Programming Methodology

in

C

Hugh Anderson

Preface

The FOLDOC[1] dictionary of computing defines C as:

> *A programming language designed by Dennis Ritchie at AT&T Bell Labs ca. 1972 for systems programming on the PDP-11 and immediately used to reimplement Unix. C is often described, with a mixture of fondness and disdain, as "a language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language".*

And then continues to define a (programming) methodology as:

> *(a) An organised, documented set of procedures and guidelines for one or more phases of the software life cycle, such as analysis or design. Many methodologies include a diagramming notation for documenting the results of the procedure; a step-by-step "cookbook" approach for carrying out the procedure; and an objective (ideally quantified) set of criteria for determining whether the results of the procedure are of acceptable quality. An example is The Yourdon methodology.*
>
> *Or (b): A pretentious way of saying "method".*

This course attempts to teach some aspects of C programming, and programming methodology. At the end of the course a student should know many of the useful features of the C language, and be able to produce a program using professional programming techniques.

[1] The Free-On-Line-Dictionary-Of-Computing is found at http://wombat.doc.ic.ac.uk/foldoc/index.html

On the lectures and notes...

Throughout the lectures, I will try to follow four principal threads:

1. Programming in C,
2. Computer science,
3. Software engineering techniques and methods including ‘good practice’, and
4. Useful information

The following four symbols are used throughout the notes, and indicate the four principal threads. The icons normally appear in the margins of the notes.

This icon indicates that this section is about C programming.

This one indicates that a “Computer Science” topic is to be explored.


This one represents an engineering aspect of the discussion, and finally,

This one represents an informative section (history, how to use a tool and so on)

In addition, code listings, specifications, algorithms and so on are consistently presented in framed, labelled boxes. The *computer science* content of this course is introduced in a gradual manner, leaving in-depth analysis until later.

Thanks to Prof Tony Adams (USP, Fiji Islands) for his C++ notes, and useful ideas.

Contents

Chapter 1

Introduction

Officially,[1] the aim of this module is to

> *"introduce students to the discipline of computing and to the problem solving process. The module stresses on good program design, and programming styles, and structured program development using a high-level programming language. Some basic concepts in procedural abstraction, structured programming and top-down design with stepwise refinement will be introduced. Topics to be covered include: algorithm design process, program development/coding/debugging, programming concepts in a high-level language including program structure, simple data types and structured types, and various control structures (sequencing, loops, conditional, etc.), and linear data structures, such as arrays and linked-lists. The utility of recursion will also be highlighted using a variety of sorting algorithms. Laboratory work is essential in this course".*

Have you got that? That is the official view, but I can see a different way of looking at this. From a student-oriented point of view, the aims could be:

1. to survive the semester, and
2. to learn some useful stuff about programming along the way.

And to *pass* of course!

In this first chapter, we begin by giving some initial tips on how to survive this course, some relevant history about C, and we make some observations about programming in general.

1.1 How to survive

The main advice I can give you is to **work consistently**. If you leave assignments till the last moment, or walk into laboratories unprepared, you will feel continually pressured. If you have difficulty understanding something it often becomes hard to work,[2] but the remedy is simple - **ask**.

[1] This description is extracted verbatim from the NUS SoC description of core modules found at http://www.comp.nus.edu.sg/~cmcurric/chapter11.PDF.

[2] I'm stuck here - what can I do now?

Programming is a practical subject. You will need to write, type in, correct and run programs. Attending lectures, and hoping to pass the final exam may work fine in some subject areas, but not this one. The good news is that you will be able to use your computer at home to develop your programming skills.

Students who do well do the following things:

- attend laboratory classes as well as lectures and tutorials,
- start laboratory assignments as soon as they are given out,
- do the tutorial exercises, and
- seek help if things get difficult.

When you get help from your friends or tutors, make sure that you understand it (for instance, if you have had help with an assignment you should be able to do it again on your own a couple of weeks later).

Aaron Tan has assembled a range of documents related to *surviving* on the web site at

- http://www.comp.nus.edu.sg/~cs1101c/5_tips/tips.html.

1.1.1 Assessment and grading

The following assessment weightings will be used:

Assessment	Weighting	
Tutorials	10%	(1/3 attendance, 2/3 presented material)
Laboratories	15%	(Taken from CourseMaster system)
MCQ tests (2)	20%	
Practical exam	15%	
Final exam	40%	

Laboratory work will be done using the CourseMaster automated C programming system, with a series of programming tasks that you may work on in the laboratories, or in your own time. At any time, you may use the CourseMaster interface to check your current assessment, and if you wish, re-submit to get a better score.

1.2 Resources

There are lots of resources available to you. There is no excuse that starts with “I could not find anything...”. First there are people resources, information in the form of books and Internet sites, and at least three sets of course notes.

1.2.1 People

We all prefer to get our information from people. Even if it is wrong. We are people-kind-of-people. You can talk to:

Colleagues: Your fellow students often have discovered things already. In a University, we encourage people to be free with their knowledge - so if you know something, share it with your colleagues. If you want to know something and the student on the next table seems to know, introduce yourself and **ask**.

Tutor, Lab assistants: Should be knowledgable in many of the areas covered, and if they can't answer your questions, they will find out!

Lecturer: I'll try to make myself available at fixed times. You may also e-mail me if you are still having problems after consulting your tutor.

Woman you met in bus: Before accepting what she says, ask for her name. If it is Thompson or Ritchie...

1.2.2 Internet

Aaron Tan: has developed a wonderful web site here at NUS with lots of pointers to various resources related to both survival, programming methodology and C.

BBS: The SoC BBS often has an active site devoted to cs1101c.

Various: There are literally hundreds of "learn C programming" courses to be found on the Internet, with a range of quality levels. You may wish to look around at them.

Announcements via e-mail: If I have announcements to make, they will be sent to you at your official NUS e-mail address. You are advised to read your e-mail as often as you can. If you have an external account, ensure that your NUS mail is redirected there.

1.2.3 The course text

C How To Program (3rd edition) - Introducing C++ and Java

by H.M. Deitel & P.J. Deitel, Prentice-Hall, ISBN 0-13-089572-5.

You are strongly urged to get the textbook. It is available at the Forum co-operative bookstore.

1.2.4 Course notes

There are at least three sets of course notes available to you at the course home site:

1. These notes are my lecture notes, and they are just an expanded version of the overheads used in class. You will need to read the textbook, and other provided course material to find out detailed information about any of the topics.
   http://www.comp.nus.edu.sg/~cs1101c/hugh/cs1101c.pdf
2. Aaron Tan's well-developed set of course notes should be read as well.
   http://www.comp.nus.edu.sg/~cs1101c/2_resources/lectaaron.html
3. The CourseMaster system also comes with (yet another) set of C programming course notes.
   http://www.comp.nus.edu.sg/~cs1101c/CourseMaster/

The overheads used in class are available at:
http://www.comp.nus.edu.sg/~cs1101c/hugh/foils.pdf

1.3 C background

The C programming language was called "C" because many features derived from an earlier compiler named "B", which in turn was developed from BCPL. BCPL was a simpler version of a language called CPL, which owed many of its features to a language called Algol 60.

The names most associated with this early development of C were Ken Thompson and Dennis Ritchie. Both were employed at Bell Telephone Laboratories and were the chief developers of the UNIX operating system. Ritchie has written a short history of C which you may find interesting. It is found at http://cm.bell-labs.com/cm/cs/who/dmr/chist.html.

UNIX has been distributed freely in the academic world since those early days, and since every UNIX system was written in C, and came with free C compilers, it became immensely popular. Even now, nearly 30 years after it was developed, C is still the main language used in systems and applications programming. The main features of C are:

- **Simplicity** - C has a relatively small number of components, although some programmers say it has too many!
- **Efficiency** - C has commands that are directly oriented towards the low level instructions, and can often produce very fast and small code. When C was developed, computers typically had 8KB of memory (8192 bytes), and so code size efficiency was a great concern.
- **Portability** - C and UNIX are available on virtually every platform.

Brian Kernighan and Ritchie published the book "The C Programming Language" in 1978, effectively setting the standard for C. This standard became ANSI C after some modifications. C itself has spawned other languages - Concurrent C, Objective C, and in 1986, C++. All of these languages accept most elements of ANSI C, but provide extra facilities.

1.3.1 C compilers and other tools

C compilers are traditionally called **cc**. There are a range of C compilers available for you to use. Here is a list:

UNIX-based: Most UNIX systems come with a C compiler called **cc**. The GNU C compilers are considered the best ones, and are available in various forms on various machines. You can use them on the UNIX systems at NUS, or if you have Linux installed on your PC you may use them on your home machine. The compiler is commonly called **gcc**, or **g++** for the language C++.

DOS-CYGWIN: There are two main ports of GNU C available for DOS/Windows systems - one is called CYGWIN, and the other DGJPP. Either may be downloaded for use. The CYGWIN distribution home site is: http://sources.redhat.com/cygwin/. Note that there are many places that sell CYGWIN, but it can be downloaded legally for free. A copy of CYGWIN for use with CourseMaster at NUS is available on the web site, along with a programmer's editor.

DOS-DJGPP: DJGPP is a free compiler for C and C++, along with support tools. It is used on MS-DOS/MS-Windows machines. The originator of DJGPP is DJ Delorie (hence the name), and the compiler is available at http://www.delorie.com/djgpp/. There are local copies of everything here at NUS at ftp://ftp.nus.edu.sg/pub/simtelnet/gnu/djgpp/.

DOS-BCC: The Borland C command-line compiler is provided with the Deitel and Deitel book. It is on the CD in the back, although when I tried to install it on my Win98 PC, I had to find the install program and execute it directly, as the install program on the disk didn't seem to work - it did not appear to interpret the extended file names correctly. The file to run is: D:\BORLAND\CBUILD~5\COMMAN~5\FREECO~6.EXE

1.4 On giving instructions

If I run into a room with an axe, everyone has enough sense to get out of the way. Even the ants and spiders will move. However a computer will not move at all. From this we can see that a computer's view of the world is quite different from that of living things.

We sometimes have difficulty dealing with the "literal monster" - the difficulty is not that it is complicated, but quite the reverse. You will not find it easy learning to deal with a complete idiot.

1.4.1 Ambiguity

A first problem is that of ambiguity. When we hear something, we interpret it differently according to context. For example:

"Stay away from the bank!"

This could be

1. A Dad telling off his 5 year-old son - meaning that there is a dangerous bank, or...
2. The Bank manager instructing her (ex) employee to stay well away from the Bank, or...

This sort of ambiguity is simple, and can be resolved fairly quickly by finding out what sort of “bank” is being referred to. There are more complex forms - for example consider this sentence:

> “Woman without her man is nothing”

Does this mean

1. “Woman, without her man, is nothing.” - or - does it mean
2. “Woman! Without her, man is nothing.”

The computer is too simple to deal with this level of structural ambiguity, and so we tend to use very specific computer languages which are insensitive to context. These languages are called *context-free*.

1.5 Stepwise refinement

If you were asked to organize the catering for a wedding, you may decide that you have a set of tasks to perform:

1. Make up a guest list
2. Invite the guests
3. Select an appropriate menu
4. Book church/hall/Minister ... and so on.

Each one of these tasks may themselves be split into component tasks:

1. Make up a guest list
   (a) Get list from Groom
   (b) Get list from Bride
   (c) Check for conflicts
      i. Bride with Groom’s list
      ii. Groom with Brides’s list
      iii. Final lists with Parents...
2. Invite the guests .... and so on

This general strategy of subdividing tasks is commonly performed when programming, and the process is known as **stepwise refinement**. It is most effective when tasks are independent. It tends to be less effective when tasks have strong inter-relationships:

Guestlist large ⇒ Book larger hall
⇒ Cost much greater
⇒ Reduce number of guests
⇒ Book smaller hall...

1.6 Summary of topics

In this section, we introduced the following topics:

- Survival techniques - resources and staying with the class
- Resources - colleagues, the Internet, books and teaching staff
- Some C history - there is lots more - have a read!
- Tools you can use for free - and ones you can buy
- Ambiguity and the general type of computer languages
- Subdividing problems - a useful technique with a fancy name

---

Questions

1. In what computer language were early C compilers first written?In what computer language are C compilers written now?
2. What is UNIX?
3. What is ambiguous about each of the following sentences? What type of ambiguity is demonstrated?

   (a) POLICE BEGIN CAMPAIGN TO RUN DOWN JAYWALKERS
   (b) DRUNK GETS NINE MONTHS IN VIOLIN CASE
   (c) SQUAD HELPS DOG BITE VICTIM
   (d) MINERS REFUSE TO WORK AFTER DEATH

4. Devise a set of instructions to help someone read the time from an analog clock. Play “devil’s advocate” with your instructions, and find an error.
5. What is ANSI?
6. What is GNU, and the GNU copyleft?
7. Find the sources for a C compiler, and count the lines in it. How did you do it?
8. Differentiate between a compiler and an interpreter

Further study

- Textbook, chapter 1, sections 1.1 to 1.18
- Textbook Chapter 2, sections 2.1 to 2.5
- Self-review exercises, chapter 1
- Aaron’s notes for Lecture 1 at http://www.comp.nus.edu.sg/~cs1101c/lect/lect01color.doc

---

Chapter 2

Programming

At this stage, a distinction is made between an algorithm, and code.

An algorithm is a set of instructions that describe a method for solving a problem. An algorithm is normally given in some mix of computer code and English. This is often called pseudo-code. When an algorithm is discussed in this text it will look like the following example:

Algorithm: 1 *Enjoying a cup of tea with a tea bag*

```
put tea bag in cup
pour boiling water over tea bag
jiggle for about 20 seconds
if like milky tea
  add milk
else
  add slice of lemon
drink
```

A code-listing (program,procedure,function,method) is an actual implementation of some algorithm. It can be compiled and executed, either by itself - as in most examples in this text, or in conjunction with other code-listings. When code is discussed in this text it will look like the following example:

CODE LISTING **HelloSingapore.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        01/11/2000
   Description: This is an initial 'Hello World' program
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    printf ("Hello Singapore!\n");
}
```

2.1 Hello Singapore - <printf>

Here we introduce our first real C program. As you can see - it is in a code-listing box, not the algorithm-box.

CODE LISTING **HelloSingapore.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        01/11/2000
   Description: This is an initial 'Hello World' program
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    printf ("Hello Singapore!\n");
}
```

- **Comment:** The first lines of the program are an (optional) comment. Comments may be either done using the symbol // (two forward slashes - which tell the C compiler to ignore the rest of the line) or by placing your comments between the **/*** and ***/** symbols. You are to put header comments something like the above one in every program you submit during this course.
- **main():** The main part of the program to be run (we say executed!) is always called **main**. This is just a tradition. It could have been called **program** or **starting-place** or just about anything, but the designers decided to use the word **main**. Not **Main**. not **MAIN**. Not **MaIn**. In C, the case (uppercase/lowercase) matters, and so **Main** and **main** are different things. So it is: **main**.
- **Body:** The body of the program contains a single statement which prints out the message **Hello Singapore!** followed by a **newline** character. The **printf** is called a statement or a command.
- **Format:** All the other brackets and semicolons are needed by the compiler so it can tell unambiguously what you want to do - there will be more explanations later in the course.

The <printf> construct:

The C statement printf can be used a few different ways, but this way is the simplest. The string within the quote marks is printed out on the console (the same place where you typed the command). The "\n" indicates that a new line is added to the output - shifting succeeding outputs onto the next line. The format given here for printf is:

printf (<string>) ;

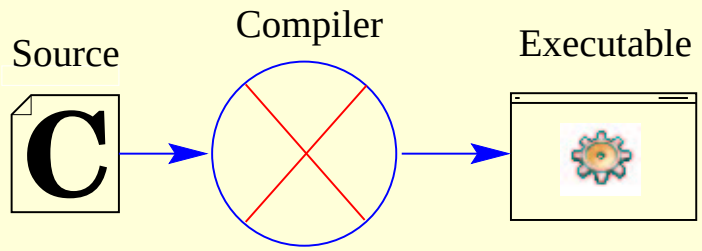
On a UNIX system, the online help can give you detailed information about each C programming construct. - If you use the command "**man printf**", you will get something like this:

```
NAME
   printf - write formatted output
SYNOPSIS
   int printf(const char *format, ...);
DESCRIPTION
   The functions in the printf family produce output
   according to a format as described below. The
   functions printf and vprintf write output to stdout,
   the standard output...
```

These manual pages take a little getting used to, but are normally a quick way to find out details of commands.

