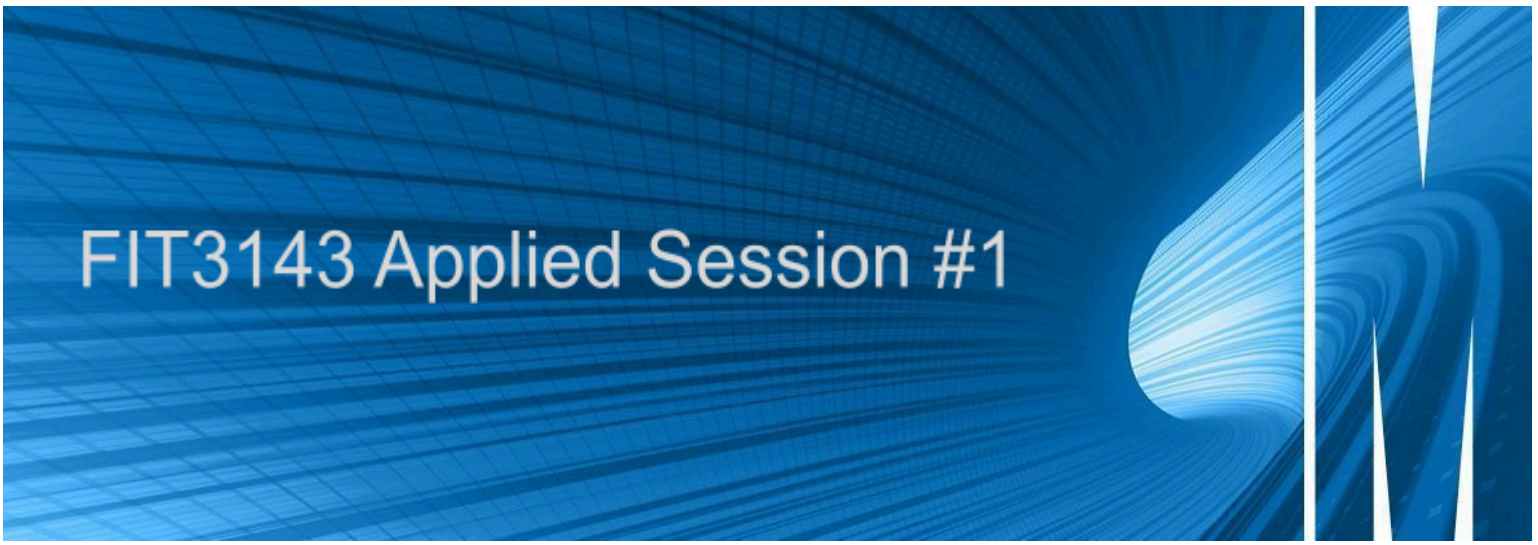
MONASH University



C PRIMER

OBJECTIVES

- The purpose of this session is to introduce you to C Programming Language

MARKS

- The applied session is worth 3 of the unit final mark.

INSTRUCTIONS

- Before class: Even though only a subset of the tasks are assessed, try to attempt all the tasks before class. Make sure you read the marking rubric.
- During class:
    - You will be given 30 minutes to prepare and refine your code & answers for the demonstration period.
    - When you have finished preparing the code & answers, you are required to submit them to Moodle.
    - During the demonstration period, the teaching staff will ask you questions on your submission(s).
- Marks will not be awarded if you skip the class, do not make any submissions, or make an empty submission to Moodle (unless a special consideration extension has been approved).
- You are allowed to use search engines or AI tools to search for information and resources during pre-class preparation. However, search engines and AI tools are not allowed during the code presentation period.
- Always double test all your codes & written answers. Make sure the codes can be run on your laptop.

MONASH University

ASSESSED TASKS

- Task 3 (20%)
- Task 5 (20%)
- Task 7 (20%)
- Task 11 (20%)
- Task 13 (20%)

MONASH University

ACTIVITIES

Task 0 – Preparation [for students missing Week 1] (No Marks)

a. Read the Linux Environment Setup Guidelines
b. Download and set up the Linux environment (with Docker if needed)
c. Get familiar with the Linux environment
d. Confirm gcc and Open MPI are available within your docker environment (for future labs/applied sessions)

MONASH University

Task 1 – C Basic - Hello World in C (No Marks)

Type the following program with your preferred text editor.

```
#include <stdio.h>


int main() {
  printf("\nHello World\n");
  return (0);
}
```

Save the code in the file hello.c, then compile it by typing:

```
gcc hello.c
```

This creates an *executable* file a.out. You can then execute simply by typing its name (assume you are in the same folder of a.out).

```
./a.out
```

The result is that the characters “Hello World” are printed out, preceded by an empty line.

