C programming for physicists

W. H. Bell

The C programming language is introduced through a set of worked examples. Linux tools for editing, compilation and linking programs are introduced. Program design is discusses using flowcharts and Pseudocode. Following the initial discussion of programming concepts, the majority of the ANSI C syntax and built in commands are demonstrated. The course concludes with a more complicated example of histogramming data from a particle physics simulation.

Contents

1 Introduction

Aims

1. Learn to solve problems by implementing computer programming structures and designs suitable for use with other structured programming languages.
2. Cover core aspects of the C programming language.
3. Introduce programming with a Linux platform.

Syllabus

- Problem analysis strategies
  - Flowcharts
  - Pseudocode
- ANSI C
  - Basic syntax and variable types
  - Arrays, pointers and functions
  - The C preprocessor
  - Input/Output operations
  - Structures and unions
- Introduction to programming on Linux
  - The GNU C Compiler (`gcc`)
  - Building executables with `make`

Material taught within the syllabus is intended to be supplemented by further reading. The recommended reference material for this course is:

- "The C Programming Language", Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, ISBN 0-13-110362-8

Further information on the GNU C compiler can be found at [1].

1.1 Motivation

Computers can be used for data acquisition, control, statistical analyses, building simulations, numerical methods and other complex systems. While many software packages have been written, it is often necessary to write or modify software to facilitate research or to meet a goal within the workplace. Those who are able to program are therefore in

an excellent position to attack complex problems.

1.2 Programming

There are a great many different computer programming languages. Thankfully, once the general process of programming has been understood it does not take a great deal of effort to apply similar strategies to other languages. The C programming language was chosen for this course for several reasons: (i) the basic syntax is the same as C++ and Java, (ii) its structured layout is similar to other common languages, (iii) the language is simple and therefore easily understood, and (iv) C is used in many modern applications.

1.3 Writing programs

Before writing a program, it is important to know exactly what is required of a program. A lot of time can be saved by thinking clearly about the internal structure of the software before any source code is written. For example, a program that will be used for data acquisition should be tailored to the specific hardware that will be used and provide a user interface that contains functionality that the user expects. Failure to properly understand the requirements of a program may result in wasted time and redesigning or patching at a later stage. Some complicated languages such as C++ can be particularly unforgiving in this respect.

Once the requirements of a program have been described, they need to be broken down into a series of steps that can be converted into a computer programming language. In this document,Pseudocode [8] and Flowcharts [7] are used to illustrate the design process.

Flowcharts are a high-level design tool that can also be used as a form of documentation for other developers. Describing a program with a flowchart promotes a logical thought process. However, documenting the fine details of a program using Flowcharts would be prohibitively time consuming. Therefore, Flowcharts should be used either to describe the overall logic of a distinct block of a program or particularly difficult to understand section of a program.

Pseudocode is a quick way of describing a program, without writing the program in a structured programming language. Pseudocode can be used to describe the internal structure of functions or think through the inner workings of complex algorithms. There is no fixed standard for Pseudocode. Therefore probably best to find a way of writing Pseudocode that feels comfortable and is consistent across a programming project. Pseudocode is more useful as a design tool, rather than as a form of documentation. However, Pseudocode may often be present as comments within a structured program.

In summary, the process of writing a program can be broken down into four steps: (i) describing the requirements of a program, (ii) designing large functional blocks, (iii) implementation of in a computer programming language, and (iv) documentation

of the final program. Throughout this process it can be useful to have a log book or electronic note pad to document the development stages.

1.4 Programming with Linux

The example programs that are discussed in this guide can be downloaded from:

`http://www.whbell.net/resources/PhysCIntro/`

Once the examples have been downloaded, they can be unpacked using the tar command:

```
]$ tar PhysCIntroSource-2015-06-22.tar.gz
```

where]$ refers to the Linux prompt. Each subdirectory contains source code and a `README.txt` text file that describes how to build and run the associated example program.

Developing structure programming languages, such as C, is easier when sections of the source code are highlighted using different colours or fonts. This functionality is present within several different editors such as `emacs` and is commonly referred to as syntax highlighting or fontification. These editors often provide other features, such as parenthesise checking and variable width tab stops.

2 Beginning to program with C

2.1 A first program

Programming languages are commonly introduced by writing a program to print a string to the standard output. The standard output is normally displayed on the terminal window or screen.

The design of the program is illustrated as a Flowchart in Figure 1 and described as Pseudocode in Pseudocode 1. The Flowchart shows that the program starts, prints one string and stops. The Pseudocode implementation is a little bit closer to the final C program and includes the returning of a status code back to the operating system. The C implementation of this program is given in Listing 1.

```
main()
  Print a string
  Return 0 to the operating system
```

Pseudocode 1: A pseudocode description of example 1

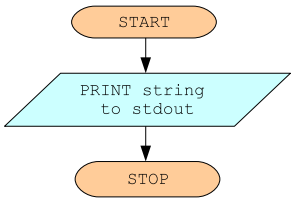

Figure 1: A flowchart description of example 1

```
/* W. H. Bell
** A very simple C program to print one line to the standard out
*/

#include <stdio.h>

int main() {
  printf ("In the beginning...\n");
  return 0;
}
```

Listing 1: The C implementation of example 1

The execution of every C program starts from a `main()` function. From this function other functions can be called. The return type of `main()` is given by the `int` prefix. Within the Linux/UNIX environment the operating system expects a program to return an exit value. The value of the return statement from the `main()` function is collected by the operating system and is available for a user to query after the program has run. Following the execution of the program, the return value can be queried by typing

```
]$ echo $?
```

where]$ is the Linux prompt.

The contents of the `main()` function are delimited by the brackets { }, which represent a compound statement. Inside this compound statement there may be several statements, each terminated by a '`;`' character, together with other compound statements. In this example, the `main()` function only contains two statements: one to print a string to the standard output and one to return the exit value to the operating system. The first of these statements prints a string to the screen by calling the standard output function `printf`. This string is terminated by the end of line character `\n`. At the top of the example, the pre-declaration of the `printf` function is included by including the header file `stdio.h`. When this program is compiled, the compiler reads the pre-declaration of `printf` from the header file and leaves an unresolved function call in the

machine code to be resolved at link time. Above the `#include` statement is a comment. Comments in C can be entered using `/* */` to surround the comment area.[1] When the compiler compiles the code it ignores any text surrounded by `/* */`.

[1]The use of `//` comments is a non-ANSI feature and is therefore not included in this course.

2.2 Loops, conditional statements and functions

Having written a first program, the next step is to introduce logic statements and other functions. A program that introduces functions, conditional statements and loops is given in the `example_02` subdirectory. This example reads from the standard input and performs several checks on integers and characters provided by the user. (The standard input normally refers to something written to a terminal, either using the keyboard or by input file redirection.) A Flowchart describing the `main()` function is given in Figure 2. A Pseudocode implementation of this program is given in Pseudocode 2.