2.1.1 Compilation

This first C program is a **source** file. That is, it is written in the C source language. It now has to be turned into a form that the computer can use - an **executable**. The process of converting a source file into an executable program is called *compilation*, and it is performed by a *compiler:*



The compiler splits the program into the parts that are important to it - called tokens. For our first program, the tokens would be:

main	(	)	{	printf	(	“Hello Singapore!\n”	)	;	}

We could write our program just like this:

```
main(){printf(“Hello Singapore!\n”);},
```

and the program would compile and run accurately. However, we do not do this. It is common programming practice to try as much as possible to make our programs readable. Here is an example of correct, but extremely hard to read code:

CODE LISTING **horrible.c**

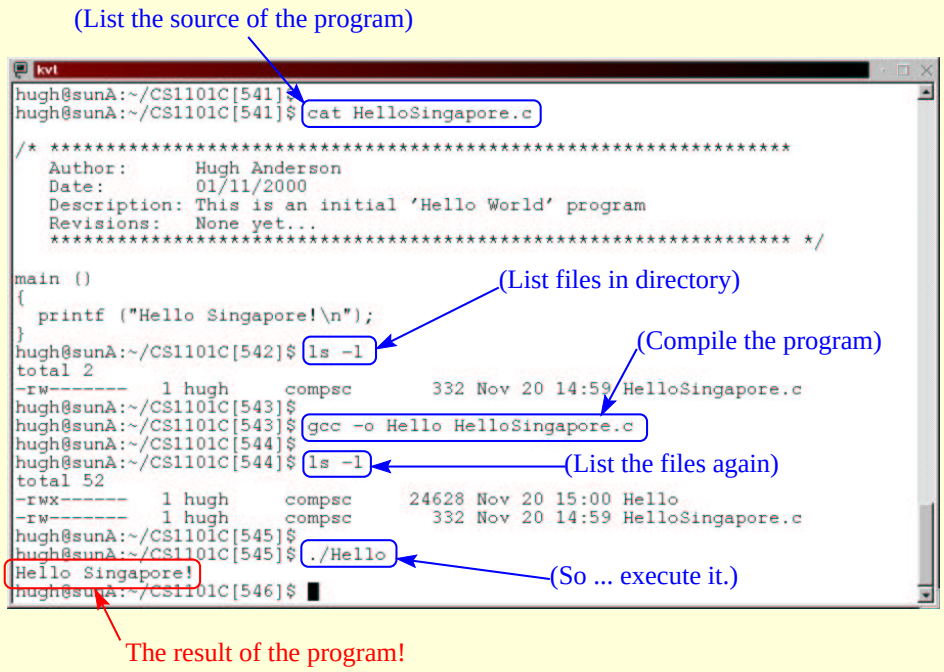
```
float s=1944,x[5],y[5],z[5],r[5],j,h,a,b,d,e;int i=33,c,l,f=1;int g(){return f=
(f*6478+1)%65346;}m(){x[i]=g()-l;y[i]=(g()-l)/4;r[i]=g()>>4;}main(){char t[1948
]=" 'MYmtw%FFlj%Jqig~%'jqig~Etsqnsj3stb",*p=t+3,*k="3tjlq9TX";l=s*20;while(i<s)
p[i++]='\n'+5;for(i=0;i<5;i++)z[i]=(i?z[i-1]:0)+l/3+!m();while(1){for(c=33;c<s;
c++){c+=!((c+1)%81);j=c/s-.5;h=c%81/40.0-1;p[c]=37;for(i=4;i+1;i--)if((b=(a=h*x
[i]+j*y[i]+z[i])*a-(d=1+j*j+h*h)*(-r[i]*r[i]+x[i]*x[i]+y[i]*y[i]+z[i]*z[i]))>0)
{for(e=b;e*e>b*1.01||e*e<b*.99;e-=.5*(e*e-b)/e);p[c]=k[(int)(8*e/d/r[i])];}}for
(i=4;i+1;z[i]-=s/2,i--)z[i]=z[i]<0?l*2+!m():z[i];while(i<s)putchar(t[i++]-5);}}
```

This code comes from an interesting web site devoted to the worst-of-the-worst-of-the-worst C code. The International Obfuscated C Code Contest at http://www.ioccc.org/. C programmers from around the world compete to write the worst C code every year. You may find it interesting to find out how *not* to write C.

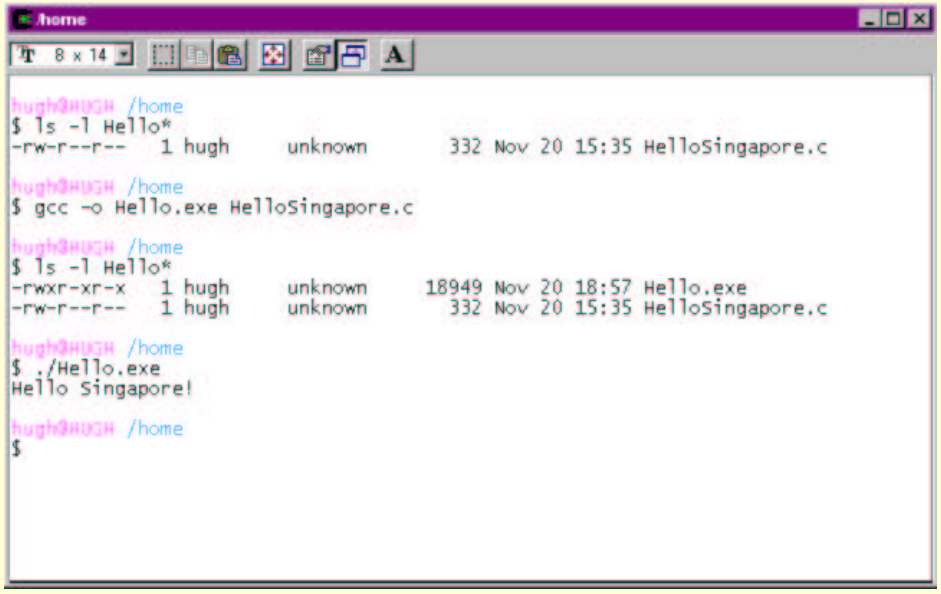
2.1.2 Compile and execute

Lets see a compilation, and first execution of this program on each of the three platforms mentioned in section 1.3.1.

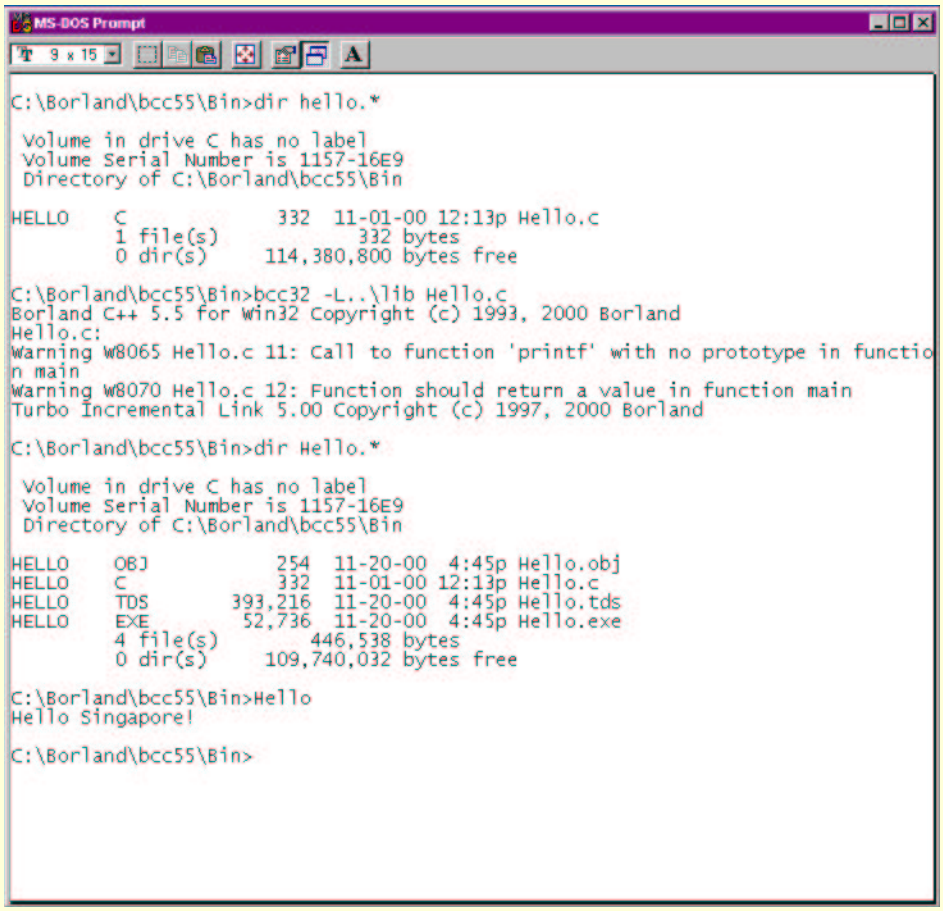
Here is an example of a compilation using the GNU C compiler on one of the UNIX machines here at NUS. The commands that I typed are marked with little rounded boxes.



We get similar results using the Cygwin environment under windows:



If you have purchased the Deitel and Deitel book, it may have a CD in the back, which contains a command line version of Borland's C compiler. This may also be used, and looks like this:



2.2 Specification of programs

The format followed for specification of programs in these notes is similar to that used by the Course-Master system. A specification looks like this:

Specification 1: Convert inches to feet and inches		**inchtofeet**
Description:	A complete program to convert an input 'inches' number to two - 'feet' and 'inches'. The values must be 'printed'.	
Preconditions:	None	TRUE
Postconditions:	The new 'feet' value times 12 plus the new 'inches' value is the same as the old'inches' value.	$(f * 12) + i = i'$
Hints:	Use two (or three) integer variables. Perhaps search for MODULO, and use it.	
Sample input/output:	suna> **inchtofeet** **Please enter inch value => 46** **46 inches is 3ft 10in.** suna>	

Title line: - includes a brief description, and the required name for the routine or executable.

Description: - a short English language description of the programming task.

Preconditions: - the minimum requirements for the code to work correctly. In the preceding sample, the program is expected to work in all circumstances - there are no preconditions. More formally we might say that the precondition is TRUE.

Postconditions: - the expected result after the program or code segment is run. This may be given informally or more formally (as shown on the right of the specification above). A short form for formally specifying this program is: $i, f : [\text{TRUE}, (f * 12) + i = i']$.

Hints: - Here are brief hints which can help you design and structure your program.

Sample input/output: This may take the form of a sample run of the program (as shown above), or it may just be a brief discussion of the likely test data and expected results. If a sample run is given, your program MUST perform exactly as the sample run does.

The CourseMaster system in addition has an "Assessment" field which gives you hints as to how the automatic assessment of your program is done, and how to get the best mark.

2.3 Further C language elements

The following sample programs introduce more statements from the C language. We follow a fixed format, starting with a specification, continuing with the code, and then concluding with a brief description of the introduced C programming construct.

Note: You must look elsewhere for *detailed* definitions and information about the constructs introduced in this section.

2.3.1 Square - <scanf>

Here is another specification - this one for a program to calculate the square of a number. We present the complete program as a pair - a specification followed by an implementation (code-listing).

Specification 2: Square a number		**sqrnum**
Description:	A complete program to calculate the square of a given input number. The values must be 'printed'.	
Preconditions:	None	TRUE
Postconditions:		
Hints:	Use <scanf>	
Sample input/output:	suna> **sqrnum** **Please enter number => 6** **6 squared is 36.** suna>	

CODE LISTING **sqrnum.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to display the square of
                an input number
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int inputnum;
    printf ("Please enter number ==> ");
    scanf ("%d", &inputnum);
    printf ("%d squared is %d.\n", inputnum, inputnum * inputnum);
}
```

Unfortunately, we have had to introduce some new things into our program. They are briefly described below.

Variable declaration - <int>

The "**int inputnum;**" declaration tells the compiler that your program is going to want a storage place in which the program can store a value. In this case, the value is an *integer* value. Within the program we can refer to this storage location by the name we specify:

- The *name* of this storage place is **inputnum**
- Its *value* is **whatever-we-put-into-it**!

We call these declarations *variable-declarations*, and the things themselves are called *variables*. We can declare as many *variables* as needed.

More on the <printf> construct

We now also see a second way of using the C **printf** statement. The rule is:

printf (<format> , <list of values/variables>) ;

The <format> part is a string which contains information about how to print the values or variables. Within the string, the %d specifier indicates that a number is to be printed in decimal. For example:

Statement	Result
printf("%d",6+12);	18
printf("The sum of %d and %d \n is%d\n",6,12,6+12);	The sum of 6 and 12 is 18

The \n specifier indicates that a new line is to be started. Here is a short summary of the most important format specifiers:

Specifier	What it does
\n	newline
%s	prints a string
%d	prints a decimal value
%x	prints a hexadecimal value

Getting input - <scanf>

A complementary function to **printf** is **scanf**. Its purpose is to read an input stream, and read input values. The **scanf** call will match the input stream with the required data types, and our initial use is very simple:

scanf(<format> , &<named variable>);

The **&** character is telling the **scanf** routine the address of the place where it should return the value to. In this case the address of some named variable.

Sequencing of C statements

If we have two or more C statements that we wish to be performed one after the other, we write them one after the other - one-to-a-line. An example:

```
printf("Hi");
printf(" there\n");
printf("Dude!\n");
```

This prints out

```
Hi there
Dude!
```

2.3.2 Sum - <assign>

Another small program - this one just being used as a way of introducing the C *assignment* statement.

Specification 3: Sum two numbers		**sum**
Description:	A complete program to calculate the sum of two given input numbers. The values must be 'printed' with the sum.	
Preconditions:	None	TRUE
Postconditions:		sum = in1 + in2
Hints:	Use <scanf> Use two or three variables	
Sample input/output:	suna> **sum** **Please enter first number => 6** **Please enter second number => 21** **The sum of 6 and 21 is 27.** suna>	

Here is a solution to this using three variables, one to hold each of the input values, and the third used to calculate the sum:

CODE LISTING **sumb.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to display the sum of
                two input numbers -this one uses three variables
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, in2, sum;
    printf ("Please enter first number ==> ");
    scanf ("%d", &in1);
    printf ("Please enter second number ==> ");
    scanf ("%d", &in2);
    sum = in1 + in2;
    printf ("The sum of %d and %d is %d.\n", in1, in2, sum);
}
```

The <assignment> construct

When a programming language has an assignment statement, we call the language a von-Neumann language, because the assignment explicitly indicates that our computer has a readable and writeable *memory* - John von-Neumann's engineering contribution to the computer. The structure of an assignment statement is:

<variable-name> = <expression> ;

The <expression> is evaluated (worked out), and assigned into the computer's memory at some location <variable-name>.

Note that we need not have used three variables - we could have just used two. Here is a solution using two variables, one to hold each of the input values:

CODE LISTING **sum.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to display the sum of
                two input numbers -this one uses two variables
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, in2;
    printf ("Please enter first number ==> ");
    scanf ("%d", &in1);
    printf ("Please enter second number ==> ");
    scanf ("%d", &in2);
    printf ("The sum of %d and %d is %d.\n", in1, in2, in1 + in2);
}
```

2.3.3 Summation - <for>

This program introduces a more complex C construct - the for-loop construct. We use this type of statement when we want to do something some number of times.

Specification 4: Calculate the sum-up-to-n		**sumton**
Description:	A complete program to ask for an input number n, and then calculate the sum from 1 to this number n. The values must be 'printed'.	
Preconditions:	The input value must be greater than 0	in > 0
Postconditions:	The result is the sum from 1 to n	result $= \sum_{i=1}^{\text{in}} i$
Hints:	Use two integer variables. Perhaps use a for loop.	
Sample input/output:	suna> **sumton** **Please enter input value => 4** **The sum from 1 to 4 is 10.** suna>	

CODE LISTING **sumton.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to display the sum from 1 to a
                specified input number
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, result, count;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    result = 0;
    for (count = 1; count <= in1; count = count + 1) {
        result = result + count;
    }
    printf ("The sum from 1 to %d is %d.\n", in1, result);
}
```

The <for-loop> construct

The structure of the for-loop statement is:

for (<beginning> ; <end test> ; <change>) { <statement> }

The <beginning> statement is done first - we often set some loop counter variable to have its beginning value here. The <end-test> expression checks if the loop is finished. The <change> part increments or changes the loop variable in some way.

If we had:

for (i=1 ; i<=10 ; i=i+1) { <statement> }

Then the <statement> would be executed 10 times.

2.4 Summary of topics

In this section, we introduced the following topics:

- Algorithms and code, and the use of pseudocode.
- Simple C constructs - including all of the following
  - Comments - commenting code using /* and */
  - The program main structure - main(<...>){<body>}
  - Declarations of simple integer variable storage spaces - int x;
  - <assignment>, <for-loop>, <printf>, <scanf>
- The compilation process - tokens, source, compilers
- Readability and code style - bad code
- Environments in which to work - CYGWIN, UNIX, DJGPP, BCC
- Specifications - informal and formal

Questions

1. Differentiate between a specification and an implementation.
2. Tabulate the size in bytes of the executables for our first listing on each of the platforms. Why are they different?
3. Write an algorithm in pseudocode for finding a word in a dictionary.
4. What is wrong with the following program? MAIN(){PRINTF(“HELLO SINGAPORE!\N”);}
5. What is wrong with the following program? main(){printF(“HELLO SINGAPORE!\n”);}
6. What is the output of this code segment: for (i=0;i<10;i++) printf(“%d\n”,i);
7. Under what conditions will the following post-condition be unobtainable? $y = a/b$
8. In the form given in these notes, give a long specification of a program to calculate the hypotenuse of a right-angled triangle given the two other sides.
9. Give the previous example in the short formal specification style as well.
10. State in words the meaning of the following specification: $x, y : [x > 0, y = x + 1]$.

Further study

- Textbook, chapter 2, sections 2.5 to 2.6
- Textbook Chapter 3, sections 3.1 to 3.8

Chapter 3

More structure

The two terms “syntax” and “semantics” are often mentioned in relation to programming languages. Some definitions:

Syntax: The structure of components of some language. It is relatively easy to get the syntax correct in our programs - we just follow a simple set of rules. A language’s syntax is described by a grammar.

Semantics: The *meaning* of a string (or sentence) in some language.

It is much harder to understand the semantics of a program, than it is to verify the correct syntax.

3.1 A useful tool - indent

A useful tool which can tidy up (or pretty-print) messy programs is the **indent** tool available on many systems. This tool takes any C, and re-formats it according to some set of rules. The tool is very flexible, but even without setting any of the options, it normally formats C programs quite nicely. For example:

CODE LISTING **formatted.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        01/11/2000
   Description: This is a test file
   Revisions:   None yet...
   **************************************************************** */
#define _REENTRANT
#include <iostream>
#include <pthread.h>

int counter = 0;
int turn = 10;

void *many (void *arg)
{
  pthread_t tid;
  cout << counter << ":";  counter = counter + 1;
  if (--turn > 0) {/* decrement turn by 1 */
pthread_create (&tid, NULL, many, (void *) 0);
cout << turn << endl;pthread_join (tid, NULL);}cout << endl;}

main (){  many (NULL);}
```

And after using the indent command: **indent formatted.c**

CODE LISTING **formatted1.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        01/11/2000
   Description: This is a test file
   Revisions:   None yet...
   **************************************************************** */
#define _REENTRANT
#include <iostream>
#include <pthread.h>

int counter = 0;
int turn = 10;

void *
many (void *arg)
{
    pthread_t tid;
    cout << counter << ":";
    counter = counter + 1;
    if (--turn > 0) {           /* decrement turn by 1 */
        pthread_create (&tid, NULL, many, (void *) 0);
        cout << turn << endl;
        pthread_join (tid, NULL);
    }
    cout << endl;
}

main ()
{
    many (NULL);
}
```

There are some points about this program. Firstly - the specific formats chosen are adjustable. If we use different parameters to the indent command, the program will be formatted differently. In this case, every command is on a different line, the target of the **if-statement** is indented, and the indenting is consistent throughout the program.

