An Introduction to C Programming

Ian D Chivers

February 18, 2019

Aims

- The aim of the notes are to provide an introduction to the basic syntax and use of C. The examples in the notes solve a common set of programming problems that should cover enough C to get started. It is assumed that you can already program.

Contents

List of Tables

Chapter 1

History

> *"We have to go to another language in order to think clearly about the problem."*
>
> Samuel R. Delany, Babel-17

1.1 Introduction

In this chapter we provide some of the history of the C programming language. There are several stages in the development of C, and they are listed below.

- The early days
- The Kernighan and Ritchie book - K and R C
- The first standard
- C99
- C11

1.2 The early days, late sixties and early seventies

There is a requirement in computing to be able to access the underlying machine directly or at least efficiently. It is therefore not surprising that computer professionals have developed high-level languages to do this. This may well seem a contradiction, but it can be done to quite a surprising degree. Some of the earliest published work was that of Martin Richards on the development of BCPL.

This language directly influenced the work of Ken Thompson and can be clearly seen in the programming languages B and C. The UNIX operating system is almost totally written in C and demonstrates very clearly the benefits of the use of high-level languages wherever possible.

With the widespread use of UNIX within the academic world C gained considerable ground during the 1970s and 1980s.

1.3 The Kernighan and Ritchie book - K&R C, 1978

UNIX systems also offered much to the professional software developer, and became widely used for large-scale software development and as Ritchie says: "C is quirky, flawed, and an enormous success. while accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

Eventually Kernighan and Richie wrote their seminal book, and most people at the beginning will have learnt C from this book

1.4 The first standard - 1989

In 1989 the American National Standards Institute published the ANSI C or C89 standard. It became an ISO standard a year later. The second edition of the K&R book covers the ANSI C standard. ISO later released an extension to the internationalization support of the standard in 1995, and a revised standard (C99) in 1999.

1.5 C99

C99 introduced several new features, including inline functions, several new data types (including long long int and a complex type to represent complex numbers), variable-length arrays, improved support for IEEE 754 floating point, support for variadic macros (macros of variable arity), and support for one-line comments beginning with // which are part of C++. This increased the compatibility of C and C++. Many of these had already been implemented as extensions in several C compilers.

1.6 C11

The current version of the standard (C11) was approved in December 2011.

The C11 standard adds several new features to C and the library, including type generic macros, anonymous structures, improved Unicode support, atomic operations, multi-threading, and bounds-checked functions. It improved compatibility with C++.

1.7 Online resources

Our home site is

`http://www.rhymneyconsulting.co.uk/`

The C material can be found at

`http://www.rhymneyconsulting.co.uk/c/`

A pdf of the notes, the examples and a copy of one of the standards can be found there.

Even though the following site targets the C++ programmer as C is a subset of C++ the information here about the C standard library is extremely useful.

`http://www.cplusplus.com/reference/`

Well worth bookmarking in your web browser.

1.8 Bibliography

ACM SIG PLAN, History of programming Languages Conference — HOPL-II, ACM Press, 1993.

- One of the best sources of information on C++, CLU, Concurrent Pascal, Formac, Forth, Icon, Lisp, Pascal, Prolog, Smalltalk and Simulation Languages by the people involved in the original design and or implementation. Very highly recommended. This is the second in the HOPL series, and the first was edited by Wexelblat. Details are given later.

Bergin T.J., Gibson R.G., History of Programming Languages, Addison-Wesley, 1996.

- This is a formal book publication of the Conference Proceedings of HOPL II. The earlier work is based on preprints of the papers.

Harbison S.P., Steele G.L., A C Reference Manual, Prentice-Hall, 2002.

- Very good coverage of the various flavours of C, including K&R C, Standard C 1989, Standard C 1995, Standard C 1999 and Standard C++

Kernighan B.W., Ritchie D.M., The C programming Language, Prentice-Hall; first edition 1978; second edition 1988.

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

Sammet J., programming Languages: History and Fundamentals, Prentice-Hall, 1969.

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

Stroustroup B., The C++ Programming Language, Addison-Wesley; third edition 1997; fourth edition 2014. 1997.

- The C++ book. Written by the designer of the language. The third edition is a massive improvement over the earlier editions. The fourth edition covers C++11. One of the best books on C++ and C++11 in particular.

Wexelblat, History of programming Languages, HOPL I, ACM Monograph Series, Academic Press, 1978.

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

Chapter 2

Compilers

2.1 Introduction

In this chapter we look at three free C compilers.

- The gnu compiler
- The Microsoft compiler
- The Oracle Solaris Studio compiler suite

2.2 The gnu compiler suite

This is available from

`https://gcc.gnu.org/`

Here is a quote from their site

> The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...). GCC was originally written as the compiler for the GNU operating system. The GNU system was developed to be 100% free software, free in the sense that it respects the user's freedom.
>
> We strive to provide regular, high quality releases, which we want to work well on a variety of native and cross targets (including GNU/Linux), and encourage everyone to contribute changes or help testing GCC. Our sources are readily and freely available via SVN and weekly snapshots.
>
> Major decisions about GCC are made by the steering committee, guided by the mission statement.

2.3 Microsoft's compiler

As C is a subset of C++ Microsoft provide a very good C compiler for the Windows platform. They do not track the C standard as well as the Gnu team, but the compiler is still capable of compiling a lot of existing C code.

Visit

`http://www.visualstudio.com/en-US/products/visual-studio-community-vs`

This provides access to what Microsoft call Visual Studio Community Edition, which is their IDE with C++ (and hence C), C#, Visual Basic and F#.

2.4 Oracle Solaris Studio

Visit

`http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`

Oracle Solaris Studio is a combined IDE and compiler suite. You get C, C++ and Fortran compilers that can be run from the command line or from within their IDE.

Here is some of their blurb from their site.

> Oracle Solaris Studio delivers a complete and comprehensive development platform, with high efficiency, high performance and high value. The tools are targeted towards making it as easy as possible to develop the best applications for Oracle Solaris and Linux operating systems. All of the components of Oracle Solaris Studio are designed, tested, and integrated to work together to help maximize developer efficiency. The Oracle Solaris Studio compilers are optimized to deliver the highest performance on the latest Oracle systems. In addition, with Oracle Solaris Studio customers are able to leverage innovations across the Oracle technology stack, including SPARC M7 Software in Silicon and custom extensions for Oracle Database and Oracle Tuxedo development. Oracle Solaris Studio contains two major suites of tools, a Compiler Suite and an Analysis Suite. The tools within each of the suites are designed to work together to provide an optimized development environment for the development of serial and parallel applications. Oracle Solaris Studio also comes with an integrated development environment (IDE) tailored for use with the compilers and tools from both suites. Oracle Solaris Studio provides a robust and reliable development environment with tools that are optimized for the underlying operating system and hardware to help you create secure, reliable, high-performance applications.

The product is free for Linux systems.

2.5 Compiling the examples

The examples can be found at

`http://www.rhymneyconsulting.co.uk/c/examples/`

As of February 17 2019 we have the following compilation details

- gcc 7.3.0 (Redhat cygwin, Windows) compiles all of the examples
- gcc 9.0.0 (equation.com, Windows) compiles all of the examples
- Microsoft C++ 19.13.26129 for x64 compiles all but 2 of the examples. Warnings are issued some of the examples. See below.
- sun cc (OpenSuse Leap) has problems with some of the examples. See below.
- gcc 7.3.1 (OpenSuse Leap) has problems with some of the examples. See below.

2.5.1 Microsoft C++ error messages

Here is a snapshot of the error messages.

```
ERRORS - no support for VLAs

C:\document\c\c_notes>cl c1401.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26129 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

c1401.c
c1401.c(4): error C2057: expected constant expression
c1401.c(4): error C2466: cannot allocate an array of constant size 0
c1401.c(11): error C2057: expected constant expression
c1401.c(11): error C2466: cannot allocate an array of constant size 0
c1401.c(11): error C2087: 'x': missing subscript
c1401.c(34): error C2057: expected constant expression
c1401.c(34): error C2466: cannot allocate an array of constant size 0
c1401.c(34): error C2133: 'x': unknown size
c1401.c(35): error C2057: expected constant expression
c1401.c(35): error C2466: cannot allocate an array of constant size 0
c1401.c(35): error C2087: 'x1': missing subscript
c1401.c(35): error C2133: 'x1': unknown size

C:\document\c\c_notes>cl c1402.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26129 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
c1402.c
c1402.c(50): error C2057: expected constant expression
c1402.c(50): error C2466: cannot allocate an array of constant size 0
c1402.c(59): error C2057: expected constant expression
c1402.c(59): error C2466: cannot allocate an array of constant size 0
c1402.c(59): error C2133: 'temp': unknown size

WARNINGS

c0401.c
c0401.c(23): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(23): note: consider using '%zd' in the format string
c0401.c(25): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(25): note: consider using '%zd' in the format string
c0401.c(27): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(27): note: consider using '%zd' in the format string
c0401.c(29): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(29): note: consider using '%zd' in the format string
c0401.c(31): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(31): note: consider using '%zd' in the format string
c0401.c(33): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(33): note: consider using '%zd' in the format string
c0401.c(35): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(35): note: consider using '%zd' in the format string
c0401.c(37): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(37): note: consider using '%zd' in the format string
c0401.c(39): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(39): note: consider using '%zd' in the format string
c0401.c(41): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(41): note: consider using '%zd' in the format string
c0401.c(43): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(43): note: consider using '%zd' in the format string
c0401.c(45): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
```

```
c0401.c(45): note: consider using '%zd' in the format string
c0401.c(47): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(47): note: consider using '%zd' in the format string
c0401.c(49): warning C4477: 'printf' : format string '%2d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0401.c(49): note: consider using '%zd' in the format string
Microsoft (R) Incremental Linker Version 14.13.26129.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c0401.exe
c0401.obj

C:\document\c\c_notes>cl c0601.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26129 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

c0601.c
c0601.c(16): warning C4477: 'printf' : format string '%d'
requires an argument of type 'int', but variadic argument 1 has type '__int64'
c0601.c(16): note: consider using '%lld' in the format string
c0601.c(16): note: consider using '%Id' in the format string
c0601.c(16): note: consider using '%I64d' in the format string
c0601.c(20): warning C4477: 'printf' : format string '%u'
requires an argument of type 'unsigned int', but variadic argument 1 has type 'unsign
c0601.c(20): note: consider using '%llu' in the format string
c0601.c(20): note: consider using '%Iu' in the format string
c0601.c(20): note: consider using '%I64u' in the format string
c0601.c(34): warning C4477: 'printf' : format string '%d'
requires an argument of type 'int', but variadic argument 1 has type '__int64'
c0601.c(34): note: consider using '%lld' in the format string
c0601.c(34): note: consider using '%Id' in the format string
c0601.c(34): note: consider using '%I64d' in the format string
c0601.c(39): warning C4477: 'printf' : format string '%d'
requires an argument of type 'int', but variadic argument 1 has type 'unsigned __int6
c0601.c(39): note: consider using '%lld' in the format string
c0601.c(39): note: consider using '%Id' in the format string
c0601.c(39): note: consider using '%I64d' in the format string
Microsoft (R) Incremental Linker Version 14.13.26129.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c0601.exe
c0601.obj

C:\document\c\c_notes>cl c0805.c
```

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26129 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

c0805.c
c0805.c(7): warning C4477: 'printf' : format string '%d'
requires an argument of type 'int', but variadic argument 1 has type '::size_t'
c0805.c(7): note: consider using '%zd' in the format string
Microsoft (R) Incremental Linker Version 14.13.26129.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c0805.exe
c0805.obj

C:\document\c\c_notes>cl c1107.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26129 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

c1107.c
c1107.c(18): warning C4477: 'scanf' : format string '%d'
requires an argument of type 'int *', but variadic argument 1 has type 'int'
Microsoft (R) Incremental Linker Version 14.13.26129.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c1107.exe
c1107.obj

C:\document\c\c_notes>cl c1202.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26129 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

c1202.c
c:\document\c\c_notes\c1202.c(7) :
warning C4700: uninitialized local variable 'p_i' used
Microsoft (R) Incremental Linker Version 14.13.26129.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c1202.exe
c1202.obj
```

2.5.2 Linux Sun cc error messages

Here is a summary of the error messages.

```
gcc version 7.3.1 20180323
==========================
```

```
[gcc-7-branch revision 258812]
(SUSE Linux)

c1103.c:(.text+0xfa): undefined reference to ‘sqrt’
c1301.c:(.text+0xd5): undefined reference to ‘sin’
c1402.c:(.text+0x2db): undefined reference to ‘sqrt’
```

i.e. 79 out of 82 compile.

2.5.3 Linux gcc error messages

Here is a summary of the error messages.

```
cc: Studio 12.6 Sun C 5.15
==========================
Linux_i386 2017/05/30
ERRORS
c0603.c:(.text+0x1fb): undefined reference to ‘pow’
c1103.c:(.text+0x2fc): undefined reference to ‘sqrt’
c1301.c:(.text+0x1fa): undefined reference to ‘atan’
c1402.c:(.text+0x524): undefined reference to ‘sqrt’
/usr/bin/ld: c1904.o: undefined reference to symbol ’atan@@GLIBC_2.2.5’

WARNINGS
"c0709.c", line 27: Warning: Likely out-of-bound read:
 *(y[((llong)i)]) in function main
"c0709.c", line 27: Warning: Likely out-of-bound read:
 *(x[((llong)i)]) in function main
"c1202.c", line 8: Warning:
Likely uninitialized read (variable p_i): main
```

i.e. 5 fail to compile and 3 issue warnings.

2.6 Summary

The gcc suite runs on most platforms, the Microsoft compiler obviously runs on Windows and the Oracle product run on Solaris and Linux.

Chapter 3

Simple examples

3.1 Introduction

In this chapter we look at some simple C examples.

3.2 Example 1 - Hello world

Here is the program.

```
#include <stdio.h>

int main()
{
  printf("Hello world \n");
  return(0);
}
```

Let us look at each line in turn.

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

C arranges language functionality using a set of standard libraries. This include line makes available functionality for standard input and output. It is a so called header file. A header file is a file with extension .h which contains C function declarations and macro definitions. The actual implementation of this functionality is provided by the compiler during the compilation process. The

```
<>
```

symbols are used for system or compiler provided header files. User supplied header files use double quotes ".

```
int main()
```

C programs are made up of functions. There are two types of functions in C, void functions and those with a type. The main program is of int type. On successfully completion or execution the program will return a value of 0 or zero.

```
{
}
```

Curly braces are used in C to organise code. The left { is used to start a block, and the right } is used to close a block.

```
  printf("Hello world \n");
```

This is an output statement. It is a printf statement. The text in double quotes will appear on the screen when you run the program. The

```
\n
```

is a control sequence that generates a line feed.

```
  return(0);
```

This terminates or ends the program. A value of 0 or zero is returned to the operating system.

3.3 Example 2 - Simple text input and output

Here is the second example which does some simple text input and output.

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

int main()
{
  char line[80];
  printf("Type in a line of text\n");
  scanf("%s",line);
  printf(" %s \n",line);
  return(0);
}
```

Let us look at each line in turn.

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

In this example we also include the string.h header file. This header file defines several functions to manipulate C strings and arrays.

```
char line[80];
```

This statement declares line to be a variable of type char, and be an array that can hold up to 80 characters. C uses the [] characters to denote arrays. We will look at arrays in more depth in a later chapter.

The

```
scanf("%s",line);
```

statement reads in a line of text using scanf. The %s says read the data as text.

3.4 Example 3 - Simple integer input and output

```
#include <stdio.h>

int main()
{
  int x;
  printf("Type in an integer\n");
  scanf("%d",&x);
  printf(" %d \n",x);
  return(0);
}
```

Let us look at this example now. First we have

```
int x;
```

the declaration of a variable x, and it is of type int, which is one of the build in C types.

The next statement of note is the

```
scanf("%d",&x);
```

statement. This statement is used to read formatted input.

The

```
%d
```

is a type specifier, and says that we are trying to read a decimal integer. There are type specifiers for all of the C supported types.

We are trying to read the integer value into the variable x. When passing parameters in C we need to pass by address or as pointers if we need to update the variable, as we do in the case of the variable x.

So this is why we have

```
&x
```

as a parameter to the scanf function.

3.5 Example 4 - Simple float input and output

```
#include <stdio.h>

int main()
{
  float x;
  printf("Type in a real number\n");
  scanf("%f",&x);
  printf(" %f \n",x);
  return(0);
}
```

This example reads a float, and we declare x to be of type float in this example, and use the f type specifier in the scanf function.

3.6 Example 5 - Simple double input and output

In the following example we try reading a double.

```
#include <stdio.h>

int main()
{
  double x;
  printf("Type in a real number\n");
  scanf("%lf",&x);
  printf(" %f \n",x);
  return(0);
}
```

Now we declare x to be of type double, and use the lf type specifier in the scanf statement.

3.7 Problems

1. Compiler and run each of the examples with whatever C compiler you have access to.

2. When reading in a line of text what happens with white space?

3. What happens if you type in more than 79 characters?

4. Experiment with the integer example. How big a number can you type in?

5. Experiment with the float example. How big a number can you type? How small?

6. Repeat 5 for the double example.

3.8 Summary

This chapter looks at a small number of examples to get you familiar with compiling and running C programs.

We have introduced several C data types including

- char
- arrays of char
- int
- float
- double

We have introduced two of the C libraries

- stdio
- string

We have a more detailed coverage of these libraries in later chapters.

We need to use the include statement to make available the functionality in these libraries using the

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

header files.

We also introduced two statements for input and output, the

```
printf
scanf
```

statements.

We also introduced the use of format strings with printf and scanf.

We introduced the following format string specifiers:

```
%s for text
%d for integers
%f for floats
%lf for doubles
```

There is a separate chapter on input and output later where we look at this whole area in much more depth.

Chapter 4

Types

4.1 Intrinsic types

There are five standard signed integer types, designated as

- signed char
- short int
- int
- long int
- long long int

For each of the signed integer types, there is a corresponding (but different) unsigned integer type.

There are three real floating types, designated as

- float
- double
- long double

The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double.

There are three complex types, designated as

- float _Complex
- double _Complex
- long double _Complex

The type char, the signed and unsigned integer types, and the floating types are collectively called the basic types. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

The three types char, signed char, and unsigned char are collectively called the character types. The implementation shall define char to have the same range, representation, and behaviour as either signed char or unsigned char.

An enumeration comprises a set of named integer constant values. Each distinct enumeration constitutes a different enumerated type.

Integer and floating types are collectively called arithmetic types. Each arithmetic type belongs to one type domain: the real type domain comprises the real types, the complex type domain comprises the complex types.

The void type comprises an empty set of values; it is an incomplete object type that cannot be completed.

Boolean type When any scalar value is converted to _Bool, the result is 0 if the value compares equal to 0; otherwise, the result is 1.

Pointers A pointer to void may be converted to or from a pointer to any object type.

4.2 Example 1 - testing what is available

This example test what is available. The source code is given below.

```
#include <stdio.h>

int main()
{
  char                    c=1;
  signed char             sc=1;
  short int               si=1;
  int                     i=1;
  long int                li=1;
  long long int           lli=1;   // c++ 11

  unsigned char           usc=1;
  unsigned short int      usi=1;
  unsigned int            ui=1;
  unsigned long int       uli=1;
  unsigned long long int  ulli=1;   // c++ 11

  float                   f=1;
  double                  d=1;
  long double             ld=1;
```

```
  printf(" char c                 ");
  printf(" %2d \n", sizeof(c));
  printf(" signed char            ");
  printf(" %2d \n", sizeof(sc));
  printf(" short int              ");
  printf(" %2d \n", sizeof(si));
  printf(" int                    ");
  printf(" %2d \n", sizeof(i));
  printf(" long int               ");
  printf(" %2d \n", sizeof(li));
  printf(" long long int          ");
  printf(" %2d \n", sizeof(lli));
  printf(" unsigned char          ");
  printf(" %2d \n", sizeof(usc));
  printf(" unsigned short int     ");
  printf(" %2d \n", sizeof(usi));
  printf(" unsigned int           ");
  printf(" %2d \n", sizeof(ui));
  printf(" unsigned long int      ");
  printf(" %2d \n", sizeof(uli));
  printf(" unsigned long long int ");
  printf(" %2d \n", sizeof(ulli));
  printf(" float                  " );
  printf(" %2d \n", sizeof(f));
  printf(" double                 " );
  printf(" %2d \n", sizeof(d));
  printf(" long double            " );
  printf(" %2d \n", sizeof(ld));

  return(0);
}
```

This example also introduces one of the comment mechanisms available in C. Single line comments are shown in the statement below.

```
long long int lli=1; // c++ 11
```

where everything after the two slashes is a comment.
A multi-line comment is shown below.

```
/*
This is a comment
spanning several lines.
It opens with a
slash asterisk
```

```
and closes with a
asterisk slash
*/
```

gcc output from this program is shown below.

```
c:\document\c\c_notes>c0401
 char c                   1
 signed char              1
 short int                2
 int                      4
 long int                 4
 long long int            8
 unsigned char            1
 unsigned short int       2
 unsigned int             4
 unsigned long int        4
 unsigned long long int   8
 float                    4
 double                   8
 long double             16
```

Microsoft output is shown below.

```
C:\document\c\c_notes>cl c0401.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.31101 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

c0401.c
Microsoft (R) Incremental Linker Version 12.00.31101.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c0401.exe
c0401.obj

C:\document\c\c_notes>c0401
 char c                   1
 signed char              1
 short int                2
 int                      4
 long int                 4
 long long int            8
 unsigned char            1
 unsigned short int       2
 unsigned int             4
```

```
 unsigned long int        4
 unsigned long long int   8
 float                    4
 double                   8
 long double              8

C:\document\c\c_notes>
```

4.3 Example 2 - testing floating point support

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

int main()
{
  printf(" %g \n",FLT_MAX);
  printf(" %d \n",FLT_DIG);
  printf(" %g \n",FLT_EPSILON);
  printf(" %g \n",DBL_MAX);
  printf(" %d \n",DBL_DIG);
  printf(" %g \n",DBL_EPSILON);
  printf(" %g \n",LDBL_MAX);
  printf(" %d \n",LDBL_DIG);
  printf(" %g \n",LDBL_EPSILON);
  return(0);
}
```

gcc output is shown below.

```
c:\document\c\c_notes>c0402
 3.40282e+038
 6
 1.19209e-007
 1.79769e+308
 15
 2.22045e-016
 1.13302e-317
 18
 1.13302e-317
```

Microsoft output is shown below.

```
C:\document\c\c_notes>cl c0402.c
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.31101 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.
```

```
c0402.c
Microsoft (R) Incremental Linker Version 12.00.31101.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:c0402.exe
c0402.obj

C:\document\c\c_notes>c0402
 3.40282e+038
 6
 1.19209e-007
 1.79769e+308
 15
 2.22045e-016
 1.79769e+308
 15
 2.22045e-016

C:\document\c\c_notes>
```

4.4 Example 3 - testing integer support

The following program tests integer support.

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

int main()
{
  printf(" %d \n",CHAR_MAX);
  printf(" %d \n",CHAR_MIN);
  printf(" %d \n",INT_MAX);
  printf(" %d \n",INT_MIN);
  printf(" %d \n",LONG_MAX);
  printf(" %d \n",LONG_MIN);
  printf(" %d \n",SCHAR_MAX);
  printf(" %d \n",SCHAR_MIN);
  printf(" %d \n",SHRT_MAX);
  printf(" %d \n",SHRT_MIN);
  printf(" %u \n",UCHAR_MAX);
  printf(" %u \n",UINT_MAX);
  printf(" %u \n",ULONG_MAX);
  printf(" %u \n",USHRT_MAX);
```

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

Here is the output from gcc on a Windows platform.

```
127
-128
2147483647
-2147483648
2147483647
-2147483648
127
-128
32767
-32768
255
4294967295
4294967295
65535
```

4.5 Problems

1. Try the examples out with your compiler(s).

Chapter 5

Operators

5.1 Introduction

In this chapter we look briefly at the C operators. The standard calls them punctuators.

5.2 Punctuators

Here is a complete list of the C punctuators taken from the standard.

```
[]  ()  {} . ->
++  --  &  *  +  - ~ !
/  %  <<  >>  <  > <= >= == != ^ | && ||
?  :  ;  ...
=  *=  /=  %=  += -= <<= >>= &= ^= |=
,  #  ##
<:  :>  <%  %>  %: %:%:
```

5.3 Operators

Tables 5.1 5.2 summarises some of the information about the C operators.

There will be more complete examples of each of the following in later chapters. Here is a simple explanation for some of these operators.

- . [member selection] object.member This operator allows us to select a member of a class.
- -> [member selection] pointer -> member As above.
- [] [subscripting] pointer [] The normal array subscripting operator.
- () [function call] expr (expr_list) The function call operator.