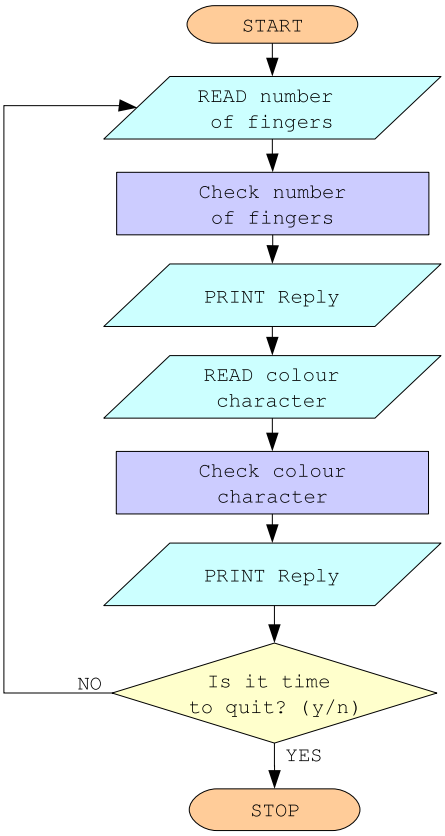

Figure 2: A flowchart representation of the `main()` function of example 2

Functions Similar to the first program, example 2 contains an `int main()` function. Within this `main()` function three functions are called: `numFingers`, `pickColour`, and `quitTime`. Each function is pre-declared before the `main()` function. Each pre-declaration is a statement where the return type, and input parameter types must be given. The `void` type simply means that no input parameter or return value is expected. All functions must be either predeclared or declared before they are used. There are three

```
main()
  DO
    Check the number of fingers.
    Check the colour.
  WHILE not time to quit

numFingers()
  Print the question
  Read a number from the stdin
  Compare the number and return an answer

pickColour()
  Print the question.
  Read a character from the stdin.
  Compare the character and return an answer.

quitTime()
  Ask the user if it is time to quit (y/n)
  Collect a character from the stdin
  Compare this with y/Y and return 1 if its is time to quit
```

Pseudocode 2: A pseudocode representation of example 2

pre-declaration statements before the `main()` function.

```
void numFingers(void);
void pickColour(void);
bool quitTime(void);
```

The implementation of these functions is given after the `main()` function. Following the same syntax as the `main()` function, the implementation of each of these three functions has a return type, a series of input types, and a compound statement enclosing the function contents. These three functions do not have any input parameters. If input parameters are defined in the function definition, then their types must be given in the pre-declaration and their types and names must be given in the implementation. If a function has been pre-declared, but has not been implemented then the program will fail to link.

In this example, the function pre-declarations are given before the `main()` function and the implementation present afterwards. However, the example would compile correctly if the pre-declaration was removed and the implementation of the functions was moved to be before the `main()` function. While this would work within this simple example, it is not possible to use implementations without pre-declarations when functions are dependent on each other. As programs become more complicated, the pre-declarations are moved into header files and the implementations are moved into associated libraries.

Header files are introduced in example 5.

Conditional statements Common ways of writing conditional statements involve either `if`, `if else`, `else` or `switch` statements. There are other ways of constructing conditional statements, but these are not covered in this course. Starting with `if`, `if else`, `else` statements, examples of their syntax are given within the `numFingers` and `quitTime` functions of example 2. Each `if` statement is evaluated such that when the contents of the logic associated with an `if` statement inside the `()` brackets is true, then the code within the following compound statement is executed.`if`, `if else`, `else` statements operate sequentially, such that each piece of logic is tested in turn. If all of the logic tests fail, then the statement following `else` is executed.

In some cases where simple sorting is needed a `switch` statement is a better choice than a long `if`, `if else`... `else` statement. An example `switch` statement is given in the `pickColour` function of example 2. While faster than an `if`, `if else`, `else` statement in some cases, a `switch` statement is limited to simple cases and therefore the logic allowed can be somewhat restrictive.

Loops Several types of loops are available to C programmers. There are `while`, `do while` and `for` loops. Each of these loops continue to loop while a condition is true. All of the logic available within an `if` conditional statement is also available within these conditional tests. Instances of these loop types can be found within some of the examples within this course. For example, example 2 contains a `do while` loop in its `main()` function. This loop continues while the boolean evaluated within the `while( );` is true. This remains true until the function `quitTime` returns true. The `while` loop tests on NOT `quitTime` return value.

2.3 Pointers and arrays

Many languages use pointers implicitly, such as implemented in FORTRAN and Java. In the C programming language pointers are implemented explicitly. Therefore, if a pointer has not been implemented, a normal variable will be used. The source code in the `example_03` subdirectory contains an example of pointer functionality and illustrates the difference between a pointer and a normal variable. The design of example 3 is given as a Flowchart in Figure 3 and as Pseudocode in Pseudocode 3.

Pointers are used to point to a memory address. Once initialised, a pointer can be used to access a memory address or the value stored in the memory address. In example 3, there are two distinct parts to the program: (i) the call to the function `fun` and (ii) the iteration over the array `v[]`.

Functions and pointers The function `fun` is declared as

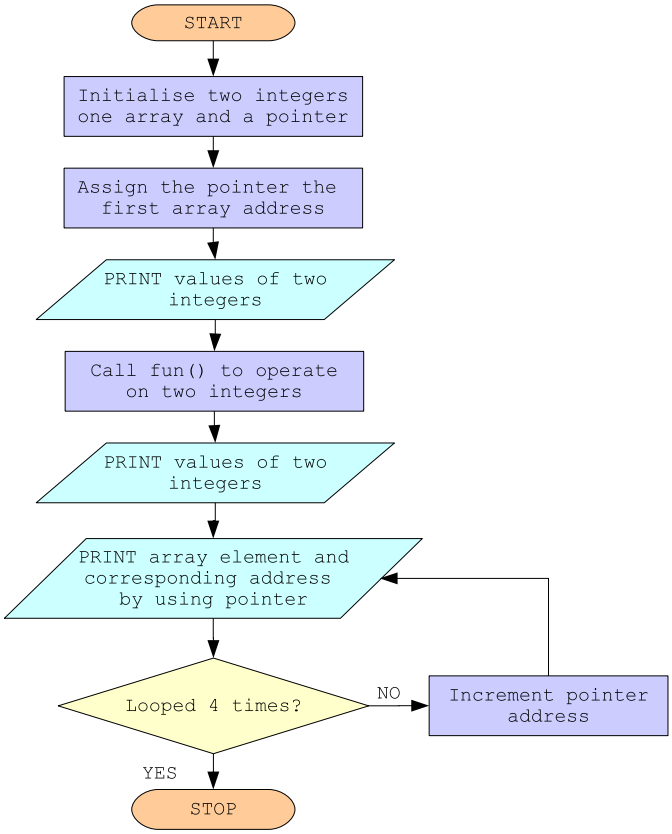

Figure 3: A flowchart illustration of the `main()` function of example 3

```
void fun(int, int *);
```

with input parameter types `int` and `int *`, where the second input parameter is a pointer. When the function is called the memory address of `p` (`&p`) is assigned to the pointer declared in `fun`. The difference between the behaviour of a pointer and a normal local variable can be seen by running the program. After the function `fun` is called, the variable `np` contains the same value that it contained before the function call. However, the value of the variable `p` is modified when the function `fun` is called. Both `np` and `p` are initialised in the `main()` function with the same value:

```
int np = 1, p = 1;
```

At the point of initialisation an `int` sized block of memory is allocated to `np` and `p`. Then the function `fun` is called with the value of `np` and the address of `p`. Within the function `fun`, a new block of memory is allocated for the local variable `np` that is distinct from the variable contained in the `main()` function. This memory is assigned the value from the variable `np` that was declared in the `main()` function. In the function `fun` the value of

```
main()
  Initialise two np and p integers with 1.
  Initialise an array v with four elements.
  Initialise a pointer pv with the address of the first element of the
  array.

  Print the values of the two integers np and p.
  CALL fun() with the value np and the address &p.
  Print the values of the two integers np and p.

  Iterate over the array indices using the pointer pv.
    Print the contents and address of each array element using pv.

fun(value and pointer)
  Assign a number to the local variable.
  Assign a number to the memory the pointer points to.
```

Pseudocode 3: A pseudocode implementation of example 3

the local variable `np` is modified and then lost as the function exits. Therefore, the value of the variable `np` declared in the `main()` function is not modified. Unlike the variable `np`, the value of the variable `p` declared within the `main()` is set by using a pointer. The pointer is initialised with the memory address of the variable `p` contained within the `main()` function. Then the memory address pointed to by the pointer `*p` is assigned the value 2. Therefore when returning to the `main()` function the value contained in the memory of `p` is still 2.