3.2 More C constructs

In this section we introduce three new C programming constructs. The **function** helps us to create more usable and readable programs. The **compound-statement** allows us to group a series of statements together and treat them as if they were one statement. Finally a statement for repetitively doing things - the **while-statement**.

3.2.1 The function

We have already met the stepwise refinement technique, and the C programming language supports this technique directly through the use of the *function* or procedure call. An example of this may be that if we were given a programming task which involved reading in some values, calculating some intermediate results, and then printing out average values, we can write this as three separate functions.

We give each function a meaningful name, such as readin, calculate and printout. We can then write the mainline of our program immediately without concerning ourselves with the contents of each of these three functions. This is exactly the tool we need to implement *stepwise refinement*.

We begin by writing the main line of the program as shown below:

CODE LISTING **thefunction.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of functions
   Revisions:   None yet...
   **************************************************************** */

void
readin ()
{
    /* Nothing here yet... */
}

void
calculate ()
{
    /* Nothing here yet... */
}

void
printout ()
{
    /* Nothing here yet... */
}

main ()
{
    readin ();
    calculate ();
    printout ();
}
```

We have now subdivided our task into three smaller tasks, which we address in turn. A C program may only have a single main, but may have many functions. The following points give guidelines for the use of functions:

1. C functions may have anything from 1 to 50 lines of code in them, but if they get too large, you should consider dividing them - in turn - into smaller functions.

2. A function should perform a single sensible subdivision of the required activity - not half of one activity and half of another.

When we do this sort of program development, we begin by writing down the mainline, subdividing our tasks into some set of smaller tasks. We can check and compile that this mainline works correctly, by just using empty functions. The term commonly used for these empty functions is *stub* procedure or *stub* function.

3.2.2 Compound-statements

Statements may be simple ones, such as the ones we have already met, or they may be compound statements. A compound statement is made up from a series of simple statements, grouped within curly braces. Here's an example:

CODE LISTING **compund1.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of compound
                Statements
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, result, count, old;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    result = 0;
    for (count = 1; count <= in1; count = count + 1) {
        old = result;
        result = result + count;
        printf (" Adding %d to %d gives %d\n", count, old, result);
    }
    printf ("The sum from 1 to %d is %d.\n", in1, result);
}
```

In this example, we have added lines of code to help us discover a possible error in the program.

3.2.3 The while-loop

We now introduce a new C programming constructs called **while**. The **while-statement** executes its body as long as some condition is true.

Here is an example of the use of the **while** programming construct:

Specification 5: Calculate the sum-up-to-n repeatedly		**while1**
Description:	A complete program to ask for an input number n, and then calculate the sum from 1 to this number n. The values must be 'printed'. t repeats until n is 0.	
Preconditions:	The input value must be greater than 0	in > 0
Postconditions:	The result is the sum from 1 to n	result $= \sum_{i=1}^{\text{in}} i$
Hints:	Use two integer variables. Perhaps use a for loop.	
Sample input/output:	suna> **sumton** **Please enter input value => 4** **The sum from 1 to 4 is 10.** **Please enter input value => 5** **The sum from 1 to 5 is 15.** **Please enter input value => 0** suna>	

CODE LISTING **while1.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                while statement
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, result, count;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    while (in1 != 0) {
        result = 0;
        for (count = 1; count <= in1; count = count + 1) {
            result = result + count;
        }
        printf ("The sum from 1 to %d is %d.\n", in1, result);
        printf ("Please enter input value ==> ");
        scanf ("%d", &in1);
    }
}
```

In the code listing, we continue to execute the statements until the user enters in the number 0. This code listing is perhaps not the best way to write this code, as we have had to repeat the same two lines in two different places within the program. Can you suggest a better way to write this program?

The syntax for the while loop is:

while (<condition>) <statement> ;

In the example program given above, the statement that belongs to the while loop is a compound statement.

Equivalence of the while and for loops

We have now met two similar C programming constructs - the **while-loop**, and the **for-loop**. They appear to do the same thing, for example if we wished to, we could use a **for-loop** to print out 9 “Hello Singapore!” messages or we could use the **while-loop.** So why do we have two different techniques for doing the same thing? There is no clear answer for this.

- If you know exactly how many loops you wish to do, you should use the **for-loop**.
- If you do not know how many loops, you should use the **while-loop**.

Let us look at two different samples. In the first example we will calculate an average from three input numbers:

Specification 6: Calculate the average		**for1**
Description:	A complete program to ask for three input numbers, and then calculate the average.	
Preconditions:	None	TRUE
Postconditions:	The result is the average	result $= (in_1 + in_2 + in_3)/3$
Hints:	Use integer variables. Perhaps use a for loop.	
Sample input/output:	suna> **for1** **Please enter input value => 4** **Please enter input value => 5** **Please enter input value => 9** **The average is 6.** suna>	

CODE LISTING **for1.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                for statement
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, result, loops;
    result = 0;
    for (loops = 0; loops < 3; loops = loops + 1) {
        printf ("Please enter input value ==> ");
        scanf ("%d", &in1);
        result = result + in1;
    }
    printf ("The average is %d.\n", result / 3);
}
```

In our second example we read until the input number is 0. In this case we use the while loop.

Specification 7: Calculate the average		**while2**
Description:	A complete program to ask for input numbers, and then calculate the average.	
Preconditions:	None	TRUE
Postconditions:	The result is the average	result $= (\sum_1^n in)/n$
Hints:	Use integer variables. Perhaps use a for and a while loop.	
Sample input/output:	suna> **for1** **Please enter input value => 5** **Please enter input value => 9** **Please enter input value => 0** **The average is ==> 7.** suna>	

CODE LISTING **while2.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                while statement
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, result, count;
    result = 0;
    count = 0;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    while (in1 != 0) {
        result = result + in1;
        count = count + 1;
        printf ("Please enter input value ==> ");
        scanf ("%d", &in1);
    }
    printf ("The average is ==> %d\n", result / count);
}
```

Alert! What is wrong with this program?

3.2.4 Libraries

Large parts of many programs do the same things over and over again. Commonly used routines are put in libraries, which may be linked in at run time. There are three distinct advantages:

1. Saving on disk space
2. Routines can be updated
3. Routines are more reliable

For example: All 'C' routines (clearscreen etc) are kept in a common place (*libc.a* in a UNIX system). When you run a program, the OS uses the latest version of *libc.a* automatically.

Note - Most compilers can also statically link in the needed libraries if necessary. The executables are larger.

Note - In general you should not bypass library or system calls. If you do, your programs date quickly.

When we use a library routine, it saves us having to create a lot of new code. For example in the following code listing, we reuse system libraries to write a graphical application:

CODE LISTING **qt.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        01/11/2000
   Description: This is an initial 'Hello World' program for Qt
   Revisions:   None yet...
   **************************************************************** */

#include <qapplication.h>
#include <qpushbutton.h>

int
main (int argc, char **argv)
{
    QApplication a (argc, argv);

    QPushButton hello ("Hello world!", 0);
    hello.resize (100, 30);

    a.setMainWidget (&hello);
    hello.show ();
    return a.exec ();
}
```

We specify header files which define the functions which we want to use, and when we compile, we instruct the compiler where to find the library:

```
g++ -o qt qt.c -lqt
```

3.3 Summary of topics

In this section, we introduced the following topics:

- Syntax and semantics
- The use of an automatic tool to pretty-print your C (indent)
- Some more C constructs
- The use of libraries

Questions

1. Differentiate between a syntax error and a semantic error.
2. Which of the following sentences follow this grammar rule?
   <sentence> ::= <noun> <verb> <noun>

   (a) CAR GO RUN
   (b) DOG EAT DOG
   (c) CATS AND DOGS
   (d) FOOD SLEEP RAIN

3. How would you make the indent program turn compound statements from

```
for (...)
{
   code
}
```

into...

```
for (...) {
   code
}
```

Further study

- Textbook Chapter 3, sections 3.9 to 3.11
- Textbook Chapter 4, sections 4.1 to 4.12
- Textbook Chapter 5, sections 5.1 to 5.15

Chapter 4

Getting our feet wet

Iteration is a name used to describe what our for-loop did in section 2.3.3. Recursion is a similar technique for performing repetitive tasks.[1] However the two techniques are normally used at different times - even though they are similar. A recursive computation may normally have speed overheads - it will be slower - due to the normal way in which recursion is implemented in a computer. C programmers normally say that the iterative computation is easier-to-understand.

However, we do use a recursive technique when the task or data itself exhibits a recursive property - in this case the software is easier to write and understand if we use recursion. As an example of a task which exhibits a recursive property, imagine specifying a function on the non-negative integers such as *sum* in terms of itself by saying that the sum of n elements is the same as the sum of $n-1$ elements plus n. When we do this we say that the function is *recursive*.

$$sum(n) = sum(n-1) + n$$

When we define something in this way, we have to also specify a base case for the function if we wish to be able to calculate its value. For example:

$$\begin{aligned} sum(0) &= 0 \\ sum(n) &= sum(n-1) + n \end{aligned}$$

We may now solve the function for any non-negative integer n. Mathematicians will recognize this as a recurrence relation, and have an array of techniques for solving such relations. Later in this chapter we will give examples of both iterative and recursive versions of the same computation.

4.1 If-statement

The if-statement is used whenever you have to make a choice between two alternatives. In the above function for example, if we were calculating the result, we may use the if-statement to differentiate between the case when the n-value was 0 (in which case the result is 0), or the case that the n-value was not 0 (in which case the result is $sum(n-1)+n$).

[1] The more mathematically inclined student may wish to research 'tail-recursion'.

The syntax for the if-statement is:

if (<condition>) <statement> ;[2]

An alternative sytax is:

if (<condition>) <statement> ; else <statement>;[3]

Here is an example of the use of the if-statement:

Specification 8: Big or small?		if1
Description:	A complete program to ask for an input number n, and then tell you if it is larger or smaller than 10	
Preconditions:	The input value must be greater than 0	in > 0
Postconditions:		
Hints:	Perhaps use a while loop. Perhaps use an if - else statement.	
Sample input/output:	suna> **if1** **Please enter input value => 4** **It is smaller than 10!** **Please enter input value => 10** **It is 10 or bigger!** **Please enter input value => 0** suna>	

CODE LISTING **if1.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                if statement
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    while (in1 != 0) {
        if (in1 >= 10) {
            printf ("It is 10 or bigger!\n");
        } else {
            printf ("It is smaller than 10!\n");
        }
        printf ("Please enter input value ==> ");
        scanf ("%d", &in1);
    }
}
```

4.2 Functions

The function is a fairly clear mathematical concept, embodying the idea of a mapping from the domain to the range of the function.

[2] I think it is better to use **if (<condition>) { <statements> }**

[3] I think it is better to use **if (<condition>) { <statements> } else { <statements }**

For example, a well known function is the $\sin$ function - perhaps used like this:

$$y = \sin(x)$$

mapping a real value x to a real value y. Note that the domain and range of the function are not necessarily the same.

In the above, we have used the terms domain and range, but you may be happier with the terms input and output. In C programming, we may define our own sections of code, and they are commonly called functions. However - they may not have inputs, and also may not have outputs! The term has stuck in the C programming world, even when the C routines do not return output values. You may also hear the following terms used in a loose fashion - methods, routines and procedures - they all may be used to refer to much the same thing - the programming construct where we define a section of code, so that later we may refer to it by name - without having to repeat all the code again. Here is an example of a C function defined so that our program is shorter - this is an example of code **re-use**:

Specification 9: Big or small?		if2
Description:	As for specification 8!	
Preconditions:	The input value must be greater than 0	in > 0
Postconditions:		
Hints:	Use a function	
Sample input/output:	suna> **if2** **Please enter input value => 4** **It is smaller than 10!** **Please enter input value => 0** suna>	

CODE LISTING **if2.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                if statement
   Revisions:   None yet...
   **************************************************************** */

int in1;

void
getnum ()
{
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
}

main ()
{
    getnum ();
    while (in1 != 0) {
        if (in1 >= 10) {
            printf ("It is 10 or bigger!\n");
        } else {
            printf ("It is smaller than 10!\n");
        }
        getnum ();
    }
}
```

4.2.1 Parameters and return values

Our C functions may have a value. As an example - let us say we wished to have a function called power, which returned a value calculated from provided data x and y such that the result of the function was x^y. We have a special statement type which returns a value from the function - it is called *return*.

Here is the syntax of the return function:

return <expression> ;

We may also pass parameters to functions that we have defined. In fact many of the library functions are of this sort - that is - functions which are passed a parameter and return a value. Here is an example of the use of a function with parameters:

Specification 10: Big or small?		if3
Description:	As for specification 8!	
Preconditions:	The input value must be greater than 0	in > 0
Postconditions:		
Hints:	Use a parameter...	
Sample input/output:	suna> **if3** **Please enter input value => 4** **It is smaller than 10!** **Please enter input value => 0** suna>	

CODE LISTING **if3.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                if statement
   Revisions:   None yet...
   **************************************************************** */

void
getnum (int *val)
{
    printf ("Please enter input value ==> ");
    scanf ("%d", val);
}

main ()
{
    int in1;
    getnum (&in1);
    while (in1 != 0) {
        if (in1 >= 10) {
            printf ("It is 10 or bigger!\n");
        } else {
            printf ("It is smaller than 10!\n");
        }
        getnum (&in1);
    }
}
```

4.3 The math library

The math library contains many functions which are generally useful:

Function name	Short description
cos()	cos - trigonometric function
acos()	inverse cos function
sin() ...	sin - trigonometric function and many more...
exp()	exponential function
sqrt()	square root
cbrt()	cube root
log()	natural logarithm
log10() ...	common logarithm and many more...

You may read more detail about each function by using the manual page for that function. When using the library, you must include the math header file in the source, and link with the math library when compiling:

gcc -o dosin -lm dosin.c

Specification 11: Calculate sin value		dosin
Description:	A complete program to calculate and display the sin value of an entered value.	
Preconditions:		
Postconditions:	Calculate the sin value	output = sin in
Hints:	Use the math library	
Sample input/output:	suna> **if1** **Please enter input value => 4** **The sin of 4.000000 is -0.756802** suna>	

CODE LISTING **dosin.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                math library
   Revisions:   None yet...
   **************************************************************** */


#include <math.h>

main ()
{
    float in1, out1;
    printf ("Please enter input value ==> ");
    scanf ("%f", &in1);
    out1 = sin (in1);
    printf ("The sin of %f is %f\n", in1, out1);
}
```

4.4 Iteration and recursion

It is a little hard to understand how recursive programs work, however we will look at a simple program which calculates the product of up to n numbers using a recursive function called fac - the factorial of a number. We begin with the specification as usual, and then give both the iterative and recursive solutions.

Specification 12: factorial		fac
Description:	A complete program to calculate the factorial of a number	
Preconditions:		in ≥ 0
Postconditions:		result $= fac(\text{in})$
Sample input/output:	suna> **fac** **Please enter input value => 7** **The factorial of 7 is 5040.** suna>	

CODE LISTING **fac.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to calculate the factorial of a
                specified input number
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int in1, result, count;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    result = 1;
    for (count = 1; count <= in1; count = count + 1)
      {
      result = result * count;
      }
    printf ("The factorial of %d is %d.\n", in1, result);
}
```

CODE LISTING **fac1.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to calculate the factorial of a
                specified input number
   Revisions:   None yet...
   **************************************************************** */

int
fac (int n)
{
    if (n == 0) {
        return 1;
    } else {
        return n * fac (n - 1);
    }
}

main ()
{
    int in1, result, count;
    printf ("Please enter input value ==> ");
    scanf ("%d", &in1);
    result = fac (in1);
    printf ("The factorial of %d is %d.\n", in1, result);
}
```

4.5 Summary of topics

In this section, we introduced the following topics:

- Iteration and recursion
- The if-statement
- Functions
- Parameters and return values

Questions

1. Calculate the value for $f(4)$ for the following recursive function definition:

$$\begin{aligned} f(0) &= 1 \\ f(n) &= (f(n-1) * n) + n \end{aligned}$$

2. Write a recursive C function which calculates the sum function defined at the beginning of this chapter.

Further study

- Textbook, Chapter 4, sections 4.7

Chapter 5

Bugs 'n stuff

For many years the software industry has referred to software *errors* as *bugs*. This trivializes the issue, and gives the software developer a false sense of security. (*"Once I've removed these last few bugs, it will be perfect!"*) In reality, the software developer may stand on very shaky ground: *"Once I've removed these last few bugs, it will be:"*

- *a system that has been known to work for 11 minutes continuously... (!)*
- *a system that still contains code that I put in last month that I don't know why it is there...*

The Internet newsgroup **comp.risks** reports on *"Risks to the Public in the Use of Computer Systems and Related Technology"*, and makes interesting reading. We find that:

- The Arizona lottery never picked the number 9 (a bad random number generator)
- The LA counties pension fund has lost US$1,200,000,000 through programming error.
- Windows NT corrupts filespace and deletes directories under some circumstances.
- A Mercedes 500SE with graceful no-skid brake computers left 200 metre skid marks. A passenger was killed.

The following points may help in improving the quality of software:

- **Design software before building it** - You would not pay much for a house without plans or that had not been designed first - why would you pay for software without similar sureties?
- **Reduce the complexity** - By localizing variables, use of modularity, well defined interfaces and so on.
- **Enforce compatibility with design** - Easier to say than to do.

And lastly - call an error an ERROR!

5.1 Error-finding (debugging) tools

There are many ways to find errors in your programs, but it is much easier to not put them there in the first place (and hence the comments given before about designing before building software). There have been many studies that demonstrate that the cost of fixing an error increases the longer it is allowed to stay in a program. This is particularly true for structural or design errors. There are a range of tools and mechanisms for finding errors:

Method 1: - put in lots of printf statements throughout your program. This allows you to match the expected behaviour of your program with its actual behaviour. You can also print out the contents of important variables as the program runs.