- () [value construction] type(expr_list) Value construction mechanism.
- ++ [post increment] lvalue ++ Increment after use.
- – Decrement after use.
- sizeof [size of object] sizeof expr Used to determine the memory size of an object.
- sizeof [size of type] sizeof (type) Used to determine the memory size of a type.
- ++ [pre increment] ++ lvalue Increment before use.
- -- [pre decrement] -- lvalue Decrement before use.
- ~ [complement] ~ expr Ones complement operator. The operand must be of integral type. Integral promotions are performed. Also used to identify a destructor.
- ! [not] ! expr Logical negation operator.
- - [unary minus] - expr As stated.
- + [unary plus] + expr As stated.
- & [address of] & expr The result of the unary & operator is a pointer to its operand.
- * [dereference] * expr The unary * operator means indirection, and the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points.
- () [cast] (type) expr An explicit type conversion can be expressed using either functional notation or the cast notation.
- .* [member selection] object.* pointer_to_member The binary operator .* binds its second operand, which must be of type pointer to member of class T to its first operand, which must be of class T or of a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand
- ->* [member selection] pointer ->* pointer_to_member The binary operator ->* binds its second operand, which must be of type pointer to member of T to its first operand, which must be of type pointer to T or pointer to a class of which T is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.
- * [multiply] expr * expr Conventional arithmetic multiplication.
- / [divide] expr / expt Conventional arithmetic division.
- % [modulo or remainder] expr

- + [plus] expr + expr Conventional arithmetic addition.
- - [minus] expr - expr Conventional arithmetic subtraction.
- << [shift left] expr ¡<< expr Shift left. The operands must be of integral type and integral promotions are performed.
- >> [shift right] expr >> expr Shift right. The operands must be of integral type and integral promotions are performed.
- < [less than] expr < expr Conventional relational operator.
- <= [less than or equal] expr <= expr Conventional relational operator.
- > [greater than] expr ¿ expr Conventional relational operator.
- >= [greater than or equal] expr >= expr Conventional relational operator.
- == [equal] expr == expr Conventional relational operator.
- != [not equal] expr != expr Conventional relational operator.
- & [bitwise AND] expr & expr The usual arithmetic conversions are performed: the result is the bitwise and function of the operands. The operator applies only to integral operands.
- ^ [bitwise exclusive OR] expr ^ expr The usual arithmetic conversions are performed: the result is the bitwise exclusive or function of the operands. The operator applies only to integral operands.
- | [bitwise inclusive OR] expr — expr The usual arithmetic conversions are performed: the result is the bitwise inclusive or function of the operands. The operator applies only to integral operands.
- && [logical AND] expr && expr The operands are converted to type bool. The result is true if both operands are true and false otherwise. Left to right evaluation is guaranteed, and the second operand is not evaluated if the first is false. All side effects of the first expression except for destruction of temporaries happen before the second expression is evaluated.
- || [logical inclusive OR] expr || expr The operands are both converted to type bool. The result is true if either of its operands is true and false otherwise. Left to right evaluation is guaranteed, and the second operand is not evaluated if the first is true. All side effects of the first expression except for destruction of temporaries happen before the second expression is evaluated.
- ?: [conditional expression] expr ? expr : expr The first expression is converted to bool. It is evaluated and if it is true the result of the conditional expression is the value of the second expression, otherwise that of the third. All side effects of the first expression except for destruction of temporaries happen before the

second or third expression is evaluated. = [conventional assignment] lvalue = expr

- = Conventional assignment.
- *= [multiply and assign]
- *= lvalue *= expr Multiply and assign, e.g. a=a*expression
- /= [divide and assign] lvalue /= expr Divide and assign, e.g. a=a/expression
- `%=` [modulo and assign] lvalue `%=` expr Modulo and assign, e.g. a=a
- += [add and assign] lvalue += expr Add and assign, e.g. a=a+expression
- -= [subtract and assign] lvalue -= expr Subtract and assign, e.g. a=a-expression
- `<<=` [shift left and assign] lvalue `<<=` expr Shift left and assign
- `>>=` [shift right and assign] lvalue `>>=` expr Shift right and assign
- &= [AND and assign] lvalue &= expr AND and assign
- `|=` [inclusive OR and assign] lvalue `|=` expr inclusive OR and assign
- `^=` [exclusive OR and assign] lvalue `^=` expr exclusive OR and assign
- , [comma] expr , expr A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. All side effects of the left expression are performed before the evaluation of the right expression. The type and value of the result are the type and value of the right operand; the result is an lvalue if its right operand is.

C has the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- the rest

The miscellaneous operators are:

```
Operator     Description
sizeof()     Returns the size of an variable.
&            Returns the address of an variable.
*            Pointer to a variable.
? :          Conditional Expression
```

5.3.1 Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>=	
	<<= &= ^= |=	Right to left
Comma	,	Left to right

5.4 Summary

This chapter is a reference chapter. You have to be aware of the operators available in C and the C family of languages.

5.5 Problems

None in this chapter.

Table 5.1: Operator summary - continued in next table

Operator	Name	Example
.	member selection	object.member
->	member selection	pointer -> member
[]	subscripting	pointer [expr]
()	function call	expr (expr_list)
()	value construction	type(expr_list)
++	post increment	lvalue ++
–	post decrement	lvalue –
sizeof	size of object	sizeof expr
sizeof	size of type	sizeof (type)
++	pre increment	++ lvalue
–	pre decrement	– lvalue
	complement	expr
!	not	! expr
-	unary minus	- expr
+	unary plus	+ expr
&	address of	& expr
	dereference	* expr
()	cast	(type) expr
.*	member selection	object.* pointer_to_member
->*	member selection	pointer ->* pointer_to_member
	multiply	expr * expr
/	divide	expr / expt
+	plus	expr + expr
-	minus	expr - expr
<<	shift left	expr << expr
>>	shift right	expr >> expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equal	expr == expr
!=	not equal	expr != expr

Table 5.2: Operator summary - Continued

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

Chapter 6

Arithmetic

6.1 Introduction

In this chapter we have examples to illustrate arithmetic in C and also have a look at some operator examples.

6.2 Example 1 - Simple integer arithmetic

Here is a simple example that looks at integer arithmetic in C, and in particular what happens when the integers overflow.

```
#include <stdio.h>

int main()
{
  short int a_s = 1;
  int a = 1;
  long int a_l = 1;
  long long int a_ll=1;
  unsigned short int a_s_u = 1;
  unsigned int a_u = 1;
  unsigned long int a_l_u = 1;
  unsigned long long int a_ll_u=1;
    printf(" %d ",a_s );
    printf(" %d ",a );
    printf(" %d ",a_l );
    printf(" %d ",a_ll);
    printf(" %u ",a_s_u );
    printf(" %u ",a_u );
    printf(" %u ",a_l_u  );
    printf(" %u\n",a_ll_u);
    for (int i=0 ; i < 40 ; ++i)
```

```
    {
      a_s = a_s*2;
      a = a *2;
      a_l = a_l*2;
      a_ll = a_ll*2;
      a_s_u = a_s_u*2;
      a_u = a_u  *2;
      a_l_u = a_l_u*2;
      a_ll_u=a_ll_u*2;
      printf(" %d ",a_s );
      printf(" %d ",a );
      printf(" %d ",a_l );
      printf(" %d ",a_ll);

      printf(" %d ",a_s_u );
      printf(" %d ",a_u );
      printf(" %d ",a_l_u  );
      printf(" %d\n",a_ll_u);
    }
  return(0);
}
```

Here is the gcc output.

```
1  1  1  1  1  1  1  1
2  2  2  2  2  2  2  2
4  4  4  4  4  4  4  4
8  8  8  8  8  8  8  8
16  16  16  16  16  16  16  16
32  32  32  32  32  32  32  32
64  64  64  64  64  64  64  64
128  128  128  128  128  128  128  128
256  256  256  256  256  256  256  256
512  512  512  512  512  512  512  512
1024  1024  1024  1024  1024  1024  1024  1024
2048  2048  2048  2048  2048  2048  2048  2048
4096  4096  4096  4096  4096  4096  4096  4096
8192  8192  8192  8192  8192  8192  8192  8192
16384  16384  16384  16384  16384  16384  16384  16384
-32768  32768  32768  32768  32768  32768  32768  32768
0  65536  65536  65536  0  65536  65536  65536
0  131072  131072  131072  0  131072  131072  131072
0  262144  262144  262144  0  262144  262144  262144
0  524288  524288  524288  0  524288  524288  524288
0  1048576  1048576  1048576  0  1048576  1048576  1048576
```

```
0  2097152  2097152  2097152  0  2097152  2097152  2097152
0  4194304  4194304  4194304  0  4194304  4194304  4194304
0  8388608  8388608  8388608  0  8388608  8388608  8388608
0  16777216  16777216  16777216
   0  16777216  16777216  16777216
0  33554432  33554432  33554432
   0  33554432  33554432  33554432
0  67108864  67108864  67108864
   0  67108864  67108864  67108864
0  134217728  134217728  134217728
   0  134217728  134217728  134217728
0  268435456  268435456  268435456
   0  268435456  268435456  268435456
0  536870912  536870912  536870912
   0  536870912  536870912  536870912
0  1073741824  1073741824  1073741824
   0  1073741824  1073741824  1073741824
0  -2147483648  -2147483648  -2147483648
   0  2147483648  2147483648  2147483648
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
```

Here is the Microsoft output.

```
1  1  1  1  1  1  1  1
2  2  2  2  2  2  2  2
4  4  4  4  4  4  4  4
8  8  8  8  8  8  8  8
16  16  16  16  16  16  16  16
32  32  32  32  32  32  32  32
64  64  64  64  64  64  64  64
128  128  128  128  128  128  128  128
256  256  256  256  256  256  256  256
512  512  512  512  512  512  512  512
1024  1024  1024  1024  1024  1024  1024  1024
2048  2048  2048  2048  2048  2048  2048  2048
4096  4096  4096  4096  4096  4096  4096  4096
8192  8192  8192  8192  8192  8192  8192  8192
```

```
 16384  16384  16384  16384  16384  16384  16384  16384
-32768  32768  32768  32768  32768  32768  32768  32768
0  65536  65536  65536  0  65536  65536  65536
0  131072  131072  131072  0  131072  131072  131072
0  262144  262144  262144  0  262144  262144  262144
0  524288  524288  524288  0  524288  524288  524288
0  1048576  1048576  1048576  0  1048576  1048576  1048576
0  2097152  2097152  2097152  0  2097152  2097152  2097152
0  4194304  4194304  4194304  0  4194304  4194304  4194304
0  8388608  8388608  8388608  0  8388608  8388608  8388608
0  16777216  16777216  16777216
   0  16777216  16777216  16777216
0  33554432  33554432  33554432
   0  33554432  33554432  33554432
0  67108864  67108864  67108864
   0  67108864  67108864  67108864
0  134217728  134217728  134217728
   0  134217728  134217728  134217728
0  268435456  268435456  268435456
   0  268435456  268435456  268435456
0  536870912  536870912  536870912
   0  536870912  536870912  536870912
0  1073741824  1073741824  1073741824
   0  1073741824  1073741824  1073741824
0  -2147483648  -2147483648  -2147483648
   0  2147483648  2147483648  2147483648
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
```

Integer overflow has occurred. Section 3.4.3 of the 2011 standard classifies this as undefined behaviour. Here is the note from the standard about this.

3.4.3.2 NOTE Possible undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

6.3 Example 2 - Simple floating point arithmetic

Here is a simple example looking at floats, doubles and long doubles.

```
#include <stdio.h>

int main()
{

  float       x_f  = 1.0;
  double      x_d  = 1.0;
  long double x_ld = 1.0;

  int i;

  for (i=1;i<350;i++)
  {
    printf(" %4d %12g %12g %12g \n",i,x_f,x_d,x_ld);
    x_f  *= 10.0;
    x_d  *= 10.0;
    x_ld *= 10.0;
  }

  return(0);

}
```

Run the program and have a look at the gcc and Microsoft output.
Here is the gcc output on a Windows system. This uses gcc 4.9.3 under cygwin.

```
   1            1            1 2.12199e-314
   2           10           10 2.12199e-314
   3          100          100 2.12199e-314
   4         1000         1000 2.12199e-314
   5        10000        10000 2.12199e-314
   6       100000       100000 2.12199e-314
   7        1e+06        1e+06 2.12199e-314
   8        1e+07        1e+07 2.12199e-314
   9        1e+08        1e+08 2.12199e-314
  10        1e+09        1e+09 2.12199e-314
  11        1e+10        1e+10 2.12199e-314
  12        1e+11        1e+11 2.12199e-314
...
...
  36        1e+35        1e+35 2.12199e-314
  37        1e+36        1e+36 2.12199e-314
```

```
   38        1e+37        1e+37 2.12199e-314
   39        1e+38        1e+38 2.12199e-314
   40          inf        1e+39 2.12199e-314
   41          inf        1e+40 2.12199e-314
   42          inf        1e+41 2.12199e-314
   43          inf        1e+42 2.12199e-314
   44          inf        1e+43 2.12199e-314
...
...
  305          inf       1e+304 2.12199e-314
  306          inf       1e+305 2.12199e-314
  307          inf       1e+306 2.12199e-314
  308          inf       1e+307 2.12199e-314
  309          inf       1e+308 2.12199e-314
  310          inf          inf 2.12199e-314
  311          inf          inf 2.12199e-314
  312          inf          inf 2.12199e-314
```

Here is the Microsoft output on the same platform.

```
    1            1            1            1
    2           10           10           10
    3          100          100          100
    4         1000         1000         1000
    5        10000        10000        10000
    6       100000       100000       100000
    7       1e+006       1e+006       1e+006
    8       1e+007       1e+007       1e+007
...
...
   36       1e+035       1e+035       1e+035
   37       1e+036       1e+036       1e+036
   38       1e+037       1e+037       1e+037
   39       1e+038       1e+038       1e+038
   40       1.#INF       1e+039       1e+039
   41       1.#INF       1e+040       1e+040
   42       1.#INF       1e+041       1e+041
   43       1.#INF       1e+042       1e+042
   44       1.#INF       1e+043       1e+043
   45       1.#INF       1e+044       1e+044
   46       1.#INF       1e+045       1e+045
...
...
  305       1.#INF       1e+304       1e+304
  306       1.#INF       1e+305       1e+305
```

```
  307       1.#INF       1e+306       1e+306
  308       1.#INF       1e+307       1e+307
  309       1.#INF       1e+308       1e+308
  310       1.#INF       1.#INF       1.#INF
  311       1.#INF       1.#INF       1.#INF
```

Both only support IEEE 32 and 64 bit reals on the Windows platform.

Here is the output from gcc 4.8.1 on openSuSe 13.1 on the same platform. This is a dual boot Windows and Linux system.

```
    1            1            1            0
    2           10           10            0
    3          100          100            0
    4         1000         1000            0
    5        10000        10000            0
    6       100000       100000            0
    7        1e+06        1e+06            0
    8        1e+07        1e+07            0
...
...
   33        1e+32        1e+32            0
   34        1e+33        1e+33            0
   35        1e+34        1e+34            0
   36        1e+35        1e+35            0
   37        1e+36        1e+36            0
   38        1e+37        1e+37            0
   39        1e+38        1e+38            0
   40          inf        1e+39            0
   41          inf        1e+40            0
   42          inf        1e+41            0
   43          inf        1e+42            0
...
...
  304          inf       1e+303            0
  305          inf       1e+304            0
  306          inf       1e+305            0
  307          inf       1e+306            0
  308          inf       1e+307            0
  309          inf       1e+308            0
  310          inf          inf            0
  311          inf          inf            0
  312          inf          inf            0
```

There is support for IEEE 32 bit float and IEEE 64 bit double.

6.4 Example 3 - Time taken to travel from the Sun to the Earth

This program calculates the time taken to travel from the Sun to the Earth.

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


int main()
{
  double light_minute;
  double light_second;
  double distance;
  double elapse_minutes;
  double elapse_seconds;
  int   minute;
  int   second;

//                      12
// light year  9.46 * 10
//
// i.e. 9.46 * 10 to the power 12

  const double light_year=9.46*pow(10.0,12);
  light_minute=light_year/(365.25*24.0*60.0);
  light_second=light_minute/60.0;
  distance = 150.0 * pow(10.0,6);
  elapse_minutes = distance/light_minute;
  elapse_seconds = distance/light_second;
  minute   = elapse_minutes;
  second   = (elapse_minutes-minute)*60;

  printf(" Light takes ");
  printf(" %d ",minute);
  printf(" minutes ");
  printf(" %d ",second);
  printf(" seconds\n");
  printf(" to reach the earth from the sun \n");
  printf(" or %7.3f seconds\n",elapse_seconds);

  return(0);
}
```

Here is the output.

```
Light takes  8  minutes  20  seconds
to reach the earth from the sun
or 500.385 seconds
```

6.5 Problems

1. Broadband speed over telephone land lines is related to the distance you are from the exchange. In rural areas many places are too far from the exchange to have a working internet connection.

A lack of masts in sparsely populated areas can also cause problems with mobile broadband.

One option therefore is satellite based broadband.

A geosynchronous satellite orbit is around 35,870 km from the Earth. Calculate the round trip time between the Earth and a satellite in a geosynchronous orbit using the last example as a starting point.

Don't bother with the time in minutes.

2. I had an interview at the observatory at UCl when I applied to do a degree in astronomy, and got to see some moon dust. This problem has been added because of that interview.

The Moon is about 384,400 km from the Earth. Modify the above program to calculate the time taken for light to travel between the Earth and the moon.

3. The next problem has been added because of the 2015 science fiction film The Martian directed by Ridley Scott.

Mars is on average 225,000,000 Km from the Earth. Modify the above program to calculate the time taken for light to travel between the Earth and Mars.

6.6 Example 4 - Truncation across the assignment operator

This example looks at mixing integers and doubles.

```
#include <stdio.h>

int main()
{
  double a,b,c;
  int i;
    a=1.5;
    b=2.0;
    c=a/b;
    i=a/b;
    printf(" a =  %g \n",a);
    printf(" b =  %g \n",b);
```

```
    printf(" c =  %g \n",c);
    printf(" i =  %d \n",i);
    return(0);
}
```

Here is the output.

```
 a =  1.5
 b =  2
 c =  0.75
 i =  0
```

6.7 Example 5 - Integer division

Here is the example.

```
#include <stdio.h>

int main()
{
  int i;
  int j;
  int k;
  double l;
  i=5;
  j=2;
  k=4;
  l=i/j*k;
  printf(" i = %d \n",i);
  printf(" j = %d \n",j);
  printf(" k = %d \n",k);
  printf(" l = %g \n",l);

  return(0);
}
```

Here is the output.

```
 i = 5
 j = 2
 k = 4
 l = 8
```

6.8 Example 6 - Fahrenheit to celsius

The following is an equation to convert from Fahrenheit to centigrade.

```
centigrade = 5/9*(fahrenheit-32)
```

The following is a program that implements this equation in C.

```
#include <stdio.h>

int main()
{
  double c;
  double f;
  f=-40;
  c=5/9*(f-32);
  printf(" -40 Fahrenheit = %g \n",c);
  return(0);
}
```

Here is the output.

```
 -40 Fahrenheit = -0
```

6.9 Example 7 - Centigrade to fahrenheit

The following is the corresponding equation to convert from centigrade to fahrenheit.

```
fahrenheit = 32 + 9/5* centigrade
```

The following is a program that implements this equation in C

```
#include <stdio.h>

int main()
{
  double c;
  double f;
  c=-40;
  f=32+9/5*c;
  printf(" -40 centigrade = %g \n",f);
  return(0);
}
```

Here is the output.

```
 -40 centigrade = -8
```

6.10 Problems

1. Correct the above two programs. Key temperatures are given below.

```
Centigrade    Fahrenheit
   -40            -40
     0             32
   100            212
```

6.11 The float.h header file

This header describes the characteristics of floating types for the specific system and compiler implementation used.

A floating-point number is composed of four elements:

- a sign: either negative or non-negative
- a base (or radix): which expresses the different numbers that can be represented with a single digit (2 for binary, 10 for decimal, 16 for hexadecimal, and so on...)
- a significand (or mantissa): which is a series of digits of the aforementioned base. The number of digits in this series is what is known as precision.
- an exponent (also known as characteristic, or scale): which represents the offset of the significand, affecting the value in the following way: value of floating-point = significand x base exponent, with its corresponding sign.

6.11.1 Macro constants

The simplest thing to do is run the following example. This will print out information about what is supported by your compiler.

Here is the output from the Microsoft compiler.

```
FLT_RADIX       = 2
FLT_MANT_DIG    = 24
DBL_MANT_DIG    = 53
LDBL_MANT_DIG   = 53
FLT_DIG         = 6
DBL_DIG         = 15
LDBL_DIG        = 15
FLT_MIN_EXP     = -125
DBL_MIN_EXP     = -1021
LDBL_MIN_EXP    = -1021
FLT_MIN_10_EXP  = -37
DBL_MIN_10_EXP  = -307
LDBL_MIN_10_EXP = -307
```

```
FLT_MAX_EXP     = 128
DBL_MAX_EXP     = 1024
LDBL_MAX_EXP    = 1024
FLT_MAX_10_EXP  = 38
DBL_MAX_10_EXP  = 308
LDBL_MAX_10_EXP = 308
FLT_MAX         = 3.402823e+038
DBL_MAX         = 1.797693e+308
LDBL_MAX        = 1.797693e+308
FLT_EPSILON     = 1.192093e-007
DBL_EPSILON     = 2.220446e-016
LDBL_EPSILON    = 2.220446e-016
FLT_MIN         = 1.175494e-038
DBL_MIN         = 2.225074e-308
LDBL_MIN        = 2.225074e-308
```

Here is the output from the gcc compiler.

The gcc compiler under cygwin on Windows supports the Intel 80 bit floating point type.

The same compiler under openSuSe 13.1 does not support this extended real type.

6.12 The math.h header file

The header declares a set of functions to compute common mathematical operations and transformations.

6.12.1 Trigonometric functions

- cos - compute cosine (function)
- sin - compute sine (function)
- tan - compute tangent (function)
- acos - compute arc cosine (function)
- asin - compute arc sine (function)
- atan - compute arc tangent (function)
- atan2 - compute arc tangent with two parameters (function)

6.12.2 Hyperbolic functions

- cosh - compute hyperbolic cosine (function)
- sinh - compute hyperbolic sine (function)

- tanh - compute hyperbolic tangent (function)
- acosh - compute arc hyperbolic cosine (function)
- asinh - compute arc hyperbolic sine (function)
- atanh - compute arc hyperbolic tangent (function)

6.12.3 Exponential and logarithmic functions

- exp - compute exponential function (function)
- frexp - Get significand and exponent (function)
- ldexp - Generate value from significand and exponent (function)
- log - compute natural logarithm (function)
- log10 - compute common logarithm (function)
- modf - Break into fractional and integral parts (function)
- exp2 - compute binary exponential function (function)
- expm1 - compute exponential minus one (function)
- ilogb - Integer binary logarithm (function)
- log1p - compute logarithm plus one (function)
- log2 - compute binary logarithm (function)
- logb - compute floating-point base logarithm (function)
- scalbn - Scale significand using floating-point base exponent (function)
- scalbln - Scale significand using floating-point base exponent (long) (function)

6.12.4 Power functions

- pow - Raise to power (function)
- sqrt - compute square root (function)
- cbrt - compute cubic root (function)
- hypot - compute hypotenuse (function)

6.12.5 Error and gamma functions

- erf - compute error function (function)
- erfc - compute complementary error function (function)
- tgamma - compute gamma function (function)
- lgamma - compute log-gamma function (function)

6.12.6 Rounding and remainder functions

- ceil - Round up value (function)
- floor - Round down value (function)
- fmod - compute remainder of division (function)
- trunc - Truncate value (function)
- round - Round to nearest (function)
- lround - Round to nearest and cast to long integer (function)
- llround - Round to nearest and cast to long long integer (function)
- rint - Round to integral value (function)
- lrint - Round and cast to long integer (function)
- llrint - Round and cast to long long integer (function)
- nearbyint = Round to nearby integral value (function)
- remainder - compute remainder (IEC 60559) (function)
- remquo - compute remainder and quotient (function)

6.12.7 Floating-point manipulation functions

- copysign - Copy sign (function)
- nan - Generate quiet NaN (function)
- nextafter - Next representable value (function)
- nexttoward - Next representable value toward precise value (function)

6.12.8 Minimum, maximum, difference functions

- fdim - Positive difference (function)
- fmax - Maximum value (function)
- fmin - Minimum value (function)

6.12.9 Other functions

- fabs - compute absolute value (function)
- abs - compute absolute value (function)fma Multiply-add (function)

6.12.10 Classification macro / functions

- fpclassify - Classify floating-point value (macro/function)
- isfinite - Is finite value (macro)
- isinf - Is infinity (macro/function)
- isnan - Is Not-A-Number (macro/function)
- isnormal - Is normal (macro/function)
- signbit - Sign bit (macro/function)

6.12.11 Comparison macro / functions

- isgreater - Is greater (macro)
- isgreaterequal - Is greater or equal (macro)
- isless - Is less (macro)islessequal Is less or equal (macro)
- islessgreater - Is less or greater (macro)
- isunordered - Is unordered (macro)

6.13 Problems

1. Calculate your body mass index or BMI value. It is given by

```
bmi = weight/(height*height)
```

where the weight measurement is in kilograms, and the height measurement is in metres. For people who think in Imperial measurements

```
one stone equals 14 pounds
one kilogram equals 2.2 pounds
one foot equals 12 inches
one inch equals 2.54 centimetres
```

According to Garrow there are the following grades of obesity.

```
  Grade     Value      Comment
    0     20 to 24.9   Desirable
    1     25 to 29.9   Overweight
    2     30 to 40     Obese
    3     40+          Morbidly obese
Ideal ranges
  Men     20.1 - 25
  Women   18.8 - 23.8
```

Where do you fit?

2. Calculate the area of a circle. Use a radius of 1 metre. Wikipedia has a value for *pi*

3. Calculate the circumference of a circle. Use a radius of 1 metre.

4. According to the CRC Handbook of Chemistry and Physics (60th Edition), a cubic foot of clay weighs 112 to 162 pounds.

Convert this to kilograms.

How much would a cubic metre of clay weigh?

Chapter 7

Arrays

7.1 Introduction

In this chapter we look at arrays in C.

- One d arrays
- The array control structure
- Two d arrays

7.2 Example 1 - simple 1 d array

```
#include <stdio.h>

#define N 12

int main()
{
  double sum=0.0,average=0.0 ;
  double rainfall[N]  =
    { 3.1 , 2.0 , 2.4 , 2.1 , 2.2 , 2.2 ,
      1.8 , 2.2 , 2.7 , 2.9 , 3.1 , 3.1 };
  int month ;
    for (month=0;month < N ;++month)
      sum = sum + rainfall[month];
    average = sum/N;

    printf("\n Rainfall monthly values are\n\n");

    for (month=0;month < N ;++month)
    {
      printf(" %5.1f \n",rainfall[month]);
```

```
    }
    printf(" Sum      = %6.2f \n",sum);
    printf(" Average  = %6.2f \n",average);

  return(0);
}
```

The first statement of interest is

```
#define N 12
```

In the C Programming Language, the #define directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code.

Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions.

We will use the symbol N throughout the program to represent the size of the array.

The following statement

```
double rainfall[N] =
  { 3.1 , 2.0 , 2.4 , 2.1 , 2.2 , 2.2 ,
  1.8 , 2.2 , 2.7 , 2.9 , 3.1 , 3.1 };
```

declares rainfall to be an array of type double. The [N] means that this is an array of size N.

We then use the numbers in curly braces {} separated by commas as the initial values of the data in the array.

The next statement

```
for (month=0;month < N ;++month)
  sum = sum + rainfall[month];
```

is a for loop that iterates over the loop variable month from 0 to 11. In the C family of languages if an array is on size N, the loop is from 0 to N-1.

K and R decided to start counting at 0.

So the loop will add each monthly rainfall value to the variable sum, which is 0 at the start.

The next loop

```
  for (month=0;month < N ;++month)
  {
    printf(" %5.1f \n",rainfall[month]);
  }
```

uses the to enclose a block of statements, which is in fact only one statement.

7.3 Problems

1. Compile and run this example.

2. Modify the program to convert the rainfall in inches to metric measurements in mm. One inch equals 2.54 cm.

3. Visit

```
http://www.metoffice.gov.uk/public/weather/
climate-historic/#?tab=climateHistoric
```

Here is a sample of data for the Cwmystwyth site.