Arrays and pointers An array of type 't' is a series of memory blocks of size according to the type. Each element of the array behaves as a separate variable of the given type of the array. Array sizes are determined at compile time and therefore must be declared somewhere within a program.[2] In example 3 the array `v` is declared with four elements:

```
int v[] = {1,2,3,4};
```

This code is equivalent in function to:

```
int v[4];
for(int i=0;i<4;i++) v[i]=i;
```

The size of the array within example 3 is determined by the number of elements within the brackets {}.

[2] C does allow dynamic allocation of memory. This will be briefly demonstrated within the problem at the end of the next section.

Within example 3 the address of the first element is assigned to the pointer `*pv`. It is important to note the point of declaration is the only place where the address is assigned to a pointer in this fashion. Equivalent in function but slightly longer hand, this could be written as:

```
int *pv;
pv=&v[0];
```

Once the pointer has been assigned the memory address of the first element of the array `v[0]` it can be used to access the elements as demonstrated in the example and illustrated in figure 4. The action of incrementing the memory address of the the pointer

```
pv++;
```

causes it to point at the next element of the array.

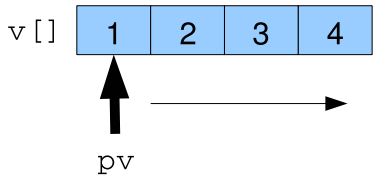

Figure 4: An illustration of a pointer being used to iterate over array elements, where the blue boxes signify the sequential sections of memory containing the values stored in the array.

2.4 Command line

A C program can either receive command line input at execution time or by reading input from a file or other data source. This example demonstrates how command line arguments are passed into a C program when the program is executed. The design of the program is given as a Flowchart in figure 5 and as Pseudocode in Pseudocode 4.

```
main(command line arguments)
  Print the number of command line arguments
  Loop over the command line inputs
    Print each command line input
```

Pseudocode 4: Example 4 in pseudocode

When programming C on a Linux platform the `main()` function is normally implemented in one of two ways:

```
int main(void)
```

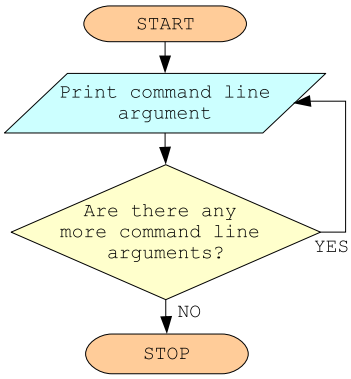

Figure 5: A flowchart describing example 4

and

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

The second form allows command line input to be passed into the program. The operating system allocates the memory for `*argv[]` and initialises it with the input command line arguments. The first element of the `argv` array contains the name of the executable. The second element exists only if there is a argument following the name of the executable, etc.. The parameter `argc` is initialised by the operating system with the number of elements in the `argv` array. While it may seem odd to include the name of the executable in this list it can in fact be very useful. For example, if a program has been written to do several tasks a symbolic link can be used to control which task it performs. The logic of this can be tested out using example 4 by typing

```
]$ ln -s command_line.exe another_cmd.exe
]$ ./another_cmd.exe
```

A simple test on the value of `argv[0]` could then be used to do something else.

2.5 File access

Most programs will need to save or read data from disk or another input/output device. Example 5 demonstrates how some simple data can be written to and read from an ASCII file. The design of the `main()` is given in figure 6 and the complete program is described in Pseudocode 5 and 6.

The program starts by checking the command line arguments. Then either `file_write` or `file_read` is called. The functions `file_write` and `file_read` are pre-declared in the header file `file_io.h` and implemented in `file_io.c`. When this code is compiled the

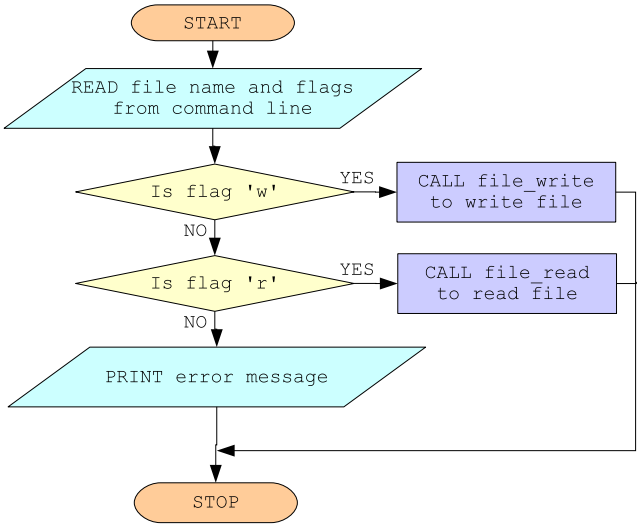

Figure 6: A flowchart describing the `main()` function of example 5

```
main(command line arguments)
  Check the command line arguments
    Make sure the file name is given
    Check for the input flag and if not given set the default to
    write.

  IF the input flag is write, write the specified file.
  ELSE IF the input flag is read, read the specified file.
  ELSE report an error
```

Pseudocode 5: The `main()` function of example 5 in pseudocode

resultant machine code `main.o` contains undefined references to these functions. Then at link time the machine code in `file_io.o` and `main.o` is linked together to produce the final executable. The practical process of building this executable is discussed in section 2.5.1.

```
file_write(file name)
  Open an output file.
  IF the file was successfully opened
    LOOP from 1 to 20
      Print the value into the file.
      IF the value is a multiple of 5 then print a newline.
      ELSE print a space.
  ELSE
    Report an error.

file_read(file name)
  Open an input file.
  IF the file was successfully opened
    Get a character until the end of file or until the buffer is full.
    IF the character is not a space or a newline
      Copy character into buffer.
    ELSE IF the buffer contains something
      Add a string terminating character.
      Scan the contents into an integer.
      Print the integer.
      IF the integer is a multiple of 5 print a new line.
      Reset the buffer.
  ELSE
    Report an error.
```

Pseudocode 6: The `file_write()` and `file_read()` functions of example 5 in pseudocode

The `file_io.h` header file of example 5 contains three pieces of precompiler syntax which surround the function predeclarations.

```
#ifndef FILE_IO_H
#define FILE_IO_H

void file_write(char *filename);
void file_read(char *filename);

#endif
```

The purpose of these statements is to prevent the functions from being pre-declared twice. While this does not happen within this example, double declaration which can result in compiler errors is more of an issue with more complicated programs. It is therefore a good idea to adopt the use of these precompiler statements at an early stage.