CODE LISTING **net.c**

```
    /* Initialization... */
printf ("Trying to open socket...\n");
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
    fprintf (stderr, "server: can't open stream socket\n");
    /* The main event loop... */
for (;;) {
    clilen = sizeof (cli_addr);
    printf ("clilen is %d\n", clilen);
    ....
```

Method 2: - think[1] about it :)

Method 3: - Use a *debugging* tool such as **gdb** which allows you to

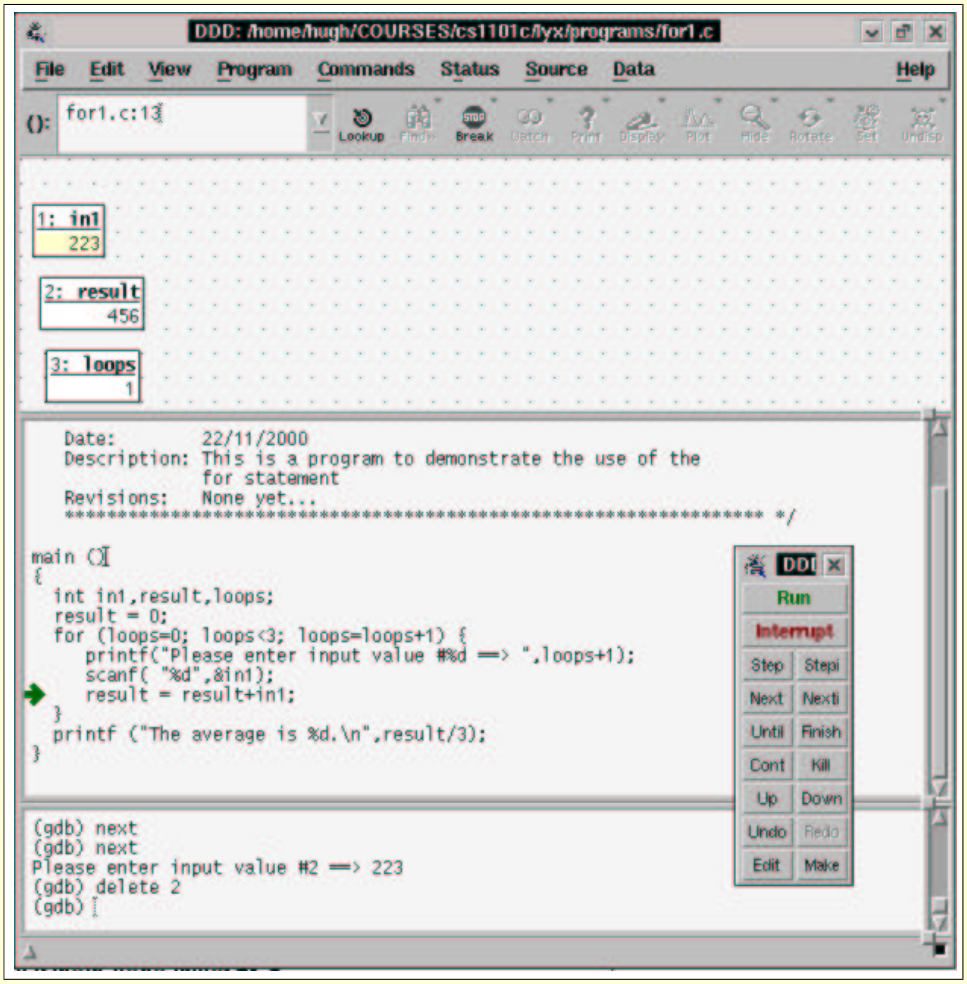
- single step programs,
- display variable values, and
- modify variable values.

A command line interface to gdb:

```
13        result = 0;
(gdb) display in1
1: in1 = 134518140
(gdb) display result
2: result = 134518120
(gdb) display loops
3: loops = 134513659
(gdb) break 17
Breakpoint 2 at 0x804845f: file for1.c, line 17.
(gdb) cont
Continuing.
Please enter input value #1 ==> 232
Breakpoint 2, main () at for1.c:17
17          result = result+in1;
3: loops = 0
2: result = 0
1: in1 = 232
(gdb)
```

[1] The best option.

A windowed interface to **gdb** - **ddd**/**xxgdb**/...

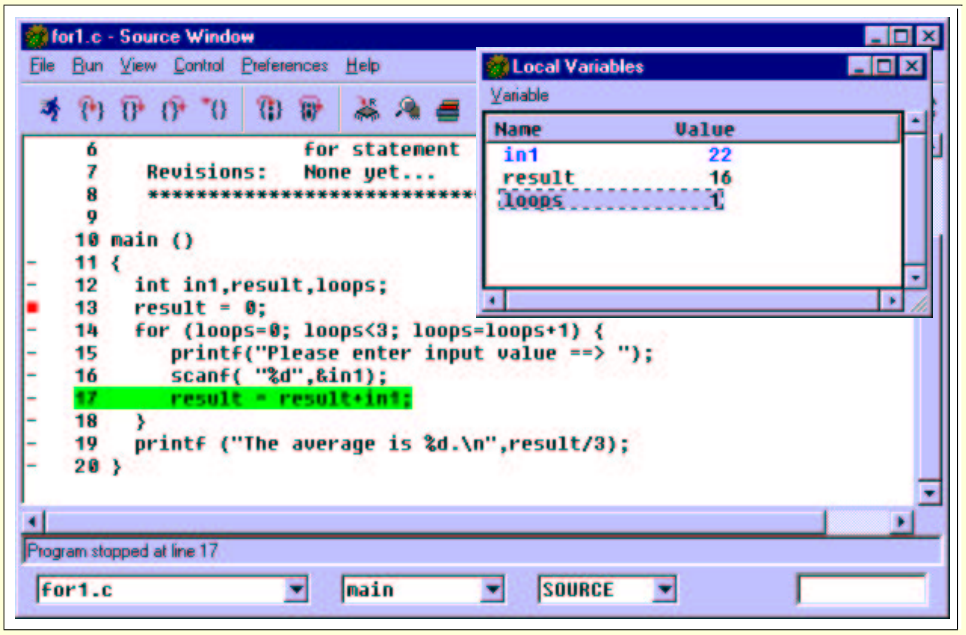


5.1.1 Insight

Here is another windowed interface to **gdb** - **insight.** This one will work on Windows platforms (as well as UNIX), and is found at http://www.comp.nus.edu.sg/~cs1101c/hugh/downloads/insight.zip. Download the file to your machine (it is about 2.5MB), and use WinZip to unzip into C:\ (or D:\. Then add the directory C:\insight\bin to your computer's PATH by modifying C:\autoexec.bat as you did before:

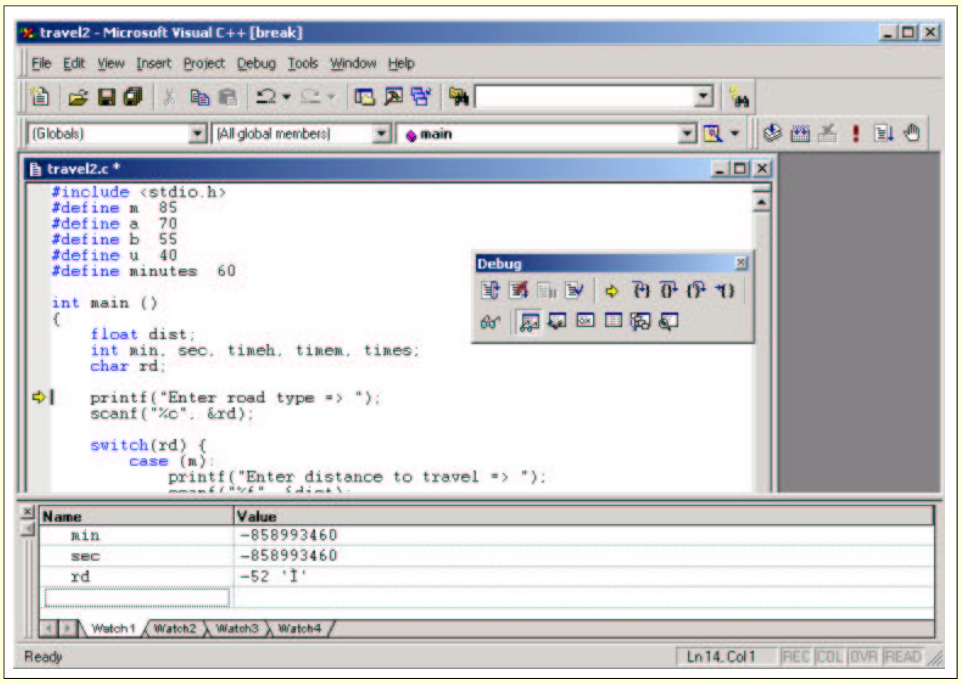
```
set PATH=C:\insight\bin;C:\pfe;C:\cmc;C:\cygnus\cygwin-b20\h-i586-cygwin32\bin;%PATH%
```

You may now run **insight** by typing **gdb** and loading up a file, or by using the latest **pfe.reg** file (which has a menu item for the debugger). Note that when you compile for **insight** you must tell the compiler that you are going to use the debugger by using the "**-g**" option: "**gcc -g -o myprog.exe myprog.c**".



5.1.2 Commercial IDE

There are also IDEs - Integrated Development Environments such as found in Microsoft or Borland products:



5.2 Manual pages

At the risk of turning this into an ‘operating systems’ course, UNIX systems include on-line help, including specifications for most C system calls. These manual pages *almost* work under CYGNUS, and are found at http://www.comp.nus.edu.sg/~cs1101c/hugh/downloads/usr.zip. Download the file to your machine (it is about 5MB), and use WinZip to unzip into C:\ (or D:\. Then modify C:\autoexec.bat, adding the directory C:\usr\local\bin to your computer’s PATH and a new line :

```
set PATH=C:\usr\local\bin;C:\insight\bin;C:\pfe;C:\cmc;C:\cygnus\cygwin-b20\h-i586-cygwin32\bin;%PATH%
set LESSCHARSET=latin1
```

You may now run the **man** command and get online help for most C functions. For example “**man printf**” will give you detailed help for the C **printf()** system call, and “**man -t printf > printf.ps**” will produce a nice (printable) postscript version - something like this:

PRINTF(3) Linux Programmer$ Manual PRINTF(3)

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf - formatted output con version

SYNOPSIS

#include <stdio.h>

int printf(const char * *format*,**.. ..);**
int fprintf(FILE * *stream*,**const char *** *format*,**.. ..);**
int sprintf(char * *str*,**const char *** *format*,**.. ..);**
int snprintf(char * *str*,**size_t** *size*,**const char *** *format*,**.. ..);**

#include <stdarg.h>

int vprintf(const char * *format*,**va_list** *ap*);
int vfprintf(FILE * *stream*,**const char *** *format*,**va_list** *ap*);
int vsprintf(char * *str*,**const char *** *format*,**va_list** *ap*);
int vsnprintf(char * *str*,**s ize_t** *size*,**const char *** *format*,**va_list** *ap*);

DESCRIPTION

The **printf** family of functions produces output according to a *format* as described belo w. The functions **printf** and **vprintf** write output to *stdout*,the standard output stream; **fprintf** and **vfprintf** write output to the given output *stream*; **sprintf**, **snprintf**, **vsprintf** and **vsnprintf** write to the character string *str*.

These functions write the output under the control of a *format* string that specifies ho w subsequent ar guments (or ar guments accessed via the v ariable-length argument facilities of **stdarg** (3)) are con verted for output.

These functions return the number of characters printed (not including the trailing ‘ \0’ used to end output strings). **snprintf** and **vsnprintf** do not write more than *size* bytes (including the trailing ’\0’), and return -1 if the output w as truncated due to this limit. (Thus until glibc 2.0.6. Since glibc 2.1 these functions return the number of characters (excluding the trailing ’\0’ which would ha ve been written to the final string if enough space had been a vailable.)

The format string is composed of zero or more directi ves: ordinary characters (not **%**), which are copied unchanged to the output stream; and con version specifications, each of which results in fetching zero or more subsequent ar guments. Each conversion specification is introduced by the character **%**

... And so on ...

5.3 Basic rules

At this stage we will revise some of the basic rules which all C programs must adhere to.

5.3.1 Reserved words

Certain words in C have special meanings and may not be redefined:

auto	**break**	**case**	**char**	**const**	**continue**
default	**do**	**double**	**else**	**enum**	**extern**
float	**for**	**goto**	**if**	**int**	**long**
register	**return**	**short**	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**	**unsigned**	**void**
volatile	**while**				

You should never re-define or use these words in any way except the intended one.

Other names we have used (such as **printf**) identify particular library functions, and may be redefined (although of course you will no longer be able to use them.)

Earlier in this course, it was mentioned that C is both a "too simple" language, and "too complex". The table above shows that the language does have a very small number of components, and thus deserves to be called simple.

However C always comes with many libraries that contain hundreds - perhaps even thousands - of routines. Names we have used (such as **printf**) identify particular library functions, and may be redefined (although of course you will no longer be able to use them.)

Note the difference between **while** and **printf()** in this context - **while** is a part of the underlying C language, but **printf()** is a library routine.

5.3.2 Rules for legal identifiers

We use identifiers to *name* our variables and functions, subject to the following rules:

1. Our identifier must not be a reserved word.
2. An identifier may consist of letters and digits and the underscore character (_).
3. Spaces are not allowed in a C identifier.
4. The first letter in an identifier must not be a digit.
5. ANSI C specifies that at least the first 31 characters of a variable name must be significant, but individual compilers have different rules.

Names that begin with the underscore character are traditionally used in libraries for values or functions, so it is better to avoid using names that begin with an underscore in simple application programs. We can choose whatever we want for an identifier name, but the main point of using an identifier is so that we can *read* our programs[2].

[2]The variable **bank_balance** is much more understandable than the variable at location **0x0112ca4e**.

This leads to some other cosmetic rules:

1. Use reasonably short meaningful identifiers.
2. It is common to use UPPERCASE for constant values and lowercase, or mixed case, for variables and functions. (for example **PI** may be **3.14159**, and **BankBalance** or **bank_balance** may be a name associated with a variable containing a bank balance.)
3. It is OK to use single-letter identifiers (**i,j,k**) for anonymous loop counter variables (rather than **count1, count2, count3**).
4. Be consistent in your use of names.

With some practice, it is possible to write very readable and understandable programs if you choose good variable and function names.

5.3.3 The preprocessor

Before compiling your program, the compiler takes your C source file, and feeds it to another program (cpp - a preprocessor), which creates a new version of your file. Any C source line which has a # as the first character on the line is interpreted by the preprocessor. These lines are called preprocessor directives.

- **#define ABC xyz**
  This redefines any occurence of ABC in your program to have the text xyz. This is a textual substitution. The facility is used to give useful names to things. For example:
  - **#define PI 3.14159**
  - **#define MAXSIZE 42**
    This sort of use of #define enhances the readability of your programs, and also allows you to make large scale changes to your program quickly. For example if you suddenly wanted to change **PI** to **3.1415926**, you only have to change it at the place where it is defined.
- **#include <stdio.h>**
  This line is replaced by a file called stdio.h found in one of the directories that the compiler knows about.
- **#include "mydefs.h"**
  This line is replaced by a file called mydefs.h found in the current directory.
- **#ifdef NOERRORS**
  The next section is only included if the name NOERRORS has been #defined (conditional compilation).

The preprocessor allows C to be extended arbitrarily - with new looking constructs, meaningful names for common code sequences, and readable names for common constants.

5.3.4 Headers vs header files

Every function in a program has a header part and a body.

1. The header has the name and type of the function, along with the parameters to that function.
2. The body is enclosed in braces, and has both delarations and executable statements.

Header files are declarations that belong to the libraries that come with C. They contain standard C defines, variable declarations, sometimes code, but more commonly declarations of external functions which are linked into our programs at link time.
Some common header files are:

<errno.h> Macros and numbers for error conditions

<math.h> The prototypes for math library functions

<stddef.h> Standard C types

<stdio.h> C input and output (such as printf)

<string.h> The prototypes for string functions

When we use **#include <stdio.h>**, or any other header file, the file is included into our program using the preprocessor.

5.3.5 Scope of names

Each identifier in a program is usable only from some areas of the program. This is called the *scope* of a variable. The rules are simple:

1. The scope of a function or variable name begins with the declaration or prototype, and ends at the end of the C source file.
2. Parameters and local variables in a function are visible from when they are declared through to the end of the function (i.e. to the closing brace of the body of the function.)

5.3.6 Function prototypes

ANSI C has a function prototype - a *declaration* of a function (in the same way as we declare variables). The prototype declaration looks like the actual function header, defining the return type of the function, its name, and the parameter types.
In ANSI, all functions used should be prototyped near the beginning of the program.

5.4 Nested for loops

Sometimes we wish to nest **for-loops**. In the following example, we use the outermost loop to printout some number of lines. The innermost loop prints out a different pattern on each line.

CODE LISTING **nestedfor.c**

```
/* ***************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of
                nested for statements
   Revisions:   None yet...
   *************************************************************** */

main ()
{
    int in1, i, outer;
    scanf ("%d", &in1);
    for (outer = 1; outer <= in1; outer = outer + 1) {
        for (i = 1; i < outer; i = i + 1) {
            printf ("+");
        }
        for (i = 1; i < 40 - (outer * 2); i = i + 1) {
            printf (" ");
        }
        for (i = 1; i < outer; i = i + 1) {
            printf ("+");
        }
        printf ("\n");
    }
}
```

When we execute this nested loop program, it displays this:

```
[hugh@stf-170 programs]$ ./nestedfor
15
+                                     +
++                                   ++
+++                                 +++
++++                               ++++
+++++                             +++++
++++++                           ++++++
+++++++                         +++++++
++++++++                       ++++++++
+++++++++                     +++++++++
++++++++++                   ++++++++++
+++++++++++                 +++++++++++
++++++++++++               ++++++++++++
+++++++++++++             +++++++++++++
++++++++++++++           ++++++++++++++
[hugh@stf-170 programs]$
```

5.5 The switch statement

The switch statement allows you to select from a large number of alternatives. We could also do this by using a series of if-then-else statements, but the switch statement is much more compact.