A C program contains *functions* and *variables*. The functions specify the tasks to be performed by the program. The “main” function establishes the overall logic of the code. It is normally kept short and calls different functions to perform the necessary sub-tasks. All C codes must have a “main” function.

Our hello.c code calls printf, an output function from the I/O (input/output) library (defined in the file stdio.h). These functions are performed by standard libraries, which are part of ANSI C. The K & R textbook lists the content of these and other standard libraries in an appendix.

The printf line prints the message “Hello World” on “stdout” (the output stream corresponding to the Terminal window in which you run the code); “\n” prints a “new line” character, which brings the cursor onto the next line. By construction, printf never inserts this character on its own. Try leaving out the “\n” lines and see what happens.

The first statement “`#include <stdio.h>`” includes a specification of the C I/O library. All variables in C must be explicitly defined before use: the “.h” files are by convention “header files” which contain definitions of variables and functions necessary for the functioning of a program, whether it be in a user-written section of code, or as part of the standard C libaries. The directive “#include” tells the C compiler to insert the contents of the specified file at that point in the code. The “< ...>“ notation instructs the compiler to look for the file in certain “standard” system directories.

The int preceding “main” indicates that main is of “int” type--that is, it has *integer* type associated with it, meaning that it will return a result of an integer type (‘0’ in the above example) on execution. Typing $? Immediately after executing this code will display the return value of ‘0’.

The “;” denotes the end of a statement. Blocks of statements are put in braces {...}, as in the definition of functions. All C statements are defined in free format, i.e., with no specified layout or column assignment. Whitespace (tabs or spaces) is never significant, except inside quotes as part of a character string. The following program would produce exactly the same result as our earlier example:

```
#include < stdio.h>
void main(){printf("\nHello World\n");}
```

The reason for arranging your programs in lines and indenting is to show structure.

MONASH University

Task 2 – C Basic - Sine Function Computations (No marks)

The following program, sine.c, computes a table of the sine function for angles between 0 and 360 degrees.

```
/************************/
/*   Table of           */
/*   Sine Function      */
/************************/
/* Michel Vallieres    */
/* Written: Winter 1995 */
#include <stdio.h>

#include <math.h>


int main() {
  int angle_degree;
  double angle_radian, pi, value;


  /* Print a header */
  printf("\nCompute a table of the sine function\n\n");

  /* obtain pi once for all */



/* or just use pi = M_PI, where M_PI is defined in math.h  */
  pi = 4.0 * atan(1.0);
  printf(" Value of PI = %f \n\n", pi);


  printf(" angle Sine \n");


  angle_degree = 0; /* initial angle value */
  /* scan over angle   */


  while (angle_degree <= 360) /* loop until angle_degree > 360 */
 {
    angle_radian = pi * angle_degree / 180.0;
    value = sin(angle_radian);
    printf(" %3d  %f \n ", angle_degree, value);
```

```
    angle_degree = angle_degree + 10; /* increment the loop index
      */
  }
  return (0);
}
```

The code starts with a series of comments indicating its purpose, as well as its author. Comments can be written anywhere in the code: any characters between /* and */ are ignored by the compiler and can be used to make the code easier to understand. The use of variable names that are meaningful within the context of the problem is also a good idea. The #include statements now also include the header file for the standard mathematics library math.h. This statement is needed to define the calls to the trigonometric functions atan and sin. Note that the compilation should include the mathematics library explicitly by typing

gcc sine.c –lm

Variable names are arbitrary (with some compiler-defined maximum length, typically 32 characters). C uses the following standard variable types:

```
int-> integer variable
short  -> short integer
long    -> long integer
float  -> single precision real (floating point) variable
double -> double precision real (floating point) variable
char   -> character variable (single byte)
```

The compiler checks for consistency in the types of all variables used in any code. This feature is intended to prevent mistakes, in particular in mistyping variable names. Calculations done in the math library routines are usually done in double precision arithmetic (64 bits on most workstations). The actual number of bytes used in the internal storage of these data types depends on the machine being used.