```
http://www.metoffice.gov.uk/pub/data/weather/
uk/climate/stationdata/cwmystwythdata.txt
```

```
Cwmystwyth
Location: 277300E 274900N, Lat 52.358 Lon -3.802, 301 metres amsl
Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic Kipp & Zonen sensor
marked with a #, otherwise sunshine data taken from a
Campbell Stokes recorder.
   yyyy  mm   tmax    tmin      af    rain     sun
              degC    degC    days      mm   hours
   1959   1    4.5    -1.9      20    ---     57.2
  1959    2    7.3     0.9      15    ---     87.2
  1959    3    8.4     3.1       3    ---     81.6
  1959    4   10.8     3.7       1    ---    107.4
  1959    5   15.8     5.8       1    ---    213.5
  1959    6   16.9     8.2       0    ---    209.4
  1959    7   18.5     9.5       0    ---    167.8
  1959    8   19.0    10.5       0    ---    164.8
  1959    9   18.3     5.9       0    ---    196.5
  1959   10   14.8     7.9       1    ---    101.1
  1959   11    8.8     3.9       3    ---     38.9
  1959   12    7.2     2.5       3    ---     19.2
  1960    1    6.3     0.6      15    ---     30.7
  1960    2    5.3    -0.3      17    ---     50.2
  1960    3    8.2     2.4       4    ---     73.9
  1960    4   11.2     2.6       7    ---    146.8
  1960    5   15.4     6.5       2    ---    153.9
  1960    6   18.5     8.2       0    ---    225.6
  1960    7   16.0     9.3       0    ---    111.3
  1960    8   16.5     9.4       0    ---    119.2
  1960    9   15.0     7.9       0    ---    120.3
  1960   10   12.0     5.3       5    ---     ---
```

```
 1960   11    8.8     2.9       5    ---     37.3
 1960   12    5.9     0.4      13    ---     33.9
 1961    1    5.4     0.2      11   144.8    31.0
 1961    2    8.7     2.9       2   112.5    45.2
 1961    3   10.2     2.1      10    77.2   102.6
 1961    4   11.9     5.0       1   130.7    83.9
 1961    5   ---     ---      ---     66.3   173.7
 1961    6   ---      7.4     ---     66.1   190.6
 1961    7   16.7     8.2       0   141.1   149.2
 1961    8   16.8    10.1       0   149.5   106.6
 1961    9   17.4     9.3       0   134.8    79.7
 1962    5            4.2       3   117.8   102.2
 1962    6            6.8       1    72.8   163.9
 1962    7   16.8     9.1       0    56.7    ---
 1962    8   15.6     9.3       0   236.2    ---
 1962    9   14.6     7.8       1   218.0    ---
 1962   10   ---     ---      ---     69.7    ---
 1962   11    7.6     1.8       9    85.2    ---
 1962   12    5.3    -1.0      18   204.4    ---
```

Choose a site and year and replace the rainfall data with rainfall for a site and year of your choice.

How wet is it in comparison with London?

7.4 Example 2 - simple variation using sizeof

```
#include <stdio.h>

int main()
{
  double sum=0.0,average=0.0 ;
  double rainfall[]  =
    { 3.1 , 2.0 , 2.4 , 2.1 , 2.2 , 2.2 ,
      1.8 , 2.2 , 2.7 , 2.9 , 3.1 , 3.1 };
  int month ;
  int n;
    n = sizeof(rainfall)/sizeof(double);
    for (month=0;month < n ;++month)
      sum = sum + rainfall[month];
    average = sum/n;

    printf("\n Rainfall monthly values are\n\n");

    for (month=0;month < n ;++month)
```

```
    {
      printf(" %5.1f \n",rainfall[month]);
    }
    printf(" Sum      = %6.2f \n",sum);
    printf(" Average  = %6.2f \n",average);

  return(0);
}
```

In this example we declare n as a conventional int variable and then calculate its value using

```
n = sizeof(rainfall)/sizeof(double);
```

where we use the sizeof intrinsic function.
The rest of the program is the same as in the first example.

7.5 Example 3 - reading in the size and dynamic allocation

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

int main()
{
  double sum=0.0,average=0.0 ;
  double * rainfall;
  int month ;
  int n;
    printf("Type in the size of the array\n");
    scanf("%d",&n);
    printf(" n = %d \n",n);
    rainfall = malloc ( n * sizeof(double) );
    for (month=0;month < n ;++month)
    {
      rainfall[month]=month;
      sum = sum + rainfall[month];
    }
    average = sum/n;

    printf("\n Rainfall monthly values are\n\n");

    for (month=0;month < n ;++month)
    {
```

```
      printf(" %5.1f \n",rainfall[month]);
    }
    printf(" Sum      = %6.2f \n",sum);
    printf(" Average  = %6.2f \n",average);

  return(0);
}
```

In this example the key statement is shown below.

```
double * rainfall;
```

which declares rainfall to be a pointer to a double. C uses the * in a declaration to indicate that the variable is a pointer.

We can then use the following statement

```
rainfall = malloc ( n * sizeof(double) );
```

to allocate an area of memory to hold an array of size n of doubles.

malloc is one of the memory allocation functions in C.

We can then use the conventional array syntax we had in the first two examples to access and manipulate the elements of the rainfall array.

7.6 Problems

1. Rewrite the above example to read the data in.
   Create a text file using the site and year data you chose earlier.
   Use the

```
<
```

redirection operator to read the data from your file.

7.7 Example 4 - simple dynamic variant using calloc

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

int main()
{
  double sum=0.0,average=0.0 ;
  double * rainfall;
  int month ;
```

```
  int n;
    printf("Type in the size of the array\n");
    scanf("%d",&n);
    printf(" n = %d \n",n);
    rainfall = (double*) calloc( n , sizeof(double) );
    for (month=0;month < n ;++month)
    {
      rainfall[month]=month;
      sum = sum + rainfall[month];
    }
    average = sum/n;

    printf("\n Rainfall monthly values are\n\n");

    for (month=0;month < n ;++month)
    {
      printf(" %5.1f \n",rainfall[month]);
    }
    printf(" Sum      = %6.2f \n",sum);
    printf(" Average  = %6.2f \n",average);

  return(0);
}
```

The examples is the same as the previous, but now we use

```
rainfall = (double*) calloc( n , sizeof(double) );
```

calloc initialises the array to zero.
The rest of the example is the same.

7.8 Example 5 - simple 2 d arrays

Here is the source code.

```
#include <stdio.h>

#define NROWS 2
#define NCOLS 3

int main()
{

  int a[NROWS][NCOLS]= { { 1,2,3 },
                         { 4,5,6 } };
```

```
  for (int row=0;row<NROWS;row++)
  {
    for (int column=0;column<NCOLS;column++)
      printf(" %d " , a[row][column]) ;
      printf("\n");
  }

  return(0);

}
```

In this example we use

```
#define NROWS 2
#define NCOLS 3
```

to set up the number of rows and columns for our 2 d array.

We then declare and initialise the array using

```
int a[NROWS][NCOLS]= { { 1,2,3 },
                       { 4,5,6 } };
```

where we use the [][] syntax to signify a 2 d array.

We then use the syntax on the right involving nested braces to actually initialise the 2 d array.

We then have a outer for row loop and an inner for column loop to process the elements of the 2 d array.

7.9 Problems

1. Consider the following example.

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

#define n 5

int main()
{

  int x[n];
  int * y;
  int * z;
  int i;
```

```
  y = malloc ( n * sizeof(int) );
  z = (int*) calloc( n , sizeof(int) );

  printf("  i       x[i]           y[i]        z[i] \n");

  for (i=0;i<n;i++)
  {
    printf(" %3d %12d %12d %12d\n",i,x[i],y[i],z[i]);
  }


  for (i=0;i<2*n;i++)
  {
    printf(" %3d %12d %12d %12d\n",i,x[i],y[i],z[i]);
  }
  return(0);

}
```

Here is the output on one system.

```
i       x[i]           y[i]        z[i]
 0   -662889607      6029353           0
 1        32759      6881367           0
 2   -662822280      6553710           0
 3        32759      7798895           0
 4            0      2097267           0
 0   -662889607      6029353           0
 1        32759      6881367           0
 2   -662822280      6553710           0
 3        32759      7798895           0
 4            0      2097267           0
 5            0      6881355           0
 6   -418512736    793195234    793326328
 7        12967  -2147480484  -2147481745
 8   -662884484      6029360      6684783
 9        32759      7209045      2097268
```

Note on this system going outside the bounds of the arrays has not caused a run time error.

What happened when you compiled and ran it?

2. Using the 2 d array program as a starting point write a program that initialises a 2d 3 * 3 array to

```
1   2   3
4   5   6
7   8   9
```

Calculate the row and column sums for the 2 d array. Use 1d arrays to hold the row and column sums.

Print out the 2 d array and row and column sums and produce output similar to the following.

```
 1   2   3     6
 4   5   6    15
 7   8   9    24

12  15  18
```

Chapter 8

Text

8.1 Introduction

This chapter looks at text in C.

- Arrays of char
- Character functions

8.2 Example 1 - Arrays of char and the sizeof intrinsic

```
#include <stdio.h>

int main()
{
  char uppercase[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char lowercase[]="abcdefghijklmnopqrstuvwxyz";
  char ch;
  int i;
  int ic;
  int usize=sizeof(uppercase)/sizeof(char);
  int lsize=sizeof(lowercase)/sizeof(char);
  printf("  upper case array size is %d \n" , usize);
  printf("  lower case array size is %d \n" , lsize);
  for (i=0;i < usize-1 ; i++)
  {
    ch=uppercase[i];
    ic=(int)(ch);
    printf(" %3d  =  %c",ic,ch);
    ch=lowercase[i];
    ic=(int)(ch);
```

```
    printf(" %3d  =  %c\n",ic,ch);
  }
  return(0);
}
```

Here is the output from one compiler.

```
upper case array size is 27
lower case array size is 27
65  =  A  97  =  a
66  =  B  98  =  b
67  =  C  99  =  c
68  =  D 100  =  d
69  =  E 101  =  e
70  =  F 102  =  f
71  =  G 103  =  g
72  =  H 104  =  h
73  =  I 105  =  i
74  =  J 106  =  j
75  =  K 107  =  k
76  =  L 108  =  l
77  =  M 109  =  m
78  =  N 110  =  n
79  =  O 111  =  o
80  =  P 112  =  p
81  =  Q 113  =  q
82  =  R 114  =  r
83  =  S 115  =  s
84  =  T 116  =  t
85  =  U 117  =  u
86  =  V 118  =  v
87  =  W 119  =  w
88  =  X 120  =  x
89  =  Y 121  =  y
90  =  Z 122  =  z
```

Note that the size of the arrays is 27, as we have zero byte terminators for arrays of characters.

With this compiler we have the ASCII character set.

8.3 Example 2 - characters available

```
#include <stdio.h>

int main()
```

```
{
  char c;
  for (int i=0;i<128;++i)
  {
    c=(char)(i);
    printf(" %3d  %c \n",i,c);
  }
  return(0);
}
```

Here is the output from one compiler.

```
  32
  33  !
  34  "
  35  #
  36  $
  37  %
  38  &
  39  ’
  40  (
  41  )
  42  *
  43  +
  44  ,
  45  -
  46  .
  47  /
  48  0
  49  1
  50  2
  51  3
  52  4
  53  5
  54  6
  55  7
  56  8
  57  9
  58  :
  59  ;
  60  <
  61  =
  62  >
  63  ?
  64  @
```

```
 65  A
 66  B
 67  C
 68  D
 69  E
 70  F
 71  G
 72  H
 73  I
 74  J
 75  K
 76  L
 77  M
 78  N
 79  O
 80  P
 81  Q
 82  R
 83  S
 84  T
 85  U
 86  V
 87  W
 88  X
 89  Y
 90  Z
 91  [
 92  \
 93  ]
 94  ^
 95  _
 96  `
 97  a
 98  b
 99  c
100  d
101  e
102  f
103  g
104  h
105  i
106  j
107  k
108  l
109  m
```

```
110  n
111  o
112  p
113  q
114  r
115  s
116  t
117  u
118  v
119  w
120  x
121  y
122  z
123  {
124  |
125  }
126  ~
```

Again we have the ASCII character set.

8.4 Problems

1. Modify the last program to produce the following output.

```
!
"#
$%&
'()*
+,-./
012345
6789:;<
=>?@ABCD
EFGHIJKLM
NOPQRSTUVW
XYZ[\]^_‘ab
cdefghijklmn
opqrstuvwxyz{
|}~
```

2. Modify the program to produce the following output.

```
   !
  "#$
 %&'()
*+,-./0
```

```
      123456789
     :;<=>?@ABCD
    EFGHIJKLMNOPQ
   RSTUVWXYZ[\]^_‘
  abcdefghijklmnopq
 rstuvwxyz{|}~
```

8.5 Example 3 - arrays of char and pointers

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

int main()
{
  char line[]="This is a line of text";
  char* ptr=line;
  while (*ptr)
  {
    *ptr=toupper(*ptr);
    ptr++;
  }
  printf("%s\n",line);
  return(0);
}
```

This is the output from running the program.

```
THIS IS A LINE OF TEXT
```

This example illustrates the way pointers and arrays are synonymous in C. We will come back to this example in the chapter on pointers.

8.6 The string.h header file

Here are some details about this header file.

This header file defines several functions to manipulate C strings and arrays.

8.6.1 Functions

- Copying
  - memcpy Copy block of memory
  - memmove Move block of memory
  - strcpy Copy string

  - strncpy Copy characters from string

- Concatenation
  - strcat Concatenate strings
  - strncat Append characters from string

- Comparison
  - memcmp Compare two blocks of memory
  - strcmp Compare two strings
  - strcoll Compare two strings using locale
  - strncmp Compare characters of two strings
  - strxfrm Transform string using locale

- Searching
  - memchr Locate character in block of memory
  - strchr Locate first occurrence of character in string
  - strcspn Get span until character in string
  - strpbrk Locate characters in string
  - strrchr Locate last occurrence of character in string
  - strspn Get span of character set in string
  - strstr Locate substring
  - strtok Split string into tokens

- Other
  - memset Fill block of memory
  - strerror Get pointer to error message string
  - strlen Get string length

8.6.2 Macros

NULL Null pointer (macro)

8.6.3 Types

size_t Unsigned integral type (type)

We next have some examples to illustrate the use of some of these functions.

8.7 Example 4 - The strcpy and strcat intrinsics

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

int main()
{
  char line[80];
  strcpy(line,"Ian");
  strcat(line," David");
  strcat(line," Chivers");
  printf("%s\n",line);
  return(0);
}
```

Let us look at the example in more depth.

```
char line[80];
end{verbatim}

This declares line to be an array variable of size 80 and of type char.

\begin{verbatim}
strcpy(line,"Ian");
```

This statement copies "Ian" into the array line, including the terminating null character (and stopping at that point).

The folllowing two statements

```
strcat(line," David");
strcat(line," Chivers");
```

appends a copy of the second argument (source) to the first argument (destination). The terminating null character in destination is overwritten by the first character of source, and a null-character is included at the end of the new string formed by the concatenation of both in destination.

Here is the output from one compiler.

```
Ian David Chivers
```

8.8 Example 5 - the strlen intrinsic

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

```
int main()
{
  char t[]="This is some text";
  printf("%d\n", strlen(t) );
  return(0);
}
```

8.9 Example 6 - malloc and dynamic string allocation

This example use strlen to calculate the sizes of several character array variables.

We then dynamically allocate a character array large enough to hold all of the text in the various character arrays.

We then use strcpy and strcat to initialise the complete array.

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

int main()
{
  char  line1[] = "A simple line of text. ";
  char  line2[] = "Flann O'Brien: At-Swim-Two-Birds. ";
  char  line3[] = "Queens Park Rangers (QPR) FC.";

  int l1 ;
  int l2 ;
  int l3 ;
  int t;

  char * total;

  l1 = strlen(line1);
  l2 = strlen(line2);
  l3 = strlen(line3);

  t=l1+l2+l3;

  printf(" line1 = %3d \n",l1);
  printf(" line2 = %3d \n",l2);
  printf(" line3 = %3d \n",l3);

  total = malloc( t * sizeof(char) ) ;
```

```
  strcpy(total,line1);
  strcat(total,line2);
  strcat(total,line3);

  printf("%s\n",total);

  return(0);
}
```

Here is the output.

```
 line1 =  23
 line2 =  34
 line3 =  29
A simple line of text. Flann O'Brien: At-Swim-Two-Birds. Queens Park Rangers (QPR) FC.
```

8.10 Example 7 - strchr

In this example we will use the strchr function. Here is some more detailed information.

```
char * strchr (       char * str, int character );

Locate first occurrence of character in string

Returns a pointer to the first occurrence
of character in the C string str.

The terminating null-character is considered
part of the C string. Therefore, it can also
be located in order to retrieve a
pointer to the end of a string.
```

Here is the source code.

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

int main()
{
  char  line1[] = "A simple line of text. ";
  char  line2[] = "Flann O'Brien: At-Swim-Two-Birds. ";
  char  line3[] = "Queens Park Rangers (QPR) FC.";
  char * total;char * t1;char * t2;
```

```
  int l1 , l2 , l3 , t, lt1 , lt2 ;
  l1 = strlen(line1);l2 = strlen(line2);
  l3 = strlen(line3);
  t=l1+l2+l3;
  total = malloc( t * sizeof(char) ) ;
  t1 = malloc( t * sizeof(char) ) ;
  t2 = malloc( t * sizeof(char) ) ;

  strcpy(total,line1);strcat(total,line2);
  strcat(total,line3);

// characters to look for

  char c1 = '.';
  char c2 = '-';

  t1  = strchr(total,c1);
  lt1 = strlen(t1);
  t2  = strchr(total,c2);
  lt2 = strlen(t2);

  printf(" Character %c found at position %3d \n",c1,(t-lt1));
  printf(" %c\n",total[t-lt1]);
  printf(" Character %c found at position %3d \n",c2,(t-lt2));
  printf(" %c\n",total[t-lt2]);

  return(0);
}
```

Here is the output.

```
 Character . found at position  21
 .
 Character - found at position  40
 -
```

8.11 Example 8 - tokenizing strings, the strtok function

Here is the source code.

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

```
int main()
{
  char  line1[] = "A simple line of text. ";
  char  line2[] = "Flann O'Brien: At-Swim-Two-Birds. ";
  char  line3[] = "Queens Park Rangers (QPR) FC.";
  char * total;
  char * tokens;

  int l1 , l2 , l3 , t;
  l1 = strlen(line1);l2 = strlen(line2);
  l3 = strlen(line3);
  t=l1+l2+l3;
  total = malloc( t * sizeof(char) ) ;

  strcpy(total,line1);strcat(total,line2);
  strcat(total,line3);

  tokens = strtok(total," .,-()");

  while ( tokens !=NULL )
  {
     printf("%s\n",tokens);
     tokens = strtok( NULL , " .,-()");
  }
  return(0);
}
```

Here is the output.

```
A
simple
line
of
text
Flann
O'Brien:
At
Swim
Two
Birds
Queens
Park
Rangers
QPR
FC
```

8.12 Example 9 - days of the week

Here is the source code.

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

int main()
{

  char days_of_the_week[7][10] = { "Sunday",
                                   "Monday",
                                   "Tuesday",
                                   "Wednesday",
                                   "Thursday",
                                   "Friday",
                                   "Sunday" };

  int i;

  for (i=0;i<7;i++)
    printf("%s\n",days_of_the_week[i]);

  return(0);

}
```

Here is the output.

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Sunday
```

8.13 The ctype.h header file

This header declares a set of functions to classify and transform individual characters.

These functions take the int equivalent of one character as parameter and return an int that can either be another character or a value representing a boolean value: an int value of 0 means false, and an int value different from 0 represents true.

There are two sets of functions:

8.13.1 Character classification functions

They check whether the character passed as parameter belongs to a certain category:

- isalnum Check if character is alphanumeric
- isalpha Check if character is alphabetic
- isblank Check if character is blank
- iscntrl Check if character is a control character
- isdigit Check if character is decimal digit
- isgraph Check if character has graphical representation
- islower Check if character is lowercase letter
- isprint Check if character is printable
- ispunct Check if character is a punctuation character
- isspace Check if character is a white-space
- isupper Check if character is uppercase letter
- isxdigit Check if character is hexadecimal digit

8.13.2 Character conversion functions

Two functions that convert between letter cases:

- tolowerConvert uppercase letter to lowercase
- toupperConvert lowercase letter to uppercase

8.14 Problems

1. Given the following text

```
"The important issue about a language, is not so "
"much what features the language possesses, but "
"the features it does possess, are sufficient, to "
"support the desired programming styles, in the "
"desired application areas."
```

Write a program that breaks the text into phrases, using the comma and full stop as breaking characters. The output expected is given below.

```
The important issue about a language
is not so much what features the language possesses
but the features it does possess
are sufficient
to support the desired programming styles
in the desired application areas
```

2. Using the days of the week program as a starting point write a program that does the same thing with the months in the year.

Chapter 9

IO in C and the stdio.h header file

9.1 Introduction

The IO functionality of C is fairly low-level by modern standards; C abstracts all file operations into operations on streams of bytes, which may be "input streams" or "output streams". Unlike some earlier programming languages, C has no direct support for random-access data files; to read from a record in the middle of a file, the programmer must create a stream, seek to the middle of the file, and then read bytes in sequence from the stream.

The stream model of file IO was popularized by Unix, which was developed concurrently with the C programming language itself. The vast majority of modern operating systems have inherited streams from Unix, and many languages in the C programming language family have inherited C's file IO interface with few if any changes (for example, PHP).

In this chapter we have a brief look at some of the headers used for io in C.

9.2 The stdio.h header file

The following is a list of functions found within the stdio.h header file:

9.2.1 Formatted Input and Output functions

Here is a list.

- fprintf - Formatted File Write
- fscanf - Formatted File Read
- printf - Formatted Write
- scanf - Formatted Read
- sprintf - Formatted String Write

- sscanf - Formatted String Read
- vfprintf - Formatted File Write Using Variable Argument List
- vprintf - Formatted Write Using Variable Argument List
- vsprintf - Formatted String Write Using Variable Argument List

9.2.2 File operation functions

Here is a list.

- fclose - Close File
- fflush - Flush File Buffer
- fopen - Open File
- freopen - Reopen File
- remove - Remove File
- rename - Rename File
- setbuf - Set Buffer (obsolete)
- setvbuf - Set Buffer
- tmpfile - Create Temporary File
- tmpnam - Generate Temporary File Name

9.2.3 Character Input and Output functions

Here is a list.

- fgetc - Read Character from File
- fgets - Read String from File
- fputc - Write Character to File
- fputs - Write String to File
- getc - Read Characters from File
- getchar - Read Character
- gets - Read String
- putc - Write Character to File

- putchar - Write Character
- puts - Write String
- ungetc - Unread Character

9.2.4 Block Input and Output functions

Here is a list.

- fread Read Block from File
- fwrite - Write Block to File

9.2.5 File positioning functions

Here is a list.

- fgetpos - Get File Position
- fseek - File Seek
- fsetpos - Set File Position
- ftell - Determine File Position
- rewind - Rewind File

9.2.6 Error handling functions

Here is a list.

- clearerr - Clear Stream Error
- feof - Test for End-of-File
- ferror - Test for File Error
- perror - Print Error Message

9.3 Formatted output - printf

The signature of the function is given below.

```
int printf ( const char * format, ... );
```

The function is used to print formatted data to stdout, and writes the C string pointed by format to the standard output (stdout). If format includes format specifiers (subsequences beginning with the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

9.3.1 Parameters

format

C string that contains the text to be written to stdout.

It can optionally contain embedded format specifiers that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A format specifier follows this prototype

```
%[flags][width][.precision][length]specifier
```

Where the specifier character at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

Table 9.1: printf format specifiers

Specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed	

The format specifier can also contain sub-specifiers: flags, width, .precision and modifiers (in that order), which are optional and follow these specifications shown below.

The length sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without length specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

On success, the total number of characters written is returned.

If a writing error occurs, the error indicator (ferror) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to EILSEQ and a negative number is returned.

Table 9.2: printf flags

flags	description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see width sub-specifier).

Table 9.3: printf width

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
asterisk	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

9.4 Example 1 - puts, scanf, sprintf and prntf

Here is an example using the puts, scanf, sprintf and printf functions.

```
#include <stdio.h>
#define SIZE 80

int main( void )
{
   char   buffer[ SIZE ];
   int    age;

   puts( "Type in your age (integer)" );
   scanf( "%d", &age );
```

Table 9.4: printf precision

.precision	description
.number	For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

```
   sprintf( buffer, " You are :%6d years old\n", age );

   printf( "%s\n%s\n",
           "The formatted output stored in character array buffer is:", buffer );
}
```

Here is sample output.

```
Type in your age (integer)
64
The formatted output stored in character array buffer is:
 You are :    64 years old
```

Note that puts adds a newline.
Here is a description of sprintf.

```
int sprintf ( char * str, const char * format, ... );

Write formatted data to string

Composes a string with the same text that would be printed if
format was used on printf, but instead of being printed, the
content is stored as a C string in the buffer pointed by str.
```

Table 9.5: printf length

			specifiers				
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double double	int int	char* char*	void* void*	int* int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

So C provides the ability to dynamically create output formatting at run time.

9.5 Example 2 - sscanf

Here is a simple example.

```
#include <stdio.h>

int main( void )
{
   char s[] = "9999 3.14159265358";
   int    i;
   double d;

   sscanf( s, "%d%lf", &i, &d );
   printf("\n%5d\n%17.11f\n",i, d );
}
```

Here is the output.

```
 9999
    3.14159265358
```

C provides the ability to read into a string variable and then parse that variable under program control at run time.

9.6 Summary

This chapter is mainly a reference chapter. More examples can be found throughout the rest of the notes. You have to be aware of what C offers in this header file.