Here is an example of a switch statement:

CODE LISTING **switch.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of the
                switch statement
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    char ch;
    scanf ("%c", &ch);
    switch (ch) {
     case 'a':
        printf ("You keyed 'a'\n");
        break;
     case 'b':
        printf ("You keyed 'b'\n");
        break;
     case 'c':
        printf ("You keyed 'c'\n");
        break;
     case 'd':
        printf ("You keyed 'd'\n");
        break;
     default:
        printf ("You did not key either 'a','b','c' or 'd'!\n");
    }
}
```

5.6 Summary of topics

In this section, we introduced the following topics:

- Error finding/debugging
- Common debugging practice
- Rules for identifiers and reserved words
- Nested for loops
- The C switch statement

Questions

1. What do the letters in **gdb** stand for?
2. Investigate the Ariane 5 and Therac 25 incidents in relation to ‘risks’. What lessons can we learn from these incidents?
3. Which of the following are valid C identifiers:
   _hello, __hello, 2much, TooLittle, nearly-enough, nearly_enough.
4. Find the header files for C (stdio.h and so on) on a UNIX or Windows system. Give the function prototypes for printf and scanf.
5. Write the first few lines of this C program:

CODE LISTING **pas.pas**

```
program mymain
begin
   writeln( "Hello World!\n" );
end
```

Further study

- Textbook, Chapter 5,6,7

Chapter 6

Low flying data structures

Another useful tool that the software engineer can use is a small program called **lint**. This program can scan a C program and discover a whole range of common programming errors. Among the things it detects are:

- Unreachable statements
- Loops not entered at the top
- Automatic variables declared and not used
- Logical expressions whose value is constant.

The **lint** utility checks for some types of incorrect use of C (although of course it is unable to discover what you *really* meant by your program). It operates in two modes: basic, which is the default, and enhanced, which includes everything done in basic mode, plus it provides additional, detailed analysis of code.

Here is an example of **lint** scanning and reporting on errors in this program segment:

```
main () {
   int in1,result,count,highest,lowest;
   result = 0;
   count = 0/highest;
   ...
```

When we run **lint** on the program, it reports the following:

```
hugh@sunA:~[515]$ lint w2.c
(6) warning: variable may be used before set: highest
(16) warning: Function has no return statement : main
variable unused in function
   (4) lowest in main
...
```

6.1 Simple data types

Here is a summary of the simple (variable) data types that we have met so far:

Data type	What it contains
int	an integer value
float	a floating point value
char	a single character

Each of these variables is stored internally in the computer as a group of bits (binary digits). For example, the letter **A** is stored in a **char** variable as the eight bits **01000001**.

6.1.1 Initializing variables

It is very common when writing the program to declare a variable and then initialize it to some value. This is so commonly done, that we have a special syntax for it. Here are some examples of variables being declared and simultaneously initialized:

```
int days=365;
float almostpi=3.14;
```

Note that if you do not initialize the variables, there is no guarantee that they will have any fixed value when your program runs, although some systems initialize them to have values of ZERO.

6.1.2 Integer storage

The number of bits used for **int** is not fixed, and depends on the implementation of the C compiler. It is common for an **int** to be 32 bits, but don't be surprised if it is only **16**, or if it is **64**!

The limited size of the storage space for values can lead to peculiar results:

CODE LISTING **maxint.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the maximum size of
                an integer
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int i;
    for (i = 0; i < 4; i = i + 1) {
        printf ("2147483646+%d is %d\n", i, 2147483646 + i);
    }
}
```

The result of this program is:

```
[hugh@stf-170 programs]$ ./a.out
2147483646+0 is 2147483646
2147483646+1 is 2147483647
2147483646+2 is -2147483648
2147483646+3 is -2147483647
[hugh@stf-170 programs]$
```

If the compiler uses 16 bits for an integer then the integer values are restricted to -32768 to 32767, (-2^{15} to $2^{15}-1$). If it uses 32 bits then the integer values are restricted to -2147483648 to 2147483647 (-2^{31} to $2^{31}-1$). If you need more (or less) range, you can use the **long** (or **short**) keyword.

6.1.3 Size of simple types

This simple program uses the **sizeof** function to show the relative sizes of these simple types on an Intel/LINUX machine, using the GNU C compiler:

CODE LISTING **sizeof.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the size of various
                simple data types.
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    char c;
    short h;
    int i;
    long int j;
    long long int k;
    float l;
    double m;
    printf ("The size of char is %d bits\n", sizeof (c) * 8);
    printf ("The size of short is %d bits\n", sizeof (h) * 8);
    printf ("The size of int is %d bits\n", sizeof (i) * 8);
    printf ("The size of long int is %d bits\n", sizeof (j) * 8);
    printf ("The size of long long int is %d bits\n", sizeof (k) * 8);
    printf ("The size of float is %d bits\n", sizeof (l) * 8);
    printf ("The size of double is %d bits\n", sizeof (m) * 8);
}
```

The result of this program is:

```
[hugh@stf-170 programs]$ ./a.out
The size of char is 8 bits
The size of short is 16 bits
The size of int is 32 bits
The size of long int is 32 bits
The size of long long int is 64 bits
The size of float is 32 bits
The size of double is 64 bits
[hugh@stf-170 programs]$
```

6.1.4 The float and double types

The **float** variable type stores a number in a kind of floating-point or scientific notation - that is, in the same way as we write 6.3e4 to mean $6.3 * 10^4$. On the computer of course we do not use powers-of-10. Also - since we are using a fixed number of bits, there is a limit to the accuracy of these floating-point numbers. With 32 bit numbers we get about 8 digits of accuracy.

The **double** type uses more bits and has higher precision (18 or so digits), and slower execution when we use them.

6.1.5 Boolean types

Boolean values of **FALSE** and **TRUE** are stored as integers in the computer. The value of **0** is taken to mean **FALSE** and the value of **1** is taken to mean **TRUE**. When we write:

```
if (a<3) {
    ...
}
```

The computer gets the value of **a** and if it is less than **3** returns a **1** and if not returns a **0**. It works very much like a function and we can replace it with a function:

```
int checksize( int x, int y ) {
   if (x<y)
     return 1;
   else
     return 0;
}
```

This could be used in the original code:

```
if (checksize(a,3)) {
   ...
}
```

The function is far longer than it needs to be. A quicker way, but possibly harder to understand initially, is:

```
int checksize( int x, int y ) {
   return x<y;
}
```

6.1.6 Character storage - the char type

Computers only store 0s and 1s, not characters. Hence we have to interpret a particular 0,1 pattern as a particular character. It is important to remember that ’b’ follows ’a’ and ’B’ follows ’A’ and ’1’ follows ’0’ and so on. Note also that many of the characters are non-printing (such as character 13, the ‘carriage-return’ character), but are extremely commonly used in C programs. To refer to these non-printing characters, we can use the following two-character sequences:

Sequence	Character		Sequence	Character
\0	Null - no character		\t	Horizontal tab
\a	Audible alarm - BELL		\v	Vertical tab
\b	Backspace		\‘	Apostrophe
\f	Formfeed		\”	Quote
\n	Newline		\?	Question mark
\r	Carriage return		\\	Backslash

To get the value of a digital character such as ’3’, we can use:

```
value = (int) ’3’ - (int) ’0’;
```

To change a lower case letter into an upper case letter, ’t’ to ’T’:

```
ucletter = (char)((int) ’t’ - (int) ’a’ + (int) ’A’);
```

Each character is given a particular 7 bit pattern - any unambiguous pattern would do but there is an international agreement that particular patters are interpreted as particular characters - ASCII - the American Standard Code for Information Interchange. This code defines 128 ($128 = 2^7$) characters:

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	^@ *nul*	32	20	*spc*	64	40	@	96	60	‘
1	01	^A *soh*	33	21	!	65	41	A	97	61	a
2	02	^B *stx*	34	22	“	66	42	B	98	62	b
3	03	^C *etx*	35	23	#	67	43	C	99	63	c
4	04	^D *eot*	36	24	$	68	44	D	100	64	d
5	05	^E *enq*	37	25	%	69	45	E	101	65	e
6	06	^F *ack*	38	26	&	70	46	F	102	66	f
7	07	^G *bel*	39	27	’	71	47	G	103	67	g
8	08	^H *bs*	40	28	(	72	48	H	104	68	h
9	09	^I *ht*	41	29	)	73	49	I	105	69	i
10	0A	^J *lf*	42	2A	*	74	4A	J	106	6A	j
11	0B	^K *vt*	43	2B	+	75	4B	K	107	6B	k
12	0C	^L *ff*	44	2C	,	76	4C	L	108	6C	l
13	0D	^M *cr*	45	2D	-	77	4D	M	109	6D	m
14	0E	^N *so*	46	2E	.	78	4E	N	110	6E	n
15	0F	^O *si*	47	2F	/	79	4F	O	111	6F	o
16	10	^P *dle*	48	30	0	80	50	P	112	70	p
17	11	^Q *dc1*	49	31	1	81	51	Q	113	71	q
18	12	^R *dc2*	50	32	2	82	52	R	114	72	r
19	13	^S *dc3*	51	33	3	83	53	S	115	73	s
20	14	^T *dc4*	52	34	4	84	54	T	116	74	t
21	15	^U *nak*	53	35	5	85	55	U	117	75	u
22	16	^V *syn*	54	36	6	86	56	V	118	76	v
23	17	^W *etb*	55	37	7	87	57	W	119	77	w
24	18	^X *can*	56	38	8	88	58	X	120	78	x
25	19	^Y *ew*	57	39	9	89	59	Y	121	79	y
26	1A	^Z *sub*	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[*esc*	59	3B	;	91	5B	[	123	7B	{
28	1C	^\ *fs*	60	3C	<	92	5C	\	124	7C	|
29	1D	^] *gs*	61	3D	=	93	5D	]	125	7D	}
30	1E	^^ *rs*	62	3E	>	94	5E	^	126	7E	~
31	1F	^_ *us*	63	3F	?	95	5F	_	127	7F	*del*

6.1.7 Type coercion - cast

In the previous section, we used a **cast** to change the type of expressions. It is useful to always use the cast to enforce the particular type that you want. Without the cast, C has a set of promotion rules, that are sometimes confusing.

6.1.8 Precedence of operators

In the C language, each arithmetic operator (such as +, -, * or /) is said to have a precedence, and for these simple operators, the precedence follows the one that we have all learnt in school - $a * 5 + 6$ is interpreted as $(a * 5) + 6$, not $a * (5 + 6)$. In general, higher precedence operators (such as * and /) are done first, and lower level ones (such as + and -) are done later.

However, C has a *large* number of operators, not just four, but more like forty-four! It is easy to get confused about which operator associates before another. Here is a table showing *some* of the C operators, ranked from highest to lowest precedence:

Operator		
( )	[ ]	->
unary+	&	sizeof
*	/	%
+	-	
>>	<<	
<	<=	>
==	!=	
&		
^		
|		
&&		
||		
?:		
=	+=	-=
,		

A simple example:

```
a=b=c+2;
```

adds **2** to **c**, and assigns the result to **b**, and then to **a**. It acts in this way:

```
a=(b=(c+2));
```

However,

```
a=(b=c)+2;
```

assigns **c** to **b**, and then adds **2**, putting the result of this calculation in **a**.

A general rule to cope with all this confusion is: **Use brackets...**

6.2 Data structures - the array

The **array** data type allows this to create an array of elements of the same type. We might wish to use this if we were reading in a series of (say 100) values and we needed to store these values in some place before performing a calculation on them. A nice syntax for this might be to store the values in **rec[0], rec[1], rec[2]** and so on up to **rec[99]**.

We could of course just use a series of variables (**rec0, rec1, rec2 .. rec99**), but the array construct turns out to be very useful:

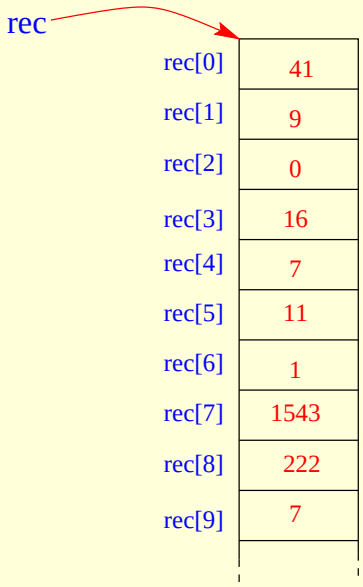
- we can refer to all the 100 values with just a single name (**rec**), and
- we can use a **for** loop to process each item:

```
for (i=0; i<100; i=i+1) {
    process( rec[i] );
}
```

instead of:

```
process( rec0 );
process( rec1 );
process( rec2 );
...
```

C arrays all start at element 0 - that is, the first item in a C array is numbered 0. C arrays are fixed (static) in size, and each of the array elements appears one-after-the-other in the memory of the computer:



When you write programs in C, it is easy to refer to (say) **rec[100]** for the 100th element of an array by mistake. This causes no compile time error, as C does not check array bounds. However it may

(or may not) cause run time errors and confusion. In general, this reference may refer to a completely different variable - perhaps a part of the one declared *next* in the program.

The array name itself refers to the machine memory address of the first element of the array.

6.2.1 Declaration and initialization of an array

When declaring a C array, we have to tell the compiler how large it is. The syntax of an array declaration is:

<type> <name>[<size>];

For example:

```
int   rec[100];
char  s1[40];
float myones[216];
```

We can pre-initialize arrays in a similar manner to the way in which we pre-initialize simple variables:

```
int recs[10] = { 26, 42, 14, 2, 1, 0, 0, 0, 45, 11223 };
char s[4]    = "abc";
```

In the **s[4]** example above, the longest string that may be stored in **s** is 3 characters. Each C string is terminated by a **NULL** (or 0) character, and so we need to leave space for this *terminating* (or *finishing*) character.

A **char** (character) array is commonly used to store strings, and we may fix the size by initializing the **char** array. The array size will be one more than the number of characters in the string:

```
char s[] = "A short message!";
```

Note: "hello" represents the string containing the five characters 'h', 'e', 'l', 'l' and 'o', and terminated with a NULL char - a total of 6 characters. By contrast, 'h' represents the single character.

6.2.2 Two dimensional arrays

C supports arrays with more than a single dimension, with similar rules for syntax as for the single dimension. Here are some sample declarations:

```
int recs[3][2] = { { 26, 42 }, { 14, 2 }, { 1, 0 } };
char s[4][40];
```

Note that a suitable way to print out (or indeed do any process on) a multi-dimensional data structure is by using a nested for-loop:

CODE LISTING **nested.for.array.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        22/11/2000
   Description: This is a program to demonstrate the use of nested
                for statements for processing multi-dimensional arrays
   Revisions:   None yet...
   **************************************************************** */

main ()
{
    int i, j;
    int rec[3][2] = { {26, 42}, {14, 2}, {1, 0} };
    for (i = 0; i < 3; i = i + 1) {
        printf ("Row %2d: ", i);
        for (j = 0; j < 2; j = j + 1) {
            printf ("%6d", rec[i][j]);
        }
        printf ("\n");
    }
}
```

The result of this program is:

```
[hugh@stf-170 programs]$ gcc nested.for.array.c
[hugh@stf-170 programs]$ ./a.out
Row 0:   26   42
Row 1:   14    2
Row 2:    1    0
[hugh@stf-170 programs]$
```

We can also use these 2D arrays to create arrays of strings, and then use a for loop to iterate through them:

CODE LISTING **array.of.string.c**

```
#include <stdio.h>

char translate[5][6] = { "zero", "one", "two", "three", "four" };

int
main ()
{
    int i;
    for (i = 0; i < 5; i = i + 1) {
        printf ("%d is %s.\n", i, translate[i]);
    }
    return 0;
}
```

The result of this program is:

```
[hugh@stf-170 programs]$ gcc array.of.string.c
[hugh@stf-170 programs]$ ./a.out
0 is zero.
1 is one.
2 is two.
3 is three.
4 is four.
[hugh@stf-170 programs]$
```

6.3 Data structures - struct

The **struct** data type is used when we want to create a data element that contains several different types of values. For example, we might wish to create an account record - it would need to contain values such as the account holder's name, the account holder's address, and the current balance.

Here is an example of such a data structure:

```
struct account {
   char name[30];
   char address[3][30];
   float balance;
} acc1;
```

A small program to illustrate the use of struct:

CODE LISTING **struct1.c**

```
#include <stdio.h>

struct account
{
    char name[30];
    char address[3][30];
    float balance;
}
acc1;

int
main ()
{
    strcpy (acc1.name, "Hugh");
    strcpy (acc1.address[0], "424242 Shenton Way");
    strcpy (acc1.address[1], "Singapore");
    strcpy (acc1.address[2], "3rd planet from sun");
    acc1.balance = -123.45;
    printf ("Account name is:   %s\n", acc1.name);
    printf ("Account address is: %s\n", acc1.address[0]);
    printf ("         %s\n", acc1.address[1]);
    printf ("         %s\n", acc1.address[2]);
    printf ("Account balance is: %.2f\n", acc1.balance);
}
```

The result of this program is:

```
[hugh@stf-170 programs]$ gcc struct1.c
[hugh@stf-170 programs]$ ./a.out
Account name is:    Hugh
Account address is: 424242 Shenton Way
                    Singapore
                    3rd planet from sun
Account balance is: -123.45
 [hugh@stf-170 programs]$
```

The typedef struct definition allows us to predeclare structs and then use them as needed:

```
typedef struct  {
   char name[30];
   char address[3][30];
   float balance;
} account;

account allaccounts[100];
```

6.4 Random numbers

We often need to use random numbers when programming. For example, if we were writing a game program, we may want to randomly pick cards to give to each of the players.

There are other examples of algorithms which work through picking random numbers. For example, one of the ways of finding your way out of a maze, is to begin by walking in a random direction, and when you hit a wall, mark it, and, keeping a hand on the wall, walk in one direction. If you ever return to your original wall position, choose a new random direction.

The C random number generator allows us to acquire random value values, but the values are not truly random - they are generated by an algorithm, and are known as pseudo-random number generators.

