with them. With its Windows C++ compilers Microsoft provide an option for strings to be expressed in their own choice of 16-bit code (Unicode) and a few Windows data structures and functions require text to be expressed in this extended format.

To deal with language insensitive messages it is good practise in large programs to avoid writing messages as strings directly embedded in the code. If all text is separated out and stored in tables then the code can refer to strings by quoting their index number in the table, and language conversion only involves re-writing the module containing the table and linking in a new version. It can even make sense to go one step further and have the text of all messages stored in a data file that gets read in either at the start of a run of your program or when it needs to display a message — then changing language just involves installing a new data file and not any recompilation. These arrangements which keep all text centrally and accessed via retrieval tables or functions

```
                                                                  char
                                                                _3141592654[3141
         ],__3141[3141];_314159[31415],_3141[31415];main(){register char*
      _3_141,*_3_1415, *_3__1415; register int _314,_31415,__31415,*_31,
    _3_14159,__3_1415;*_3141592654=__31415=2,_3141592654[0][_3141592654
   -1]=1[__3141]=5;__3_1415=1;do{_3_14159=_314=0,__31415++;for( _31415
  =0;_31415<(3,14-4)*__31415;_31415++)_31415[_3141]=_314159[_31415]= -
1;_3141[*_314159=_3_14159]=_314;_3_141=_3141592654+__3_1415;_3_1415=
__3_1415    +__3141;for                (_31415 = 3141-
           __3_1415  ;                 _31415;_31415--
           ,_3_141 ++,                 _3_1415++){_314
           +=_314<<2 ;                 _314<<=1;_314+=
          *_3_1415;_31                  =_314159+_314;
          if(!(*_31+1)                  )* _31 =_314 /
          __31415,_314                  [_3141]=_314 %
          __31415 ;* (                  _3__1415=_3_141
         )+= *_3_1415                    = *_31;while(*
         _3__1415 >=                     31415/3141 ) *
         _3__1415+= -                    10,(*--_3__1415
        )++;_314=_314                    [_3141]; if ( !
        _3_14159 && *                    _3_1415)_3_14159
        =1,__3_1415 =                    3141-_31415;}if(
        _314+(__31415                     >>1)>=__31415 )
        while ( ++ *                      _3_141==3141/314
       )*_3_141--=0                       ;}while(_3_14159
       ) ; { char *                       __3_14= "3.1415";
       write((3,1),                       (--*__3_14,__3_14
       ),(_3_14159                         ++,++_3_14159))+
      3.1415926; }                         for ( _31415 = 1;
     _31415<3141-                          1;_31415++)write(
    31415% 314-(                           3,14),_3141592654[
  _31415    ] +                           "0123456789","314"
  [ 3]+1)-_314;                           puts((*_3141592654=0
,_3141592654))                             ;_314= *"3.141592";}
```

Figure 10: Code by Roemer Lievaart.

also make it fairly easy to arrange that the message text is kept in some compressed form, and expanded on demand - for large programs the saving in space achieved by keeping strings compressed far outweighs the bulk of the decompression code. Keeping messages compressed also keeps them just a little more secure from prying eyes.

Arranging those functions that print diagnostics so that parameterised messages will make sense whichever language the message is to be displayed in is not something that happens without thought, and a great many error messages need part of the user's input data (or something else) merged in with the text that is displayed. One possible approach is (rather than having all messages in a string table) to use the pre-processor to do the work as follows artificial example:

```
// this is part of a file "error-messages.h"
#ifdef ENGLISH
#define msg1 "Wrong colour used"
#define msg2(a, b) "Message %s with %d in it", a, b
#else
#define msg1 "Wrong color used"
#define msg2(a, b) "Case %d in context %s", b, a
#endif

// now for some of the code that uses these
#include "error-messages.h"
...
if (...) printf(msg1);
else if (...) printf(msg2("blah", 42));
...
```

Observe that even the order of the values to be included in messages can easily be controlled in this form of parameterisation, and that it takes advantage of the fact that the preprocessor does not have to expand things into syntactically complete forms — for instance `msg2` expands to give a list of three items.