9.7 Problems

Run the examples.

Chapter 10

The stdlib.h header file

10.1 The stdlib.h header file

This header defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and - Converting.

10.1.1 String conversion

- atof - Convert string to double (function)
- atoi - Convert string to integer (function)
- atol - Convert string to long integer (function)
- atoll - Convert string to long long integer (function)
- strtod - Convert string to double (function)
- strtof - Convert string to float (function)
- strtol - Convert string to long integer (function)
- strtold - Convert string to long double (function)
- strtoll - Convert string to long long integer (function)
- strtoul - Convert string to unsigned long integer (function)
- strtoull - Convert string to unsigned long long integer (function)

10.1.2 Pseudo-random sequence generation

- rand - Generate random number (function)
- srand - Initialize random number generator (function)

10.1.3 Dynamic memory management

- calloc - Allocate and zero-initialize array (function)
- free - Deallocate memory block (function)
- malloc - Allocate memory block (function)
- realloc - Reallocate memory block (function)

10.1.4 Environment

- abort - Abort current process (function)
- atexit - Set function to be executed on exit (function)
- at_quick_exit - Set function to be executed on quick exit (function)
- exit - Terminate calling process (function)
- getenv - Get environment string (function)
- quick_exit - Terminate calling process quick (function)
- system - Execute system command (function)
- _Exit - Terminate calling process (function)

10.1.5 Searching and sorting

- bsearch - Binary search in array (function)
- qsortSort elements of array (function)

10.1.6 Integer arithmetics

- abs - Absolute value (function)
- div - Integral division (function)
- labs - Absolute value (function)
- ldiv - Integral division (function)
- llabs - Absolute value (function)
- lldiv - Integral division (function)

10.1.7 Multibyte characters

- mblen - Get length of multibyte character (function)
- mbtowc - Convert multibyte sequence to wide character (function)
- wctomb - Convert wide character to multibyte sequence (function)

10.1.8 Multibyte strings

- mbstowcs - Convert multibyte string to wide-character string (function)
- wcstombs - Convert wide-character string to multibyte string (function)

10.1.9 Macro constants

- EXIT_FAILURE - Failure termination code (macro)
- EXIT_SUCCESS - Success termination code (macro)
- MB_CUR_MAX - Maximum size of multibyte characters (macro)
- NULL - Null pointer (macro)RAND_MAXMaximum value returned by rand (macro)

10.1.10 Types

- div_t - Structure returned by div (type)
- ldiv_t - Structure returned by ldiv (type)
- lldiv_t - Structure returned by lldiv (type)
- size_t - Unsigned integral type (type)

10.2 Summary

This chapter is a reference chapter. Examples can be found throughout the rest of the notes. You have to be aware of what C offers in this header file.

10.3 Problems

None in this chapter.

Chapter 11

Control structures

11.1 Introduction

In this chapter we look at control structures in C.

- if statement
- if else statement
- switch statement
- for statement
- while statement
- do while statement
- break statement
- continue statement

We also look at boolean expressions and blocks of statements.

11.2 Boolean expressions

There are expressions that evaluate to true of false.

In C any integer type may be used to represent boolean values. The value zero represents false, and all non zero values represent true.

Boolean expressions evaluate to 0 if false and 1 if true.

C99 introduces a true boolean type - _Bool. It also a header file stdbool.h which defines the more convenient type name bool and the boolean values true and false. These names are consistent with the boolean type in C++, but different from the macro names traditionally used in C - FLASE and TRUE.

11.3 Blocks

Blocks of statements in C use braces .

11.4 Example 1 - the if statement

```
#include <stdio.h>

int main()
{
  int x;
  printf("Type in an integer\n");
  scanf("%d",&x);
  if (x>0)
    printf(" x is positive \n");
  return(0);
}
```

Compile and run this example. The program will execute the printf statement if the number is greater than 0.

11.5 Example 2 - The else and elseif statements

```
#include <stdio.h>

int main()
{
  int x;
  printf("Type in an integer\n");
  scanf("%d",&x);
  if ( x<0 )
    printf(" x is negative  \n");
  else if (x > 0)
    printf(" x is positive \n");
  return(0);
}
```

In this example we test for positive and negative numbers.

11.6 Example 3 - quadratic roots

A quadratic equation is:

$$ax^2 + bx + c = 0$$

This program has a simple structure. The roots of the quadratic are either real, equal and real, or complex depending on the magnitude of the term `b ** 2 - 4 * a * c`. The program tests for this term being greater than or less than zero: it assumes that the only other case is equality to zero (from the mechanics of a computer, floating point equality is rare, but we are safe in this instance):

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

int main()
{
  double a, b, c, term, a2, root1, root2;

  // a b and c are the coefficients of the terms
  // a*x**2+b*x+c
  // find the roots of the quadratic, root1 and
  // root2

  printf(" Type in the coefficients a, b and c\n");
  scanf("%lf %lf %lf", &a, &b, &c);
  printf(" a = %lf\n",a);
  printf(" b = %lf\n",b);
  printf(" c = %lf\n",c);

  term = b*b - 4.*a*c;
  a2 = a*2.;

// if term < 0, roots are complex
// if term = 0, roots are equal
// if term > 0, roots are double and different

  if (term<0.0)
  {
    printf(" roots are complex\n");
  }
  else if (term>0.0)
  {
    term = sqrt(term);
    root1 = (-b+term)/a2;
    root2 = (-b-term)/a2;
    printf(" roots are %lf %lf\n", root1, root2);
  }
  else
  {
    root1 = -b/a2;
```

```
    printf(" roots are equal, at %lf\n", root1);
  }
  return(0);
}
```

11.7 Example 4 - date calculation

```
#include <stdio.h>

int main()
{

  int year, n, month, day, t;

// calculates day and month from year and
// day-within-year
// t is an offset to account for leap years.
// Note that the first criteria is division by 4
// but that centuries are only
// leap years if divisible by 400
// not 100 (4 * 25) alone.

  printf(" year, followed by day within year\n");
  scanf("%d %d", &year, &n);
// checking for leap years
  if ((year/4)*4==year)
  {
    t = 1;
    if ((year/400)*400==year)
    {
      t = 1;
    }
    else if ((year/100)*100==year)
    {
      t = 0;
    }
  }
  else
  {
    t = 0;
  }
// accounting for February
  if (n>(59+t))
  {
```

```
    day = n + 2 - t;
  }
  else
  {
    day = n;
  }
  month = (day+91)*100/3055;
  day = (day+91) - (month*3055)/100;
  month = month - 2;
  printf(" calendar date is %d %d %d\n", day, month, year);
  return(0);

}
```

11.8 The switch statement

The switch statement is a multiway branch based on the value of the control variable.

The control expression that follows the keyword switch must have an intergral type and is subject to the usual unary conversions.

The expression following the keyword case must be an integral constant expression.

A switch statement is executed as follows

- The control expression is evaluated.
- If the value matches that of a case label program control is transferred to the point indicated by the case label.
- If the values does not match any case label control is passed to the default label.
- If the value does not match any case label and there is no default label transfer is to whatever is after the switch statement.

Let us look at some examples.

11.9 Example 5 - simple switch

```
#include <stdio.h>

int main()
{
  int i;
  printf(" Type in an integer value \n");
  scanf("%d", &i);
  switch (i)
  {
```

```
    case 1 : printf("  one entered \n") ;
    break;
    case 2 : printf("  two entered \n") ;
    break;
    case 3 : printf("  three entered \n") ;
    break;
    default: printf("  number other than 1,2 or 3 entered \n") ;
    break;
  }
  return(0);
}
```

11.10 Example 6 - more complex switch statement

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

int main()
{
  char line[80] = "This is !$ some text";
  strcat(line," and this is a bit @more");
  strcat(line," here is some 12345 punctuation .,;:");
  int ndigits = 0;
  int nvowels = 0;
  int nconsonants = 0;
  int npunctuation = 0;
  int nthrerest = 0;
  int nblank = 0;
  int i;
  int l;
  char c;
  l = strlen(line);
  printf(" %4d characters in the line \n",l);
  for (i = 0; i < l; i++)
  {
    c = line[i];
    switch (c)
    {
    case 'a': case 'A': case 'e': case 'E':
    case 'i': case 'I': case 'o': case 'O':
    case 'u': case 'U':
      nvowels++;
      break;
    case 'b': case 'B': case 'c': case 'C':
```

```
    case 'd': case 'D': case 'f': case 'F':
    case 'g': case 'G': case 'h': case 'H':
    case 'j': case 'J': case 'k': case 'K':
    case 'l': case 'L': case 'm': case 'M':
    case 'n': case 'N': case 'p': case 'P':
    case 'q': case 'Q': case 'r': case 'R':
    case 's': case 'S': case 't': case 'T':
    case 'v': case 'V': case 'w': case 'W':
    case 'x': case 'X': case 'y': case 'Y':
    case 'z': case 'Z':
      nconsonants++;
      break;
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
      ndigits++;
      break;
    case '.': case ',': case '!': case '"':
    case ':': case ';': case '(': case ')':
      npunctuation++;
      break;
    case ' ':
      nblank++;
      break;
    default:
      nthrerest++;
    } // switch loop
  } // for loop
  printf(" %4d digits \n",ndigits);
  printf(" %4d vowels \n",nvowels);
  printf(" %4d consonants \n",nconsonants);
  printf(" %4d punctuation \n",npunctuation);
  printf(" %4d blanks \n",nblank);
  printf(" %4d the rest \n",nthrerest);
  return(0);
}
```

11.11 Example 7 - while statement and sentinel usage

In this example we have a conventional counter controlled for loop and a zero trip loop, or while statement.

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

```
int main()
{
  int a[11];
  int mark;
  int end;
  int i;
  printf(" What number are you looking for?\n");
  scanf("%d", &mark);
  printf("%d\n",mark);
  printf(" How many numbers?\n");
  scanf("%d", &end);
  printf("%d\n",end);
  printf(" Type the numbers in\n");
  for (i=0; i < end ; ++i)
  {
    scanf("%d",  a[i]);
    printf("%d\n",a[i]);
  }
  i=0;
  a[end]=mark;
  while (mark != a[i]) ++i;
  if (i == end )
    printf("  Item not in list \n");
  else
    printf("  Item at position %d \n" , i );
  return(0);
}
```

11.12 Example 8 - do while and e**x evaluation

The function `etox` illustrates one use of the repeat until construct. The function evaluates e^x This may be written as

$$1 + x/1! + x^2/2! + x^3/3! \ldots$$

or

$$1 + \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!} x/n$$

Every succeeding term is just the previous term multiplied by x/n. At some point the term x/n becomes very small, so that it is not sensibly different from zero, and successive terms add little to the value. The function therefore repeats the loop until x/n is smaller than the tolerance. The number of evaluations is not known beforehand, since this is dependent on x:

```
#include <stdio.h>

int main()
{
  float term,x,etox;
  int nterm;
  float tol=1.0E-6;
  etox=1.0;
  term=1.0;
  nterm=0;
  x=1.0;
  do
  {
    nterm+=1;
    term = (x/nterm) * term;
    etox+=term;
  }
  while (term > tol);
  printf(" After %3d iterations\n",nterm);
  printf(" %f \n" , etox );
  return(0);
}
```

11.13 Example 9 - Counter controlled loops - the for statement

```
#include <stdio.h>

int main()
{
  float x[]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
  int i;
  int n;
  n=sizeof(x)/sizeof(float);
  printf(" Numbers are \n");
  for ( i=0 ; i < n ; i++)
    printf(" %5.1f\n", x[i] );
  return(0);
}
```

11.14 Example 10 - The for, continue and break statements

```
#include <stdio.h>

int main()
{
  int i;
  int n;
  int j[]={0,1,2,3,4,5,6,7,8,9};
  n=sizeof(j)/sizeof(int);
  for (i=0;i<n;i++)
  {
    if (j[i]>5) goto end;
    printf(" %3d\n", j[i] );
    continue;
    end:
    printf(" %3d is greater than 5\n",j[i]);
    break;
  }
  return(0);
}
```

11.15 Problems

1. Rewrite the program for the period of a pendulum. The new program should print out the length of the pendulum and period, for pendulum lengths from 0 to 100 cm in steps of 0.5 cm. The program should incorporate a function for the evaluation of the period.

2. Write a program to read an integer that must be positive.
Hint. use a `do while` to make the user re-enter the value.

3. Using functions, do the following:

- Evaluate $n!$ from $n = 0$ to $n = 10$
- Calculate 76!
- Now calculate $(x^n)/n!$, with $x = 13.2$ and $n = 20$.
- Now do it another way.

4. The function `etox` has been given in this chapter. The standard Fortran function `exp` does the same job. Do they give the same answers? Curiously the Fortran standard does not specify how a standard function should be evaluated, or even how accurate it should be.

5. The physical world has many examples in which processes require that some threshold be overcome before they begin operation: critical mass in nuclear reactions, a given slope to be exceeded before friction is overcome, and so on. Unfortunately, most of these sorts of calculations become rather complex and not really appropriate here. The following problem tries to restrict the range of calculation, whilst illustrating the possibilities of decision making.

If a cubic equation is expressed as

$$ax^3 + bx^2 + cx + d = 0$$

and we let

$$\Delta = 18abcd - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2$$

We can determine the nature of the roots as follows
$\Delta > 0$: three distinct real roots
$\Delta = 0$: has a multiple root and all roots are real
$\Delta < 0$: 1 real root and 2 non real complex conjugate roots
Incorporate this into a program, to determine the nature of the roots of a cubic from suitable input.

11.16 Bibliography

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

- This is the original text, and a must. The quote at the start of the chapter by Dijkstra summarises beautifully our limitations when programming and the discipline we must have to master programming successfully.

Knuth D.E., Structured programming with goto Statements, in Current Trends in programming Methodology, Volume 1, Prentice-Hall, 1977.

- The chapter by Knuth provides a very succinct coverage of the arguments for the adoption of structured programming, and dispels many of the myths concerning the use of the goto statement. Highly recommended.

Chapter 12

Pointers

12.1 Introduction

In this chapter we look at pointers in C.

- Basic pointer syntax
- Pointers and arrays

12.2 Example 1 - basic pointer syntax

```
#include <stdio.h>

int main()
{
  int i=999 ;
  int * p_i ;
  p_i = &i  ;
  printf(" Size of the pointer is ");
  printf(" %zd bytes \n",sizeof(p_i));
  printf("   i             %ld \n" , i    );
  printf("   address of i  %p \n" , &i   );
  printf("   value of p_i  %p \n" , p_i  );
  printf("   value of &p_i %p \n" , &p_i );
  printf("   value of *p_i %ld \n" , *p_i );
  *p_i=1010; // alter the value at the address p_i
  printf("   i             %ld \n" , i    );
  printf("   address of i  %p \n" , &i   );
  printf("   value of p_i  %p \n" , p_i  );
  printf("   value of &p_i %p \n" , &p_i );
  printf("   value of *p_i %ld \n" , *p_i );
  return(0);
}
```

The first key statement in this program is the declaration

```
int * p_i;
```

which declares p_i to be a pointer to an int.

The next statement

```
p_i = &i;
```

is an assignment statement and assigns the address of the variable i to the pointer p_i - we are assigning a pointer a value, which is an address.

The next set of statements print out i, the address of i, the pointer p_i, the address of the pointer p_i, and the value that p_i points to.

Here is the output of the complete program.

First we have the output from the Microsoft compiler.

```
 Size of the pointer is  8 bytes
   i             999
   address of i  000000810475FA60
   value of p_i  000000810475FA60
   value of &p_i 000000810475FA68
   value of *p_i 999
   i             1010
   address of i  000000810475FA60
   value of p_i  000000810475FA60
   value of &p_i 000000810475FA68
   value of *p_i 1010
```

Note that a pointer is 8 bytes on this platform, with this version of the Microsoft compiler.

Note that the output provides some information about how the compiler does things in memory.

Next we have the output from gcc under cygwin on Windows.

```
 Size of the pointer is  8 bytes
   i             999
   address of i  0xffffcc0c
   value of p_i  0xffffcc0c
   value of &p_i 0xffffcc00
   value of *p_i 999
   i             1010
   address of i  0xffffcc0c
   value of p_i  0xffffcc0c
   value of &p_i 0xffffcc00
   value of *p_i 1010
```

Again we have the pointer being 8 bytes.

12.3 Example 2 - pointers and assignment of a numeric literal

Look at the following example.

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

int main()
{
  int * p_i;
  *p_i=10;
  printf(" Value that p_i points to is %d \n",*p_i);
  return(0);
}
```

In this example we have declared the pointer and are then trying to assign a numeric literal.

```
  int * p_i;
  *p_i=10;
```

This programs crashes with the Microsoft compiler and causes a segmentation fault with the gcc compiler.

Whilst we have declared the pointer there is no memory to store the literal numeric value of 10 into.

You do not get a compilation error message with this program.

12.4 Example 3 - pointers and assignment of a numeric literal variant

Now consider the following variant.

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

int main()
{
  int * p_i = malloc (sizeof(int));
  *p_i=10;
  printf(" Value that p_i points to is %d \n",*p_i);
  return(0);
}
```

The difference now is that we have both declared the pointer and assigned memory so that the following assignment will now work.

Compile and run these two examples.

12.5 Example 4 - simple memory leak

It is very easy when using pointers to write a program that has a memory leak.

Consider the following example.

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

int main()
{
  int z=999;
  int * p_i = malloc (sizeof(int));
  *p_i=10;
  printf(" Value that p_i points to is %d \n",*p_i);
  printf(" Address of p_i           is %p \n",p_i);
  p_i=&z;
  printf(" Value that p_i points to is %d \n",*p_i);
  printf(" Address of p_i           is %p \n",p_i);
  return(0);
}
```

Here is the output from the Microsoft compiler.

```
 Value that p_i points to is 10
 Address of p_i           is 0000023379766350
 Value that p_i points to is 999
 Address of p_i           is 0000003BF94FF900
```

Here is the output from the gcc compiler.

```
 Value that p_i points to is 10
 Address of p_i           is 0x600010430
 Value that p_i points to is 999
 Address of p_i           is 0xffffcc04
```

In the above case the amount of memory being lost is small. Care needs to be taken when using pointers to avoid memory leaks.

12.6 Example 5 - arrays and pointers

In the following example we solve a problem in three different syntactic ways.

The first summation using conventional array syntax throughout.

The second summation initialises pstart using the base address of the x array, and then uses pointer arithmetic throughout.

The third summation assigns a value to pstart using the name of the array and illustrates the equivalence syntactically in C of and array and a pointer or address.

This was a decision that Kernighan and Ritchie made when designing C. It helped to simplify the implementation of the compiler.

The paper by Kernighan and Ritchie in the HOPL book goes into the design and implementation of C and is a fascinating read. There is a reference in the first chapter.

```
#include <stdio.h>

int main()
{
  int i;
  float sum=0.0,x[10],*pstart,*pend;
  printf("  Standard array access notation \n");
  for (i=0;i<10;++i)
  {
    x[i]=i;
    sum += x[i];
    printf("  %3d  %6.2f  %6.2f \n",i,x[i],sum);
  }
  sum=0.0;
  pstart = &x[0];
  pend = pstart+10;
  printf("  Using &x[0] for start address \n");
  while (pstart < pend)
  {
    sum += *pstart++;
    printf("  %6.2f\n",sum);
  }
  sum=0.0;
  pstart = x;
  pend = pstart+10;
  printf("  Using x for start address \n");
  while (pstart < pend)
  {
    sum += *pstart++;
    printf("  %6.2f\n",sum);
  }
  return(0);
}
```

12.7 Example 6 - Character arrays and pointers

The following example

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

```
int main()
{
  char line1[]=" This is a line of text";
  char line2[80];
  char* p1=line1;
  char* p2=line2;
  while ( *p2++ = *p1++);
  printf(" %s\n", line1);
  printf(" %s\n", line2 );
  return(0);
}
```

illustrates character array manipulation using pointer arithmetic and shows clearly the use of the zero byte terminator to control the process.

The key statement is

```
while ( *p2++ = *p1++ )
```

which copies from line1 to line 2 using pointer arithmetic, and the whole expression becomes false (0) when the zero byte terminator is encountered.

This also illustrates the equivalence in C of boolean true and false with integer values.

12.8 Example 7 - ragged 2 d arrays

In this example we look at creating a ragged 2d array in C, i.e. the number of columns in each row varies.

Here is the source code.

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

#define NROWS 3

int main()
{
  int *a[NROWS] ;
  int r;
  int ncols=3;
  int c;
  int i=1;

  for ( r=0 ; r<NROWS ; r++ )
  {
```

```
    for ( c=0 ; c<=r ; c++ )
    {
      a[r] = malloc( (r+1) * sizeof(int) );
    }
  }

  for ( r=0 ; r<NROWS ; r++ )
  {
    for ( c=0 ; c<=r ; c++ )
    {
      a[r][c] = i;
      i++;
    }
  }

  for ( r=0 ; r<NROWS ; r++ )
  {
    for ( c=0 ; c<=r ; c++ )
    {
      printf(" %3d ",a[r][c]);
    }
    printf("\n");
  }

  return(0);
}
```

Here is the output.

```
 1
 2   3
 4   5   6
```

So in this example

- row 0 has 1 column
- row 1 has 2 columns
- row 2 has 3 columns

Let us look at the source code in more depth.
The first statement of interest is

```
int *a[NROWS] ;
```

a is declared as an array of pointers to int.

The following statement

```
a[r] = malloc( (r+1) * sizeof(int) );
```

allocates each element of the array to be of size (r+1) * sizeof(int). So the first row has 1 column, the second row has two columns and the third row has three columns.

The next loop initialises each element of the array a using a conventional 2 d array syntax.

The next loop then prints the 2 d array out.

So C has a syntactic mechanism to create ragged 2 d arrays.

12.9 Problems

Compile and run the examples in this chapter.

Chapter 13

Functions

13.1 Introduction

In this chapter we look at functions in C.

- Intrinsic functions
- Parameter passing
- User defined functions

13.2 Example 1 - Basic intrinsic trig function usage

In this example we use some of the basic trig functions available in C.
We use the sin, cos and tan functions.
We have to include the math.h header file to make these functions available.
The chapter on arithmetic has more details on the math.h header file.

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

int main()
{

  double pi = 4.0 * atan(1.0);

  int angles[] = { -1 ,  0 ,  1 ,
                   29 , 30 , 31 ,
                   44 , 45 , 46 ,
                   59 , 60 , 61 ,
                   89 , 90 , 91 };
  int i;
```

```
  double r;

  for (i=0;i<15;i++)
  {
    r = angles[i] * pi /180.0;
    printf(" %3d ",angles[i]);
    printf("%22.18f " , sin(r) );
    printf("%22.18f " , cos(r) );
    printf("%22.18f\n" , tan(r) );
  }

  return(0);

}
```

Here is a sample of the output for the Microsoft compiler.

```
 -1  -0.017452406437283512   0.999847695156391270  -0.017455064928217585
  0   0.000000000000000000   1.000000000000000000   0.000000000000000000
  1   0.017452406437283512   0.999847695156391270   0.017455064928217585
 29   0.484809620246337059   0.874619707139395741   0.554309051452768986
 30   0.499999999999999944   0.866025403784438708   0.577350269189625731
 31   0.515038074910054156   0.857167300702112334   0.600860619027560383
 44   0.694658370458997254   0.719339800338651192   0.965688774807073935
 45   0.707106781186547573   0.707106781186547573   0.999999999999999889
 46   0.719339800338651081   0.694658370458997365   1.035530313790569368
 59   0.857167300702112223   0.515038074910054267   1.664279482350517370
 60   0.866025403784438597   0.500000000000000111   1.732050807568876749
 61   0.874619707139395741   0.484809620246337114   1.804047755271423581
 89   0.999847695156391270   0.017452406437283376  57.289961630759876243
 90   1.000000000000000000   0.000000000000000061 16331239353195370.000000000000000000
 91   0.999847695156391270  -0.017452406437283477 -57.289961630759549394
```

Doubles have between 15 and 18 digits of precision so we have used formatting to display most of the digits. The cosine of 90 degrees is nearly zero, and the tangent is large but not infinite. These results are consistent with IEEE double precision arithmetic in any programming language.

When using functions from the math.h library it is recommended that you look up detailed information like that shown below.

Visit

`http://en.cppreference.com/w/c/numeric`

for one on line source.

13.3 sin, sinf, sinl

```
Defined in header <math.h>

float       sinf( float arg );            (1)  (since C99)
double      sin( double arg );            (2)
long double sinl( long double arg );      (3)  (since C99)

Defined in header <tgmath.h>

#define sin( arg )                        (4)  (since C99)
```

13.3.1 Notes

1-3 Computes the sine of arg (measured in radians). 4 Type-generic macro: If the argument has type long double, sinl is called. Otherwise, if the argument has integer type or the type double, sin is called. Otherwise, sinf is called. If the argument is complex, then the macro invokes the corresponding complex function (csinf, csin, csinl).

13.3.2 Parameters

arg - floating point value representing an angle in radians

13.3.3 Return value

- If no errors occur, the sine of arg (sin(arg)) in the range [-1 ; +1], is returned.
- The result may have little or no significance if the magnitude of arg is large. (until C99)
- If a domain error occurs, an implementation-defined value is returned (NaN where supported).
- If a range error occurs due to underflow, the correct result (after rounding) is returned.

13.3.4 Error handling

Errors are reported as specified in math_errhandling.

- If the implementation supports IEEE floating-point arithmetic (IEC 60559),
- if the argument is ± 0, it is returned unmodified

- if the argument is $\pm\infty$, NaN is returned and FE_INVALID is raised
- if the argument is NaN, NaN is returned

13.4 Example 2 - 1 d arrays as parameters

Functions that work with one dimensional arrays as parameters are relatively straightforward in C.

In the following example the sum function can take a one d array of any size. The function prototype or signature is shown below.

```
int sum(int x[],int n)
```

So the array x can be of any size.

Here is the complete program.

```
#include <stdio.h>

int sum(int x[],int n)
{
  int t=0;
  for (int i=0;i<n;++i)
  t += x[i];
  return t;
}

int main()
{
  int a[10]={1,2,3,4,5,6,7,8,9,10};
  int
  b[20]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  int nn=10;
  int nnn=20;
  printf(" %5d\n" , sum(a,nn)  );
  printf(" %5d\n" , sum(b,nnn) );
  return(0);
}
```

Here is the output.