The C **rand()** function returns a pseudo-random integer between 0 and **RAND_MAX**. If you want to generate a random integer between 1 and 10, you can do it by:

```
j = 1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

6.5 Summary of topics

In this section, we introduced the following topics:

- Program checking - lint
- Revision of simple data types
- Internal storage of simple data types
- Two dimensional arrays and more complex structures
- Random numbers

Questions

1. Which of the following numbers may be represented exactly in a **float**:
   1, 100, 0.5, 0.3, 0.25
2. Given the expression $y = ax^3 + 7$, which of the following (if any) are correct C statements for this equation?
   y = a*x*x*x + 7;
   y = a*x*x*(x + 7);
   y = (a*x)*x*(x + 7);
   y = (a*x)*x*x + 7;
   y = a*(x*x*x) + 7;
   y = a*x*(x*x + 7);
3. Write a short function **upcase()** which is passed a **char** parameter, and returns the same character unless it is a lowercase ASCII character. If it is a lowercase ASCII character, the function returns the equivalent uppercase ASCII character.

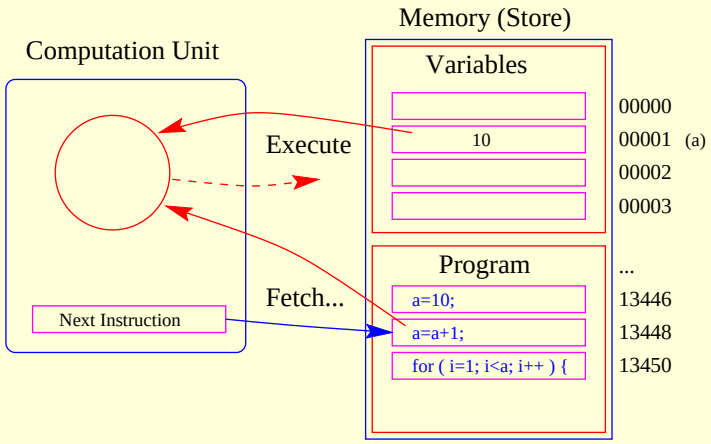
Further study

- Textbook, Chapters 5&6.

Chapter 7

Pointers and strings

When trying to understand the operation of C programs, it is useful to have a representation of the underlying machine. The following simplified model helps us understand the behaviour of C programs.



The elements of the model are:

- A linear memory, each location addressed or identified by number. In the diagram above, memory location 1 is used for the C variable a.
- An execution unit, which has a pointer to the current C instruction to be executed.
- An area of memory used for storing the program (which is in a *machine* language, not C).
- A fetch-execute cycle.

During normal operation of the C machine, the first instruction to execute is the one at the beginning of the **main()**.

7.1 Pointers

Our model of the C machine includes a memory space, with a large number of memory slots into which we store our program variables. In the programming language, we address these slots by name, but internally the computer refers to them by number. The specific location of a variable in memory is called its address, and in some situations it is useful to refer to variables by their address - particularly when trying to get the most speed out of some code.

For example, if we were trying to zero-out all the elements of an array, we may have code something like this:

```
for ( i=0; i<sizeof( myarray ); i=i+1 ) {
    myarray[ i ] = 0;
}
```

This code is fine, and very readable, but requires the calculation of the addess of the ith element of the array every time through the loop, along with the loop increment. We can use pointers to get the same effect

```
for ( ptr=&myarray; ptr<(&myarray+sizeof( myarray )); ptr=ptr+1 ) {
   *ptr = 0;
}
```

When this code is translated to machine code by the compiler, the resultant code is smaller and faster, without the continual recalculation of the address of each element of the array.

We refer to the address of a variable by prefixing the variable name with the **&** character. We have been using pointers whenever we use **scanf()** - the **&varname** in **scanf("%f", &varname)** refers to the address of varname.

Addresses are 16 bits in size on a 16 bit machine, 32 bits in size on a 32 bit machine and so on.

We can declare variables that contain other variable addresses. For example:

```
int *pmyint;
```

is a pointer to an integer variable, and

```
char *pmychar[20];
```

is an array of 20 pointers to characters.

We dereference a pointer variable by prefixing the name with the * character. For example:

```
int a=1,b=2;
int *p;
p = &a;
printf( “The address of a is %d, and the value there is %d\n”, p, *p );
p = &b;
printf( “The address of b is %d, and the value there is %d\n”, p, *p );
```

Pointers cause a lot of problems in C programs - if you attempt to access the memory pointed at by an arbitrary pointer, and that memory is not available to you, then you may get a “segmentation fault”. This means that your program has tried to access memory that does not belong to it, and the OS has trapped this reference, and terminated the program.

Why do we use pointers?

1. For speed, and
2. because we want dynamic structures - that are created only at run time.

A very common fault in C programs with pointers is to forget to initialize the pointer, leading to a segmentation fault.

7.2 Strings

Standard C contains no special type for a string, and so a string is made from arrays of characters. The C string is a character array in which the last character is a NULL (‘\0’). C strings are of unlimited size, and may contain any ASCII character - except for the NULL character (for obvious reasons).

We can create a string easily:

CODE LISTING **string1.c**

```
int
main ()
{
    char mystr[10];

    mystr[0] = 'H';
    mystr[1] = 'u';
    mystr[2] = 'g';
    mystr[3] = 'h';
    mystr[4] = '\0';

    printf ("My name is %s.\n", mystr);
    return 0;
}
```

The result of this program is:

```
[hugh@stf-170 programs]$ gcc string1.c
[hugh@stf-170 programs]$ ./a.out
My name is Hugh.
[hugh@stf-170 programs]$
```

We can initialise a string in the declaration:

```
char str[] = "abc";
```

and the result will be the same as above except that it will have made its size to be 4 since that is all that is needed. What we cannot do is:

```
char str[5];
str = "abc";
```

In order to do that we have to *copy* one string to another. We could write a function:

```
void copystrng(char s1[], char s2[]) {
        int i=0;
        while (s2[i] != '\0') {
                s1[i] = s2[i];
                i++;
        }
        s1[i] = '\0';
}
```

and this will copy the string in **s2** and put it into **s1**. However, this sort of thing is so commonly done that a similar function exists in a standard library accessed via the **<string.h>** header file:

7.2.1 Sample string functions - string copy

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

The **strcpy()** function copies the string pointed to by **src** (including the terminating '**\0**' character) to the array pointed to by **dest**. The strings may not overlap, and the destination string **dest** must be large enough to receive the copy. The **strncpy()** function is similar, except that not more than **n** bytes of **src** are copied. Thus, if there is no NULL byte among the first **n** bytes of **src**, the result will not be null-terminated.

7.2.2 Sample string functions - string concatenate

```
#include <string.h>
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

The **strcat()** function appends the **src** string to the **dest** string overwriting the '**\0**' character at the end of **dest**, and then adds a terminating '**\0**' character. The strings may not overlap, and the **dest** string must have enough space for the result. The **strncat()** function is similar, except that only the first **n** characters of **src** are appended to **dest**.

7.2.3 Sample string functions - string compare

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

The **strcmp()** function compares the two strings **s1** and **s2**. It returns an integer less than, equal to, or greater than zero if **s1** is found, respectively, to be less than, to match, or be greater than **s2**. The **strncmp()** function is similar, except it only compares the first **n** characters of **s1**.

7.2.4 Sample string functions - string length

```
#include <string.h>
size_t strlen(const char *s);
```

The **strlen()** function calculates the length of the string **s**, not including the terminating **'\0'** character.

7.2.5 Sample string functions - print-to-string

This one is in **<stdio.h>** - it is just like **printf()**, except that the output is placed in a string.

```
#include <stdio.h>
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

The **snprintf()** function does not write more than **size** bytes.

7.3 Memory allocation

There is a standard library for memory allocation in C:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

The **calloc()** system call allocates memory for an array of **nmemb** elements of **size** bytes each and returns a pointer to the allocated memory. The memory is set to zero.

The **malloc()** system call allocates **size** bytes and returns a pointer to the allocated memory. The memory is not cleared.

The **free()** system call frees the memory space pointed to by **ptr**, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(ptr)** has already been called before, undefined behaviour occurs. If **ptr** is NULL, no operation is performed.

7.4 Summary of topics

In this section, we introduced the following topics:

- The C programming model
- Pointers
- Strings
- Memory allocation

Questions

1. Give initializer declarations for an array of 12 integers containing the days in each month of 2001.
2. On some computers, the bytes of an integer are stored in different orders. This is called the endianness of a computer. Write a short program using a char pointer and an integer which discovers the ordering of successive bytes of an integer.
3. Write a function to concatenate two strings, which accepts two input strings as parameters, and returns a pointer to a new string. Your routine should use malloc().

Further study

- Textbook, Chapters 7 and 8

Chapter 8

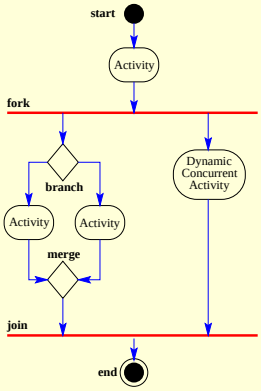
Files 'n stuff

A possible[1] technique for developing software is to progress through the following stages:

- Task specification - research the situation, developing a clear specification or definition of the task to be performed.
- Analysis - look at the problem from the point of view of the user. Develop a logical plan of the system - how it should appear to the user.
- Design - Develop the algorithm(s) to execute the logical plan. Pseudocode, or other representations.
- Implementation - Code in the language of your choice.
- Testing - Use a test plan, and document the testing process.

To aid in documenting and designing software, we use diagrams of different sorts. A very old and very simple technique is the flow diagram. The flow diagram uses differently shaped boxes to represent different programming constructs. The principal problem with this diagramming technique is that it is not scalable - the components of the diagram always represent the lowest level program structures.

Nowadays we use better tools - the UML (Unified Modelling Language) has a range of modelling and diagramming techniques, and some tool support to check the correctness of designs before implementation.



[1] Derived from Lawlor's five steps.

8.1 File input and output

A simple initial definition of a file[2] might be:

> *A named block of data.*

However, this definition proves inadequate when we look at the wider usage of the term in modern operating systems - as we will see.

8.1.1 Stream of characters vs structured

A fundamental file type found on DOS, UNIX, and WinXX systems is the *stream of characters*. In these files, each 8-bit or 16-bit unit represents a single character.

On UNIX and NT systems, we build structured files (for example - files of structs or records), *on-top-of* these stream of character files.

The *Macintosh* and *NeXt* computers use a more developed view of a file (which they call a *document*). A Macintosh file is a fork containing two elements:

1. The data belonging to the file
2. A *resource fork* containing information about the data

Typically the *resource fork* is used to keep file information that may change. If the file is an application, the *resource fork* may contain the text for menu items, or the colours and backgrounds of popups. The file fork contains the program itself. The resource fork may be separately editted using a *Macintosh* program called *resedit*.

8.1.2 Names

The file names can be of different sizes, and use different characters.

DOS file names: DOS files use an 8-byte name with a 3-byte extension - XXXXXXXX.YYY. There are limits on the characters allowed in file names.

UNIX file names: UNIX systems vary, but typically allow 255 character file names, using nearly any characters, and being case sensitive. The name "a.b.c.d.e()[]\" is a perfectly valid UNIX file name. So is "a name with spaces!".

NT file names: NT systems support a range of file systems, including the older DOS based file systems. When using NTFS, file names are case insensitive, and may be up to 255 characters long, using nearly any characters.

8.2 The C stream file API

In the standard C view of files, a file is a simple thing, which we open, write to (or read from) and then close. When we open the file, we are returned a file handle, which we may use to later specify which file we wish to write to. The *stream* file system API is:

Function	System call
Create file	fopen()
Close file	fclose()
Read or Write to file	fprintf(), fscanf()

A typical use of the *stream* file system API is:

CODE LISTING **file1.c**

```
#include <stdio.h>

main ()
{
    FILE *f1;
    int i;
    f1 = fopen ("myfile.dat", "w");
    for (i = 0; i < 10; i = i + 1) {
        fprintf (f1, "Line %d – Hi there!\n", i);
    }
    fclose (f1);
}
```

8.2.1 Create file - fopen()

```
#include <stdio.h>
FILE *fopen (const char *path, const char *mode);
FILE *fdopen (int fildes, const char *mode);
FILE *freopen (const char *path, const char *mode, FILE *stream);
```

The fopen function opens the file whose name is the string pointed to by path and associates a stream with it. The argument **mode** points to a string containing the ‘mode’ in which the file is to be opened.

8.2.2 Close file - fclose()

```
#include <stdio.h>
int fclose( FILE *stream);
```

The fclose function dissociates the named stream from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using **fflush(3)**.

8.2.3 Read and write files - fprintf() and fscanf()

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int fscanf( FILE *stream, const char *format, ...);
```

These system calls act in a similar way to the **printf()** system call.

[2]IBM use the term *dataset* to refer to a file.

8.3 The C raw file API

A lower level API is this one:

Function	System call
Create file	open(), creat(), dup()
Close file	close()
Read or Write to file	read(), write()
Get or Set attributes	fcntl()

8.3.1 Create file - open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int oflag, ... /* mode_t mode */);
```

The string path points to a path name naming a file. open opens a file descriptor for the named file and sets the file status flags according to the value of oflag. oflag values are constructed by OR-ing Flags

8.3.2 Close file - close()

```
#include <unistd.h>
int close(int fildes);
```

The parameter *fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, *pipe*, or *iocntl* system call. The system call *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

8.3.3 Read and write files - read(), write()

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

The system call *read* attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. If *nbyte* is zero, *read* returns zero and has no other results. The parameter *fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, *pipe*, or *ioctl* system call.

The system call *write* attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with *fildes*. If *nbyte* is zero and the file is a regular file, *write* returns zero and has no other results.

8.3.4 Control attributes - fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl (int fildes, int cmd, ... /* arg */);
```

The *fcntl* system call provides for control over open descriptors. A large number of operations are possible using the *fcntl* call.

8.4 Network programming

Most network programming involves using an API which provides a *session layer* service. The API provides a set of system calls, and software which implements some *view* or *abstraction* of the network.

It is normal to use these programming *abstractions* when doing network programming. They allow you to model the behaviour of the system, and understand its behaviour.

The *'socket'* is an abstraction for each endpoint of a communication link. The abstraction allows data communication paths to be treated in the same way as UNIX files. When a socket is initialized, the API system calls return a *file descriptor* number which may be used to read and write in the same way as those returned by file stream API calls.

Here is an example of a simple server written in C:

CODE LISTING **server.c**

```
/* ****************************************************************
   Author:      Hugh Anderson
   Date:        01/11/2000
   Description: This is the mainline of the server software for
                a simple server for teaching...
   Revisions:   None yet...
   **************************************************************** */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>

#define SERV_TCP_PORT 9000
#define MAXLINE 512

main (argc, argv)
     int argc;
     char *argv[];
{
    /* Declarations... */
    int loc, sockfd, newsockfd, clilen;
    struct sockaddr_in cli_addr, serv;

    /* Initialization... */
    printf ("Trying to open socket...\n");
    if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
        fprintf (stderr, "server: can't open stream socket\n");
    bzero ((char *) &serv, sizeof (serv));
    serv.sin_family = AF_INET;
    serv.sin_addr.s_addr = htonl (INADDR_ANY);
    serv.sin_port = htons (SERV_TCP_PORT);
    if (bind (sockfd, (struct sockaddr *) &serv, sizeof (serv)) < 0)
        fprintf (stderr, "server: can't bind local address\n");
    listen (sockfd, 5);

    /* The main event loop... */
    for (;;) {
        clilen = sizeof (cli_addr);
        printf ("clilen is %d\n", clilen);
        newsockfd = accept (sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0)
            fprintf (stderr, "server: accept error\n");
        write (newsockfd, "Hugh's Server\n", 14);
        process (newsockfd);
        close (newsockfd);
    }
}
```

8.5 More operators

C has a large number of operators, and here we look at some more of them.

8.5.1 Boolean operators

If we wish to combine Boolean expressions we can use two operators AND and OR. To see what they do we look at a *truth* table:

A	B	A OR B	A AND B
F	F	F	F
F	T	T	F
T	F	T	F
T	T	T	T

In C these operators are written as **&&** and **||**. There is no implementation of the exclusive-or operator in C. The unary operator NOT is '!' in C.

Up to now we have had Boolean expressions such as **(a>=3)** which is TRUE if the value of **a** is greater than or equal to **3** and FALSE otherwise.

If we wanted an integer to be within a range such as greater than 3 and less than or equal to 10, we can write this as **(a>=3 && a<=10).**

8.5.2 Increment and decrement operators

C is well known for being terse (as is UNIX). Certain common assignments have shorthand versions in C. For example - a common C assignment is to add 1 to a variable:

```
count = count+1;
```

This is so commonly done, that C has a special syntax specifically for this - the *increment* operator:

```
count++;
or
++count;
```

If the assignment is on its own there isn't a difference between them but if the increment operator is used in an expression then the order may matter.

pre-increment: "++a" means increment and then use the value of a.

post-increment: "a++" means use the value of a and then increment.

We have a matching decrement operator: "–".

8.5.3 Operator assignments

Another common expression is **a = a + n**. This can be written as: **a += n;** The following expressions are all equivalent:

```
counter = counter + 1;
counter++;
++counter;
counter +=1;
```

8.6 Summary of topics

In this section, we introduced the following topics:

- Software development process
- Diagramming methods
- File operations
- More operators

Questions

1. Declare four char variables in a program, and determine the addresses of each.
2. Write a small program which asks for the names of two files, opens the first one and copies any lines that start with a digit to the end of the other file.
3. Give a function prototype for a function which does not return any value.
4. What is the value of x after “**x = 1; x=(x == (x+=1)+1);**”.
5. What will be the values of the three parameters to this function?

```
a=1;
myfunc( ++a, ++a, ++a );
```

Further study

- Textbook, Chapter 11

Chapter 9

Sorting

The study of sorting algorithms is traditional in introductory programming courses. The reason for this is that the study gives us a vehicle for showing a range of different algorithms for the same problem. The *sort* problem can be clearly and simply expressed (something like "sort all those numbers in ascending or descending order").

As we will see though, there are many solutions - each using a different algorithm. There are many ways of performing sorts on unsorted data, with no *one* best way, as some sorts are faster when the data is almost completely sorted, but very slow when the data is a randomly distributed.

Comparison sorting algorithms have a range of speeds - mergesort, heapsort, and the best case of quicksort will take $O(n \log n)$, while insertion sort, selection sort, and the worst case of quicksort all take $O(n^2)$.

The **Big-O** notation given here describes the amount of *time* taken by an algorithm. When we use the term $O(n^2)$ we mean that the time for the algorithm is of-the-order-of n^2 - in this case the square of the number of things to be sorted.