Finally an ANSI C library comes complete with some features to support code for international markets. The place where a program is to be used is known as a *locale*, and the library provides skeleton support for getting times, dates and amounts of money displayed, and for controlling the alphabetic order used when comparing strings that are in non-American character sets (eg ones that use codes to stand for accented characters (ñ, ç) or things like the German ßor Nordic Å. Despite being part of the ANSI standard `setlocale` is generally not very well supported by most C and C++ systems, and you may well do better to write and use your own code. But read the relevant section of the ANSI document (and the accompanying rationale) if you can find a copy to discover what they expected to count as standard layouts in a variety of countries: it goes beyond the traditional muddle as to whether the date 1-2-95 is the first of February or the second of January!

19 The ANSI Standard For C

The major feature of C which is not shared by all the languages that you come across is that there is a formal international standard for it. Despite the fact that ML refers to itself as "Standard ML" the dialect so labelled was decided upon by a medium sized group of language enthusiasts. Modula-3 is defined by its implementation and a reference manual. The amount of manpower and effort that went into the standarisation of C was of a quite different order of magnitude, involving not just a central committee of experts working over a period of around seven years, but also a consultation process where comments and suggestions were send in by many many thousands of other interested people. Following on from the issue of the official ANSI C standard various commercial organisations have developed stringent test suites that National certification bodies and others can use to check if a supposedly conforming implementation of C does in fact meet the specification.

Without all of this it is effectively impossible for two independent implementations of a computer language (any language, not just C) to agree in all the fine and murky corners where they ought to, and it is also impossible for a language implementation team or user to draw a proper line that separates differences in behaviour that are compiler bugs from those that are necessary consequences of running on a different computer or operating system.

The only proper way to find out in detail about the ANSI C standard is the defining document[7] itself. The standards body obtains part of its funding by selling printed copies of the documents that it produces, so this document should not be available on-line anywhere. It also hardly counts as an ordinary book, so regular bookshops tend not to have copies on their shelves. There ought to be a copy available from the Computer Laboratory Library: if you want your own copy expect to have to order it (possible through your bookshop) from the American National Standards Institute (ANSI), 1430 Broadway, New York, NY 10018, USA (phone: 1-212-642-4900).

A major issue in preparing a standard for a language is ensuring that it is quite clear what is defined and what is left to the discretion of the compiler-writer. Not leaving any flexibility at all would be possible if the standardisation was done by selecting one existing implementation running on one real computer system, and declaring that to be the reference with all other versions expected to match its behaviour **exactly**. Quite apart from severe commercial offence that such an approach would cause, a consequence would be that one particular release of the reference compiler would be the standard — including all the bugs in it — and anybody wanting to meet the standard exactly would need to discover and perpetuate those bugs. It is said that when, in Cambridge, the Titan computer was installed as a replacement for the earlier EDSAC-2, an Autocode[29] compiler for Titan was prepared following this philosophy, and thus carefully and deliberately (and dare one suggest in a spirit of mild fun) re-creating the known bugs in the EDSAC compiler. That was in the mid 1960's and I do not have

[29]The high level programming language in use then.

and good but more recent examples of the approach.

At a slightly more abstract level over-definition of a language can still be unduly restrictive. Users of C generally expect competent optimisations from their compiler, and drawing a good line between defining enough that code can be reliably ported and leaving enough freedom (eg with regard to the order of evaluation of the separate parts of a complicated expression) is difficult. There are also fundamental differences between various brands of computers that would all like to support standard C compilers. As well as the obvious matter of 16 and 32 bit systems (and now 64-bit ones), some computer like to lay out integers in memory with the least significant byte at the lowest memory address, and some with the most significant byte at the lowest address[30]. Characters sets also vary. Most Unix systems agree on the ASCII character set, but this is only really adequate for the representation of English[31]: even other European languages require a variety of extra or accented letters so computers for use in such places must support them. Thus a programming language that is for realistic international use can not survive if defined to use just the 96 characters that standard ASCII knows about.