```
   55
  210
```

The two statements

```
printf(" %5d\n" , sum(a,nn) );
printf(" %5d\n" , sum(b,nnn) );
```

show how to call or use the sum function.

13.5 Example 3 - Recursive factorial function

Here is an example of a recursive function in C. It evaluates the factorial function.

So for a value of 5 we have

```
  5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1
```

Here is the program.

```
#include <stdio.h>

int factorial(int i)
{
  if (i==0)
    return(1);
  else
    return(i*factorial(i-1));
}

int main()
{
  int i=5;
  printf(" Factorial of 5 is %10d\n",  factorial(i) );
  return(0);
}
```

The key statement in the factorial function is shown below.

```
return(i*factorial(i-1));
```

where the function calls itself.

Adding the following statement

```
printf(" i = %d\n"',i);
```

will show what is happening dynamically as the program executes.

We have another example of a recursive function in a later chapter.

13.6 Example 4 - Passing 2 d arrays as parameters

Passing 2 d arrays in C is slightly more complex than passing one d arrays.

We have several examples which illustrate some of the options available to us.

In the first example we hard code the size of the second dimension.

Here is the source code.

```
#include <stdio.h>

int sum(int x[][5],int n,int m)
{
  int s;
  int i;
  int j;
  s=0;
  for (i=0;i<n;++i)
    for (j=0;j<m;++j)
      s+= x[i][j];
  return(s);
}

int main()
{
  int nr;
  int nc;
  int x1[2][5] = {{ 1, 2, 3, 4, 5},
                  { 6, 7, 8, 9,10}};
  int x2[3][5] = {{ 1, 2, 3, 4, 5},
                  { 6, 7, 8, 9,10},
                  {11,12,13,14,15}};
  nr=2;
  nc=3;
  printf(" Sum of 2 * 5 = %5d\n",sum(x1,nr,nc));
  nr=3;
  printf(" Sum of 3 * 5 = %5d\n",sum(x2,nr,nc));
  return(0);
}
```

The first dimension can vary, but the second dimension is fixed.

13.7 Example 5 - Passing 2 d arrays as parameters and integer summation

In this example we set an upper bound at compiler time on the first and second dimensions, and then can vary the number of rows and columns up to these compile

time upper bounds.

In the first example below we use integer arrays.

```
#include <stdio.h>

#define NROW 100
#define NCOL 100

int sum(int x[NROW][NCOL],int n,int m)
{
  int s;
  int i;
  int j;
  s=0;
  for (i=0;i<n;++i)
    for (j=0;j<m;++j)
      s+= x[i][j];
  return(s);
}

int main()
{
  int nr;
  int nc;
  int r;
  int c;

  int x1[NROW][NCOL] ;
  int x2[NROW][NCOL] ;

  nr=2;
  nc=3;

  for (r=0;r<nr;r++)
    for (c=0;c<nc;c++)
      x1[r][c]=r+c;

  printf(" Sum of 2 * 5 = %5d\n",sum(x1,nr,nc));

  nr=3;
  nc=4;

  for (r=0;r<nr;r++)
    for (c=0;c<nc;c++)
      x2[r][c]=r+c;
```

```
  printf(" Sum of 3 * 5 = %5d\n",sum(x2,nr,nc));

  return(0);

}
```

13.8 Example 6 - Passing 2 d arrays as parameters and double summation

In this example we use double arrays.

```
#include <stdio.h>

#define NROW 100
#define NCOL 100

double sum(double x[NROW][NCOL],int n,int m)
{
  double s;
  int i;
  int j;
  s=0;
  for (i=0;i<n;++i)
    for (j=0;j<m;++j)
      s+= x[i][j];
  return(s);
}

int main()
{
  int nr;
  int nc;
  int r;
  int c;

  double x1[NROW][NCOL] ;
  double x2[NROW][NCOL] ;

  nr=2;
  nc=3;

  for (r=0;r<nr;r++)
    for (c=0;c<nc;c++)
```

```
      x1[r][c]=(double)(r+c);

  printf(" Sum of 2 * 5 = %7.2f\n",sum(x1,nr,nc));

  nr=3;
  nc=4;

  for (r=0;r<nr;r++)
    for (c=0;c<nc;c++)
      x2[r][c]=(double)(r+c);

  printf(" Sum of 3 * 5 = %7.2f\n",sum(x2,nr,nc));

  return(0);

}
```

13.9 Example 7 - Passing 2 d integer dynamic arrays as parameters

C provides with a way of specifying the number of rows and columns at run time, but now we have lost the ability to use an array syntax and must use pointer arithmetic.

Here is the first example that works with integer arrays.

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

int sum(int* a,int nr,int nc)
{
  int s=0;
  int r;
  int c;
  for (r=0;r<nr;r++)
  {
    for (c=0;c<nc;c++)
    {
      printf(" %5d ",*(a+nc*r+c));
      s+=*(a+nc*r+c);
    }
    printf("\n");
  }
  return(s);
}
```

```
int main()
{
  int nr;
  int nc;
  int r;
  int c;
  int matrix_size;
  int s;
  int* x;
  int* base;
  nr=2;
  nc=3;
  matrix_size=nr*nc;
  x = (int *) calloc(matrix_size,sizeof(int));
  base=x;
  for (r=0;r<nr;r++)
  {
    for (c=0;c<nc;c++)
    {
      *x=(int)(1+nc*r+c);
      printf(" %5d ",*x);
      x++;
    }
    printf("\n");
  }
  x=base;
  s=0;
  s=sum(x,nr,nc);
  printf(" sum is %5d\n" , s);
  return(0);
}
```

The key statement in the calling routine or main program is

```
x = (int *) calloc(matrix_size,sizeof(int));
```

where x is a pointer to an int.

We use the above statement to dynamically allocate memory at run time to the correct size.

We could of course read in the values for the number of rows and columns.

The next code section

```
for (r=0;r<nr;r++)
{
  for (c=0;c<nc;c++)
```

```
  {
    *x=(int)(1+nc*r+c);
    printf(" %5d ",*x);
    x++;
  }
printf("\n");
}
```

initialises the matrix.

In the summation routine the following code

```
s+=*(a+nc*r+c);
```

does the summation, stepping through memory one integer at a time.

The expression in brackets

```
(a+nc*r+c);
```

is an address calculation, using pointer arithmetic.

We use the * operator to retrieve that matrix element at that address.

Here is an explicit evaluation of the nested loops for the values of the number of rows and columns in this program.

```
Assume a base address of 100 for a

  r   c   ( a + nc * r + c )

  0   0    100 + 3 * 0  + 0   = 100
  0   1    100 + 3 * 0  + 1   = 101
  0   2    100 + 3 * 0  + 2   = 102
  1   0    100 + 3 * 1  + 0   = 103
  1   1    100 + 3 * 1  + 0   = 104
  1   2    100 + 3 * 1  + 0   = 105
```

13.10 Example 8 - Passing 2 d double dynamic arrays as parameters

Here is a simple variant using doubles.

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

double sum(double* a,int nr,int nc)
```

```
{
  double s=0;
  int r;
  int c;
  for (r=0;r<nr;r++)
  {
    for (c=0;c<nc;c++)
    {
      printf(" %7.2f ",*(a+nc*r+c));
      s+=*(a+nc*r+c);
    }
    printf("\n");
  }
  return(s);
}

int main()
{
  int nr;
  int nc;
  int r;
  int c;
  int matrix_size;
  double s;
  double* x;
  double* base;
  nr=2;
  nc=3;
  matrix_size=nr*nc;
  x = (double *) calloc(matrix_size,sizeof(double));
  base=x;
  for (r=0;r<nr;r++)
  {
    for (c=0;c<nc;c++)
    {
      *x=1+nc*r+c;
      printf(" %7.2f ",*x);
      x++;
    }
    printf("\n");
  }
  x=base;
  s=0;
  s=sum(x,nr,nc);
  printf(" sum is %7.2f\n" , s);
```

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

13.11 Example 9 - Passing functions as parameters

Many problem solutions in programming require the ability to pass a function as a parameter.

In this example we show how to do it in C.

Here is the program source.

```
#include <stdio.h>

float f1(int i);
float f2(int i);
float f3(float (*f)(int i), int j);

float f1(int i)
{
  return 1.0/i;
}

float f2(int i)
{
  return 1.0/(i*i);
}

float f3(float (*f)(int i),int j)
{
  float t=0;
  for (int k=1 ; k<j ; ++k)
    t+=(*f)(k);
  return t;
}

int main()
{
  float (*t)(int i);
  printf("  Using f1 - simple reciprocal \n");
  printf("  %7.2f\n",f3(f1,5));
  printf("  Using f2 - 1/(i*i) \n");
  printf("  %7.2f\n",f3(f2,5));
  t=f1;
  printf("  using t=f1 \n");
  printf("  %7.2f\n",f3(t,5));
```

```
  t=f2;
  printf("  using t=f2 \n");
  printf("  %7.2f\n",f3(t,5));
  return(0);
}
```

Here is the function prototype for the function that takes a function as a parameter.

```
float f3(float (*f)(int i), int j);
```

So we have a function f3 whose first parameter is a function f that takes an integer argument and returns a float value. C requires us to pass a pointer to a function to make this work.

In the f3 routine the

```
t+=(*f)(k);
```

is the statement that calls the function f from within the function f3.

In the main program the following statement is the declaration of a variable which is a pointer to a function.

```
float (*t)(int i);
```

The function takes an integer argument and returns a float.

Here are some calls to f3 from within the main program.

f1 as argument.

```
printf(" %7.2f\n",f3(f1,5));
```

f2 as argument.

```
printf(" %7.2f\n",f3(f2,5));
```

Assign t1 to pointer variable t1.

```
t=f1;
```

t as argument.

```
printf(" %7.2f\n",f3(t,5));
```

Most programming languages have a mechanism to pass functions as parameters. C does it using pointers.

13.12 Function Arguments

The default parameter passing mechanism in C is pass by value, i.e. a copy of the value of each argument is taken and the function works with that.

The default parameter passing mechanism for array arguments in C is to pass the base address of the array. No copy is involved.

13.13 Example 10 - Swapping arguments - pass by address

Some problem solutions require changing the values of parameters in a function call.

Here is the implementation of a swap function in C.

```
#include <stdio.h>

void swap(int* p_i, int* p_j)
{
  int t;
  t=*p_i;
  *p_i=*p_j;
  *p_j=t;
}

int main()
{
  int i=1,j=2;
  printf("  i= %4d   j= %4d\n",i,j);
  swap(&i,&j);
  printf("  i= %4d   j= %4d\n",i,j);
  return(0);
}
```

13.14 Example 11 - Scope and duration

The scope of a declaration is the region of a program text over which the declaration is visible.

The scope of every identifier is limited to the C source file in which it occurs. An identifier can be declared to be external, in which case the same identifier can be referred to in two or more files.

An automatic variable declared at the beginning of a function is available in the body of the function.

There is also the concept of block scope.

The following example should help illustrate the scope rules in C.

```
#include <stdio.h>

void updatelocal();
void updatestatic();
void updateglobal();

int x=1;

void updatelocal()
{
  int x=10;
  printf(" Enter local  x = %5d\n",x);
  x++;
  printf(" Exit  local  x = %5d\n",x);
}

void updatestatic()
{
  static int x=100;
  printf(" Enter static x = %5d\n",x);
  x++;
  printf(" Exit  static x = %5d\n",x);
}

void updateglobal()
{
  printf(" Enter global x = %5d\n",x);
  x++;
  printf(" Exit  global x = %5d\n",x);
}

int main()
{
  int x=1000;
  printf(" main x = %5d\n",x);

  {
    int x=10000;
    printf(" main local block scope x = %5d\n",x);
  }

  printf(" main x = %5d\n",x);

  updatelocal();
  updatelocal();
```

```
  updatestatic();
  updatestatic();
  updateglobal();
  updateglobal();

  return(0);
}
```

13.15 Example 12 - Using header files

Here is the header file.

```
void updatelocal();
void updatestatic();
void updateglobal();
void updateextern();

int x=1;
int y=1;
```

Here is the C source file.

```
void updatelocal()
{
  int x=10;
  printf(" Enter local  x = %5d\n",x);
  x++;
  printf(" Exit  local  x = %5d\n",x);
}

void updatestatic()
{
  static int x=100;
  printf(" Enter static x = %5d\n",x);
  x++;
  printf(" Exit  static x = %5d\n",x);
}

void updateglobal()
{
  printf(" Enter global x = %5d\n",x);
  x++;
  printf(" Exit  global x = %5d\n",x);
}
```

Here is the main program.

```
#include <stdio.h>

#include "c1312.h"

#include "c1312.c"

int main()
{
  int x=1000;
  printf(" main x = %5d\n",x);

  {
    int x=10000;
    printf(" main local block scope x = %5d\n",x);
  }

  printf(" main x = %5d\n",x);

  updatelocal();
  updatelocal();
  updatestatic();
  updatestatic();
  updateglobal();
  updateglobal();

  return(0);
}
```

13.16 Storage class

C has the following storage classes

- auto - Permitted only in declarations of variables within blocks.
- extern - May appear in declarations of external functions and variables, at the top level or within blocks. The name is known to the linker.
- register - maybe used for variables of parameter declarations. Equivalent to auto and a hint to the compiler that the object will be heavily used and should be allocated in a way to minimise access time.
- static - May appear on function and variable declarations. On function names it means that the name is not exported to the linker. For data declarations not

exported to the linker. Variables with static storage class have static extent, as opposed to local extent.

- typedef - new name for an existing type.

13.17 Problems

Compile and run the examples in this chapter.

Chapter 14

C99 Variable length arrays

14.1 Introduction

In C99, the size or bounds of an array can be a run-time expression. Such arrays are called variable length arrays or VLAs for short. VLAs can simplify storage management in a program and allow the use of the normal array notation even when the problem to be solved requires arrays to have different sizes at different times.

We have some examples in this chapter to show how to use them.

14.2 Example 1 - simple vla usage

Here is the source code.

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

void f1( int size, int x[ size ] )
{
  for ( int i = 0; i < size; i++ )
    printf( " %d  ", x[ i ] );
  printf("\n");
}

void f2( int row, int col, int x[ row ][ col ] )
{
  for ( int r = 0; r < row; r++ )
  {
    for ( int c = 0; c < col; c++ )
      printf( "%5d", x[ r ][ c ] );
    printf("\n");
  }
}
```

```
int main()
{

  int n;
  int nrows, ncols;
  int r;
  int c;

  n=10;
  nrows=2;
  ncols=3;

  int x[ n ];
  int x1[ nrows ][ ncols ];

  int t=1;

  for ( r=0;  r< n; r++ )
    x[ r ] = r + 1;

  for ( r = 0; r < nrows; r++ )
    for ( c = 0; c < ncols; c++ )
    {
      x1[ r ][ c ] = t;
      t++;
    }

  printf("\n Calling 1 d function \n\n");
  f1( n, x );

  printf("\n Calling 2 d function \n\n");
  f2( nrows, ncols, x1 );

  return(0);

}
```

Here is the output from compiling and running this program with gcc 4.9.3.
You need to compile with the -std=c99 switch.
The Microsoft compiler does not currently support C99 vlas.

```
 Calling 1 d function
```

```
 1   2   3   4   5   6   7   8   9   10

 Calling 2 d function

    1    2    3
    4    5    6
```

14.3 Example 2 - calculating the mean, standard deviation and median

This program uses vlas in the find routine.
Here is the source code.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double find(double y[] , int n , int k)
{

  int  l, r, i, j;
  double  t1, t2;

  l = 1;
  r = n;
  while (l<r)
  {
    t1 = y[k];
    i = l;
    j = r;
    for(;;)
    {
      while (y[i]<t1)
      {
        i = i + 1;
      }
      while (t1<y[j])
      {
        j = j - 1;
      }
      if (i<=j)
      {
```

```
        t2 = y[i];
        y[i] = y[j];
        y[j] = t2;
        i = i + 1;
        j = j - 1;
      }
      if (i>j) break;
    }
    if (j<k)
    {
      l = i;
    }
    if (k<i)
    {
      r = j;
    }
  }
  return(y[k]);
}

void statistics( int n , double x[ n ] ,
  double *mean, double *std_dev, double *median)
{

  double sumxi;
  double sumxi2;

  double variance;

  double temp[n];
  int i;
  int k1;
  int k2;

  sumxi    = 0.0;
  sumxi2   = 0.0;
  variance = 0.0;

  for (i=0;i<n;i++)
  {
    sumxi  += x[i];
    sumxi2 += x[i]*x[i];
  }

  *mean     = sumxi/n;
```

```
  variance = (sumxi2 - sumxi*sumxi/n)/(n-1);
  *std_dev  = sqrt(variance);

  for(i=0;i<n;i++)
    temp[i]=x[i];

  if ( (n%2) == 0 )
  {
    k1=(n/2);
    k2=(n/2)+1;
    *median = ( find(temp,n,k1) + find(temp,n,k2) ) /2;
  }
  else
  {
    k1 = (n/2)+1;
    *median = find(temp,n,k1);
  }
}

int main()
{
  int n=1000;
  int r;
  int i;

  double *x;
  double mean=0.0;
  double std_dev = 0.0;
  double median = 0.0;

  printf(" RAND_MAX            = %d \n",RAND_MAX);
  printf(" Approximate average = %d \n",RAND_MAX/2);

  for (i=1;i<4;i++)
  {
    x =  malloc( n * sizeof(double) );
    for ( r=0;  r< n; r++ )
      x[ r ] = (double) rand();
    printf("\n Calling statisics routine \n\n");
    printf(" N = %d \n",n);
    statistics( n , x , &mean , &std_dev, & median);
    printf(" Mean               = %f \n",mean);
    printf(" Standard deviation = %f \n",std_dev);
    printf(" Median             = %f \n",median);
    mean=0.0;
```

```
    std_dev = 0.0;
    median = 0.0;
    n = n * 10;
    free(x);
  }

  return(0);

}
```

Here is the output.

```
RAND_MAX            = 2147483647
Approximate average = 1073741823

Calling statisics routine

N = 1000
Mean               = 1094518412.452000
Standard deviation = 610594916.933025
Median             = 1104055479.500000

Calling statisics routine

N = 10000
Mean               = 1064364950.095900
Standard deviation = 619576908.569395
Median             = 1053229961.500000

Calling statisics routine

N = 100000
Mean               = 1071004067.657930
Standard deviation = 618866980.320345
Median             = 1070966905.000000
```

This was using gcc 4.9.3 under cygwin.

RAND_MAX is based on a 32 integer. The mean and median should be approximately half of RAND_MAX.

14.4 Notes

These arrays are normally implemented on the stack. You may have problems with large array sizes.

14.5 Problems

Compile and run the examples in this chapter.

Chapter 15

Structs

Russells theory of types leads to certain complexities in the foundations of mathematics,... Its interesting features for our purposes are that types are used to prevent certain erroneous expressions from being used in logical and mathematical formulae; and that a check against violation of type constraints can be made purely by scanning the text, without any knowledge of the value which a particular symbol might happen to have.

C. A. R. Hoare, Structured Programming.

It is said that Lisp programmers know that memory management is so important that it cannot be left to the users and C programmers know that memory management is so important that is cannot be left to the system.

anon

15.1 Introduction

User defined data types are an essential part of general purpose programming languages. Early languages provided this functionality via concrete data types, later languages by abstract data types.

C provides concrete data types using structs. It was possible to emulate abstract data typing in C by providing access to the data in the struct via functions.

In this chapter we look at some examples illustrating the use of structs in C.

15.2 Example 1 - Basic struct syntax and use

There are two stages in the process of creating and using our own data types, we must first define that type and second create variables of that type.

Here is the source of the first example.

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

```
#define FILE_NAME_LENGTH 80
#define LINE_LENGTH      81

int main()
{

  char  input_file_name[FILE_NAME_LENGTH]="c1501_in.txt";
  char output_file_name[FILE_NAME_LENGTH]="c1501_out.txt";

  char line[LINE_LENGTH];

  FILE *in_ptr;
  FILE *out_ptr;

  in_ptr  = fopen(input_file_name  , "r");
  out_ptr = fopen(output_file_name , "w");

  int i;
  int n_lines=3;

  for(i=0;i<n_lines;i++)
  {
    fgets(line,LINE_LENGTH-1,in_ptr);
    fprintf(out_ptr , "%s" , line);
  }

  printf(" Program ends\n");

  return(0);

}
```

The

```
struct date
{
int day;
int month;
int year;
};
```

statements define date to be a data type or struct with three integer components called day, month and year.

The

```
struct date d;
```

declares d to be a variable of type date.
We need to repeat the struct keyword.
The next three statements

```
d.day=1;
d.month=1;
d.year=1996;
```

initialise the components of the date variable d (day, month and year) to have values of 1, 1 and 1996 respectively. d.day, d.month and d.year are visible or public.
The following statement

```
printf(" Date = %2d/%2d/%4d\n",d.day,d.month,d.year);
```

prints out the values of the date variable d, as we have direct access to the components of this variable.

15.3 Example 2 - Passing structs as parameters

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

#define FILE_NAME_LENGTH 80
#define LINE_LENGTH      81

int main()
{

  char  input_file_name[FILE_NAME_LENGTH]="cwmystwythdata.txt";
  char output_file_name[FILE_NAME_LENGTH]="cwmystwyth.txt";

  char line[LINE_LENGTH];

  FILE *in_ptr;
  FILE *out_ptr;

  in_ptr  = fopen(input_file_name  , "r");
  out_ptr = fopen(output_file_name , "w");

  int i;
  int n_lines=0;

  while (fgets(line,LINE_LENGTH-1,in_ptr) != NULL)
  {
    n_lines++;
```

```
    if (n_lines<=8) continue;
//    if (line[0]=='S') continue;
    fprintf(out_ptr , "%s" , line);
  }

  fclose(in_ptr);
  fclose(out_ptr);

  printf(" Lines read = %6d\n",n_lines);
  printf(" Program ends\n");

  return(0);

}
```

15.4 Example 3 - Passing arrays of structs as parameters - 1

```
#include <stdio.h>

struct date
{
  int day;
  int month;
  int year;
};

void print_date(struct date d[],int n)
{
  int i;
  for(i=0;i<n;i++)
  {
    printf(" Date = %2d/%2d/%4d\n",d[i].day,d[i].month,d[i].year);
  }
}

int main()
{
  int n=2;
  int i;
  struct date d[2];
  d[0].day=1;
  d[0].month=1;
```

```
  d[0].year=1996;
  d[1].day=11;
  d[1].month=2;
  d[1].year=1952;
  print_date(d,n);
  return(0);
}
```

15.5 Example 4 - Passing arrays of structs as parameters - 2

```
#include <stdio.h>

struct date
{
  int day;
  int month;
  int year;
};

void print_date(struct date d[],int n)
{
  int i;
  for(i=0;i<n;i++)
  {
    printf(" Date = %2d/%2d/%4d\n",d[i].day,d[i].month,d[i].year);
  }
}

void update(struct date d[],int n)
{
  d[0].day=11;;
  d[0].month=2;
  d[0].year=1952;
  d[1].day=1;
  d[1].month=3;
  d[1].year=1956;
}

int main()
{
  int n=2;
  int i;
```

```
  struct date d[2];
  d[0].day=1;
  d[0].month=1;
  d[0].year=1996;
  d[1].day=11;
  d[1].month=2;
  d[1].year=1952;
  print_date(d,n);
  update(d,n);
  print_date(d,n);

  return(0);
}
```

15.6 Example 5 - Example - date data type using get and set functions

The following example provide access to the day month and year components of a date data type using get and set functions.

Here is the source code.

```
#include <stdio.h>

struct date
{
  int day;
  int month;
  int year;
};

void set_day(struct date * d,int day)
{
  d->day=day;
}

int get_day(struct date d)
{
  return(d.day);
}

void set_month(struct date * d,int month)
{
  d->month=month;
}
```

```
int get_month(struct date d)
{
  return(d.month);
}

void set_year(struct date * d,int year)
{
  d->year=year;
}

int get_year(struct date d)
{
  return(d.year);
}

int main()
{
  struct date d;
  set_day(&d,1);
  set_month(&d,1);
  set_year(&d,1996);
  printf(" Date = %2d/%2d/%4d\n",get_day(d),get_month(d),get_year(d));
  return(0);
}
```

15.7 Problems

Compile and run the examples in this chapter.

Chapter 16

Data structures

16.1 Introduction

This chapter looks at two simple data structure examples in C, a linked list and a binary tree.

16.2 Example 1 - a simple singly linked list

Here is the source code.

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

#define WORDLENGTH  20
#define NWORDS     100

int main()
{
  struct item
  {
    char word[WORDLENGTH];
    struct item *next;
  };

  struct item x[NWORDS];
  struct item *first=NULL;
  struct item *p,**ptr;

  char word[WORDLENGTH];

  int in = 0;
```

```
  /* read all the words in */

  while (scanf("%s",word) == 1)
  {
    ptr = &first;

    /* Find where to insert */

    while ((p=*ptr) != NULL)
    {
      if (strcmp(p->word,word) > 0)
        break;
      ptr = &(p->next);
    }

    /* Add in correct position */

    strcpy(x[in].word,word);
    x[in].next = p;
    *ptr = &(x[in++]);
  }

  /* Now print them out */

  p = first;

  while (p!= NULL)
  {
    printf("%s\n",p->word);
    p = p->next;
  }

  return(0);

}
```

Here is the text input file.

```
This is the first line of text

This is the last
```

Here is the output.