The functions `file_write` and `file_read` are implemented in `file_io.c`. The `file_write` function calls `fopen` to open a file for writing

```
outputfile = fopen(filename,"w");
```

This command returns a *file pointer*, which is a pointer to a `struct typedef` called `FILE`. `FILE` contains information about the file which needs to be passed between the different file I/O function calls. If the file cannot be opened for writing then `null` is returned. This is logically equivalent to false and can therefore be used to check if the file was opened correctly or not.

```
if (outputfile) {
```

Once the file has been opened successfully data can be written in ASCII format by calling the `fprintf` function. This function follows the same syntax as the `printf` function with the exception of the *file pointer* as the first argument. When all the data have been written to the file `fclose` is called to flush any remaining data in memory to the file and to close the file. If `fclose` is not called, then when a program exits the file will only be partially written to disk.

Once the output file is present on disk the function `file_read` is called to read the data back in and print them to the standard out. `file_read` starts by opening a file for reading

```
inputfile = fopen(filename,"r");
```

Then single characters are read from the file until the End Of File (EOF) is reached or the character buffer is full.

```
while((c=fgetc(inputfile)) != EOF  && j<9)
```

This is much more robust than just using `fscanf` to read data, which can easily result in errors or crashes. If the character is not a space or a new line it is appended to

the character buffer. When a space or a new line is found then if the buffer contains something it is scanned into an `int` by calling

```
sscanf(str, "%d", &i);
```

where `&i` is the memory address of `i` and `%d` tells `sscanf` to treat the string as an integer. After each `int` is read the data are printed back to the standard output in the same form as they were saved to disk. Finally the file is closed.

2.5.1 Make

As the number of files involved or dependencies increases simply typing `gcc` would become an lengthy and complicated task. Therefore from this example onwards `Make`[2] is used to aid with compilation and linking of examples. The `Makefile` in example 5 has three targets which can be called from the command line: `file_io.exe`, `clean` and `veryclean`. The default target is the first one `file_io.exe`. The default target it called by typing

```
]$ make
```

The other targets are called by using the target name

```
]$ make veryclean
```

Above the default target are the definition and initialisation of three variables

```
CC=gcc
TARGET=file_io
OBJECTS=main.o file_io.o
```

The last one of these is a list. When the default target is called the dependencies `$(OBJECTS)` are checked. If these files are not found then `Make` looks through the file for a suitable target to make these files with. If there is a `.c` file with the correct name it calls

```
%.o: %.c
        @echo "**"
        @echo "** Compiling C Source"
        @echo "**"
        $(CC) -c $(INCFLAGS) $<
```

This target compiles the `.c` file into a `.o` file. Then when all of the `$(OBJECTS)` are present on disk `Make` executes the rest of the default target

```
$(TARGET).exe: $(OBJECTS)
        @echo "**"
```

```
	@echo "** Linking Executable"
	@echo "**"
	$(CC) $(OBJECTS) -o $(TARGET).exe
```

which links the object files together to make the executable.

When any of the `.c` files referred to in the `Makefile` are present but are newer than the `.o` file of the prefix name, `make` just rebuilds the files concerned and completes the link step. When any of the `$(OBJECTS)` are newer than `file_io.exe`, `make` just completes the link step. This can be tested out by using `touch`

```
]$ make
]$ touch main.c
]$ make
]$ touch file_io.o
]$ make
```

More examples of `make` syntax are given in later examples.

2.6 Problem

While working in research it can often be the case that a piece of equipment or software produces its output in the wrong form for use as an input to another application. The program `problem/generator/generator.c` writes random numbers into a `sample.txt` file. Follow the instructions in `generator/README` file to produce a `sample.txt`. Then write a program to convert `sample.txt` into a Comma Separated Values file (CSV)[6] suitable for use as input to a spreadsheet. Run your program and read the resultant output into a spreadsheet and plot the Value column as a histogram.

Start by drawing a simple Flowchart outline of the program. Then write the program in Pseudocode. Finally write your program in C using code from each of this section's examples as necessary. Remember to add comments.

3 Data structures

In this section more complex data structures `struct` and `union` will be discussed. An example of how to interface to FORTRAN77 code will be given and the section finishes with a complicated example including much of the syntax introduced so far.

3.1 Pointers and structs

The purpose of example 6 program is to demonstrate `struct` syntax and to show how arrays and `struct`s can be passed into a function. The program is described by Pseudocode 7.

```
main()
  Instantiate two ints, an array of two ints and two structs.
  Assign 1 to all data members
  Print the values contained in the ints, the array and the structs
  Call fun() to modify some of the values
  Print the values contained in the ints, the array and the structs

fun(int np, int *p, int *arr, struct st dat, struct st *dat_ptr)
  Change the value of the local variable np
  Change the value in the memory p points to.
  Change the values of the two elements of arr
  Change the data members of the local struct dat
  Change the data members of the struct dat_ptr points to.
```

Pseudocode 7: Example 6 in pseudocode

At the top of the file `pointers2.c` a struct is defined

```
struct st {
  int i;
  int array[2];
};
```

where `st` is the name of the `struct`. The data members of the struct are enclosed in the {} brackets. This definition can be given in a header file instead of in a source file. The definition defines the `struct` but does not create an instance of it. Then in the `main()`two `st` `struct`s are instantiated

```
struct st dat, dat_p;
```

and their data members are initialised.

```
dat.i=1;
dat.array[0]=1;
```

```
dat.array[1]=1;
dat_p.i=1;
dat_p.array[0]=1;
dat_p.array[1]=1;
```

The syntax is <*instantiation*>.<*data member*>. Each instantiation of a `struct` is stored in a separate block of memory. Unlike an array the data members can have different lengths. The total memory used by a `struct` instantiation is determined by the sum of the memory used for the data members. The order of the `struct` members in the `struct` definition determines the order of their allocation in memory. The member types can include other `struct`s, `union`s, arrays and basic variables etc..

After the `struct`s, basic variables and the array have all been initialised the function `fun()` is called. This function is called with the value of `np`, the address of `p`, the array `arr`, the value of `dat` and the address of `dat_p`. Notice that the name of the array `arr` is actually equivalent to passing an address of the first element `&arr[0]`. Therefore the function declaration and implementation of `fun` includes this as `int *`. Inside the function `fun` each of the variables is given a different value. The syntax of accessing data members of a `struct` pointer is <*pointer name*> − ><*data member*>

```
dat_ptr->i=5;
```

or (*<*pointer name*>).<*data member*>

```
(*dat_ptr).i=5;
```

When the function `fun` is called the argument `struct st dat` caused a new instantiation of the `struct st` to be made.

```
void fun(int np, int *p, int *arr, struct st dat, struct st *dat_ptr)
  {
```

The memory assigned to this instantiation is local to the function `fun` and will go out of scope when the program leaves this function. When the function is called the data members of `dat` are initialised with the values of the data members of `dat` from the `main()`. The behaviour is therefore similar to the variable `np`.

Unlike `dat`, the address of `dat_p` is passed into the function fun. The pointer `struct st *dat_ptr` is therefore initialised with this address. When the data members of the object that `*dat_ptr` points to are modified

```
dat_ptr->i=5; dat_ptr->array[0]=6; dat_ptr->array[1]=7;
```

then the change is still present when the execution returns to the `main()`.