Here is a chart showing how $O(n \log n)$ and $O(n^2)$ are related:

n	1	10	100	1000	10000
$O(n \log n)$	0	10	200	3000	40000
$O(n^2)$	1	100	10000	1000000	100000000

As you can see, $O(n \log n)$ is faster than $O(n^2)$.

You can see animations of different sorting algorithms at http://www.epaperpress.com/s_man.html.

In the following examples, we assume that the initial data to be sorted is stored in an array, but of course the algorithms might be used to sort any data structures.

9.1 Bubble sort

This one gets its name from the idea of bubbles rising up from the bottom of champagne (or coke if you prefer). The larger bubbles get to the top first[1].

1. Make a pass over the values from left to right, comparing adjacent pairs of elements and moving the larger one to the left by swapping the current pair of elements - this puts the largest one at the left.
2. Do it again - but this time you can do one less comparison - because the last element is already known to be the largest.
3. Keep doing it until there are no swaps done.

The software makes successive passes over the array, comparing and exchanging items so that the largest item ends up at the left hand side of the array. This is repeated for the remaining items until the array is sorted.

Algorithm: 2 *Bubble Sort*

```
foreach element P from left to right (unless there were no swaps last time)
  foreach element j from right-1 downto P
    if this element is greater than the next one then
      swap the items
    end if
  end foreach
end foreach
```

Here is the code written in C:

CODE LISTING **BubbleSort.c**

```
void
BubbleSort (int A[], int N)
{
    int j, P;
    int switched = TRUE;
    int tmp;

    for (P = 0; (P < N - 1) && (switched == TRUE); P = P + 1) {
        switched = FALSE;
        for (j = N - 1; j > P; j = j - 1) {
            if (A[j] > A[j - 1]) {
                switched = TRUE;
                tmp = A[j];
                A[j] = A[j - 1];
                A[j - 1] = tmp;
            }
        }
    }
}
```

The bubble sort has a worst case running time of $O(n^2)$.

[1] In this algorithm they do...

9.2 Insertion sort

The insertion sort works by repeatedly examining the next item and inserting it into the final array in its correct order with respect to items already examined. Using arrays, the insertion sort requires the other values to be moved to make room for the inserted value.

1. Sort the first two values in the array if needed
2. Insert the third value into the appropriate position relative to the first two values - moving the second value if needed.
3. Insert the fourth value, and so on...

The algorithm is:

Algorithm: 3 *Insertion Sort*

```
foreach element P from left(+1) to right
  insert P into correct place in array from left to P (moving elements as needed)
end foreach
```

Here is the code written in C:

CODE LISTING **InsertionSort.c**

```
void
InsertionSort (int A[], int N)
{
    int j, P;
    int tmp;

    for (P = 1; P < N; P = P + 1) {
        tmp = A[P];
        for (j = P; (j > 0) && (A[j - 1] > tmp); j = j - 1)
            A[j] = A[j - 1];
        A[j] = tmp;
    }
}
```

The insertion sort has a worst case running time of $O(n^2)$.

9.3 Shell sort

The shell sort is an example of a divide-and-conquer approach to a task, and is more efficient than insertion sort for larger arrays, but less efficient than some other methods. Shellsort is similar to insertion sort but allows non-adjacent elements to be swapped. The algorithm arranges the array so that every k^{th} element (starting anywhere) yields a sorted array. Such an array is said to be k-sorted.

By doing this for a sequence of values of k which end in 1 (a k-sequence) we produce a sorted array. For example - if we had this array:

65	219	4	54	5	62	7	9	92	6	15	3

We can arrange this to be 5-sorted:

Before			**After**		
65	62	15	15	62	65
219	7	13	7	13	219
4	9		4	9	
54	92		54	92	
5	6		5	6	

This could then be rearranged to be 2-sorted:

Before						**After**					
15	4	5	13	92	65	4	5	13	15	65	92
7	54	62	9	6	219	6	7	9	54	62	219

We now have (many of) the smaller elements clustered near the left side of the array. If we perform a 1-sort now, there will be few exchanges, as elements do not have to move all the way from one end to the other of the array.

Here is the code written in C:

CODE LISTING **ShellSort.c**

```
void
ShellSort (int A[], int N)
{
    int P, j, inc;
    int tmp;

    for (inc = 1; inc <= N; inc = inc * 2);
    for (inc = (inc / 2) - 1; inc > 0; inc = (inc - 1) / 2) {
        for (P = inc; P < N; P = P + 1) {
            tmp = A[P];
            for (j = P; j >= inc && A[j - inc] > tmp; j = j - inc)
                A[j] = A[j - inc];
            A[j] = tmp;
        }
    }
}
```

To get the most effective shell sort, it is best if there are no factors common between the numbers used for the values of k in the k-sequence.

The shell sort efficiency depends on the choice of the k-sequence, and has a best case running time of $O(n \log n)$, and a worst case running time of $O(n^2)$.

9.4 Selection sort

The selection sort algorithm involves scanning the whole remaining unsorted part of the array repeatedly in the manner described in this algorithm:

Algorithm: 4 *Selection Sort*

```
foreach unsorted part of the array
  choose leftmost card as 'provisional highest value'
  step through rest of array, keeping track of 'provisional highest value'
  move the provisional highest value to the left
end foreach
```

Here is the code written in C:

CODE LISTING **SelectionSort.c**

```
void
SelectionSort (int A[], int N)
{
    int minIndex, j, P;
    int tmp;

    for (P = 0; P < N - 1; P = P + 1) {
        minIndex = P;
        for (j = P + 1; j < N; j = j + 1) {
            if (A[j] < A[minIndex])
                minIndex = j;
        }
        tmp = A[P];
        A[P] = A[minIndex];
        A[minIndex] = tmp;
    }
}
```

The selection sort has a worst case running time of $O(n^2)$.

9.5 Quick sort

Perhaps the fastest sort algorithm is quicksort - another divide-and-conquer algorithm. Quicksort is a recursive procedure, developed by C.A.R Hoare in 1962. The algorithm works like this:

1. If there is only one item - do nothing (it is already sorted).
2. Choose a point in the array at random - this is called the *pivot*.
3. Make two sub-arrays - the one on the left containing items lower than the pivot, the one on the right containing items higher than the pivot. Quicksort each of the sub-arrays.

Note that this has a recursive definition, and we have specified a base case.

Algorithm: 5 *Quick Sort*

```
if array has only one item then
  do nothing
else
  Choose pivot and form the left and right arrays
  Quicksort( leftarray )
  Quicksort( rightarray )
  Join the leftarray, pivot and right array
endif
```

Here is the code written in C:

CODE LISTING **QuickSort.c**

```
void
QuickSort (int A[], int N)
{
    int pivotIndex;
    if (N > 1) {
        pivotIndex = Partition (A, N);
        QuickSort (A, pivotIndex);
        QuickSort (&A[pivotIndex + 1], (N - pivotIndex - 1));
    }
}

int
Partition (int A[], int N)
{
    int i, j, incr = 0, decr = 1, swap;
    int tmp;
    i = 0;
    j = N - 1;
    while (i != j) {
        if (A[i] > A[j]) {
            tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
            swap = incr;
            incr = decr;
            decr = swap;
        }
        i = i + incr;
        j = j - decr;
    }
    return j;
}
```

The quicksort algorithm has an average running time of $O(n \log n)$, although it is possible to choose very bad pivots and get a worst case running time of $O(n^2)$.

9.6 Summary of topics

In this section, we introduced the following topics:

- Sorting!

Questions

1. Investigate the heap sort algorithm. What is its running time? How does it work?
2. What is another divide-and-conquer algorithm used in programming? If you don't know one, find one.
3. Try out each of the sorting algorithms given here, by writing a main() which calls them with arrays of unsorted data. Put counters in the code to measure how many inner loops each procedure takes.
4. Write and test a program which sorts the characters in a string into ascending (ASCII) order.
5. Write and test a program which sorts the words in a string into ascending order.
6. Using the algorithm given in the code listing on page 80, give the k-sequence for an array with 47 elements.

Further study

- Textbook, Chapter 6, section 6.6
- Textbook, Chapter 7, section 7.6

Chapter 10

GUI programming

Pronounced GOOEY programming!

GUI stands for Graphical User Interface, and its use here refers to programs which use the *WIMP*[1] interface found in most workstations. - as opposed to the *command-line* interface found in a DOS-prompt, or telnet session to a UNIX machine. Unfortunately, (again) there is no one standard for GUI programming, although the techniques that you learn in one standard are generally transportable to another.

In the programs given so far in this text, there is a single thread-of-control, which we can examine by reading the main(). This code determines how a user is expected to interact with the program. This general program architecture is satisfactory for small programs with simple command-line user interfaces.

However, graphical user interfaces have a much more complex thread-of-control. For example, at any time we may have a number of possible events about to occur - the user may ask for the window to be minimized, or resized, or click a button, or select a menu, or ...

Our programs must respond to each of these events. The normal way to do this is by restructuring our programs as a group of functions - each of which responds to an event. These functions are called *callbacks*.

Our GUI mainline code looks like this:

CODE LISTING **GUICode.c**

```
#include <any GUI header files needed>

int
main ()
{
    RegisterAllCallbacks ();
    LoopForever ();
}
```

An interesting area of GUI programming is the development of abstract windowed environments, where we program using an abstract API. These systems normally allow the development of software which can be compiled for any environment.

[1] Window, Icon, Mouse, Pointer...

10.1 Window systems

Contrary to popular public opinion, “Windows” is not the only GUI window system. The Macintosh system, and the UNIX X window system both predate Microsoft’s GUI system, and each have interesting features that have yet to be added to “Windows”. For example the UNIX X window system allows a clear separation between the display and the processing, whereas “Windows” display and processing must be on the same machine.

By far the most common GUI windowed environment on the desktop is the one found in Win95/98, and our programming API for it is known as Win32.

We will briefly look at three window systems.

10.1.1 Win32

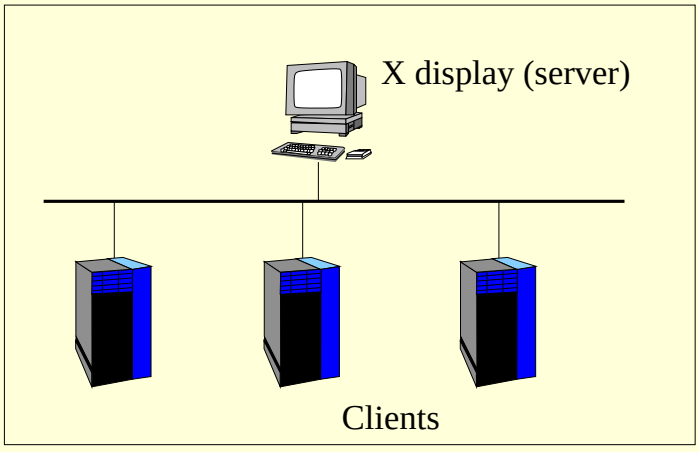
Win32 is a generic name for 4 (slightly) different APIs for Microsoft operating systems. The Win32 API provides standard function calls for accessing the GUI, file system, processes and so on[2]. The Win32 API on Win95 is a subset of those on WinNT, so applications written for Win95 should be portable to WinNT. The reverse is not always true, but most WinNT applications can run on WIn95/98.

The normal way for you to access Win32 functions is by using a precompiled library from a C program.

10.1.2 X

The X window system[3] is a well developed system which allows for multimedia software and hardware components to be distributed around a network. At its simplest level, it allows a program and its display to be on different computers.

The architectural view of X is a little peculiar. The designers view the display as central to the system, and the software running on the display is called the X-server:



[2] Not just the GUI.

[3] The system is called X, or the X window system. It is **not** called X-windows!

From the diagram, we can easily identify three essential components of X:

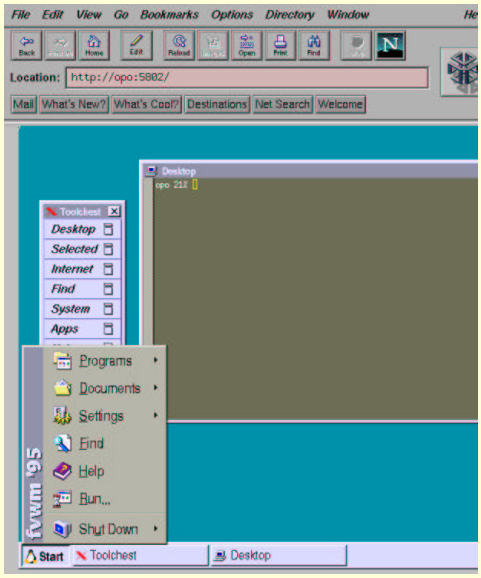
1. **The X server** - providing the high resolution graphical display(s), keyboard and mouse.
2. **The X protocol** - providing standard communication methods between the distributed software components.
3. **X clients** - programs with graphical display output, running on (perhaps) many machines.

However, there are two other components:

- **The Window manager(s)** - providing decorations around each window.
- **The Display manager(s)** - providing access to the system.

10.1.3 Thin client systems

A relatively recent development involves moving the X-server (or equivalent) from the machine with the display to a larger machine, and then using a smaller computer to actually display the data. Here is a thin client written in Java, running as an applet in a web page:



These systems allow applications to be distributed to any desktop.

10.2 Direct calls to the Win32 API

It is possible to write applications that use the Win32 API directly. These programs tend to be long (that is - they have a lot of source lines).

In the same vein as our HelloSingapore example, we start with this program:

CODE LISTING **SimpleWin32.c**

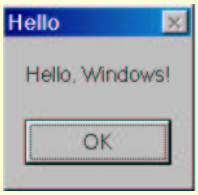
```
#include <windows.h>

int STDCALL
WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpCmd, int nShow)
{
    MessageBox (NULL, "Hello, Windows!", "Hello", MB_OK);
    return 0;
}
```

This small example may be compiled using CYGWIN as follows:

```
gcc -oSimpleWin32 SimpleWin32.c -mwindows
```

And it produces the following application:



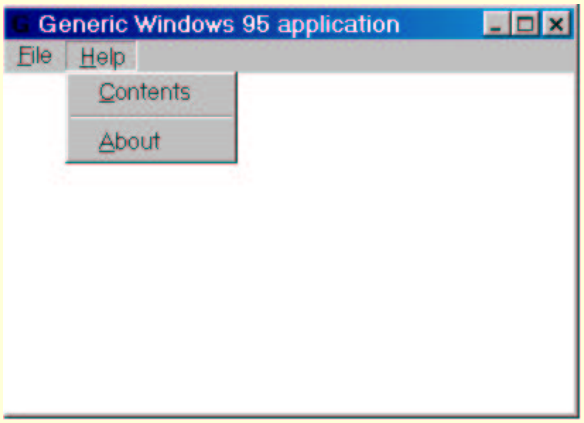
Another more complex example shows the flavour of raw Win32 programming, although I have removed about 380 lines for clarity:

CODE LISTING **BiggerWin.c**

```
#include <windows.h>
#include <string.h>

int STDCALL
WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpCmd, int nShow)
{
    HWND hwndMain;              /* Handle for the main window. */
    MSG msg;                    /* A Win32 message structure. */
    WNDCLASSEX wndclass;        /* A window class structure. */
    char *szMainWndClass = "WinTestWin";
    memset (&wndclass, 0, sizeof (WNDCLASSEX));
    wndclass.lpszClassName = szMainWndClass;
    wndclass.cbSize = sizeof (WNDCLASSEX);
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = MainWndProc;
    wndclass.hInstance = hInst;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    RegisterClassEx (&wndclass);
    hwndMain = CreateWindow (szMainWndClass, "Hello", WS_OVERLAPPEDWINDOW,
                             CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                             CW_USEDEFAULT, NULL, NULL, hInst, NULL);
    ShowWindow (hwndMain, nShow);
    UpdateWindow (hwndMain);
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}
```

The full source code and a makefile is available at http://www.comp.nus.edu.sg/~cs1101c/hugh/generic.tgz. When this is compiled, we get this:



Unfortunately, the full source code totals over 400 lines of C, and still doesn't actually do anything! If you are interested in learning more about Win32 programming, there is an interesting tutorial at http://www.winprog.org/tutorial.

10.3 OO GUI toolkits

If we can reduce the size of code, we can be more assured that our code is correct. We are already using one mechanism to do this - we call Win32 or C functions which do quite complex things (such as the call to CreateWindow() in the Win32 code above. A view of this might be that we are *hiding* the difficult parts from our application.

However, this sort of functional hiding has some flaws - our programs must maintain various sets of data representing our GUI environment, and can easily cause errors. Object-oriented technology provides a better mechanism, hiding the data from our programs - so that the only way in which the programs can modify the data is by using “approved” routines. Discussion of object-oriented technology is outside the framework of this course, but we can summarize the key features of OO as:

- OO code is considered to be more reuseable - due to a *software component* orientation.
- OO components are related through a new relationship - *inheritance*.
- OO objects may take different forms during run-time. This is called *polymorphism*.

Unfortunately, there is no one object-oriented standard for GUI applications, leading to a fragmented situation - there are literally hundreds of commercial and free OO GUI framework/toolkits, with no one clear leader in the market place, although the Microsoft Foundation Classes (MFC) should be mentioned.

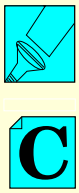
Here is a part of a small C GUI application built using the Qt GUI toolkit:

CODE LISTING **SmallQtApp.c**

```
#include <qapplication.h>
#include "application.h"

int
main (int argc, char **argv)
{
    QApplication a (argc, argv);
    ApplicationWindow *mw = new ApplicationWindow ();
    mw->setCaption ("Document 1");
    mw->show ();
    a.connect (&a, SIGNAL (lastWindowClosed ()), &a, SLOT (quit ()));
    return a.exec ();
}
```

10.4 Scripting languages



Scripting languages which can produce GUI interfaces are relatively easy to use. An effective strategy for building GUI applications is to write the GUI part in a scripting language, and to write the core 'difficult' part in C.

In the following example, a Tcl/Tk program is integrated with a C program, giving a very small code-size GUI application, that can be compiled on any platform - windows, UNIX or even the Macintosh platform without changes.