```
This
```

```
This
first
is
is
last
line
of
text
the
the
```

16.3 Example 2 - binary tree

Here is the source code.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct tree_node
{
  struct tree_node *left_ptr;
  int data;
  struct tree_node *right_ptr;
};

typedef struct tree_node tree_node;
typedef tree_node *tree_node_ptr;

void insert_node( tree_node_ptr *tree_ptr, int value );
void in_order( tree_node_ptr tree_ptr );
void pre_order( tree_node_ptr tree_ptr );
void post_order( tree_node_ptr tree_ptr );
void print_tree( tree_node_ptr t,int h);

int main()
{
  int i;
  int item;
  int n=10;
  int x[10] = {1,10,2,9,3,8,4,7,5,6};

  tree_node_ptr root_ptr = NULL;
```

```
  printf("Creating the tree with %d entries \n",n );

  for ( i = 0; i < n; ++i )
  {
    item = x[i];
    insert_node( &root_ptr, item );
  }

  printf( "\n Pre_order traversal  " );
  pre_order( root_ptr );

  printf( "\n In_order traversal   " );
  in_order( root_ptr );

  printf( "\n Post_order traversal " );
  post_order( root_ptr );

  printf("\n Print tree \n");

  print_tree( root_ptr,n);

  return(0);

}

void insert_node( tree_node_ptr *tree_ptr, int value )
{
  if ( *tree_ptr == NULL )
  {
    *tree_ptr = malloc( sizeof( tree_node ) );
    if ( *tree_ptr != NULL )
    {
      ( *tree_ptr )->data = value;
      ( *tree_ptr )->left_ptr = NULL;
      ( *tree_ptr )->right_ptr = NULL;
    }
    else
    {
      printf( "%d not inserted. No memory available.\n", value );
    }
  }
  else
  {
    if ( value < ( *tree_ptr )->data )
    {
```

```
      insert_node( &( ( *tree_ptr )->left_ptr ), value );
    }
    else if ( value > ( *tree_ptr )->data )
    {
      insert_node( &( ( *tree_ptr )->right_ptr ), value );
    }
    else
    {
      printf( "%s", "duplicate" );
    }
  }
}

void in_order( tree_node_ptr tree_ptr )
{
  if ( tree_ptr != NULL )
  {
    in_order( tree_ptr->left_ptr );
    printf( "%3d", tree_ptr->data );
    in_order( tree_ptr->right_ptr );
  }
}

void pre_order( tree_node_ptr tree_ptr )
{
  if ( tree_ptr != NULL )
  {
    printf( "%3d", tree_ptr->data );
    pre_order( tree_ptr->left_ptr );
    pre_order( tree_ptr->right_ptr );
  }
}

void post_order( tree_node_ptr tree_ptr )
{
  if ( tree_ptr != NULL )
  {
    post_order( tree_ptr->left_ptr );
    post_order( tree_ptr->right_ptr );
    printf( "%3d", tree_ptr->data );
  }
}

void print_tree(tree_node_ptr t,int h)
{
```

```
  int i;
  if( t != NULL )
  {
    print_tree( t->left_ptr , h+1);
    for(i=1;i<h;i++)
      printf(" ");
    printf("%3d\n",t->data);
    print_tree( t->right_ptr , h+1);
  }
}
```

Here is the output.

```
Creating the tree with 10 entries

 Pre_order traversal    1 10  2  9  3  8  4  7  5  6
 In_order traversal     1  2  3  4  5  6  7  8  9 10
 Post_order traversal   6  5  7  4  8  3  9  2 10  1
 Print tree
          1
            2
              3
                4
                  5
                   6
                 7
               8
             9
          10
```

16.4 Problems

Compile and run the examples in this chapter.

Chapter 17

Miscellaneous examples

17.1 Introduction

This chapter looks at a small number of additional examples.

17.2 Example 1 - using the data and time intrinsics

Here is a simple example using the data and time intrinsic functionality provided by C.

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

int main()
{
  time_t local_time;
  struct tm *today;
  int day=0,month=0,year=0;

  time(&local_time);
  printf("%s",ctime(&local_time));
  today = localtime(&local_time);
  day   = today->tm_mday;
  month = today->tm_mon;
  year  = (today->tm_year)+1900;
  printf("%2d/%2d/%4d\n",day,month,year);

  return(0);
}
```

The variable local_time is of type time_t. time_t is a long integer and is used to represents time values in time.

The variable today is a pointer to the tm structure. The tm structure is used by asctime, gmtime, localtime, mktime, and strftime to store and retrieve time information.The fields of the structure type tm store the following values, each of which is an int:

```
tm_sec - Seconds after minute (0  59)
tm_min - Minutes after hour (0  59)
tm_hour - Hours after midnight (0  23)
tm_mday - Day of month (1  31)
tm_mon - Month (0  11; January = 0)
tm_year - Year (current year minus 1900)
tm_wday - Day of week (0  6; Sunday = 0)
tm_yday - Day of year (0  365; January 1 = 0)
tm_isdst - Positive value if daylight saving time is in effect;
  0 if daylight saving time is not in effect;
negative value if status of daylight saving time is unknown.
```

The C run-time library assumes the United Statess rules for implementing the calculation of Daylight Saving Time (DST).

The time function gets the system time. Its prototype is

```
time_t time( time_t *timer )
```

It returns a value of type time_t and takes one argument which is a pointer to a variable of type time_t.

The localtime function converts a time value and corrects for the local time zone. Its prototype is:

```
struct tm *localtime( const time_t *timer );
```

It takes one argument which is a pointer to a constant variable of type time_t. It returns a pointer to a structure tm. If the value in timer represents a date before midnight, January 1, 1970, localtime returns NULL. The explanation for this lies with Kernighan and Ritchie.

Here is the output.

```
Fri Apr  8 12:29:34 2016
 8/ 3/2016
```

17.3 Example 2 - simple quicksort

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

```
#define N 100000000

void swap(int array[],int i, int j)
{
  int tmp=array[i];
  array[i]=array[j];
  array[j]=tmp;
}

void quicksort(int array[], int l, int r)
{
  int i=l;
  int j=r;
  int v=array[(int)((l+r)/2)];
  for (;;)
  {
    while (array[i] < v) i=i+1;
    while (v < array[j]) j=j-1;
    if (i<=j) { swap(array,i,j); i=i+1 ; j=j-1; }
    if (i>j) goto ended ;
  }
  ended: ;
  if (l<j) quicksort(array,l,j);
  if (i<r) quicksort(array,i,r);
}

int main()
{

  time_t local_time;
  struct tm *today;
  int i;
  int *x;
  time(&local_time);

  printf("%s",ctime(&local_time));

  x = (int *) calloc(N,sizeof(int));

  printf("%s",ctime(&local_time));

  for (i=1;i<=N;++i)
  {
    x[i]=rand();
```

```
  }

  time(&local_time);
  printf("%s",ctime(&local_time));

  for(i=0;i<10;i++)
    printf(" %10d\n",x[i]);

  printf(" Quicksort starts\n");
  quicksort(x,0,N-1);
  printf(" Quicksort ends\n");

  time(&local_time);
  printf("%s",ctime(&local_time));

  for(i=0;i<10;i++)
    printf(" %10d\n",x[i]);

  return(0);

}
```

17.4 Example 3 - sorting with timing

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

#define N 100000000

void swap(int array[],int i, int j)
{
  int tmp=array[i];
  array[i]=array[j];
  array[j]=tmp;
}

void quicksort(int array[], int l, int r)
{
  int i=l;
  int j=r;
  int v=array[(int)((l+r)/2)];
  for (;;)
```

```
  {
    while (array[i] < v) i=i+1;
    while (v < array[j]) j=j-1;
    if (i<=j) { swap(array,i,j); i=i+1 ; j=j-1; }
    if (i>j) goto ended ;
  }
  ended: ;
  if (l<j) quicksort(array,l,j);
  if (i<r) quicksort(array,i,r);
}

int main()
{

  time_t local_time;
  struct tm *today;
  int i;
  int *x;
  time(&local_time);

  printf("%s",ctime(&local_time));

  x = (int *) calloc(N,sizeof(int));

  printf("%s",ctime(&local_time));

  for (i=1;i<=N;++i)
  {
    x[i]=rand();
  }

  time(&local_time);
  printf("%s",ctime(&local_time));

  for(i=0;i<10;i++)
    printf(" %10d\n",x[i]);

  printf(" Quicksort starts\n");
  quicksort(x,0,N-1);
  printf(" Quicksort ends\n");

  time(&local_time);
  printf("%s",ctime(&local_time));

  for(i=0;i<10;i++)
```

```
    printf(" %10d\n",x[i]);

  return(0);

}
```

Here are some sample runs.

17.5 Problems

Compile and run the examples in this chapter.

Chapter 18

Files

18.1 Introduction

In this chapter we look at examples illustrating some basic file concepts of C.

18.2 Example 1 - copying a text file

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

#define FILE_NAME_LENGTH 80
#define LINE_LENGTH      81

int main()
{

  char  input_file_name[FILE_NAME_LENGTH]="c1501_in.txt";
  char output_file_name[FILE_NAME_LENGTH]="c1501_out.txt";

  char line[LINE_LENGTH];

  FILE *in_ptr;
  FILE *out_ptr;

  in_ptr  = fopen(input_file_name  , "r");
  out_ptr = fopen(output_file_name , "w");

  int i;
  int n_lines=3;

  for(i=0;i<n_lines;i++)
  {
```

```
    fgets(line,LINE_LENGTH-1,in_ptr);
    fprintf(out_ptr , "%s" , line);
  }

  printf(" Program ends\n");

  return(0);

}
```

18.3 Example 2 - Reading a Met Office file

The site is

```
http://www.metoffice.gov.uk/public/
weather/climate-historic/#?tab=climateHistoric
```

The program below reads a Met Office station file and strips off the header.

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

#define FILE_NAME_LENGTH 80
#define LINE_LENGTH      81

int main()
{

  char  input_file_name[FILE_NAME_LENGTH]="cwmystwythdata.txt";
  char output_file_name[FILE_NAME_LENGTH]="cwmystwyth.txt";

  char line[LINE_LENGTH];

  FILE *in_ptr;
  FILE *out_ptr;

  in_ptr  = fopen(input_file_name  , "r");
  out_ptr = fopen(output_file_name , "w");

  int i;
  int n_lines=0;

  while (fgets(line,LINE_LENGTH-1,in_ptr) != NULL)
  {
    n_lines++;
```

```
    if (n_lines<=8) continue;
//    if (line[0]=='S') continue;
    fprintf(out_ptr , "%s" , line);
  }

  fclose(in_ptr);
  fclose(out_ptr);

  printf(" Lines read = %6d\n",n_lines);
  printf(" Program ends\n");

  return(0);

}
```

18.4 Problems

Compile and run the examples in this chapter.

Chapter 19

Parallel programming using openmp

19.1 Introduction

The main OpenMP site is

`http://openmp.org/wp/`

and this has details about the various specifications

`http://openmp.org/wp/openmp-specifications/`

We recommend downloading the documentation if you are going to do OpenMP programming. You should visit

`http://openmp.org/wp/openmp-compilers/`

to see an up to date list of what compilers support the OpenMP specification, and at what level.

The OpenMP site has a range of resources available, check out

`http://openmp.org/wp/resources`

for more information.

We've run the examples in this chapter with one or more of the following compilers

- gcc
- Microsoft

19.2 OpenMP memory model

OpenMP is a shared memory programming model. It has several features including

- All threads have access to the same shared memory
- Data can be shared or private
- Data transfer is transparent to the programmer
- Synchronization takes place and is generally implicit

We will look at a small number of examples to highlight some of the key features. We provide a brief coverage of some of the OpenMP glossary to provide a basic background to OpenMP.

- Threading Concepts
  - Thread - An execution entity with a stack and associated static memory, called threadprivate memory.
  - OpenMP thread - A thread that is managed by the OpenMP runtime system.
  - Thread-safe routine - A routine that performs the intended function even when executed concurrently (by more than one thread).
- OpenMP language terminology
  - Structured block - For Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.
  - Loop directive - An OpenMP executable directive whose associated user code must be a loop that is a structured block. For Fortran, only the do directive and the optional end do directive are loop directives.
  - Master thread - The thread that encounters a parallel construct, creates a team, generates a set of tasks, then executes one of those tasks as thread number 0.
  - Worksharing construct - A construct that defines units of work, each of which is executed exactly once by one of the threads in the team executing the construct. For Fortran, worksharing constructs are do, sections, single and workshare.
  - Barrier - A point in the execution of a program encountered by a team of threads, beyond which no thread in the team may execute until all threads in the team have reached the barrier and all explicit tasks generated by the team have executed to completion.
- Data Terminology

  - Variable - A named data object, whose value can be defined and redefined during the execution of a program. Only an object that is not part of another object is considered a variable. For example, array elements, structure components, array sections and substrings are not considered variables.
  - Private variable - With respect to a given set of task regions that bind to the same parallel region, a variable whose name provides access to a different block of storage for each task region.
  - Shared variable - With respect to a given set of task regions that bind to the same parallel region, a variable whose name provides access to the same block of storage for each task region.

- Execution Model

  - The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library).

The above coverage should be enough to get started with OpenMP and understand the examples that follow.

19.3 Example 1

Here is the first program. You will need to look at the compiler options later.

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

int main()
{
  int nthreads;
  int thread_number;
  int i;

  nthreads = omp_get_max_threads();
  printf(" Number of threads = %3d\n",nthreads);

#pragma omp parallel for
  for (i=0;i<nthreads;i++)
  {
    printf(" $%3d Hello from thread %3d \n",i,omp_get_thread_num());
```

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

The first statement of interest is

```
#include <omp.h>
```

This statement makes available the OpenMP environment. OpenMP statements are treated as comments without this statement.

The following two statements

```
nthreads = omp_get_max_threads();
printf(" Number of threads = %3d\n",nthreads);
```

The first statement sets the variable nthread to the value returned by the OpenMP function omp_get_max_threads(). We then print out this value.

```
#pragma omp parallel for
```

OpenMP directives in C start with the # character.

The parallel for words indicate that the code that follows is a parallel region construct. In this case a for loop. Here is a small table listing some of the OpenMP directives.

```
               Parallel region construct

#omp parallel [clause]
structured block

               Work sharing constructs

#omp for [clause] ...
do loop
#omp sections [clause] ...
[#omp section
structured block ] ...
#omp single [clause]
structured block

               Combined parallel work
               sharing constructs

#omp parallel for [clause]
structured block
```

```
#omp parallel sections [clause] ...
[#omp section
structured block ] ...

               Synchronisation constructs

#omp master
structured block
#omp critical [(name)]
structured block
#omp barrier
#omp atomic
expression list
#omp flush
#omp ordered
structured block

               Data environment

#omp threadprivate (/c1/,/c2/)
```

We next have the parallel for.

```
#pragma omp parallel for
for (i=0;i<nthreads;i++)
  {
    printf(" $%3d Hello from thread %3d \n",i,omp_get_thread_num());
  }
```

This loop prints out a message from each thread showing the thread number.

This closing { marks the end of the OpenMP parallel loop.

So at the start of the loop the OpenMP run time system does a fork and creates multiple threads. At the end of the loop we have a join operation and we are back to one thread of execution.

Here is the output from an AMD Phenom system with 6 cores.

```
$ gcc  -fopenmp openmp01.c

ian@Dell-7100 /cygdrive/c/document/c/c_notes
$ ./a.exe
 Number of threads =   6
 $  1 Hello from thread   1
 $  4 Hello from thread   4
 $  2 Hello from thread   2
 $  0 Hello from thread   0
```

```
 $  3 Hello from thread   3
 $  5 Hello from thread   5

ian@Dell-7100 /cygdrive/c/document/c/c_notes
```

Here is the Microsoft output.

```
c:\document\c\c_notes>cl openmp01.c /openmp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

openmp01.c
Microsoft (R) Incremental Linker Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:openmp01.exe
openmp01.obj

c:\document\c\c_notes>openmp01
 Number of threads =   6
 $  0 Hello from thread   0
 $  2 Hello from thread   2
 $  1 Hello from thread   1
 $  3 Hello from thread   3
 $  4 Hello from thread   4
 $  5 Hello from thread   5
```

19.4 Example 2

This is a simple variation on the first example. At first sight it appears to be identical in effect to example one.

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

int main()
{
  int nthreads;
  int thread_number;
  int i;

  nthreads = omp_get_max_threads();
  printf(" Number of threads = %3d\n",nthreads);

#pragma omp parallel for
```

```
  for (i=0;i<nthreads;i++)
  {
    thread_number = omp_get_thread_num();
    printf(" $%3d Hello from thread %3d \n",i,thread_number);
  }
  return(0);
}
```

However we have introduced a variable thread_number and are using the OpenMP default data scoping rules, i.e. we have said nothing.

We appear to have a working program. Here is the output from the gcc and Microsoft compilers.

```
$ gcc -fopenmp openmp02.c

ian@Dell-7100 /cygdrive/c/document/c/c_notes
$ ./a.exe
 Number of threads =   6
 $  2 Hello from thread   2
 $  0 Hello from thread   0
 $  3 Hello from thread   3
 $  5 Hello from thread   5
 $  4 Hello from thread   4
 $  1 Hello from thread   1


c:\document\c\c_notes>cl openmp02.c /openmp
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.23506 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

openmp02.c
Microsoft (R) Incremental Linker Version 14.00.23506.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:openmp02.exe
openmp02.obj

c:\document\c\c_notes>openmp02
 Number of threads =   6
 $  0 Hello from thread   0
 $  5 Hello from thread   5
 $  3 Hello from thread   3
 $  2 Hello from thread   2
 $  4 Hello from thread   4
 $  1 Hello from thread   1
```

The default variable scoping rules mean that the variable thread_number is available to all threads - in OpenMP terminology it is shared.

The opposite of shared is private and each thread has their own copy. Example 3 corrects this problem.

19.5 Example 3

Here is the source code.

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

int main()
{
  int nthreads;
  int thread_number;
  int i;

  nthreads = omp_get_max_threads();
  printf(" Number of threads = %3d\n",nthreads);

#pragma omp parallel for private(thread_number)
  for (i=0;i<nthreads;i++)
  {
    thread_number = omp_get_thread_num();
    printf(" $%3d Hello from thread %3d \n",i,omp_get_thread_num());
  }
  return(0);
}
```

19.6 Example 4 - parallel solution for PI calculation

We choose numerical integration in this example. The following integral

$$\int_0^1 \frac{4}{1+x^2}\, dx$$

is one way of calculating an approximation to π, and is a problem that is easy to parallelise. The integral can be approximated by

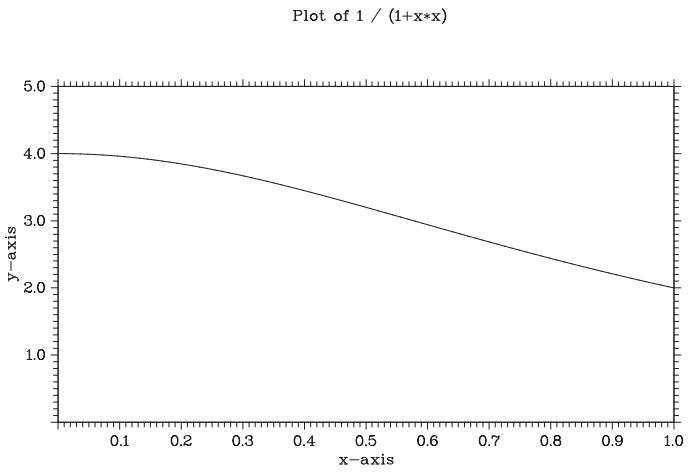
$$1/n \sum_1^n \frac{4}{1+\left(\frac{i-0.5}{n}\right)^2}$$

According to Wikipedia π to 50 digits is

3.14159265358979323846264338327950288419716939937510

Another way of calculating π is using the formula 4 $tan^{-1}(1)$ and in C this is 4.0*atan(1.0).

Consider the following plot of the above equation.



To do the evaluation numerically we divide the interval between 0 and 1 into n sub intervals. The higher the value of n the more accurate our value of π will be, or should be.

Here is the source.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <omp.h>

double f(double x)
{
  double y;
  y = 4.0/(1.0+x*x);
  return(y);
}

int main()
{
  double cplusplus_internal_pi;
  double partial_pi;
  double pi_difference;
  double openmp_pi;
  double width;
  double x;
```

```
  int nthreads;
  int i;
  int j;
  int k;
  int n;

  char buff[100];
  time_t now = time (NULL);

  printf(" Program starts\n");

  strftime (buff, 100, "%Y-%m-%d %H:%M:%S.000", localtime (&now));
  printf (" %s\n", buff);

  time_t t1 = time( NULL );
  time_t t2 = time( NULL );

  nthreads = omp_get_max_threads();
  cplusplus_internal_pi = 4.0*atan(1.0);

  printf(" Maximum number of threads = %3d\n",nthreads);

  k = 1;
  for(;;)
  {
    n = 100000;
    omp_set_num_threads(k);
    printf(" ************************\n");
    printf("  Number of threads = %2d\n" , k );
    printf(" ************************\n" );

    for (j = 1;j<= 5;j++)
    {
      width = 1.0/n;
      partial_pi = 0.0;

#pragma omp parallel for private(x) shared(width) reduction(+:partial_pi)

      for(i = 1;i<= n;i++)
      {
        x = width*((double)(i)-0.5);
        partial_pi = partial_pi + f(x);
      }

      openmp_pi = width*partial_pi;
```

```
      t2 = time(NULL);
      printf( " %4.0f seconds  ",
        difftime( t2, t1 ) );
      pi_difference = fabs(openmp_pi-cplusplus_internal_pi);
      printf(" %10d  %20.16f  %20.16f %20.16f \n",n,openmp_pi,cplusplus_internal_pi,p
      t1=t2;
      n = n*10;

    }

    k = k+1;

    if (k>nthreads) goto finish;

  }

  finish:printf(" Program ends\n" );
  now = time (NULL);
  strftime (buff, 100, "%Y-%m-%d %H:%M:%S.000", localtime (&now));
  printf (" %s\n", buff);

  return(0);
}
```

Here is the Microsoft compiler output.

```
c:\document\c\c_notes>openmp04
 Program starts
 2016-04-04 16:03:53.000
 Maximum number of threads =   6
 ************************
  Number of threads =  1
 ************************
    0 seconds       100000    3.1415926535981615    3.1415926535897931   0.0000000000
    0 seconds      1000000    3.1415926535897643    3.1415926535897931   0.0000000000
    0 seconds     10000000    3.1415926535897309    3.1415926535897931   0.0000000000
    3 seconds    100000000    3.1415926535904264    3.1415926535897931   0.0000000000
   26 seconds   1000000000    3.1415926535899708    3.1415926535897931   0.0000000000
 ************************
  Number of threads =  2
 ************************
    0 seconds       100000    3.1415926535981464    3.1415926535897931   0.0000000000
    0 seconds      1000000    3.1415926535898993    3.1415926535897931   0.0000000000
    0 seconds     10000000    3.1415926535899228    3.1415926535897931   0.0000000000
    1 seconds    100000000    3.1415926535899099    3.1415926535897931   0.0000000000
```

```
  13 seconds   1000000000    3.1415926535899010    3.1415926535897931   0.000000000000
 ************************
  Number of threads =  3
 ************************
    0 seconds       100000    3.1415926535981384    3.1415926535897931   0.000000000008
    0 seconds      1000000    3.1415926535899041    3.1415926535897931   0.000000000000
    0 seconds     10000000    3.1415926535897274    3.1415926535897931   0.000000000000
    1 seconds    100000000    3.1415926535895702    3.1415926535897931   0.000000000000
    9 seconds   1000000000    3.1415926535899614    3.1415926535897931   0.000000000000
 ************************
  Number of threads =  4
 ************************
    0 seconds       100000    3.1415926535981269    3.1415926535897931   0.000000000008
    0 seconds      1000000    3.1415926535898757    3.1415926535897931   0.000000000000
    0 seconds     10000000    3.1415926535896697    3.1415926535897931   0.000000000000
    1 seconds    100000000    3.1415926535896825    3.1415926535897931   0.000000000000
    7 seconds   1000000000    3.1415926535898211    3.1415926535897931   0.000000000000
 ************************
  Number of threads =  5
 ************************
    0 seconds       100000    3.1415926535981269    3.1415926535897931   0.000000000008
    0 seconds      1000000    3.1415926535899112    3.1415926535897931   0.000000000000
    0 seconds     10000000    3.1415926535897860    3.1415926535897931   0.000000000000
    1 seconds    100000000    3.1415926535897367    3.1415926535897931   0.000000000000
    6 seconds   1000000000    3.1415926535895955    3.1415926535897931   0.000000000000
 ************************
  Number of threads =  6
 ************************
    0 seconds       100000    3.1415926535981318    3.1415926535897931   0.000000000008
    0 seconds      1000000    3.1415926535898833    3.1415926535897931   0.000000000000
    0 seconds     10000000    3.1415926535898127    3.1415926535897931   0.000000000000
    1 seconds    100000000    3.1415926535896457    3.1415926535897931   0.000000000000
    5 seconds   1000000000    3.1415926535896830    3.1415926535897931   0.000000000000
 Program ends
 2016-04-04 16:05:07.000
```

Here is the ggc output.

```
$ ./a.exe
 Program starts
 2016-04-04 16:08:42.000
 Maximum number of threads =   6
 ************************
  Number of threads =  1
 ************************
```

```
   0 seconds       100000    3.1415926535981615    3.1415926535897931   0.0000000000
   0 seconds      1000000    3.1415926535897643    3.1415926535897931   0.0000000000
   0 seconds     10000000    3.1415926535897309    3.1415926535897931   0.0000000000
   2 seconds    100000000    3.1415926535904264    3.1415926535897931   0.0000000000
  18 seconds   1000000000    3.1415926535899708    3.1415926535897931   0.0000000000
************************
 Number of threads =  2
************************
   0 seconds       100000    3.1415926535981464    3.1415926535897931   0.0000000000
   0 seconds      1000000    3.1415926535898993    3.1415926535897931   0.0000000000
   0 seconds     10000000    3.1415926535899228    3.1415926535897931   0.0000000000
   1 seconds    100000000    3.1415926535899099    3.1415926535897931   0.0000000000
   9 seconds   1000000000    3.1415926535899010    3.1415926535897931   0.0000000000
************************
 Number of threads =  3
************************
   0 seconds       100000    3.1415926535981384    3.1415926535897931   0.0000000000
   0 seconds      1000000    3.1415926535899041    3.1415926535897931   0.0000000000
   0 seconds     10000000    3.1415926535897274    3.1415926535897931   0.0000000000
   0 seconds    100000000    3.1415926535895702    3.1415926535897931   0.0000000000
   6 seconds   1000000000    3.1415926535899619    3.1415926535897931   0.0000000000
************************
 Number of threads =  4
************************
   0 seconds       100000    3.1415926535981269    3.1415926535897931   0.0000000000
   0 seconds      1000000    3.1415926535898753    3.1415926535897931   0.0000000000
   0 seconds     10000000    3.1415926535896697    3.1415926535897931   0.0000000000
   1 seconds    100000000    3.1415926535896825    3.1415926535897931   0.0000000000
   5 seconds   1000000000    3.1415926535898211    3.1415926535897931   0.0000000000
************************
 Number of threads =  5
************************
   0 seconds       100000    3.1415926535981269    3.1415926535897931   0.0000000000
   0 seconds      1000000    3.1415926535899112    3.1415926535897931   0.0000000000
   0 seconds     10000000    3.1415926535897860    3.1415926535897931   0.0000000000
   0 seconds    100000000    3.1415926535897367    3.1415926535897931   0.0000000000
   5 seconds   1000000000    3.1415926535895955    3.1415926535897931   0.0000000000
************************
 Number of threads =  6
************************
   0 seconds       100000    3.1415926535981318    3.1415926535897931   0.0000000000
   0 seconds      1000000    3.1415926535898828    3.1415926535897931   0.0000000000
   0 seconds     10000000    3.1415926535898127    3.1415926535897931   0.0000000000
   0 seconds    100000000    3.1415926535896457    3.1415926535897931   0.0000000000
   3 seconds   1000000000    3.1415926535896821    3.1415926535897931   0.0000000000
```

```
 Program ends
 2016-04-04 16:09:32.000
```

19.7 Example 5 - parallel array initialisation and summation

Here is the program source.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <omp.h>

#define n 1000000000

int main()
{
  double * x;
  double openmp_sum;
  double real_sum;
  int nthreads;
  int i;
  int j;
  int k;

  x = malloc( n * sizeof(double) );

  char buff[100];
  time_t now = time (NULL);

  printf(" Program starts\n");

  strftime (buff, 100, "%Y-%m-%d %H:%M:%S.000", localtime (&now));
  printf (" %s\n", buff);

  time_t t1 = time( NULL );
  time_t t2 = time( NULL );

  nthreads = omp_get_max_threads();
  printf(" Maximum number of threads = %3d \n",nthreads);

  k = 1;
```

```
  real_sum=n*1.0;

  for(i=0;i<nthreads;i++)
  {
    omp_set_num_threads(k);
    openmp_sum=0.0;
    printf(" ***********************\n");
    printf(" Number of threads = %3d\n"  ,k);
    printf(" ***********************\n");

#pragma omp parallel for private(j) shared(x)
    for(j = 0;j< n;j++)
    {
      x[j]=1.0;
    }

    t2 = time (NULL);
    printf( " Initialisation took %4.0f seconds \n",
      difftime( t2, t1 ) );
    t1=t2;

#pragma omp parallel for private(k) shared(x) reduction(+:openmp_sum)
    for(k = 0;k< n;k++)
    {
      openmp_sum+=x[k];
    }

    t2 = time (NULL);
    printf( " Summation took      %4.0f seconds \n",
      difftime( t2, t1 ) );
    t1=t2;
    printf(" Real sum   = %14.0f \n",real_sum );
    printf(" Openmp sum = %14.0f \n",openmp_sum);
    t1=t2;
    k=k+1;
  }

  now = time (NULL);
  printf(" Program ends\n");

  strftime (buff, 100, "%Y-%m-%d %H:%M:%S.000", localtime (&now));
  printf (" %s\n", buff);

  return(0);
}
```

Here is the output from the Microsoft compiler on an I7 system.

These systems have 4 real cores and through hyperthreading appear to have 8 cores.