3.2 Binary I/O and unions

In example 7 the data concerning several fundamental particles are written in binary form into an output file. Then the program reads these data back and prints out the 7^{th}

particle data record. The design of the program is given in Pseudocode 8.

```
main()
  Print the size of one union entry.
  Print the size of each union member.
  Write 10 records of fundamental particles to a binary file.
  Read the 7th record back from the binary file.
  Print the values of the 7th record.

status write_records()
  Create an array of 12 records.
  Initialise the array of records with fundamental particle data.
  Open a file for writing.
  Write each record element out in binary form.
  Close the file.

status find_record()
  Open an input file.
  Fast forward to the 7th record.
  Read the 7th record.
  Close the file.
```

Pseudocode 8: Example 7 in pseudocode

At the beginning of the `main()` a `record dat` is instantiated.

```
record dat;
```

`record` is a `typedef` declared in the header file `data_record.h`.

```
typedef entry record[4];
```

This statement means that a `record` is a `typedef` of an array of `entry`s of length 4. Therefore

```
record dat;
```

is equivalent to

```
entry dat[4];
```

The type `entry` is itself a `typedef` of a union

```
typedef union {
  int id;
  double mass;
  char name[16];
  int charge;
} entry;
```

In the previous example `struct`s were declared with the syntax:

```
struct st {

};
```

This syntax is allowed with `union`s and the `union` syntax used in this example is allowed with `struct`s.

```
typedef struct {

} st;
```

The advantage of the `typedef` form used in this example is that the `union` prefix does not have to be carried around the code. It is therefore much more common that the `typedef` form of a `struct` or `union` definition is used.

A `union` contains a set of members which are listed within the {} brackets. The members of a `union` share the same memory allocation. The size of the `union` is therefore set by the largest member. This can easily be demonstrated by running example 7. The syntax for accessing the members of a `union` is the same as the syntax used for accessing the members of a `struct`.

In this example an array of the `entry union` is created with 4 elements, this is a `record`. The `record` stores a particle's `id`, `mass`, `name`, and `charge`. To make the code more readable a set of `#define` statements are used instead of the indices.

```
#define RECORD_ID 0
#define RECORD_MASS 1
#define RECORD_NAME 2
#define RECORD_CHARGE 3
```

When the code compiles the precompiler replaces each definition with its value. Then the compiler turns the result into machine code. It is a common convention that `#define` statements use all upper case letters to avoid confusion with real variables. What follows the `#define NAME` does not have to be a simple number, the precompiler just does a find and replace.

The first use of the precompiler definitions is in `write_records`. In this function an array of `record`s is instantiated

```
record particle_data[12];
```

and each `entry` is filled by simple assignment.

```
particle_data[0][RECORD_ID].id = 22;
particle_data[0][RECORD_MASS].mass = 0.E+00;
strcpy(particle_data[0][RECORD_NAME].name,"gamma");
particle_data[0][RECORD_CHARGE].charge = 0;
```

This demonstrates that the array index contained in the `typedef record` statement comes after the array index defined by defining the `particle_data` array.

Following the initialisation of the `particle_data` array the data are written to a binary file. The binary file is opened in the same way as the ASCII file in example 5.

```
file_ptr = fopen(particle_file,"w");
```

This time however the error messages are handled by printing them to the standard error. To print to the standard error the `fprintf` function is used with the standard error *file pointer* `stderr`.

```
fprintf(stderr," Error: unable to open \'%s\' for writing.\n",
  particle_file);
```

The standard error is printed to the console or shell window in the same manner as the standard output but is contained in a different stream. When running a program on Linux the standard output an be redirected to a file.

```
]$ ./prog.exe > output
```

This does not redirect the standard error though, which can be left for the user to read.[3]

Once the file has been opened the `particle_data` records are written to it in binary form.

```
fwrite(&particle_data[i][j], 16, 1, file_ptr);
```

The arguments of `fwrite` are the address of one `union` instantiation, the size of the `union`, the number of `union` instantiations to be written and the *file pointer* to which the data should be written. Since the size of the `union` is set by the largest member each data element is the same size. This means that the code to write out these data is very simple and only requires two loops over the `fwrite` statement.

After the data have been written to disk `find_record` is called to find and read the 7th record. `find_record` opens the binary file in the same way as the ASCII file was opened in example 5.

```
file_ptr = fopen(particle_file,"r");
```

Then after checking the file was opened successfully it fast forwards to the correct position in the file

```
if(fseek(file_ptr, (long)(sizeof(*dat)*offset), 0) != 0)
```

where `fseek` returns a non-zero value if the seek fails. This then leaves the file-position indicator set to point at the value given by the offset. The record is then read by calling `fread`.

[3] It is possible to redirect both standard error and standard output to one file.

```
fread(dat, sizeof(*dat), 1, file_ptr);
```

Finally in the `main()` these data are printed to the standard out

```
printf("\t Id \t Mass     \t Name           \t Charge \n");
printf("\t %d \t %3.3e \t %-12s \t %d \n",
       dat[RECORD_ID].id,
       dat[RECORD_MASS].mass,
       dat[RECORD_NAME].name,
       dat[RECORD_CHARGE].charge);
```

Throughout example 7 possible errors are caught and result in a non-zero value being returned to the operating system. When writing a program it is important to make sure the implementation is able to prevent crashes by handling error conditions properly. This means that the program should exit in a controlled manner and provide the user with a description of the error, rather than a cryptic message or nasty crash.

3.3 Linking with FORTRAN77

Although many modern programs have been written in C it is sometimes very useful to be able to link C code to another programming language. When the code to be linked to is compiled into machine code this can be achieved by knowing how the two compilers concerned work. For example, FORTRAN77 a programming language often used for physics and engineering applications, can be compiled by `gfortran` into object files. These object files can be browsed with `nm`.

```
]$ gfortran -c fortran.for
]$ nm -g fortran.o
00000251 T call_back_
00000000 T commons_
         U do_lio
         U e_wsle
00000062 C forcom_
         U mult_a_
         U s_wsle
```

C code is also compiled into object files, which can also be browsed with `nm`.

```
]$ gcc -c main.c
]$ nm -g main.o
         U call_back_
         U commons_
00000065 T fill_common
         U forcom_
```

```
00000000 T main
000000ed T mult_a_
         U sprintf
```

All that is needed to join FORTRAN77 with C is for the undefined references in FORTRAN77 or C to be present in the object files generated from the other language. In this example `call_back_`, `commons_`, `forcom_` and `mult_a_` are linked between the two languages. The other undefined symbols can be found in the system libraries. When `gfortran` compiles the FORTRAN code it uses a lower case version of the name and appends one underscore to the end of the name.[4]

Example 8 demonstrates all of the features of linking C and FORTRAN77 together. The design of the program is given in Pseudocode 9.

```
main()
  Fill the FORTRAN common block with numbers and a string.
  Print the contents of the FORTRAN common block with FORTRAN code.
  Multiply a number and print a string by calling the FORTRAN code.

fill_common()
  Fill the int and float arrays of the common plot with
    sequential numbers.
  Fill the FORTRAN string with a C string.

mult_a_()
  Multiply the input value by 10 and return it.

SUBROUTINE COMMONS
  Print the contents of the common block.

FUNCTION CALL_BACK
  Print the input string.
  Call the C function to multiply the input value by 10.
  Print the resulting value and return it.
```

Pseudocode 9: Example 8 in pseudocode

There are some important differences to point out between the two languages. Firstly C strings are character arrays terminated by the '\0' character. This means that the maximum length of a string is equal to the number of elements of the character array minus one. In FORTRAN character arrays are also used to store strings but unlike C '\0' is not used. The maximum length of a string in FORTRAN is therefore set by the number of elements in the character array. FORTRAN keeps track of the length of a

[4]This behaviour can be controlled by using compiler options.

string by implicitly passing its length with the string. When FORTRAN is compiled with gfortran these implicit variables become explicit in the machine code output. Therefore when calling a FORTRAN function from C, the lengths of each of the strings must follow each of the character arrays:

```
float call_back__(float *,char *, int);
```

where the int is the implicit string length variable.