The printf function can be instructed to print integers, floats and strings properly.

```
The general syntax is
     printf( "format", variables );
```

where "format" specifies the conversion specification and variables is a list of quantities to print. Some useful formats are

%.nd integer (optional n = number of columns; if 0, pad with zeros)

m.nf float or double (optional m = number of columns, n = number of decimal places)
%ns string (optional n = number of columns)

%c character

\n \t to introduce new line or tab

\g ring the bell (“beep”) on the terminal

MONASH University

Task 3 – C Basic - Loops (20%)

Most real programs contain some construct that loops within the program, performing repetitive actions on a stream of data or a region of memory. There are several ways to loop in C. Two of the most common are the while loop:

```
while (expression) {
  ...block of statements to execute...
}
```

and the for loop:

```
for (expression_1; expression_2; expression_3)
{
 ...block of statements to execute...
}
```

The while loop continues to loop until the conditional expression becomes false. The condition is tested upon entering the loop. Any logical construction (see below for a list) can be used in this context.

The for loop is a special case, and is equivalent to the following while loop:

```
expression_1;
while (expression_2)
{
 ...block of statements...
 expression_3;
}
```

For instance, the following structure is often encountered:

```
i = initial_i;
while (i <= i_max)
{
      ...block of statements...
      i = i + i_increment;
}
```

This structure may be rewritten in the easier syntax of the for loop as:

```
for (i = initial_i; i <= i_max; i = i + i_increment)
{ ...block of statements... }
```

Infinite loops are possible (e.g. for(;;)). C permits you to write an infinite loop, and provides the break statement to "breakout " of the loop. For example, consider the following (admittedly not-so-clean) re-write of the previous loop:

```
angle_degree = 0;
for ( ; ; )
{
 ...block of statements...
 angle_degree = angle_degree + 10;
 if (angle_degree == 360) break;
}
```

The conditional if simply asks whether angle_degree is equal to 360 or not; if yes, the loop is stopped.

Write your own C codes containing two different types of loops.

Submit the file as loop.c.

MONASH University

Task 4 – C Basic - **Symbolic Constants (No marks)**

You can define constants of any type by using the #define compiler directive. Its syntax is simple--for instance

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

would define ANGLE_MIN and ANGLE_MAX to the values 0 and 360, respectively. C distinguishes between lowercase and uppercase letters in variable names. It is customary to use capital letters in defining global constants.

MONASH University

Task 5 – C Basic - Conditionals (20%)

Conditionals are used within the if and while constructs:

```
if (conditional_1)
{
 ...block of statements executed if conditional_1 is true...
}
else if (conditional_2)
{
 ...block of statements executed if conditional_2 is true...
}
else
{
 ...block of statements executed otherwise...
}
```

and any variant that derives from it, either by omitting branches or by including nested conditionals.

Conditionals are logical operations involving comparison of quantities (of the same type) using the conditional operators:

```
 <  smaller than
 <= smaller than or equal to
 == equal to
 != not equal to


 >= greater than or equal to
 >  greater than
```

and the boolean operators

```
 && and
 || or
 !  not
```

Another conditional use is in the switch construct:

```
switch (expression)
{
 case const_expression_1:
 {
  ...block of statements...
  break;
 }
```

```
    case const_expression_2:
    {
     ...block of statements...
     break;
    }
    default:
    {
        ...block of statements..
    }
   }
```

The appropriate block of statements is executed according to the value of the expression, compared with the constant expressions in the case statement. The break statements insure that the statements in the cases following the chosen one will not be executed. If you would want to execute these statements, then you would leave out the break statements. This construct is particularly useful in handling input variables.

Write your own C codes containing at least two types of conditionals.
Submit the file as conditionals.c.

MONASH University

Task 6 – C Basic – Character Arrays (No marks)

A *string constant*, such as
"I am a string"

is an array of characters. It is represented internally in C by the ASCII characters in the string, i.e., “I”, blank, “a”, “m”,... for the above string, and terminated by the special null character “\0” so programs can find the end of the string.