```
Program starts
2016-04-06 11:14:23.000
Maximum number of threads =   8
**********************
Number of threads =   1
**********************
Initialisation took    5 seconds
Summation took         5 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   2
**********************
Initialisation took    1 seconds
Summation took         3 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   3
**********************
Initialisation took    1 seconds
Summation took         1 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   4
**********************
Initialisation took    1 seconds
Summation took         2 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   5
**********************
Initialisation took    1 seconds
Summation took         1 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   6
**********************
```

```
Initialisation took    1 seconds
Summation took         1 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   7
**********************
Initialisation took    1 seconds
Summation took         1 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
**********************
Number of threads =   8
**********************
Initialisation took    1 seconds
Summation took         0 seconds
Real sum   =     1000000000
Openmp sum =     1000000000
Program ends
2016-04-06 11:14:49.000
```

19.8 Problems

Compile and run the examples in this chapter.

Chapter 20

The preprocessor

20.1 Introduction

In this chapter we look at the preprocessor. Information in this chapter has been taken from the wikipedia entry.

The C preprocessor or cpp is the macro preprocessor for the C and C++ computer programming languages. The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.

In many C implementations, it is a separate program invoked by the compiler as the first part of translation.

20.2 Phases

Preprocessing is defined by the first four (of eight) phases of translation specified in the C Standard.

- 1.Trigraph replacement: The preprocessor replaces trigraph sequences with the characters they represent.
- 2.Line splicing: Physical source lines that are continued with escaped newline sequences are spliced to form logical lines.
- 3.Tokenization: The preprocessor breaks the result into preprocessing tokens and whitespace. It replaces comments with whitespace.
- 4.Macro expansion and directive handling: Preprocessing directive lines, including file inclusion and conditional compilation, are executed. The preprocessor simultaneously expands macros and, in the 1999 version of the C standard,[clarification needed] handles _Pragma operators.

20.3 Including files

One of the most common uses of the preprocessor is to include another file. With the

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

The preprocessor replaces the line #include ¡stdio.h¿ with the text of the file 'stdio.h', which declares the printf() function among other things.

This can also be written using double quotes, e.g. #include "stdio.h". If the filename is enclosed within angle brackets, the file is searched for in the standard compiler include paths. If the filename is enclosed within double quotes, the search path is expanded to include the current source directory. C compilers and programming environments all have a facility which allows the programmer to define where include files can be found. This can be introduced through a command line flag, which can be parameterized using a makefile, so that a different set of include files can be swapped in for different operating systems, for instance.

By convention, include files are given a .h extension, and files not included by others are given a .c extension. However, there is no requirement that this be observed. Files with a .def extension may denote files designed to be included multiple times, each time expanding the same repetitive content; #include "icon.xbm" is likely to refer to an XBM image file (which is at the same time a C source file).

#include often compels the use of #include guards or #pragma once to prevent double inclusion.

20.4 Conditional compilation

The if-else directives #if, #ifdef, #ifndef, #else, #elif and #endif can be used for conditional compilation.

```
#if VERBOSE >= 2
  print("trace message");
#endif
```

Most compilers targeting Microsoft Windows implicitly define _WIN32. This allows code, including preprocessor commands, to compile only when targeting Windows systems. A few compilers define WIN32 instead. For such compilers that do not implicitly define the _WIN32 macro, it can be specified on the compiler's command line, using -D_WIN32.

```
#ifdef __unix__
#/* __unix__ is usually defined by compilers targeting Unix systems */
# include <unistd.h>
#elif defined _WIN32
#/* _Win32 is usually defined by compilers targeting 32 or 64 bit Windows systems */
# include <windows.h>
#endif
```

The example code tests if a macro

```
__unix__
\end{verbatim>


is defined. If it is, the file <unistd.h> is then included. Otherwise, it tests if a

\begin{verbatim}
macro _WIN32
```

is defined instead. If it is, the file ¡windows.h¿ is then included.

A more complex #if example can use operators, for example something like:

```
#if !(defined __LP64__ || defined __LLP64__) || defined _WIN32 && !defined _WIN64
// we are compiling for a 32-bit system
#else
// we are compiling for a 64-bit system
#endif
```

Translation can also be caused to fail by using the #error directive:

```
#if RUBY_VERSION == 190
# error 1.9.0 not supported
#endif
```

20.5 Macro definition and expansion

There are two types of macros, object-like and function-like. Object-like macros do not take parameters; function-like macros do (although the list of parameters may be empty). The generic syntax for declaring an identifier as a macro of each type is, respectively:

```
#define <identifier> <replacement token list>
#// object-like macro
#define <identifier>(<parameter list>) <replacement token list>
#// function-like macro, note parameters
```

The function-like macro declaration must not have any whitespace between the identifier and the first, opening, parenthesis. If whitespace is present, the macro will be interpreted as object-like with everything starting from the first parenthesis added to the token list.

A macro definition can be removed with #undef:

```
#undef <identifier>
#// delete the macro
```

Whenever the identifier appears in the source code it is replaced with the replacement token list, which can be empty. For an identifier declared to be a function-like macro, it is only replaced when the following token is also a left parenthesis that begins the argument list of the macro invocation. The exact procedure followed for expansion of function-like macros with arguments is subtle.

Object-like macros were conventionally used as part of good programming practice to create symbolic names for constants, e.g.,

```
#define PI 3.14159
```

instead of hard-coding the numbers throughout the code. An alternative in both C and C++, especially in situations in which a pointer to the number is required, is to apply the const qualifier to a global variable. This causes the value to be stored in memory, instead of being substituted by the preprocessor.

An example of a function-like macro is:

```
#define RADTODEG(x) ((x) * 57.29578)
```

This defines a radians-to-degrees conversion which can be inserted in the code where required, i.e., RADTODEG(34). This is expanded in-place, so that repeated multiplication by the constant is not shown throughout the code. The macro here is written as all uppercase to emphasize that it is a macro, not a compiled function.

The second x is enclosed in its own pair of parentheses to avoid the possibility of incorrect order of operations when it is an expression instead of a single value. For example, the expression RADTODEG(r + 1) expands correctly as ((r + 1) * 57.29578); without parentheses, (r + 1 * 57.29578) gives precedence to the multiplication.

Similarly, the outer pair of parentheses maintain correct order of operation. For example, 1 / RADTODEG(r) expands to 1 / ((r) * 57.29578); without parentheses, 1 / (r) * 57.29578 gives precedence to the division.

20.6 Special macros and directives

Certain symbols are required to be defined by an implementation during preprocessing. These include __FILE__ and __LINE__, predefined by the preprocessor itself, which expand into the current file and line number. For instance the following:

```
// debugging macros so we can pin down message origin at a glance
#define WHERESTR  "[file %s, line %d]: "
#define WHEREARG  __FILE__, __LINE__
#define DEBUGPRINT2(...)       fprintf(stderr, __VA_ARGS__)
#define DEBUGPRINT(_fmt, ...)  DEBUGPRINT2(WHERESTR _fmt, WHEREARG, __VA_ARGS__)
//...

  DEBUGPRINT("hey, x=%d\n", x);
```

prints the value of x, preceded by the file and line number to the error stream, allowing quick access to which line the message was produced on. Note that the WHERESTR argument is concatenated with the string following it. The values of __FILE__ and __LINE__ can be manipulated with the #line directive. The #line directive determines the line number and the file name of the line below. E.g.:

```
#line 314 "pi.c"
printf("line=%d file=%s\n", __LINE__, __FILE__);
```

generates the printf function:

```
printf("line=%d file=%s\n", 314, "pi.c");
```

Source code debuggers refer also to the source position defined with

```
__FILE__
and
__LINE__
```

This allows source code debugging, when C is used as target language of a compiler, for a totally different language. The first C Standard specified that the macro

```
__STDC__
```

be defined to 1 if the implementation conforms to the ISO Standard and 0 otherwise, and the macro

```
__STDC_VERSION__
```

defined as a numeric literal specifying the version of the Standard supported by the implementation.

Standard C++ compilers support the

```
__cplusplus
```

macro. Compilers running in non-standard mode must not set these macros, or must define others to signal the differences.

Other Standard macros include

```
__DATE__ -  the current date
__TIME__ - the current time
```

The second edition of the C Standard, C99, added support for

```
__func__
```

which contains the name of the function definition within which it is contained, but because the preprocessor is agnostic to the grammar of C, this must be done in the compiler itself using a variable local to the function.

Macros that can take a varying number of arguments (variadic macros) are not allowed in C89, but were introduced by a number of compilers and standardised in C99. Variadic macros are particularly useful when writing wrappers to functions taking a variable number of parameters, such as printf, for example when logging warnings and errors.

One little-known usage pattern of the C preprocessor is known as X-Macros.[2][3][4] An X-Macro is a header file. Commonly these use the extension ".def" instead of the traditional ".h". This file contains a list of similar macro calls, which can be referred to as "component macros". The include file is then referenced repeatedly.

Many compilers define additional, non-standard macros, although these are often poorly documented. A common reference for these macros is the Pre-defined C/C++ Compiler Macros project, which lists "various pre-defined compiler macros that can be used to identify standards, compilers, operating systems, hardware architectures, and even basic run-time libraries at compile-time".

20.7 Token stringification

The # operator (known as the "Stringification Operator") converts a token into a string, escaping any quotes or backslashes appropriately.

```
#define str(s) #s

str(p = "foo\n";) // outputs "p = \"foo\\n\";"
str(\n)           // outputs "\n"
```

If you want to stringify the expansion of a macro argument, you have to use two levels of macros:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4

str (foo)  // outputs "foo"
xstr (foo) // outputs "4"
```

You cannot combine a macro argument with additional text and stringify it all together. You can however write a series of adjacent string constants and stringified arguments: the C compiler will then combine all the adjacent string constants into one long string.

20.8 Token concatenation

The ## operator (known as the "Token Pasting Operator") concatenates two tokens into one token.

```
#define DECLARE_STRUCT_TYPE(name) typedef struct name##_s name##_t

DECLARE_STRUCT_TYPE(g_object); // Outputs: typedef struct g_object_s g_object_t;
```

20.9 Implementations

20.9.1 Compiler-specific preprocessor features

The #pragma directive is a compiler-specific directive, which compiler vendors may use for their own purposes. For instance, a #pragma is often used to allow suppression of specific error messages, manage heap and stack debugging and so on. A compiler with support for the OpenMP parallelization library can automatically parallelize a for loop with #pragma omp parallel for.

C99 introduced a few standard #pragma directives, taking the form #pragma STDC ..., which are used to control the floating-point implementation. Many implementations do not support trigraphs or do not replace them by default. Many implementations (including, e.g., the C compilers by GNU, Intel, Microsoft and IBM) provide a non-standard directive to print out a warning message in the output, but not stop the compilation process. A typical use is to warn about the usage of some old code, which is now deprecated and only included for compatibility reasons, e.g.:

(GNU, Intel and IBM)

```
#warning "Do not use ABC, which is deprecated. Use XYZ instead."
```

(Microsoft)

```
#pragma message("Do not use ABC, which is deprecated. Use XYZ instead.")
```

Chapter 21

Make

21.1 Introduction

In this chapter we look at make. Information in this chapter has been taken from the wikipedia entry.

In software development, Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix.

Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

21.2 Origin

There are now a number of dependency-tracking build utilities, but Make is one of the most widespread, primarily due to its inclusion in Unix, starting with the PWB/UNIX 1.0, which featured a variety of tools targeting software development tasks. It was originally created by Stuart Feldman in April 1976 at Bell Labs.[1] Feldman received the 2003 ACM Software System Award for the authoring of this widespread tool.

Before Make's introduction, the Unix build system most commonly consisted of operating system dependent "make" and "install" shell scripts accompanying their program's source. Being able to combine the commands for the different targets into a single file and being able to abstract out dependency tracking and archive handling was an important step in the direction of modern build environments.

21.3 Derivatives

Make has gone through a number of rewrites, including a number of from-scratch variants which used the same file format and basic algorithmic principles and also provided a number of their own non-standard enhancements. Some of them are:

- SunPro make is a rewrite of the UNIX make program that appeared in 1986 with SunOS-3.2. With SunOS-3.2, SunPro make was delivered as optional program; with SunOS-4.0, SunPro make was made the default make program.[3] In December 2006, it was made OpenSource as it is needed to compile OpenSolaris.

- dmake or Distributed Make that comes with Sun Solaris Studio as its default make, but not the default one on the Solaris Operating System (SunOS). It is required to build Apache OpenOffice.

- BSD Make, which is derived from Adam de Boor's work on a version of Make capable of building targets in parallel, and survives with varying degrees of modification in FreeBSD, NetBSD and OpenBSD. Distinctively, it has conditionals and iterative loops which are applied at the parsing stage and may be used to conditionally and programmatically construct the makefile, including generation of targets at runtime.

- GNU Make (short gmake) is the standard implementation of make for Linux and OS X. It provides several extensions over the original make, such as conditionals. It also provides many built-in functions which can be used to eliminate the need for shell-scripting in the makefile rules as well as to manipulate the variables set and used in the makefile. For example, the foreach function sets a variable to the list of all files in a given directory. GNU Make has been required for building gcc since version 3.4.[8] It is required for building the Linux kernel.

- Glenn Fowler's nmake is unrelated to the Microsoft program of the same name. Its input is similar to make, but not compatible. This program provides shortcuts and built-in features, which according to its developers reduces the size of makefiles by a factor of 10.

- Microsoft nmake, a command-line tool which normally is part of Visual Studio. It supports preprocessor directives such as includes and conditional expressions which use variables set on the command-line or within the makefiles. Inference rules differ from make; for example they can include search paths. The make tool supplied with Embarcadero products has a command-line option that "Causes MAKE to mimic Microsoft's NMAKE."

- Mk replaced Make in Research Unix, starting from version 9. A redesign of the original tool by Bell Labs programmer Andrew G. Hume, it features a different syntax. Mk became the standard build tool in Plan 9, Bell Labs' intended successor to Unix.

POSIX includes standardization of the basic features and operation of the Make utility, and is implemented with varying degrees of completeness in Unix-based versions of Make. In general, simple makefiles may be used between various versions of Make with reasonable success. GNU Make, BSD Make and Makepp can be configured[citation needed] to look first for files named "GNUmakefile", "BSDmakefile" and "Makeppfile" respectively, which allows one to put makefiles which use implementation-defined behaviour in separate locations.

21.4 Behaviour

Make is typically used to build executable programs and libraries from source code. Generally though, Make is applicable to any process that involves executing arbitrary commands to transform a source file to a target result. For example, Make could be used to detect a change made to an image file (the source) and the transformation actions might be to convert the file to some specific format, copy the result into a content management system, and then send e-mail to a predefined set of users indicating that the above actions were performed.

Make is invoked with a list of target file names to build as command-line arguments:

```
make [TARGET ...]
```

Without arguments, Make builds the first target that appears in its makefile, which is traditionally a symbolic "phony" target named all.

Make decides whether a target needs to be regenerated by comparing file modification times.[22] This solves the problem of avoiding the building of files which are already up to date, but it fails when a file changes but its modification time stays in the past. Such changes could be caused by restoring an older version of a source file, or when a network file system is a source of files and its clock or timezone is not synchronized with the machine running Make. The user must handle this situation by forcing a complete build. Conversely, if a source file's modification time is in the future, it triggers unnecessary rebuilding, which may inconvenience users.

21.5 Makefiles

Make searches the current directory for the makefile to use, e.g. GNU make searches files in order for a file named one of GNUmakefile, makefile, Makefile and then runs the specified (or default) target(s) from (only) that file.

The makefile language is similar to declarative programming. This class of language, in which necessary end conditions are described but the order in which actions are to be taken is not important, is sometimes confusing to programmers used to imperative programming.

One problem in build automation is the tailoring of a build process to a given platform. For instance, the compiler used on one platform might not accept the same options as the one used on another. This is not well handled by Make. This problem is typically handled by generating platform specific build instructions, which in turn are processed by Make. Common tools for this process are Autoconf and CMake.

21.6 Rules

A makefile consists of rules. Each rule begins with a textual dependency line which defines a target followed by a colon (:) and optionally an enumeration of components

(files or other targets) on which the target depends. The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon). It is common to refer to components as prerequisites of the target.

```
target [target ...]: [component ...]
Tab [command 1]
        .
        .
        .
Tab [command n]
```

Usually each rule has a single unique target, rather than multiple targets.

For example, a C .o object file is created from .c files, so .c files come first (i.e. specific object file target depends on a C source file and header files). Because Make itself does not understand, recognize or distinguish different kinds of files, this opens up a possibility for human error. A forgotten or an extra dependency may not be immediately obvious and may result in subtle bugs in the generated software. It is possible to write makefiles which generate these dependencies by calling third-party tools, and some makefile generators, such as the Automake toolchain provided by the GNU Project, can do so automatically.

Each dependency line may be followed by a series of TAB indented command lines which define how to transform the components (usually source files) into the target (usually the "output"). If any of the prerequisites has a more recent modification time than the target, the command lines are run. The GNU Make documentation refers to the commands associated with a rule as a "recipe".

The first command may appear on the same line after the prerequisites, separated by a semicolon,

```
targets : prerequisites ; command
```

for example,

```
hello: ; @echo "hello"
```

Make can decide where to start through topological sorting.

Each command line must begin with a tab character to be recognized as a command. The tab is a whitespace character, but the space character does not have the same special meaning. This is problematic, since there may be no visual difference between a tab and a series of space characters. This aspect of the syntax of makefiles is often subject to criticism.

However, the GNU Make since version 3.82 allows to choose any symbol (one character) as the recipe prefix using the .RECIPEPREFIX special variable, for example:

```
.RECIPEPREFIX := :
all:
:@echo "recipe prefix symbol is set to '$(.RECIPEPREFIX)'"
```

Each command is executed by a separate shell or command-line interpreter instance. Since operating systems use different command-line interpreters this can lead to unportable makefiles. For instance, GNU Make by default executes commands with /bin/sh, where Unix commands like cp are normally used. In contrast to that, Microsoft's nmake executes commands with cmd.exe where batch commands like copy are available but not necessarily cp.

A rule may have no command lines defined. The dependency line can consist solely of components that refer to targets, for example:

```
realclean: clean distclean
```

The command lines of a rule are usually arranged so that they generate the target. An example: if "file.html" is newer, it is converted to text. The contents of the makefile:

```
file.txt: file.html
```

lynx -dump file.html ¿ file.txt

The above rule would be triggered when Make updates "file.txt". In the following invocation, Make would typically use this rule to update the "file.txt" target if "file.html" were newer.

```
make file.txt
```

Command lines can have one or more of the following three prefixes: a hyphen-minus (-), specifying that errors are ignored an at sign (@), specifying that the command is not printed to standard output before it is executed a plus sign (+), the command is executed even if Make is invoked in a "do not execute" mode

Ignoring errors and silencing echo can alternatively be obtained via the special targets ".IGNORE" and ".SILENT".[27]

Microsoft's NMAKE has predefined rules that can be omitted from these makefiles, e.g. "c.obj (CC)(CFLAGS)".

21.7 Macros

A makefile can contain definitions of macros. Macros are usually referred to as variables when they hold simple string definitions, like "CC=clang". Macros in makefiles may be overridden in the command-line arguments passed to the Make utility. Environment variables are also available as macros.

Macros allow users to specify the programs invoked and other custom behaviour during the build process. For example, the macro "CC" is frequently used in makefiles to refer to the location of a C compiler, and the user may wish to specify a particular compiler to use.

New macros (or simple "variables") are traditionally defined using capital letters:

```
MACRO = definition
```

A macro is used by expanding it. Traditionally this is done by enclosing its name inside ().*Anequivalentformusescurlybracesratherthanparenthesis*, *i.e.*, which is the style used in the BSDs.

```
NEW_MACRO = $(MACRO)-$(MACRO2)
```

Macros can be composed of shell commands by using the command substitution operator, denoted by backticks (‘).

```
YYYYMMDD  = ‘ date ‘
```

The content of the definition is stored ”as is”. Lazy evaluation is used, meaning that macros are normally expanded only when their expansions are actually required, such as when used in the command lines of a rule. An extended example:

```
PACKAGE   = package
VERSION   = ‘ date +"%Y.%m%d" ‘
ARCHIVE   = $(PACKAGE)-$(VERSION)

dist:
        #  Notice that only now macros are expanded for shell to interpret:
        #      tar -cf package-‘date +"%Y%m%d"‘.tar

        tar -cf $(ARCHIVE).tar .
```

The generic syntax for overriding macros on the command line is:

```
make MACRO="value" [MACRO="value" ...] TARGET [TARGET ...]
```

Makefiles can access any of a number of predefined internal macros, with ’?’ and ’@’ being the most common.

```
target: component1 component2
        # contains those components, which need attention (i.e. they ARE YOUNGER th
        echo $?
        # evaluates to current TARGET name from among those left of the colon.
        echo $@
```

21.8 Suffix rules

Suffix rules have ”targets” with names in the form .FROM.TO and are used to launch actions based on file extension. In the command lines of suffix rules, POSIX specifies that the internal macro

```
$<
```

refers to the first prerequisite and

```
$@
```

refers to the target. In this example, which converts any HTML file into text, the shell redirection token > is part of the command line whereas

```
$<
```

is a macro referring to the HTML file:

```
.SUFFIXES: .txt .html

# From .html to .txt
.html.txt:
        lynx -dump $<   >   $@
```

When called from the command line, the above example expands.

```
$ make -n file.txt
lynx -dump file.html > file.txt
```

21.9 Pattern rules

Suffix rules cannot have any prerequisites of their own. If they have any, they are treated as normal files with unusual names, not as suffix rules. GNU make supports suffix rules for compatibility with old makefiles but otherwise encourages usage of pattern rules.

A pattern rule looks like an ordinary rule, except that its target contains exactly one character '%'. The target is considered a pattern for matching file names: the '%' can match any substring of zero or more characters, while other characters match only themselves. The prerequisites likewise use '%' to show how their names relate to the target name.

The above example of a suffix rule would look like the following pattern rule,

```
# From %.html to %.txt
%.txt : %.html
        lynx -dump $< > $@
```

21.10 Other elements

Single-line comments are started with the hash symbol (#).

Some directives in makefiles can include other makefiles.