The allocation of memory to FORTRAN arrays is different to that of C. For example when a two dimensional array is declared in FORTRAN

```
INTEGER INTARRAY(3,2)
REAL REALARRAY(2,3)
```

in C the array indices of the equivalent memory mapping are:

```
int intarray[2][3];
float realarray[3][2];
```

The first index of a FORTRAN array is by default 1, but C always uses 0 as the first index of an array. These differences are particularly important when dealing with `common` blocks. In the FORTRAN include file `FORTRAN.INC` a common block is declared.

```
INTEGER INTARRAY(3,2)
REAL REALARRAY(2,3)
CHARACTER*50 SOMESTRING
COMMON/FORCOM/INTARRAY,REALARRAY,SOMESTRING
```

This defines the common block and creates one global instance of the common block in memory. In a similar fashion to a `struct`, the order of the member variables denotes there order in memory, and the total size of the common block is just the sum of the sizes of the members. Once a common block has been declared FORTRAN or C code can access its data members. The FORTRAN common block can be included in several different FORTRAN files, but only one instance of it will be created in memory. This global memory can be accessed from C by creating a global un-resolved `struct` of the correct name.

```
typedef struct {
  int intarray[2][3];
  float realarray[3][2];
  char somestring[50];
} forcom;

extern forcom forcom_;
```

The `extern` command means that the compiler will look through the object files for the memory definition of `forcom_` and link to it, rather than allocate another block of memory. Without the `extern` prefix the memory allocated in the C program would be

different to that of the FORTRAN common block. The order, types and sizes of the C `struct` must match the order, types and sizes of the FORTRAN common block.

In addition to the memory mapping and variable type name differences, FORTRAN uses pointers implicitly in function calls. A function of the form

```
FUNCTION CALL_BACK(A,NAME)
IMPLICIT NONE
REAL A, C, CALL_BACK, MULT_A
CHARACTER*(*) NAME
```

is therefore equivalent to

```
float call_back_(float *,char *, int);
```

The last variable in the C version of this function is the length of the string, which is implicitly present in the FORTRAN code.

3.4 Sine wave generator

As programs become larger the initial design becomes more important. Therefore to encourage thought about program structure the example programs given in this course will become more and more complicated. In example 9 there are 9 C files containing a total of 300 lines. When writing a program of this size it needs to be clear what the design requirements and components are.

Design

Requirements A program to either generate sine wave spectra and save them to a binary file or read the spectra from a binary file. The last spectrum either written to or read from the file should be plotted with gnuplot [3].

Components

- `main` implement in `main.c`
  Handles the users command line arguments and selects either write or read with the selected number of events if this is given.

- `gen_data` implement in `gen_data.c`
  Generates the sine wave from a flat random number generator

- `gen_value` implement in `gen_data.c`
  A function to generate a random floating point number between two limits using

a flat random number generator.

- `write_data` implement in `file_io.c`
  A function to write a data sample to a binary file

- `read_data` implement in `file_io.c`
  A function to read a data sample from a binary file

- `sample` define in `gen_data.h`
  A struct to contain sine wave samples.

- `plot_data` implement in `plot_data.c`
  A function to plot a data sample with gnuplot

- `gnuplot` implement in `gnuplot.c`
  A function to provide an interface with gnuplot by using a system call.

Discussion of the implementation

The implementation of example 9 is described in Pseudocode 10, 11, 12 and 13.

This program uses several concepts that were introduced with previous examples. The discussion of this example therefore only covers additional points.

The program starts by checking if a binary file is to be written or read. It then checks the number of events which should be either written or read. Following this the program writes or reads sine wave data to or from the binary file and plots the last sine wave sample with gnuplot. In example 9 the each sine wave sample is stored in an instantiation of a `struct typedef sample`.

```
typedef struct {
  float x[MAX_PTS]; /* The x value */
  float y[MAX_PTS]; /* The y value */
  int pts; /* Number of points in the sample */
} sample;
```

In the `main()`an instance of `sample` is declared.

```
sample dat;
```

Then `dat` is either filled with generated data

```
gen_data(&dat);
```

or with data read from a file.

```
read_data(file_ptr , &dat);
```

While sizes of the data member arrays `x` and `y` are fixed the number of elements used varies. Therefore the data member `pts` is use to keep track of the number of useful elements in memory. When these data are written to file the record length of each sine wave sample varies in accordance with the value of `pts`. The value of `pts` is therefore written to the file for each sine wave sample. An illustration of the resultant file structure is given in figure 7.

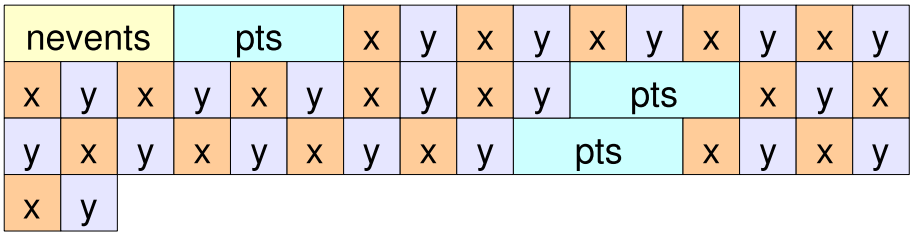

Figure 7: An illustration of the binary file structure used in example 9

The last sine wave sample that is either read from file or generated is plotted with gnuplot. The `main()`calls `plot_data`, passing it a pointer to the last sample. `plot_data` then writes a temporary file for gnuplot to read. This file has to be of the form: "<`number`> <`number`>". Then after plot_data has written the file the gnuplot command is assembled.

```
sprintf(gnuplot_command, "plot \'%s\'\n", filename);
```

(`sprintf` follows the same syntax as `printf` but starts with a pointer to a string buffer.) The command assembled corresponds to what would be typed at the interactive gnuplot command line to plot these data. This command is passed to the `gnuplot` function to plot the data. The `gnuplot` function inserts another string in front of the gnuplot command

```
sprintf(syscommand, "echo \"%s\" | gnuplot -persist", gnucommand);
```

and uses a system call to execute the command in a Linux shell.

```
system(syscommand);
```

The affect of the "`-persist`" is that gnuplot continues to run after the C program has exited.

```
main(commmand line inputs)
  Check the command line inputs
    IF the file name and flag are not given report an error
    IF the flag is -w check for the number of events
      IF the number of events has not been given set it to 10 events.
      ELSE read the number of events.
      Open an output file.
      Write the number of events into the file as a header.
      Generate the required number of events needed.
         Write each one to the output file.

      Plot the last event with gnuplot.
      close the output file.

    ELSE IF the flag is -r open an input file.
      Read the number of events, recorded at the start of the file.
      IF the number of events has not been given use the number at the
        top of the file.
      ELSE check if the argument is a number and if it is set the
        number of events to read to be the number given.
      IF the number of events selected is greater than the number of
        events in the file
        Set the number of events selected to be the number of events
          in the file.

      Read the events from the file.
      Plot the last event.
      Close the input file.
   ELSE return an error reporting the flag is invalid.
```