From these and many more issues, ANSI C builds up to the idea that a standard for a programming language should be viewed as a contract between the language implementor and user. The implementor **must** support some language constructs, will be given license to make definite decisions about other ones on a local basis, is not required to give any guarantees at all in some further cases, but will be prohibited from making some sorts of extension to the language that a compiler will accept. The user on the other hand may choose to try to write a strictly conforming program that adheres should work correctly on all possible conforming compilers. Or for better performance a programmer may choose to rely on some of the implementation defined aspects of one particular compiler.

ANSI specify that some sorts of things that a programmer might write will have "undefined" consequences. It is often very tempting to think of such cases by imagining how several different compiler strategies could lead to wildly different behaviours, and assume that "undefined" means an arbitrary choice from among those behaviours. That is **not** what is meant! If you submit a program that contains code with undefined interpretation to a fully validated and utterly conforming ANSI C compiler it would be entitled to spot what you had done and generate in the place of your dubious code (or indeed at any other place in your whole program that it saw fit) new and original code that deleted all your files, send abusive e-mail to your manager, or dialled a phone hotline to order that a hit-squad come and rearrange your features and/or programming style. The term "undefined" in the ANSI standard really is supposed to indicate that **all** bets should be off about what might happen when a program is run!

Reading and understanding the full consequences (and the reasoning behind) the

[30] The distinction is usually referred to as the "byte sex" one, but it is harder to find agreed names to characterise the two configurations. Following Gulliver, little-endian and big-endian are probably the most common terms in use.

[31] And even then it has only been recently that such spellings as "æroplane" have gone out of fashion.

ANSI standard is a fascinating exercise in applied pedantry! The standards committee needed to balance a desire to keep to the existing spirit of the C language, and preserve the usefulness of as much existing code as possible. Equally they needed to end up with a definition that was acceptably unambiguous (the original edition of Kernighan and Ritchie[3] is notable for gaps in its precision) and which made C into a better and more modern language. Some of the issues they addressed are covered further in Section 21.

20 Forthcoming Standardisation of C++

As mentioned before, the committee that is working towards a C++ standard has issued its first review document. Rather than describe what is in that (it is much too long!) I will try here to give an overview of what can be expected from the final standard:

Explore ambiguities: The existing C++ descriptions at least come close to leaving the language ambiguous. A standard may choose either to give very precise rules to define which meaning should be ascribed to each program, or adjust the language to make it a little less delicate. Those of you who have not been involved in C or C++ implementation projects may not have a good feeling for how delicate the languages are and how difficult it is to have a standard that leaves flexibility for compiler writers to generate good code on all brands of computer while defining the exact behaviour that can be expected in all reasonable cases.

Define libraries: The `iostream` library used in example programs here is a very minimal start towards what can be expected by way of library support defined in a C++ standard. To preserve backwards compatibility it may be hoped that the existing C library functions will remain available, but then the exact rules for mixing use of the C `printf` and C++ "`<<`" operations will need to be documented. In the draft ANSI specification the section discussing the library is itself book-sized[6] and will represent a significant leaning effort for users.

Firm up newer features: Exception handling in C++ is (at the time of writing) still an area that has not stabilised, and there will be a great many other areas of the language where people will want to suggest small changes to the rules in order that the language finally defined ends up as coherent and useful as possible. The nature of any 700+ page document is that there will be inconsistencies between statements made in different sections, and the process of detecting and resolving them will be quite protracted.