String constants are often used in making the output of code intelligible using printf ;

```
printf("Hello, world\n");
printf("The value of a is: %f\n", a);
```

String constants can be associated with variables. C provides the char type variable, which can contain one character--1 byte--at a time. A character string is stored in an array of character type, one ASCII character per location. Never forget that, since strings are conventionally terminated by the null character “\0”, we require *one extra storage location* in the array!

C does not provide any operator, which manipulate entire strings at once. Strings are manipulated either via pointers or via special routines available from the standard *string* library string.h. Using character pointers is relatively easy since the name of an array is just a pointer to its first element. Consider the following code:

```
#include <stdio.h>
int main()
{
     char text_1[100], text_2[100], text_3[100];
     char *ta, *tb;
     int i;
/* set message to be an array */
/* of characters; initialize it */
/* to the constant string "..." */
/* let the compiler decide on    */
/* its size by using []*/

     char message[] = "Hello, I am a string; what are you?";

     printf("Original message: %s\n", message);

/* copy the message to text_1    */
/* the hard way   */
     i=0;
while ( (text_1[i] = message[i]) != '\0' )
```

```
    i++;
     printf("Text_1: %s\n", text_1);



     /* use explicit pointer arithmetic */
     ta=message;
     tb=text_2;
     while ( ( *tb++ = *ta++ ) != '\0' )
                                ;
     printf("Text_2: %s\n", text_2);


return(0);
}
```

The standard “string” library contains many useful functions to manipulate strings; a description of this library can be found in an appendix of the K & R textbook. Some of the most useful functions are:

```
char *strcpy(s,ct)-> copy ct into s, including "\0"; return s char
*strncpy(s,ct,n)  -> copy ncharcater of ct into s, return s char
*strncat(s,ct)    -> concatenate ct to end of s; return s char
*strncat(s,ct,n)       -> concatenate n character of ct to end
                             of s, terminate with "\0"; return s int
strcmp(cs,ct)-> compare cs and ct; return 0 if cs=ct,
                              <0 if cs0 if cs>ct
char *strchr(cs,c)-> return pointer to first occurrence of c
                             in cs or NULL if not encountered
size_t strlen(cs) -> return length of cs (s and t are char*, cs
and ct are const char*, c is a char converted to type int, and n
is an int.)
```

Consider the following code, which uses some of these functions:

```
#include <stdio.h>
#include <string.h>
int main()
{
     char line[100], *sub_text;
                          /* initialize string */
     strcpy(line,"hello, I am a string;");
     printf("Line: %s\n", line);
                          /* add to end of string */
     strcat(line," what are you?");
```

```
        printf("Line: %s\n", line);
                            /* find length of string */
                            /* strlen brings back  */
                            /* length as type size_t */


        printf("Length of line: %d\n", (int)strlen(line));


                        /* find occurrence of substrings */
        if ( (sub_text = strchr ( line, 'W' ) )!= NULL )
               printf("String starting with \"W\" ->%s\n", sub_text);


if ( ( sub_text = strchr ( line, 'w' ) )!= NULL ) printf("String
       starting with \"w\" ->%s\n", sub_text);

if ( ( sub_text = strchr ( sub_text, 'u' ) )!= NULL ) printf("String
       starting with \"w\" ->%s\n", sub_text);
  return(0);
  }
```

MONASH University

Task 7 – C Input/Output – Character I/O (20%)

C provides (through its libraries) a variety of I/O routines. At the character level, getchar() reads one character at a time from stdin, while putchar() writes one character at a time to stdout. For example, consider

```
#include <stdio.h>

int main()
{
    int i, nc;


    nc = 0;
    i = getchar();
    while (i != EOF) {
     nc = nc + 1;



     i = getchar();
    }
    printf("Number of characters in file = %d\n", nc);
return(0);
}
```

This program counts the number of characters in the input stream (e.g. in a file piped into it at execution time). The code reads characters (whatever they may be) from stdin (the keyboard), uses stdout (the terminal you run from) for output, and writes error messages to stderr (usually also your terminal). These streams are always defined at run time. EOF is a special return value, defined in stdio.h, returned by getchar() when it encounters an *end-of-file* marker when reading. Its value is computer dependent, but the C compiler hides this fact from the user by defining the variable EOF. Thus the program reads characters from stdin and keeps adding to the counter nc, until it encounters the “end of file”.