Pseudocode 10: Example 9 in pseudocode

```
gen_value(limits)
  Generate a random float between two limits using a flat random
    distribution.
  Return this value

gen_data(data sample pointer)
  Set the limits for the random parameters: number of points, a, b, and c.
  Generate values for: the number of points, a, b, and c.
  Use the number of points to determin the step size, where the range
    of x is 0 to 2PI.
  Use the step size, and the values for a, b, and c to generate the
    sine wave event.
```

Pseudocode 11: Example 9 in pseudocode

```
write_data(file pointer, data record)
  Write the number of points
  Loop over all points
    Write the x value
    Write the y value

read_data(file pointer, data record)
  Read the number of points
  Loop over all points
    Read the x value
    Read the y value
```

Pseudocode 12: Example 9 in pseudocode

```
plot_data(data record)
  Open a temporary output file.
  Loop over all points
    Write "x y" as text to the file.
  Create the gnuplot command.
  Plot the data with gnuplot.
  Remove the temporary output text file.
```

Pseudocode 13: Example 9 in pseudocode

3.5 Problem

Computers can be used to monitor other hardware or themselves. In the `problem_02` subdirectory there is a program that requests memory using the `malloc` function. Then after some period of time, the program then frees the dynamically allocated memory by calling the `free` function. Write a program to monitor the total memory used every second for a given number of minutes. The program should finish by plotting the total memory usage as a function of time. Once the pseudocode and implementation has been finished run the monitoring program at the same time as the memory loading program:

```
]$ ./system_monitor.exe 2 &
]$ ../problem/resource_hog/resource_hog.exe
```

where 2 is the number of minutes the monitoring program should run.

Hints

The memory allocation status can be obtained by calling `sysinfo`

```
int sysinfo (struct sysinfo *info)
```

This function can be called by including

```
#include <sys/sysinfo.h>
```

This include file also includes a definition of the `sysinfo` struct:

```
struct sysinfo {
  long uptime;             /* Seconds since boot */
  unsigned long loads[3];  /* 1, 5, and 15 minute load averages */
  unsigned long totalram;  /* Total usable main memory size */
  unsigned long freeram;   /* Available memory size */
  unsigned long sharedram; /* Amount of shared memory */
  unsigned long bufferram; /* Memory used by buffers */
  unsigned long totalswap; /* Total swap space size */
  unsigned long freeswap;  /* swap space still available */
  unsigned short procs;    /* Number of current processes */
  unsigned long totalhigh; /* Total high memory size */
  unsigned long freehigh;  /* Available high memory size */
  unsigned int mem_unit;   /* Memory unit size in bytes */
  char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding for libc5 */
};
```

Use the `sleep` command used in `problem/resource_hog/main.c` to sleep for one second between measurements. This will prevent the monitoring program from using up a lot of CPU, which would affect the results. More information can be found in the manual page by typing

```
]$ man -s 3 sleep
```

Use the `time` command used in `problem/resource_hog/main.c`. More information can be found in the in the manual page by typing

```
]$ man time.h
```

4 Simple analyses

During previous sections much of the C programming language has been introduced. This section is therefore dedicated to solving more complicated problems.

4.1 Histogramming data

When processing large volumes of data it is often very useful to accumulate data values in histograms. Histograms provide an important tool for observing fluctuations in a data sample. For example, a histogram could be used to find a mass peak above a flat background.

Example 10 is a program to histogram the output of two random number generators. Its design and implementation are discussed in the following sub-sections.

Design

Requirements A program to histogram random numbers generated according to uniform and Gaussian distributions. The program should provide these random numbers by using the integer random number generator `rand` from `stdlib.h`. The histrogramming code should provide visual output using `gnuplot`.

Components

- `main` implement in `main.c`: create two histograms and fill them with 10000 uniform and Gaussian random numbers respectively. Display the results with `gnuplot`.
- Provide histogram functionality.
  - `hist_create` implement in `histogram.c`: a function to create a histogram
  - `hist_book` implement in `histogram.c`: a function to add a value to an existing histogram
  - `hist_plot` implement in `histogram.c`: a function to plot a histogram's contents using gnuplot.
  - `hist_entry` define in `histogram.c`: a `struct` to contain the information of each histogram.
  - `gnuplot` implement in `gnuplot.c`: A function to provide an interface with gnuplot by using a system call.
- Provide random number functionality.
  - `set_seed` implement in `random_dist.c`: a function to set the seed of the

random number generator.

- `random_dist_flat` implement in `random_dist.c`: a function to return a floating point random number between 0. and 1.
- `random_dist_gaus` implement in `random_dist.c`: a function to return a floating point random number following a Gaussian distribution with a given sigma.

Discussion of the implementation

The implementation of example 9 is described in Pseudocode 14, 15, and 16.

```
main()
  Create two histograms.
  Loop for 10000
    Generate a random number following a Gaussian distribution.
    Histogram the Gaussian random number.
    Generate a random number following a uniform distribution.
    Histogram the uniform random number.
  Plot both histograms using gnuplot
```

Pseudocode 14: Example 10 in pseudocode

The program starts by creating two histograms.

```
hist_create("Gaus",20,-3.0,3.0);
hist_create("Flat",20,0.0,1.0);
```

The `hist_create` function assigns the histogram information to members of an element of the `h_dat` array. The variable `num_hist` is used to keep track of which elements of `h_dat` array have been filled. Both `h_dat` and `num_hist` are globally available within `histogram.c`.

```
static int num_hist = 0;
static hist_entry h_dat[MAX_HIST];
```

The `static` prefix means that no function outside `histogram.c` is able to access these variables. Without the `static` prefix the variables could be accessed by a function implemented outside `histogram.c` provided an `external` instantiation was used. An example of an `external` instantiation is given in example 8.

The advantage of instantiating `h_dat` and `num_hist` globally in `histogram.c` is that they do not go out of scope. Therefore every time one of the histogram functions is called `h_dat` and `num_hist` are still in memory.

The variable `h_dat` is an array of `typedef struct hist_entry`, which is defined in `histogram.c` rather than in a header file. The reason for this is that `hist_entry` is

```
hist_create(histogram name, number of bins, lower limit, upper limit)
  Fill static memory with histogram information.
  Calculate bin size and store it in static memory.
  Initialise all bins to be 0.
  RETURN this histogram's index

hist_book(histogram index, value, weight)
  IF the value is less than the lower limit increment the underflow
    bin by the weight.
  ELSE IF the value is greater or equal to the upper limit increment
    the overflow bin by the weight.
  ELSE find which bin the value is inside and increment its value by
    the weight.

hist_plot(histogram index)
  Create a temporary file for gnuplot
  Open the temporary file
  Save the data as "<value> <value>" to the file.
  Assemble a gnuplot command to plot the histogram.
  Call gnuplot.
  Remove the temporary file.
```

Pseudocode 15: Example 10 in pseudocode

intended just for storing histogram information. When writing a large program it is highly advisable to break it down into building blocks. In this program the histogram functions and the associated data are one building block.