Address C standard failures: In ANSI C provision was made to allow programmers to write C code even if the keyboard they used was deficient (lacking various punctuation characters such as `[]{}\&^|`. This was done using things called trigraphs, each starting with the string "`??`". The arrangement seems ugly and

is not obviously popular — the C++ committee is suggesting better solutions to this problem and to other areas where five or so years of experience with ANSI C have revealed room for improvement.

21 A Few More Pitfalls and Traps

It has already been explained that compiling and running a program that has "undefined" behaviour is permitted to have most curious effects. And furthermore those effects might depend on the date or time of day that the program was run[32]. This section lists a collection of tolerably common C (mostly) and C++ mistakes to give you an idea of what to be on guard against.

21.1 Compiler Bugs

It would feel reasonable to expect that a very heavily used language like C would now have compilers that were 100% reliable, and that the ANSI standard would be met in all serious implementations. One would hope that C++ would only give trouble in areas where the specification is still under review. If you develop large C or C++ programs and try to run them on a significant number of different architectures you will find that the world is not so kind. Even when a single compiler implementation (`gcc`) is available on many computers you will find that not all have the same release installed, and there can be target-specific compiler bugs. I have encountered C compiler or library bugs that have caused me at least some frustration on PCs (at least 3 different compilers), Macintosh, Sun, HP, SGI, Acorn and Apollo computers. You should not get the impression that C will be worse than any other language in this respect (it is probably better) or that compiler bugs are the major cause of non-working code, but they are out there and they can hurt.

21.2 Sequence Points

Within one expression a C compiler may generally evaluate sub-expressions in whatever order it sees fit. At so called *sequence points* everything must be brought back into a stable condition: the main constructs that give rise to sequence points are the semicolon that terminates a command, the comma that joins a succession of expressions into a sequence (but not the comma that separates arguments in a function call) and the `&&` and `||` logical connectives. In the presence of side effects evaluation order may matter. Consider where `f` is some function of two arguments. The programmer may have expected that the arguments passed will have been 1 and 2. But the ANSI C standard says that the behaviour is undefined. To see why it may be reasonable

[32] If the undefinedness is a result of the program attempting to read from un-initialised memory or from outside the space allocated to it it is even reasonable for the exact details of the behaviour to vary from run to run.

to make such things undefined, imagine a computer where `a++` is implemented as code that picks up the value of `a` into a register, makes a copy into a second register, increments one of the register (while keeping the other for use as the value of the expression) and finally writes the incremented value back to memory. Especially on a high-performance RISC machine it could make sense to interleave the streams of instructions that did that for the first and second argument, and after scheduling the instructions to keep the CPU fully busy almost any arrangement could result, and it will not even be clear that `a` will end up with the value 3. By making the meaning of the code undefined the standard makes it legal for a compiler to generate whatever code happens to come out of its optimiser, or to detect the oddity and generate code that prints a warning and stops (or anything else the compiler writer likes the idea of). Other problems arise with indirect addressing:

```
int a = 1;
int *b = &a;   // take address of variable a
cout << (a + *b++); // a bit like (a + a++)?
```

21.3 Macro expansion woes

Consider the code

```
#define print(x) { cout << x; cout << x; }
if (something) print(x);
else ...
```

The important issue here is that there is a macro that is defined so as to expand to a block of commands enclosed in braces — the fact that printing is involved is unimportant. It then looks natural to use this macro in other arbitrary contexts, however there is a slight trap: it looks correct to write a semicolon after the use of `print(x)` in the `if` statement above, but because the macro expands to `{ ...}` that is in fact incorrect. It is necessary to write one of

```
if (something) print(x)
else ...
if (something) { print(x); }
else ...
```

THis illustrates how much caution is sometimes needed with preprocessor macros: it C++ inline functions are used instead the problem might not arise. However even then all is not well, since even when a function is inline all the values it needs to access must be passed as arguments, so there will remain places where macros are needed. Some people suggest that the solution to this problem is to go to the trouble of writing all macros that would expand to a block as

```
#define my_macro(a,b,c)           \
    do {   /* start of block */   \
      ...                         \
      ...                         \
    } while (0)
```

and then the curious looking use of `do...while` has no effect on execution behaviour but leaves the syntax more secure!