An experienced C programmer would probably code this example as:

```
#include <stdio.h>

int main()
{
    int c, nc = 0;

    while ( (c = getchar()) != EOF ) nc++;

    printf("Number of characters in file = %d\n", nc);
```

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

C allows great brevity of expression, usually at the expense of readability!

The () in the statement (c = getchar()) says to execute the call to getchar() and assign the result to c before comparing it to EOF; the brackets are necessary here. Recall that nc++ (and, in fact, also ++nc) is another way of writing nc = nc + 1. (The difference between the prefix and postfix notation is that in ++nc, nc is incremented before it is used, while in nc++, nc is used before it is incremented. In this particular example, either would do.) This notation is more compact (not always an advantage, mind you), and it is often more efficiently coded by the compiler.

The UNIX command wc counts the characters, words and lines in a file. The program above can be considered as your own wc. Let's_ add a counter for the lines.

```
#include <stdio.h>



int main()
{
    int c, nc = 0, nl = 0;

    while ( (c = getchar()) != EOF )
    {
     nc++;
     if (c == '\n') nl++;
    }

    printf("Number of characters = %d, number of lines = %d\n",
nc, nl);
return(0);
}
```

Note: CTRL-D will act as EOF character.

Can you think of a way to count the number of words in a text file? Write down your answers and submit as a file (Task7.txt) to Moodle.

MONASH University

Task 8 – C Input/Output – Higher-level I/O (No marks)

We have already seen that printf handles formatted output to stdout. The counterpart statement for reading from stdin is scanf. The syntax

scanf("format string", variables);

resembles that of printf. The format string may contain blanks or tabs (ignored), ordinary ASCII characters, which must match those in stdin, and conversion specifications as in printf.

Equivalent statements exist to read from or write to character strings. They are:

sprintf(string, "format string", variables);

scanf(string, "format string", variables);

The “string” argument is the name of (i.e. a pointer to) the character array into which you want to write the information.

MONASH University

Task 9 – C Input/Output – File I/O (No marks)

Similar statements also exist for handling I/O to and from files. The statements are

```
#include <stdio.h>

    FILE *fp;
    fp = fopen(name, mode);
    fscanf(fp, "format string", variable list);
    fprintf(fp, "format string", variable list);
    fclose(fp );
```

The logic here is that the code must

define a local “pointer” of type FILE (note that the uppercase is necessary here), which is defined in < stdio.h>

“open” the file and associate it with the local pointer via fopen perform the I/O operations using fscanf and fprintf
disconnect the file from the task with fclose

The “mode” argument in the fopen specifies the purpose/positioning in opening the file: “r” for reading, “w” for writing, and “a” for appending to the file. Try the following:

```
#include <stdio.h>

int main()
{
    FILE *fp;
    int i;

    fp = fopen("foo.dat", "w");  /* open foo.dat for writing
*/

    fprintf(fp, "\nSample Code\n\n"); /* write some info */
    for (i = 1; i <= 10 ; i++)
     fprintf(fp, "i = %d\n", i);

    fclose(fp);   /* close the file */
return(0);
}
```

Compile and run this code; then use any editor or “cat” to read the file foo.dat.

MONASH University

Task 10 – C Functions - Calling a function (No marks)

Functions are easy to use; they allow complicated programs to be parcelled up into small blocks, each of which is easier to write, read, and maintain. We have already encountered the function main and made use of I/O and mathematical routines from the standard libraries. Now let's look at some other library functions, and how to write and use our own.

The call to a function in C simply entails referencing its name with the appropriate arguments. The C compiler checks for compatibility between the arguments in the calling sequence and the definition of the function.

Library functions are generally not available to us in source form. Argument type checking is accomplished through the use of header files (like stdio.h), which contain all the necessary information. For example, as we saw earlier, in order to use the standard mathematical library you must include math.h via the statement

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

at the top of the file containing your code. The most commonly used header files are:

```
<stdio.h>  -> defining I/O routines
<ctype.h>  -> defining character manipulation routines
<string.h> -> defining string manipulation routines
<math.h>-> defining mathematical routines
<stdlib.h> -> defining number conversion, storage allocation and
similar tasks
g.h> -> defining libraries to handle routines with variable numbers of
        arguments
        <time.h>-> defining time-manipulation routines

        In addition, the following header files exist:
        <assert.h> -> defining diagnostic routines
        <setjmp.h> -> defining non-local function calls
        <signal.h> -> defining signal handlers
        <limits.h> -> defining constants of the int type

        <float.h> -> defining constants of the float type
```

Appendix B in the K & R book describes these libraries in great detail.