Once the histograms have been created a Gaussian random number is generated by calling `random_dist_gaus()`. When this function is called it either generates two random numbers and returns one, or uses the spare one stored from the last time it was called. To store the spare random number `random_dist_gaus()` uses two static variables:

```
static int random_dist_gaus_status = 0;
```

and

```
static double spare_num;
```

The variable `random_dist_gaus_status` is defined outside the function `random_dist_gaus()` as `static`. It is therefore private to the functions within `random_dist.c`. The variable `spare_num` is defined as `static` within the function `random_dist_gaus()`. Unlike normal local variables the variable `spare_num` is defined once in memory and does not go out of scope when the program leaves the `random_dist_gaus()` function. The value stored in `spare_num` at the end of `random_dist_gaus()` is therefore available when the function is next called.

```
random_dist_flat()
  Generate an integer random number
  Use the limit of the integer to calculate a floating point number
    between 0 and 1.
  RETURN the floating point number.

random_dist_gaus(double sigma)
  IF a spare random number is not present
    Generate two random numbers within the unit circle.
    Use a Box-Muller transformation to get two Gaussian random
      numbers.
    Save one of the random numbers in static memory and RETURN the
      other.
  ELSE
    RETURN the spare random number.
```

Pseudocode 16: Example 10 in pseudocode

There are some program operations that can cause a serious error. For example, if an array index is outside of the arrays memory allocation or if a function is passed a variable outside of the limits allowed. Wherever program structures or function calls are used that might cause such an error they should be protected. In example 10 there is a simple catch to prevent an array index from going out of bounds.

```
if(i<0) {
  fprintf(stderr,"CRITICAL ERROR: something has gone very wrong!\n");
  exit(1);
}
h_dat[h_index].bins[i] += weight;
```

The program should not ever get inside this `if` statement. If it does there is a serious bug in the code. The `exit(1)` function call causes the program to terminate and return 1 to the operating system. This is a rather extreme case and other error prevention code may not need to cause the program to exit. For example, if a section of code is written to calculate the hypotenuse of a right angled triangle, then the code should catch the case where the sides have no length.

```
double a, b=0., c=0.;
double a_sqd; /* a squared */
a_sqd = pow(b,2.0)+pow(c,2.0); /* b^2 + c^2 */
if(a_sqd<=0) { /* Prevent a possible sqrt error */
  a = 0;
}
else {
  a = sqrt(a_sqd);
}
```

After the histograms have been filled `hist_plot` is called to plot this histogram using gnuplot. In example 9 the temporary file was written in the present working directory with a fixed name. This could cause problems were the present working directory can not be written to or when two instances of the program are running in the same directory at the same time. To solve both of these issues it is common that temporary files are written in the `tmp` directory with unique file names. Rather than write a piece of code to produce unique file names the program calls `mkstemp`. This function is part of `stdlib.h` and creates a unique file following the supplied template. The file is left open and a *file descriptor* is returned. A file stream is then opened with this *file descriptor*.

```
tmpfile = fdopen(file_descriptor, "w");
```

Data are then written to the file and the file and file descriptor are closed with `fclose`. *File descriptor*s are low level I/O because they are closer to the underlying operating system.

The function `random_dist_gaus` contains three mathematical functions: `pow`, `sqrt`, and `log`. These functions are all defined in the header file `math.h` and require the library `libm.a` to be linked in to the final executable. `libm.a` is a standard library and is therefore already in the search path for the linker. (If the library was not in the standard search path a `-L<directory name>` would have to be added to the link line.) The `libm.a` library is included in the link step by simply adding `-lm` to the link line.

4.2 Problem

The `problem_03/generator` directory contains a program that generates simulated B meson pairs, using the PYTHIA[5] event generator. The program should be compiled and run as specified in the `README` file to generate at least 5000 events.

Write a program to read the `HEPEVT` event records from the binary file `data/pptobbar_hepevt.dat`. Histogram the transverse momentum (p_{T}) of the B mesons and their proper lifetime τ_{true} as the length $c\tau_{\mathrm{true}}$. Calculate the mean value of $c\tau_{\mathrm{true}}$ for the data sample and compare this with the PDG[4] world averages given in table 1.

Meson	$c\tau$ (μm)
$\mathrm{B}^{\pm}$	491.1
B^{0}	458.7
B_{s}	439

Table 1: A table of the mean lifetimes of B mesons expressed as $c\tau_{true}$, taken from the PDG.

Start by writing down a Pseudocode implementation. Then implement a solution. Remember to comment your code. Marks will be given for pseudocode and implementation.

Background information

The transverse momentum p_T is defined as

$$p_T = \sqrt{p_x^2 + p_y^2}$$

where p_x and p_y are the x and y components of the momentum respectively.

The transverse displacement of the secondary vertex with respect to the primary vertex can be calculated from

$$L_{xy} = \sqrt{v_x^2 + v_y^2}$$

where v_x and v_y are the x and y components of the vertex position respectively.

The true lifetime τ_{true} is related to the displacement of the secondary vertex, as stated in equation 1.

$$c\tau_{\mathrm{true}} = L_{\mathrm{xy}}^{\mathrm{B}} \frac{m_{\mathrm{B}}}{p_{\mathrm{T}}^{B}}, \tag{1}$$

where m_{B} is the mass of the B meson, $p_{\mathrm{T}}^{\mathrm{B}}$ is the transverse momentum of the B meson and $L_{\mathrm{xy}}^{\mathrm{B}}$, is the displacement of the secondary vertex from the primary vertex within the transverse plane.

Hints

Start by reading over `problem/generator/main.c` and its associated `Makefile`.

The `HEPEVT` event record is defined in `include/hepevt.h`.

```
/* Maximum number of particles */
#define NMXHEP 4000

typedef struct {
  int nevhep; /* The event number */
  int nhep; /* The number of particles in the event */
  int isthep[NMXHEP]; /* Particle status code */
  int idhep[NMXHEP]; /* Particle identifier (PDG standard) */
  int jmohep[NMXHEP][2]; /* Mother index/indices ranges */
  int jdahep[NMXHEP][2]; /* Daughter index/indices ranges */
  double phep[NMXHEP][5]; /* Four vector and mass */
  double vhep[NMXHEP][4]; /* Production vertex */
} HEPEVT;
```

Documentation of the `HEPEVT` member variables is given in `include/hepevt.h`. The decay vertex of the B mesons can be obtained by using the first daughter particle's production vertex.

Use the function

```
void read_hepevt(FILE *file_ptr, HEPEVT *hepevt);
```

to read each `HEPEVT` event record. This function is pre-declared in `include/hepevt/hepevt_io.h` and implemented in `lib/libhepevt.a`. To link to `lib/libhepevt.a` start from `lab3/problem/generator/Makefile`.

Use the histogram functions pre-declared in `include/histo/histogram.h` and implemented in `lib/libhisto.a`. Add `-lhisto` to the link line to add this library to the link step. The two histograms can be created by calling `hist_create` twice.

```
hist_create("B pt", 50, 0., 50.);
hist_create("B ctau", 50, 0., 1.);
```

References

[1] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[2] GNU Make. http://www.gnu.org/software/make/manual/make.html.

[3] Gnuplot. http://www.gnuplot.info/.

[4] Particle Data Group. http://pdg.lbl.gov/.

[5] PYTHIA. http://www.thep.lu.se/∼torbjorn/Pythia.html.

[6] Wikipedia definition of Comma-separated values. http://en.wikipedia.org/wiki/Comma-separated_values.

[7] Wikipedia definition of Flowcharts. http://en.wikipedia.org/wiki/Flow_chart.

[8] Wikipedia definition of Pseudocode. http://en.wikipedia.org/wiki/Pseudocode.