21.4 Out of memory failures

In C chunks of memory are obtained by the use of the `malloc` function[33]. In C++ the corresponding facility is provided by the `new` operator. If you use these a lot it can become hard to remember that eventually you may run out of memory and the attempt to allocate more will fail. `malloc` and `new` do not provoke an immediate crash when they are unable to satisfy a request — they just return a `NULL` pointer. If you fail to check for this case and eventually you do run out of memory the effects may be uncomfortable.

21.5 Memory allocation

If attempting to use a `NULL` pointer as if it were the address of the start of a useful block of memory was a bad prospect, then the consequences of not getting uses of `delete` properly matches with those of `new` can be much worse. Such errors can sometimes confuse the internals of the freestore allocation package, and result in newly allocated structures overlapping. And these new structures may not just be ones that the user allocates directly — they could be control blocks or buffer areas allocated by the library for its own internal use. Running foul of this problem is almost enough cause to give up C++ totally and move over to Lisp where storage allocation has been properly thought out.

21.6 Other library function failures

As well as running out of memory, there are many other possible ways that library functions may not work properly for you. In general the C tradition is that the function just returns, possibly with some slightly unusual result, but that there will not be any automatic generation of warning messages. The programmer has full responsibility for checking that all goes well. A few nasty examples: `sqrt(-1.0)` does not generate an exception, it will just hand back some silly value. If an output stream has been directed to a file on a floppy disc (say) then if the user removes the disc while the program is running, or if the disc is damaged, or runs out of space, then simple output operations can fail. The functions that support direct access (`fseek`) may not be supported on all sorts of file (eg on Unix `/dev/null`!). There may be system-imposed limits on how many files you can access at one time. The list of possibilities is almost endless, but in C and (so far) in C++ checking each individual library call for success can be very tedious and ugly.

[33] Especially under Unix it can sometimes make sense to use a lower level and system-specific function `sbrk`, while if you are writing code for a Macintosh or for regular Windows 3.1 then you usually need to take great care to fit in with the operating system's choice of memory allocation calls.

21.7 Unsigned values

If you have, in your code, a mixture of short and long integers and signed and unsigned variables, the compiler will apply rules to bring quantities to a proper common type when an operation is to be performed. This can involve widening a short value to a long one, and depending on whether it is signed or unsigned this either propagates the sign bit or pads with zero bits. Usually what happens will be just what you expect, and most operations (in particular `+`, `-`, `*`, `&`, `|` and `<<` are not sensitive to whether their arguments are signed or unsigned. However the comparison operators `<` and so on are. If `x` is an unsigned value then the boolean expression `(x < 0)` can never be true. Beware cases where comparisons may involve values of different widths or where some arguments may be signed and others unsigned, or read the standard **very carefully** so make sure you understand what will happen. Explicit casts as in

```
if ((unsigned long int)x < 0x8000LU) ...
```

can avoid any possible confusion.

21.8 Unexpected overflows

If you perform arithmetic in C in such a way that there could be overflow then the Standard declares that all bets are off. With almost all compilers arithmetic will in fact be done using 16 or 32-bit 2's complement and the overflow will be silently ignored. Sometimes this is very useful, but on others it is a source of bugs. My favourite nasty in this regard are the harmless-looking shift in

```
#define bit_n(n)   (1 << (n))
{   long int x = ...
    if (x & bit_n(24)) ...
```

which is attempting to see if the bit `0x01000000` is set in the variable `x`. On a 16-bit computer despite `x` being a long integer the shift may be done in 16-bit arithmetic, leading to a (probably silent) overflow and the a test that masks with zero. 1¡¡24 when sizeof(int)==2. My preferred correction is to have a type of my own called `int32`, with a `typedef` to map it onto a suitable system integer type, and then define the macro as

```
#define bit_n(n)   (((int32)1) << (n))
```