Line continuation is indicated with a backslash
character at the end of a line.

```
target: component \
        component
Tab command ;          \
Tab command |          \
Tab piped-command
```

21.11 Example makefiles

Makefiles are traditionally used for compiling code (*.c, *.cc, *.C, etc.), but they can also be used for providing commands to automate common tasks. One such makefile is called from the command line:

```
make                        # Without argument runs first TARGET
make help                   # Show available TARGETS
make dist                   # Make a release archive from current dir
```

The makefile:

```
PACKAGE  = package
VERSION  = ‘ date "+%Y.%m%d%" ‘
RELEASE_DIR  = ..
RELEASE_FILE = $(PACKAGE)-$(VERSION)

# Notice that the variable LOGNAME comes from the environment in
# POSIX shells.
#
# target: all - Default target. Does nothing.
all:
echo "Hello $(LOGNAME), nothing to do by default"
        # sometimes: echo "Hello ${LOGNAME}, nothing to do by default"
echo "Try ’make help’"

# target: help - Display callable targets.
help:
egrep "^# target:" [Mm]akefile

# target: list - List source files
list:
# Won’t work. Each command is in separate shell
cd src
ls

# Correct, continuation of the same shell
cd src; \
ls
```

```
# target: dist - Make a release.
dist:
tar -cf  $(RELEASE_DIR)/$(RELEASE_FILE) && \
gzip -9  $(RELEASE_DIR)/$(RELEASE_FILE).tar
```

Below is a very simple makefile that by default (the "all" rule is listed first) compiles a source file called "helloworld.c" using the system's C compiler and also provides a "clean"target to remove the generated files if the user desires to start over. The

```
$@
and
$<
```

are two of the so-called internal macros (also known as automatic variables) and stand for the target name and "implicit" source, respectively. In the example below,

```
$^
```

expands to a space delimited list of the prerequisites. There are a number of other internal macros.

```
CFLAGS ?= -g

all: helloworld

helloworld: helloworld.o
# Commands start with TAB not spaces
$(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
$(CC) $(CFLAGS) -c -o $@ $<

clean: FRC
rm -f helloworld helloworld.o

# This pseudo target causes all targets that depend on FRC
# to be remade even in case a file with the name of the target exists.
# This works with any make implementation under the assumption that
# there is no file FRC in the current directory.
FRC:
```

Many systems come with predefined Make rules and macros to specify common tasks such as compilation based on file suffix. This lets users omit the actual (often unportable) instructions of how to generate the target from the source(s). On such a system the above makefile could be modified as follows:

```
    all: helloworld

    helloworld: helloworld.o
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

    clean: FRC
rm -f helloworld helloworld.o

    # This is an explicit suffix rule. It may be omitted on systems
    # that handle simple rules like this automatically.
    .c.o:
$(CC) $(CFLAGS) -c $<

    FRC:
    .SUFFIXES: .c
```

That "helloworld.o" depends on "helloworld.c" is now automatically handled by Make. In such a simple example as the one illustrated here this hardly matters, but the real power of suffix rules becomes evident when the number of source files in a software project starts to grow. One only has to write a rule for the linking step and declare the object files as prerequisites. Make will then implicitly determine how to make all the object files and look for changes in all the source files.

Simple suffix rules work well as long as the source files do not depend on each other and on other files such as header files. Another route to simplify the build process is to use so-called pattern matching rules that can be combined with compiler-assisted dependency generation. As a final example requiring the gcc compiler and GNU Make, here is a generic makefile that compiles all C files in a folder to the corresponding object files and then links them to the final executable. Before compilation takes place, dependencies are gathered in makefile-friendly format into a hidden file ".depend" that is then included to the makefile. Portable programs ought to avoid constructs used below.

```
# Generic GNUMakefile

# Just a snippet to stop executing under other make(1) commands
# that won't understand these lines
ifneq (,)
This makefile requires GNU Make.
endif

PROGRAM = foo
C_FILES := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(C_FILES))
CC = cc
CFLAGS = -Wall -pedantic
```

```
LDFLAGS =
LDLIBS = -lm

all: $(PROGRAM)

$(PROGRAM): .depend $(OBJS)
$(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM) $(LDLIBS)

depend: .depend

.depend: cmd = gcc -MM -MF depend $(var); cat depend >> .depend;
.depend:
@echo "Generating dependencies..."
@$(foreach var, $(C_FILES), $(cmd))
@rm -f depend

-include .depend

# These are the pattern matching rules. In addition to the automatic
# variables used here, the variable $* that matches whatever % stands for
# can be useful in special cases.
%.o: %.c
$(CC) $(CFLAGS) -c $< -o $@

%: %.c
$(CC) $(CFLAGS) -o $@ $<

clean:
rm -f .depend $(OBJS)

.PHONY: clean depend
```

Chapter 22

Terms

22.1 C terminology

The following is taken from the C standard.

- access (execution-time action) to read or modify the value of an object
- alignment requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address
- argument actual argument actual parameter (deprecated) expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation
- behavior external appearance or action
- implementation-defined behavior unspecified behavior where each implementation documents how the choice is made
- locale-specific behavior behavior that depends on local conventions of nationality, culture, and language that each implementation documents
- undefined behavior behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements
- unspecified behavior use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance
- bit unit of data storage in the execution environment large enough to hold an object that may have one of two values
- byte addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

- character ¡abstract¿ member of a set of elements used for the organization, control, or representation of data
- character single-byte character ¡C¿ bit representation that fits in a byte
- multibyte character sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment
- wide character bit representation that fits in an object of type wchar_t, capable of representing any character in the current locale
- constraint restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted
- correctly rounded result representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision
- diagnostic message message belonging to an implementation-defined subset of the implementation message output
- forward reference reference to a later subclause of this International Standard that contains additional information relevant to this subclause
- implementation particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment
- implementation limit restriction imposed upon programs by the implementation
- memory location either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width
- object region of data storage in the execution environment, the contents of which can represent values
- parameter formal parameter formal argument (deprecated) object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition
- recommended practice specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations
- runtime-constraint requirement on a program when calling a library function

- value precise meaning of the contents of an object when interpreted as having a specific type
- implementation-defined value unspecified value where each implementation documents how the choice is made
- indeterminate value either an unspecified value or a trap representation
- unspecified value valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance
- trap representation an object representation that need not represent a value of the object type
- perform a trap interrupt execution of the program such that no further operations are performed

Chapter 23

More syntax

23.1 Keywords

Here are the C keywords in C11.

```
alignof
auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
inline
int
long
register
restrict
return
short
signed
sizeof
static
struct
```

```
switch
typedef
union
unsigned
void
volatile
while
_Alignas
_Atomic
_Bool
_Complex
_Generic
_Imaginary
_Noreturn
_Static_assert
_Thread_local
```

23.1.1 C89

```
alignof
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
```

23.1.2 C99

```
_Bool
_Complex
_Imaginary
inline
restrict
```

23.1.3 C11

```
_Alignas
_Alignof
_Atomic
_Generic
_Imaginary
_Noreturn
_Static_assert
_Thread_local
```

23.2 Identifiers

```
Syntax
identifier:
  identifier-nondigit
  identifier identifier-nondigit
  identifier digit
identifier-nondigit:
  nondigit
  universal-character-name
  other implementation-defined characters

nondigit: one of

_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
```

```
N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9
```

23.2.1 Semantics

An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.

23.3 Standard headers

The standard headers are

- assert.h
- complex.h
- ctype.h
- errno.h
- fenv.h
- float.h
- inttypes.h
- iso646.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stdalign.h
- stdarg.h
- stdatomic.h
- stdbool.h

- stddef.h
- stdint.h
- stdio.h
- stdlib.h
- string.h
- tgmath.h
- threads.h
- time.h
- uchar.h
- wchar.h
- wctype.h

Chapter 24

Background introduction to parallel programming

"'Can you do addition?' the White Queen asked. 'What's one and one and one and one and one and one and one and one and one?'

'I don't know' said Alice. 'I lost count.'

'She can't do addition,' the Red Queen interrupted."

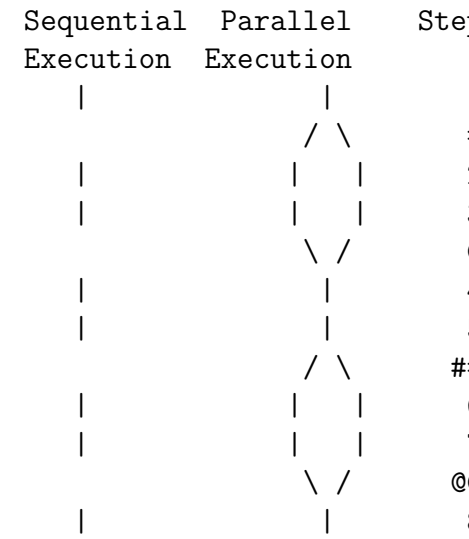
Lewis Carroll, Through the Looking Glass and What Alice Found There.

Aims

The aims of this chapter is to provide a short introduction to parallel programming.

24.1 Introduction

Parallel programming involves breaking a program down into parts that can be executed concurrently. Here is a simple diagram to illustrate the idea.



On the left hand side we have a sequential program and this steps through linearly from beginning to end. The right hand side has the same program that has been partially parallelised. There are two parallel regions and the work here is now shared between two processes or threads. At each parallel part of the program we have the following

```
                      Parallel    Parallel
                      Region 1    Region 2
Set up cost           Step #      Step ##
Parallel section      Steps 2,3   Steps 6,7
Synchronisation cost  Step @      Step @@
```

The theory is that the overall run time of the program will have been reduced or we will have been able to solve a larger problem by parallelising our code. In the above example we have divided the work between two processes or threads. Here are some details of a range of processors which support multiple cores. Visit the AMD and Intel sites for up to date information.

```
  Processor              Cores  Hyper
                                Threading
  AMD Phenom II X6         6
  Intel Core i7 920        4    * 2
  Intel Core i7 2600K      4    * 2
  AMD Opteron Shanghai     4
        Istanbul           6
        Magny Cours        8
        Magny Cours       12
  Intel E5-2697           12    * 2
```

Intel introduced hyperthreading technology in 2002. For each physical processor core the Intel chip has the operating system can see or address two virtual or logical cores, and can share the workload between them when possible. See the Wikipedia entry for more information.

`http://en.wikipedia.org/wiki/Hyper-threading`

There are several ways of doing parallel programming, and this chapter will look at three ways of doing this in Fortran. There are a common set of concepts and terminology that are useful to know about, whichever method we use, and we will cover these first.

24.2 Parallel computing classification

Parallel computing is often classified by the way the hardware supports parallelism. Two of the most common are:

- multi-processor and multi-core computers having multiple processing elements within a single system
- clusters or grids with multiple computers connected to work together.

Modern large systems are increasingly hybrids of the two above.

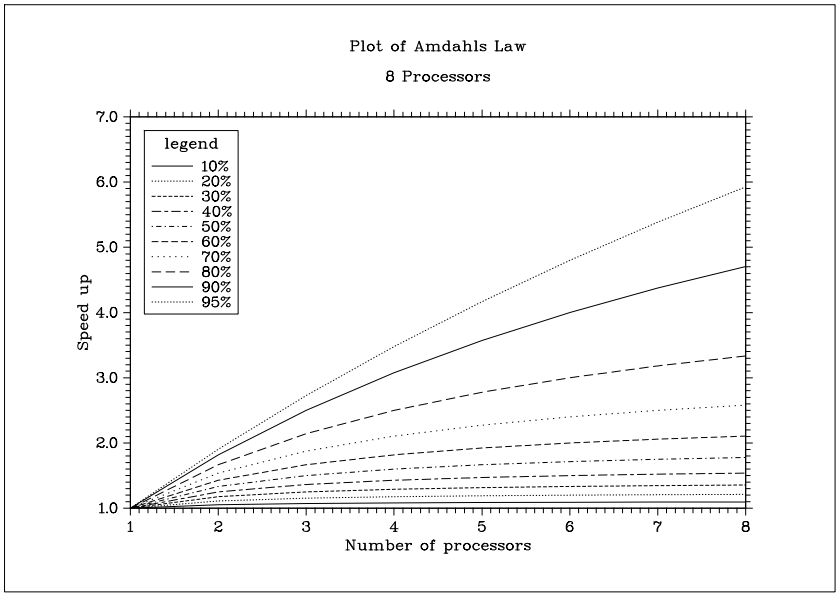
24.3 Amdahl's Law

Amdahl's law is a simple equation for the speedup of a program when parallelised. It assumes that the problem size remains the same when parallelised. In the equation below

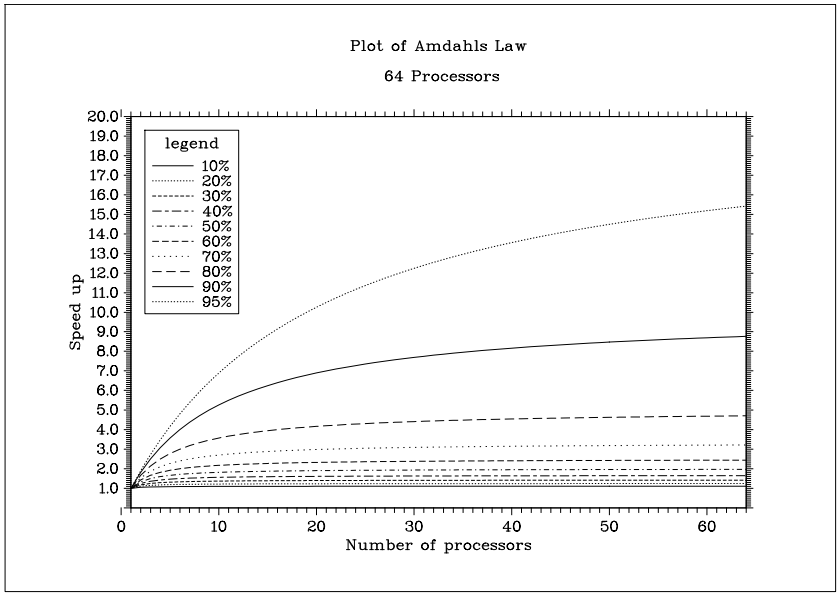
- P is the proportion of the program that can be parallelised
- (1-P) is the serial proportion
- N is the number of processors
- speedup = 1 / ((1-P) + P/N)

We have included a couple of graphs to illustrate the above. We have written programs that use the dislin graphics library to do the plots. More information on these programs can be found in chapter 35, where we have a look at third party numeric and graphics libraries.

24.3.1 Amdahl's Law graph 1 - 8 processors or cores



24.3.2 Amdahl's Law graph 2 - 64 processors or cores



24.4 Gustafson's law

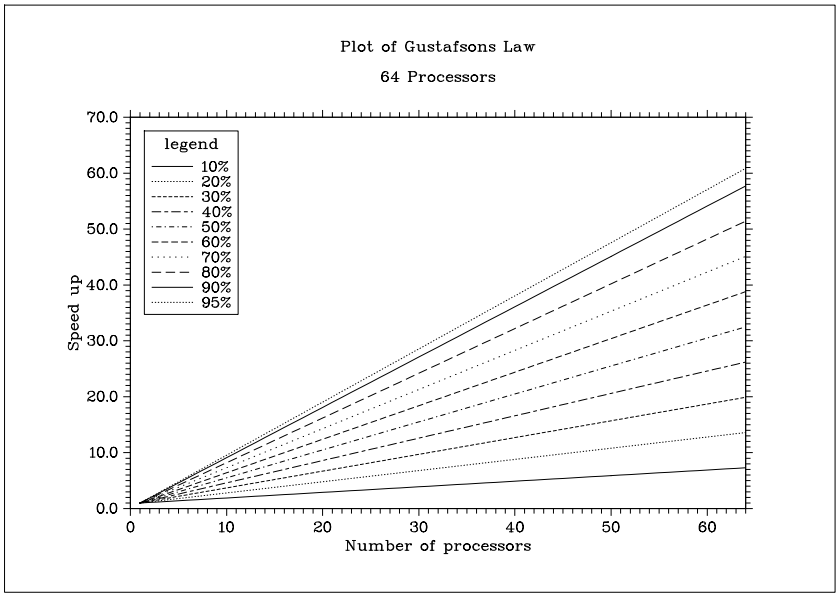
Gustafson's Law is often seen as a contradiction of Amdahl's Law. Simplistically it states that programmers solve larger problems when parallelising programs.

The equation for Gustafson's Law is given below.

- N is the number of processors
- Serial is the proportion that remains serial
- Speedup(N) = N - Serial * (N - 1)

We have again included a graph to illustrate the above.

24.4.1 Gustafson's Law graph 1 - 64 processors or cores



24.5 Memory access

Memory access times fall into two main categories that are of interest in parallel computing

- uma - uniform memory access. Each element of main memory can be accessed with the same latency and bandwidth. Multi-processor and multi-core computers typically have this behaviour.
- numa - non uniform memory access. Distributed memory systems have non-uniform memory access. Clusters or grids with multiple computers connected to work together have this behaviour.

24.6 Cache

Modern processors have a memory hierarchy. They typically have two or more levels:

- main memory
- cpu memory

and there is a speed and cost link. Main memory is cheap and relatively slow in comparison to the cpu memory.

The cpu memory or cache is used to reduce the effective access time to memory. If the information that the program requires is in the cpu cache then the average latency of memory accesses will be closer to the cache latency than to the latency of main memory. Getting high performance from a computer normally means writing

cache friendly programs. This means that the data and instructions that the program needs are already in the cache and don't need to be accessed from the much slower main memory.

In a multi-core and multi-cpu system each core and cpu will have their own memory or cache. This introduces the problem of cache coherency - i.e. the consistency of data stored in local caches compared to the data in the common shared memory. This problem must obviously be addressed when doing parallel programming.

24.7 Bandwidth and latency

Bandwidth is the rate at which data can be transferred. Latency is the start up time for a data transfer. We normally want a high bandwidth and low latency. Table 24.1 looks at some figures for several interconnects.

Table 24.1: Bandwidth and latency

	MPI bandwidth or theoretical maximum GB/s	latency μs
Gigabit ethernet	0.125	≈ 100
Infiniband	1.3	4.0
Myrinet 10-G	1.2	2.1
Quadrics QsNet II	0.9	2.7
Cray SeStar2	2.1	4.5

24.8 Flynn's taxonomy

Flynn's taxonomy is an old, but still widely used, classification scheme for computer architecture.

- Single Instruction, Single Data stream (SISD) A sequential computer which exploits no parallelism in either the instruction or data streams. Term rarely used.
- Single Instruction, Multiple Data streams (SIMD) A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelised. For example, an array processor or GPU.
- Multiple Instruction, Single Data stream (MISD) Multiple instructions operate on a single data stream. Term rarely used.
- Multiple Instruction, Multiple Data streams (MIMD) Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either

exploiting a single shared memory space or a distributed memory space. Essentially separate computers working together to solve a problem.

We also have the term

- Single Program Multiple Data - An identical program executes on a MIMD computer system. Conditional statements in the code mean that different parts of the program execute on each system.

24.9 Consistency models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

24.10 Threads and threading

In computing a thread of execution is often regarded as the smallest unit of processing that can be scheduled by an operating system. The implementation of threads and processes generally varies with operating system.

24.11 Threads and processes

From a strict computer science point of view threads and processes are different. However when looking simply at parallel programming the term can often be used interchangeably. In the following we use the term thread.

24.12 Data dependencies

A data dependency is when one statement in a program depends on a calculation from a previous statement. This will obviously hinder parallelism.

24.13 Race conditions

Race conditions can occur in programs when separate threads depend on a shared state or variable.

24.14 Mutual exclusion - mutex

A mutex is a programming construct that is used to allow multiple threads to share a resource. The sharing is not simultaneous. One thread will acquire the mutex and then lock the other threads from accessing it until it has completed.

24.15 Monitors

In concurrent programming, a monitor is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

24.16 Locks

In computing a lock is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies.

24.17 Synchronization

The concept of synchronisation is often split into process and data synchronisation.

In process synchronisation several processes or threads come together at a certain part of a program.

Data synchronisation is concerned with keeping data consistent.

24.18 Granularity and types of parallelism

Granularity is a useful concept in parallel programming. A common classification is

- Fine-grained - a lot of small components, larger amounts of communication and synchronisation
- Coarse-grained - a small number of larger components, hence smaller amounts of communication and less synchronisation

The terms are of course relative.
We also have the concept of

- Embarrassingly parallel - very little effort is required to partition the task and there is little or no communication and synchronisation.

A simple example of this would be a graphics processor processing individual pixels.

24.19 Partitioned global address space - PGAS

PGAS is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each processor. The PGAS model is the basis of Unified Parallel C, Coarray Fortran, Titanium, Fortress, Chapel and X10.

24.20 Fortran and Parallel Programming

Most Fortran compilers now offer support for parallel programming. We next provide a brief coverage of three methods

- MPI - Message Passing Interface
- OpenMP - Open Multi-Processing
- CoArray Fortran

Subsequent chapters look at simple examples using each method.

24.21 MPI

MPI started with a meeting that was held at the Supercomputing 92 conference. The attendants agreed to develop and implement a common standard for message passing. The first MPI standard, called MPI-1 was completed in May 1994. The second MPI standard, MPI-2, was completed in 1998.

MPI is effectively a library of C and Fortran callable routines. It has become widely used and is available on a number of platforms. Some useful web addresses are given below. The first is hosted at Argonne National Laboratory.

`http://www.mcs.anl.gov/research/projects/mpi/`

MPI was designed by a broad group of parallel computer users, vendors, and software writers. These included

- Vendors - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
- Library writers - PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
- Companies - ARCO, Convex, Cray Research, IBM, Intel, KAI, Meiko, NAG, nCUBE, Parasoft, Shell, TMC
- Laboratories - ANL, GMD, LANL, LLNL, NOAA, NSF, ORNL, PNL, Sandia, SDSC, SRC

- Universities - UC Santa Barbara, Syracuse University, Michigan State University, Oregon Grad Inst, University of New Mexico, Mississippi State University, University of Southampton, University of Colorado, Yale University, University of Tennessee, University of Maryland, Western Michigan University, University of Edinburgh, Cornell University, Rice University, University of San Francisco

So whilst MPI is not a formal standard like Fortran, C or C++, its development has involved quite a wide range of people. The following site has details of MPI meetings.

`http://meetings.mpi-forum.org/`

The steering committee (March 2015) and affiliations are given below

- Jack Dongarra - Computer Science Department, University of Tennessee
- Al Geist - Group Leader, Computer Science Research Group, Oak Ridge National Laboratory
- Richard Graham
- Bill Gropp - Computer Science Department, University of Illinois Urbana-Champaign
- Andrew Lumsdaine - Computer Science Department, Indianna University
- Ewing Lusk - Mathematics and Computer Science Division, Argonne National Laboratory
- Rolf Rabenseifner - High Performance Computing Center, Germany

Another useful site is the Open MPI site.

`http://www.open-mpi.org/`

The following is taken from their site.

The Open MPI Project is an open source MPI implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

Both sites provide free down loadable implementations. Commercial implementations are available from

- Cray
- IBM

- Intel
- Microsoft

amongst others.

MPI is, at the time of writing, the dominant parallel programming method used in Fortran. MPI and Fortran currently account for over 80% of the code running on the Archer Service in Edinburgh. Archer is the UK's national supercomputing resource, funded by the UK Research Councils. Visit

`http://www.archer.ac.uk`

for more information.

24.22 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface that supports shared memory multiprocessing programming in three main languages (C, C++, and Fortran) on a range of hardware platforms and operating systems. It consists of a set of compiler directives, library routines, and environment variables that determine the run time behaviour of a program.

The OpenMP Architecture Review Board (ARB) has published several versions

- October 1997 - OpenMP for Fortran 1.0. October the following year they released the C/C++ standard.
- 2000 - Fortran version
- 2005 - Fortran 2.5
- 2008 - OpenMP 3.0. Included in the new features in 3.0 is the concept of tasks and the task construct.
- 2011 - OpenMP 3.1
- 2013 - OpenMP 4.0 was released in July 2013

A number of compilers from various vendors or open source communities implement the OpenMP API, including

- Absoft
- Cray
- gnu
- Hewlett Packard

- IBM
- Intel
- Lahey/Fujitsu
- Nag
- Oracle/Sun
- PGI

The main OpenMP web site is:

`http://www.openmp.org/`

24.23 Coarray Fortran

Coarrays became part of Fortran in the 2008 standard. The original ideas came from work by Robert Numrich and John Reid in the 1990s. They are based on a single program multiple data model. A coarray Fortran program is interpreted as if it were duplicated several times and all copies execute asynchronously. Each copy has its own set of data objects and is termed an image. The array syntax of Fortran is extended with additional trailing subscripts in square brackets to provide a concise representation of references to data that is spread across images.

The syntax is architecture independent and may be implemented on:

- Distributed memory machines.
- Shared memory machines.
- Clustered machines.

Work is underway for additional Coarray functionality for the next standard.

24.24 Other parallel options

There are a number of additional parallel methods. They are covered for completeness.

24.24.1 PVM

Parallel Virtual Machine consists of a library and a run-time environment which allow the distribution of a program over a network of (even heterogeneous) computers. Visit

- `http://www.epm.ornl.gov/pvm/`
- `http://www.netlib.org/pvm3/`

for more details.

24.24.2 HPF

To quote their home page

`http://hpff.rice.edu/index.htm`

'The High Performance Fortran Forum (HPFF), a coalition of industry, academic and laboratory representatives, works to define a set of extensions to Fortran 90 known collectively as High Performance Fortran (HPF). HPF extensions provide access to high-performance architecture features while maintaining portability across platforms.'

They also provide details of:

- Surveys of HPF compilers and tools.
- Currently available commercial HPF compilers.
- public domain HPF compilation systems.
- Research prototypes of HPF and HPF-related compilation systems.
- Mailing list.

24.25 Top 500 supercomputers

Have a look at

`http://www.top500.org/`

for a lot of links to supercomputing centres and information on parallel computing in general. To see what can be done with all this processing power visit:

`http://www.met-office.gov.uk/`

24.26 Summary

Fortran has long been one of the main languages used in parallel programming. This chapter has provided a brief coverage of some of the background to parallel programming in general, and Fortran in particular.

In the next three chapters we will look at a small number of programs that introduce some of the basic syntax of parallel programming with MPI, OpenMP and Coarray Fortran. We will also look at solving one problem serially and then solve it using the parallel features provided by MPI, OpenMP and Coarray Fortran. We provide timing details so that we can see the benefits that parallel solutions offer.

24.27 Bibliography

The ideas involved in parallel computing are not new and we've included a couple of references about computer hardware and operating systems, which provide information for the more inquisitive reader. Wikipedia is an on-line source of information in this area.

Up to date hardware information can be found at most hardware vendor sites. Here are the web sites for AMD, IBM and Intel.

```
AMD
http://developer.amd.com/pages/default.aspx
IBM
http://www.ibm.com/products/us/en/
Intel
http://www.intel.com/en_UK/
products/processor/index.htm
```

Baer J.L., Computer Systems Architecture, Computer Science Press, 1980.

The chapters on the memory hierarchy and memory management are old, but well written.

Deitel H.M., Operating Systems, Addison Wesley, 1990.

Part two of the book (process management) has chapters on process concepts, asynchronous concurrent processes, concurrent programming and deadlock and indefinite postponement. The bibliographies at the end of each chapter are quite extensive.

The following four books provide a good coverage of the essentials of MPI and OpenMP.

Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R., Parallel Programming in OpenMP, Morgan Kaufmann.

Chapman B., Jost G., Van Der Pas R., Using OpenMP, MIT Press.

Gropp W., Lusk E., Skjellum A., Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press.

Pacheco P., Parallel Programming with MPI, Morgan Kaufmann.