CODE LISTING **CplusTclTk.c**

```
#include <stdio.h>
#include <tcl.h>
#include <tk.h>

char tclprog[] = "\
     proc fileDialog {w} {\
       set types {\
         { \"Image files\" {.gif} }\
         { \"All files\" *}\
       };\
       set file [tk_getOpenFile −filetypes $types −parent $w];\
       image create photo picture −file $file;\
       set glb_tx [image width picture];\
       set glb_ty [image height picture];\
       .c configure −width $glb_tx −height $glb_ty;\
       .c create image 1 1 −anchor nw −image picture −tags \"myimage\";\
     };\
     frame .mbar −relief raised −bd 2;\
     frame .dummy −width 10c −height 0;\
     pack .mbar .dummy −side top −fill x;\
     menubutton .mbar.file −text File −underline 0 −menu .mbar.file.menu;\
     menu .mbar.file.menu −tearoff 1;\
     .mbar.file.menu add command −label \"Open...\" −command \"fileDialog .\";\
     .mbar.file.menu add separator;\
     .mbar.file.menu add command −label \"Quit\" −command \"destroy .\";\
     pack .mbar.file  −side left;\
     canvas .c −bd 2 −relief raised;\
     pack .c −side top −expand yes −fill x;\
     bind . <Control−c> {destroy .};\
     bind . <Control−q> {destroy .};\
     focus .mbar";

int
main (argc, argv)
     int argc;
     char **argv;
{
    Tk_Window mainWindow;
    Tcl_Interp *tcl_interp;

    setenv ("TCL_LIBRARY", "/cygnus/cygwin−b20/share/tcl8.0");
    tcl_interp = Tcl_CreateInterp ();
    if (Tcl_Init (tcl_interp) != TCL_OK || Tk_Init (tcl_interp) != TCL_OK) {
        if (*tcl_interp->result)
            (void) fprintf (stderr, "%s: %s\n", argv[0], tcl_interp->result);
        exit (1);
    }
    mainWindow = Tk_MainWindow (tcl_interp);
    if (mainWindow == NULL) {
        fprintf (stderr, "%s\n", tcl_interp->result);
        exit (1);
    }

    Tcl_Eval (tcl_interp, tclprog);
    Tk_MainLoop ();

    exit (1);
}
```

The first half of the listing is a C string containing a Tcl/Tk program. The second part of the listing is C code which uses this Tcl/Tk.

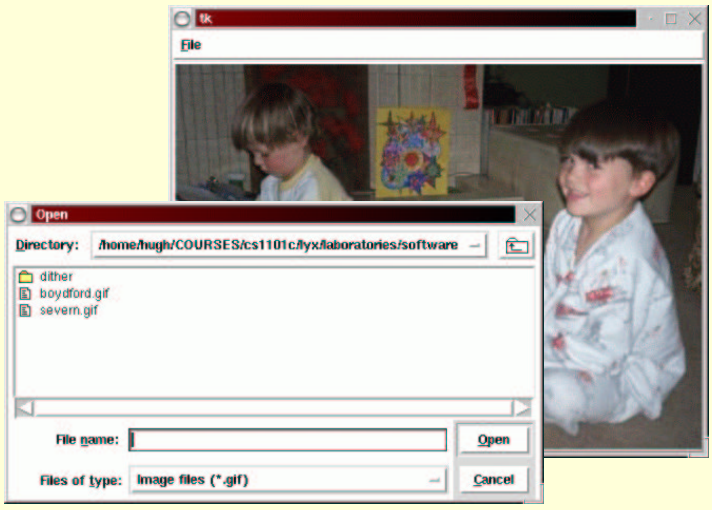
On a Win32 system, we compile this as:

```
gcc -o CplusTclTk CplusTclTk.c -mwindows -ltcl80 -ltk80
```

On a UNIX system we use:

```
gcc -o CplusTclTk CplusTclTk.c -ltk -ltcl -lX11 -lm -ldl
```

And the result is a simple viewer for GIF images. The total code size is 57 lines. The application looks like this when running:



10.5 Summary of topics

In this section, we introduced the following topics:

- Different software architectures
- Graphical user interface techniques
- WIN32 programming
- Object Oriented GUI toolkits
- Scripting languages

Questions

1. Try out each of the programs given here.
2. Investigate the MFC. What is the simplest “Hello-World” program that can be written using it?
3. In object-oriented terms, differentiate between a class and an object.
4. Given the *callback* system architecture outlined in this chapter, what does the software do when no events are happening?

Further study

- There is nothing in the textbook specifically related to C and GUI, but chapter 29 has a discussion on GUIs, and programming them in Java.

Chapter 11

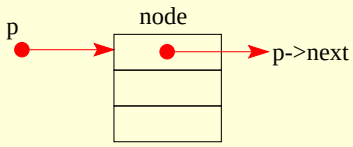
Arbor day

In this section, we look at ways in which we can combine simple variables, arrays, structs and pointers to make more complex user-defined data structures.

The reason we build these more complex structures is to assist our program structure to match the structure of our problem. If our problem has a complex structure, then it is best if our program structure maps onto this structure in some obvious way.

Before showing sample code to manipulate these structures we introduce the C syntax for referring to structure elements using pointers. If we had a struct with components **key** and **next**, and a pointer **p** to that structure, then we refer to **p->key** and **p->next**.

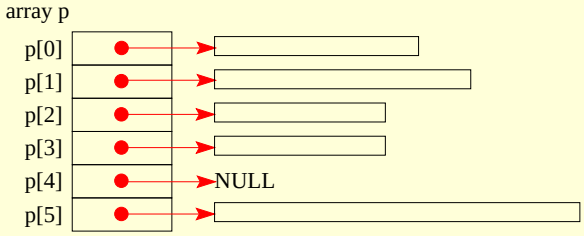
We also introduce a diagramming technique to help us remember the shape of user-defined data structures. A pointer is drawn as an arrow, a struct as a box.



In the above diagram we are looking at a pointer to a struct which itself has a pointer to something else.

11.1 Array of pointers

We use arrays of pointers when we know some upper bound on the number of items, but the items may be of different sizes (for example - arrays of strings). A suitable diagram is:



If a pointer does not point to anything (yet) then we label it as NULL or NIL. We should always remember to initialize our array elements as NULL.

In this example program we use an array of pointers to store some strings, which are created dynamically as needed using the **malloc()** call.

CODE LISTING **arrayofpointers.c**

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

int
main ()
{
    char str[80];
    char *strings[10];
    int i;

    for (i = 0; i < 10; i = i + 1) {
        fgets (str, 79, stdin);
        str[strlen (str) - 1] = '\0';
        strings[i] = (char *) malloc (strlen (str) + 1);
        strcpy (strings[i], str);
    }

    for (i = 9; i >= 0; i = i - 1) {
        printf ("line %d was: %s\n", i + 1, strings[i]);
    }

    return 0;
}
```

11.1.1 Parameters to programs

A *real* example of the use of an array of pointers is in the parameters to a program. The **argc** value is the count of the number of (text) arguments. The parameter **argv** is an array of pointers:

CODE LISTING **argcargv.c**

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    printf ("There are %d arguments (parameters) \n", argc);
    for (i = 0; i < argc; i = i + 1) {
        printf ("Argument number %d is %s.\n", i, argv[i]);
    }
    return 0;
}
```

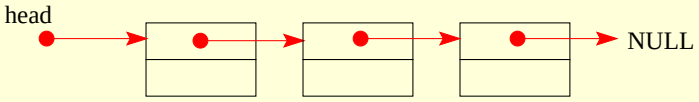
If we run this program, it displays its own arguments:

```
[hugh@stf-170 programs]$ gcc -o argcargv argcargv.c
[hugh@stf-170 programs]$ ./argcargv first second 3 -4
There are 5 arguments (parameters)
Argument number 0 is ./argcargv.
Argument number 1 is first.
Argument number 2 is second.
Argument number 3 is 3.
Argument number 4 is -4.
[hugh@stf-170 programs]$
```

When we type a command like "**del file1.txt**", the **del** command looks at its own argument to find out which file to delete.

11.2 Lists

The list structure can be extended as needed - when we need more items, we use malloc to create a new item, and then link it into the list. Our diagram is:



The list is initialized with

```
head = NULL;
```

To manipulate nodes on a list, we have basic operations which insert and delete nodes. However it is common for our lists to be kept in sorted order to speed searching and if our data needs to be kept sorted. We will look first at a simple list, and then at a sorted list. In the following codes we assume a **head** variable which is a pointer to the head of the list.

11.2.1 Simple singly-linked list

To insert nodes on the head of a simple list, we can just use:

```
mynode = malloc( sizeof( node ) );
mynode->next = head;
head = mynode;
```

To remove the first item from a list we can use[1]:

```
mynode = head;
head = mynode->next;
free( mynode );
```

To put an item at the end of the list[2]:

```
p = head;
while (p->next!=NULL)
   p = p->next;
mynode->next = NULL;
p->next = mynode;
```

The previous code segments are not an exhaustive list of needed code, but introduce the way in which we can manipulate data structures linked by pointers.

[1] This code fails to work if the head is NULL.
[2] This code also fails if the head is NULL.

11.2.2 Sorted singly-linked list

It is often useful to keep items in a list sorted according to some criteria. As a simple example, assume we have a key field in each of our list nodes:

```
struct node {
    struct node *next;
    int key;
    char *mydata;
} node;

typedef struct node *NODEPTR;

NODEPTR head=NULL;
```

If we wish to keep our list sorted in (say) ascending order, we do this by only inserting in the correct position[3]:

```
p1 = head;
p2 = p1->next'
while ( (p2!=NULL) && (p2->key<newitem->key) {
    p1 = p2;
    p2 = p1->next;
}
p1->next = newitem;
newitem->next = p2;
```

We will also need delete and search functions, but these are left as an exercise at the end of the chapter.

11.3 Binary tree

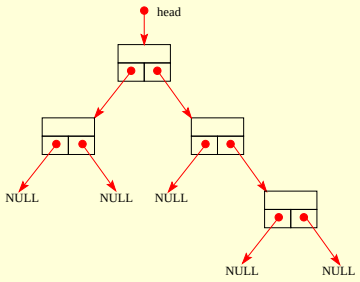
One property of a list structure is that the larger a list gets, the slower it is to search. In general, it goes up as $O(n)$. A balanced tree structure search time goes up as $O(\log_2 n)$, which is a much slower rate:

n	2	4	8	16	32	64
$\log_2 n$	1	2	3	4	5	6

Maintaining a *balanced* binary tree is a little difficult - and the simple tree structure given here may result in an unbalanced tree - so we may not get search times as good as $O(\log_2 n)$.

[3] This code also does not work if the head is NULL.

We visualize our binary tree as:



In the following code, we use these declarations:

```
struct node {
   struct node *lhs,*rhs;
   int key;
   char *mydata;
} node;

typedef struct node *NODEPTR;
NODEPTR head=NULL;
```

11.3.1 Creating a node

This code segment shows one method of creating a new node for the tree. Note the use of **malloc()** to create both the new node, and the string it contains. The function returns a *pointer* to the newly created node.

CODE LISTING **makenode.c**

```
NODEPTR
makenode (int key, char *mydata)
{
    NODEPTR p;
    p = (NODEPTR) malloc (sizeof (node));
    p->lhs = p->rhs = NULL;
    p->key = key;
    p->mydata = (char *) malloc (strlen (mydata) + 1);
    strcpy (p->mydata, mydata);
    return p;
}
```

This code segment shows one method of displaying all nodes in the tree. This is a recursive routine, as the tree is a recursive structure (as discussed in class). Our algorithm is:

Algorithm: 6 *Display binary tree*

If the tree is NULL
  do nothing,
but otherwise
  display the leftmost part of the tree,
  followed by this node,
  followed by the rightmost part of the tree".

Here is the (recursive) code for displaying the elements in a tree:

CODE LISTING **inordertree.c**

```
void
inorderdisplay (NODEPTR this)
{
    if (this != NULL) {
        inorderdisplay (this->lhs);
        printf ("Key: %6d, mydata: %s\n", this->key, this->mydata);
        inorderdisplay (this->rhs);
    }
}
```

11.3.2 Insert node in tree

This code segment shows one method of inserting a new node in the tree. If the keys are the same, it over-writes the existing record, and returns the storage no longer needed to the operating system.

CODE LISTING **insertnode.c**

```
NODEPTR
insert (NODEPTR hd, NODEPTR rec)
{
    if (hd == NULL)
        return rec;
    if (hd->key == rec->key) {
        free (hd->mydata);
        hd->mydata = rec->mydata;
        free (rec);
    } else {
        if (hd->key > rec->key) {
            hd->lhs = insert (hd->lhs, rec);
        } else {
            hd->rhs = insert (hd->rhs, rec);
        }
    }
    return hd;
}
```

11.3.3 Find node in tree

This code segment shows one method of finding a node in the tree.

CODE LISTING **findnode.c**

```
char *
find (int key, NODEPTR hd)
{
    if (hd == NULL) {
        return "Not found!";
    } else {
        if (hd->key == key) {
            return hd->mydata;
        } else {
            if (hd->key > key) {
                return find (key, hd->lhs);
            } else {
                return find (key, hd->rhs);
            }
        }
    }
}
```

11.4 Summary of topics

In this section, we introduced the following topics:

- A diagramming technique to help visualize complex data structures
- Code for managing arrays of pointers
- Code for managing simple singly-linked lists
- Algorithms and code for binary trees

Questions

1. In section 11.2.1, it is mentioned that the *delete* code fails to work if the head is NULL. Investigate this - why does it not work?
2. Fix the *delete* code in section 11.2.1 so that it does work even if the head is NULL.
3. In section 11.2.2, we give a flawed insert routine. Write and test a delete and search routine for the same list structure.
4. In section 11.3, a recursive routine to print out a tree is given. Rewrite this using a non-recursive method.

Further study

- Textbook, Chapter 12

Chapter 12

Building larger C programs

The programming style and techniques outlined in CS1101C are suitable for small(ish) software systems, but not for large scale software developments. In the laboratory exercises in this course, the largest programs are normally only 100 lines or so - and by now I am sure you are aware of how easy it is to make errors even in these small programs.

Medium sized software developments range up to about 50,000 lines of C source, and these developments will require tool support beyond the editor/compiler/debugger/make tools described so far. Tools such as configure, make, version control, automatic documentation and so on assist greatly with these sort of projects.

Large sized software developments may have 50,000,000 lines of code[1] or more, and generally involve significant investments. The software is engineered more carefully, with techniques ranging from strict design methodologies in which each step must be followed rigorously, through to analysis of human factors - workplace environment, ethos and so on.

In “The Mythical Man-Month”, Frederick P. Brooks gives a lot of advice about managing large software projects, and despite the book being 25 years old, the advice in it is still sound. The assertion that

*1 programmer * 9 months does not equal 9 programmers * 1 month*

is still absolutely correct[2].

In this chapter we will briefly introduce some of the smaller tools that may be used to assist in the management of medium-sized software developments. The notes are taken from the on-line documentation for the tools.

[1] The open source Mozilla web browser has around 30,000,000 lines of C.

[2] Isn’t there a joke about the New Zealander who was in a hurry to start raising a family - so rather than wait 9 months with one woman, he planned to only wait one month...

12.1 Revision control systems

There are many revision control systems. They help a programmer, or group of programmers, keep older revisions of software - particularly when there are multiple files as in larger software developments. Here is some information about one revision control system (RCS), taken from the documentation.

The Revision Control System (RCS) manages multiple revisions of files. It automates the storing, retrieval, logging, identification, and merging of revisions, and is useful for text that is revised frequently, for example programs, documentation, graphics, papers, and form letters. The functions of RCS are to:

- **Store and retrieve** multiple revisions of text. RCS saves all old revisions in a space efficient way. Changes no longer destroy the original, because the previous revisions remain accessible.
- **Maintain a complete history** of changes. RCS logs all changes automatically. Besides the text of each revision, RCS stores the author, the date and time of check-in, and a log message summarizing the change.
- **Resolve access conflicts.** When two or more programmers wish to modify the same revision, RCS alerts the programmers and prevents one modification from corrupting the other.
- **Maintain a tree of revisions.** RCS can maintain separate lines of development for each module.
- **Merge revisions** and resolve conflicts. Two separate lines of development of a module can be coalesced by merging. If the revisions to be merged affect the same sections of code, RCS alerts the user about the overlapping changes.
- **Control releases** and configurations. Revisions can be assigned symbolic names and marked as released, stable, experimental, etc.
- **Minimize disk storage.** RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding *deltas* are compressed accordingly.

RCS is avaliable (for free) on all platforms and is the core technology in many of the commercial revision control systems.

12.2 Makefile

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.

To use make, you must write a file called the makefile that describes the relationships among files in your program, and the states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this command:

```
make
```

will perform all necessary recompilations.

The make program uses the *makefile* data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

Normally you should call your *makefile* either makefile or Makefile.

Here is a simple makefile which manages the recompilation of three C text files (one.c, two.c and main.c):

CODE LISTING **makefile**

```
all: main.exe

main.exe: one.o two.o main.o
        gcc -o main.exe one.o two.o main.o

one.o: one.c
        gcc -c one.c

two.o: two.c
        gcc -c two.c

main.o: main.c
        gcc -c main.c
```

Note that:

- The unindented lines are dependency descriptions
- The indented lines are compile-link descriptions - you must use a TAB, not SPACES.

12.3 Configure

The first step in compiling a new GNU package is often running the **configure** script in the source directory. This script is an autoconfiguration system which detects the system idiosyncracies of your computer - such as which operating system, which compiler and so on. The **configure** script automatically generates a Makefile which you can then use to build the new system.

The **configure** system is comprised of several different tools, and program developers must build and install all of these tools. However - people who just want to build programs from distributed sources normally do not need anything except a make program, and a C compiler.

autoconf provides a general portability framework, based on testing the features of the host system at build time.

automake a system for describing how to build a program, permitting the developer to write a simplified ‘Makefile’.

libtool a standardized approach to building shared libraries.

gettext provides a framework for translation of text messages into other languages.

m4 autoconf requires the GNU version of m4 - a macro processing language

perl automake requires perl.

12.4 Summary of topics

In this section, we introduced the following topics:

- Software development in-the-large
- Revision control systems
- The makefile
- The GNU configure system

Questions

1. Write a Makefile which manages the recompilation of laboratory exercise u4ex6.
2. Download the chess game program from ftp://ftp.gnu.org/pub/gnu/chess/. Run "./configure –unix" and then "make".

Further study

- Textbook, Chapter 14, sections 14.5

Contents