21.9 Arithmetic Right Shifts

ANSI C permits an implementation to implement right shifts on signed values either so that they replicate the sign bit, or so that they fill bit positions vacated by the shift with zero. Thus right shifts on signed values need to be coded with a degree of caution. While mentioning shifts, observe that (if overflow is indeed always ignored), the following three are equivalent if `x` is an integer:

```
2 * x;
x + x;
x << 1;
```

and occasionally the addition or shift may be faster than a multiply (but a good compiler would make the transformation for you if it was really useful). For positive values on a binary machine `(x/2)` and `(x>>1)` are also equivalent, but shifting negative values right will not in general halve them even if the shift is arithmetic.

21.10 Indirection through NULL pointers

A pointer variable will normally contain a proper pointer, but it is always valid to store `NULL` there. If the value is `NULL` then indirecting on it is a bad idea. With some operating systems the attempt will be trapped promptly, while with others only attempts to write to the bad address cause a fault, and reading just retrieves some stray value. Yet cruder machines would trap neither reads or writes, and so the erroneous program could possibly corrupt system memory!

21.11 Representation of NULL

Because on many implementations of C the value `NULL` is stored as a zero bit-pattern some programmers come to believe that this is necessarily so, and rely on it. `NULL` is what you get when your compiler casts zero to a pointer type, and it may be different. For instance one could imagine a computer where all pointers were 48 bits long while integers were various other widths, then `NULL` would be a 48-bit quantity and not quite like any integer at all.

21.12 Local variables after `setjmp`

`setjmp` can be understood by imagining that when it is called it just dumps the machine's registers in the `jmp_buf` and returns. `longjmp` can then just reload all the registers, which may have a side effect of resetting the stack and allowing it to return from `setjmp` for a second time. Just how much state gets restored by `longjmp` may be slightly system dependent, and the standard is careful to specify that the state of local variables whose value has been altered between the original call to `setjmp` and the `longjmp` is not determinate. Beware!

21.13 Code and Data memories separate

It could be that a computer has quite separate address spaces for code and data, in which case an attempt to cast from a pointer to a function to get a pointer to data will be pretty ineffective. Also code space may be read only (in some cases it may even eventually map onto ROM), and some compilers may like to store literal constants

(especially string constants) in code space, while others will put them together with other data. So although on some systems you may get away with updating the contents of a string literal (yuk!) and by so doing save some space or time, do not count on it. If the same text appears in several strings in a program some compilers may try to save you space by storing the string just once, while others will naively store multiple copies — with C you can easily write code that would behave differently depending on the strategy used!

21.14 `printf`, `scanf` and long integers

Users of the C library functions `printf` and `scanf` should take some care to ensure that the types of values passed match the format directives present in the format strings — especially perhaps when `scanf` is used to read a long integer.

21.15 Assuming 16 or 32-bit arithmetic

I was caught out when I first used a compiler that had support for 64-bit integers by the fact that `~0x80000000` was not the same as `0x7fffffff` but could be `0xffffffff7fffffffL` It is very easy to suppose that you know just what sort of arithmetic you are relying on!

21.16 Comment nesting

`/*` Here is the start of a comment, where due to clumsiness with the editor the intended end of the comment, here it is: `*?` has got slightly mangled. I had `?` where I meant `/` The effect is that the text you are reading now is begin swallowed up as part of the comment, probably without any load message from the compiler. Things may recover when I get to the start of the next comment — ah here it comes `/*` Here is the start of my next comment - and its end `*/` The text from here on is now not inside a comment, and if I am unlucky the material inadvertently swallowed will not disrupt my syntax and I will be none the wiser about its loss.

21.17 Scope of `struct` tags

A gross oddity in ANSI C that may get changed in C++ is that if the first time you see a structure tag is in the header line of a function, as in

```
extern void f(struct xx *y);
```

then the structure tag is treated as having a scope that is just the body of the function defined. Even if you write `struct xx` elsewhere it will be viewed as a different tag. In consequence is not possible to call the function giving it an argument of exactly the correct type! To avoid this lunacy, declare all your structures (and classes) before functions that use them.

21.18 Byte Ordering

Quite often one wants to write whole data structures out to a file, or copy them wholesale from one part of memory to another. The raw memory copying operations are not sensitive to the order of the bytes that make up integers (or floating point values). A case where this sort of issue really matters is if you are writing a compiler that ought to generate exactly the same binary object files whatever brand on machine it happened to be hosted upon.

21.19 Six-character mono-case names

As a matter of extreme caution, ANSI C suggests that names of external variables and functions should be limited to six letters and that you so not assume that case can be used to distinguish letters. It is perhaps reasonable to hope that you will never come across an environment that makes this restriction come alive! One way to cope if you ever do is to compile your whole program with a header file that maps the sensible meaningful names that you will use into horrid short ones suitable for your backwards-looking computer. Here is a slightly modified version of a small fragment of a header file that I use for just that purpose:

```
#define alloc_dispose         g01all
#define alloc_init            g02all
#define alloc_noteaestoreuse  g03all
#define alloc_reinit          g04all
#define alloc_unmark          g05all
#define builtin_init          g06bui
#define cautious_mcrepofexpr  g07cau
#define cautious_mcrepoftype  g08cau
#define cc_err                g09cc_
#define cc_err_l              g10cc_
```

A The GNU Library General Public License

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor’s protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don’t assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

GNU LIBRARY GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

    c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

    (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

  Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that

uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying

the Library or works based on it.

  10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

  11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

  12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

  13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

  14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is

copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the library's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.
>
> You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if

```
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That's all there is to it!
```

B DOS GCC redistribution notice

```
This is the file "copying.dj".  It does not apply to any sources
copyrighted by UCB Berkeley or the Free Software Foundation.

   Copyright Information for sources and executables that are marked
   Copyright (C) DJ Delorie
                 24 Kirsten Ave
                 Rochester NH  03867-2954

This document is Copyright (C) DJ Delorie and may be distributed
verbatim, but changing it is not allowed.

Source code copyright DJ Delorie is distributed under the terms of the
GNU General Public Licence, with the following exceptions:

* Any existing copyright or authorship information in any given source
file must remain intact.  If you modify a source file, a notice to that
effect must be added to the authorship information in the source file.

* binaries provided in djgpp may be distributed without sources ONLY if
the recipient is given sufficient information to obtain a copy of djgpp
themselves.  This primarily applies to go32.exe, emu387, stub.exe, and
the graphics drivers.

* modified versions of the binaries provided in djgpp must be
distributed under the terms of the GPL.

* objects and libraries linked into an application may be distributed
without sources.

-----

Changes to source code copyright BSD or FSF are copyright DJ Delorie, but
fall under the terms of the original copyright.

A copy of the file "COPYING" is included with this document.  If you did not
receive a copy of "COPYING", you may obtain one from whence this document
was obtained, or by writing:
      Free Software Foundation
      675 Mass Ave
      Cambridge, MA  02139
      USA
```

References

[1] Jim Conger. *Microsoft Foundation Class Primer*. Waite Group, 1993.

[2] Samuel Harbison and Guy Steele. *C, a Reference Manual*. Prentice Hall, 1987.

[3] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1 edition, 1978.

[4] Don Libes. *Obfuscated C and Other Mysteries*. Wiley, 1993.

[5] Stanley Lippman. *C++ Primer*. Addison Wesley, 2 edition, 1991.

[6] P. J. Plauger. *The Standard C++ Library*. Prentice Hall, 1994.

[7] X3J11. *ANSI X3.159, ISO/IEC 9899:1990*. American National Standards Institute, International Standards Organisation, 1990.