MONASH University

Task 11 – C Functions - Writing a function (20%)

A function has the following layout:

return-type function-name (argument-list-if-necessary)

```
{
    ...local-declarations...

    ...statements...

    return return-value;
}
```

If return-type is omitted, C defaults to int. The return-value must be of the declared type.

A function may simply perform a task without returning any value, in which case it has the following layout:

```
void function-name ( argument-list-if-necessary )
{
    ...local-declarations...

    ...statements...
}
```

As an example of function calls, consider the following code:

```
/* include headers of library */
/* defined for all routines   */
/* in the file */
#include <stdio.h>
#include <string.h>
/* prototyping of functions   */
/* to allow type checks by    */
/* the compiler     */

int main()
{
    int n;
    char string[50];
   /* strcpy(a,b) copies string b into a */
```

```
    /* defined via the stdio.h header  */ strcpy(string, "Hello
World");

    /* call own function */
     n = n_char(string);
     printf("Length of string = %d\n", n);
  return(0);
  }


    /* definition of local function n_char */
  int n_char(char string[])
  {

    /* local variable in this function */

     int n;

    /* strlen(a) returns the length of */
    /* string a    */
    /* defined via the string.h header */

     n = strlen(string);
     if (n > 50)
      printf("String is longer than 50 characters\n");


         /* return the value of integer n */
     return n;
  }
```

Arguments are always passed *by value* in C function calls. This means that local "copies" of the values of the arguments are passed to the routines. Any changes made to the arguments internally in the function are made only to the local copies of the arguments. In order to change (or define) an argument in the argument list, this argument must be passed as an address, thereby forcing C to change the "real" argument in the calling routine.

As an example, consider exchanging two numbers between variables. First let's illustrate what happen if the variables are passed by value:

```
#include <stdio.h>

void exchange(int a, int b);
```

```
int main()
{   /* WRONG CODE */
    int a, b;

    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);

    exchange(a, b);
    printf("Back in main: ");
    printf("a = %d, b = %d\n", a, b);
return(0);
}

void exchange(int a, int b)
{
    int temp;

    temp = a; a = b;
    b = temp;
    printf(" From function exchange: ");
    printf("a = %d, b = %d\n", a, b);
}
```

Run this code and observe that a and b are NOT exchanged! Only the copies of the arguments are exchanged. The RIGHT way to do this is of course to use pointers:

```
#include <stdio.h>
void exchange ( int *a, int *b );

int main()
{   /* RIGHT CODE */
    int a, b;

    a = 5;
    b = 7;
    printf("From main: a = %d, b = %d\n", a, b);

    exchange(&a, &b);
```

```
    printf("Back in main: ");
    printf("a = %d, b = %d\n", a, b);
return(0);
}


void exchange ( int *a, int *b )
{
    int temp;


    temp = *a;
    *a = *b;
    *b = temp;
    printf(" From function exchange: ");
    printf("a = %d, b = %d\n", *a, *b);
}
```

The rule of thumb here is that

You use regular variables if the function does not change the values of those arguments
You MUST use pointers if the function changes the values of those arguments

Write your own C codes defining at least two functions that use pointers as arguments. Submit the file as functions.c.

MONASH University

Task 12 – Command Line Arguments - (No marks)

The full statement of the first line of the C program is

main(int argc, char** argv)

The syntax char** argv declares argv to be a pointer to a pointer to a character, that is, a pointer to a character array (a character string) – in other words, an array of character strings. You could also write this as char* argv[].

When you run a program, the array argv contains, in order, *all* the information on the command line when you entered the command (strings are delineated by whitespace), *including the command itself*. The integer argc gives the total number of strings, and is therefore equal to equal to the number of arguments *plus one*. For example, if you typed

a.out -i 2 -g -x 3 4

the program would receive

```
argc = 7
argv[0] = "a.out"
argv[1] = "-i"
argv[2] = "2"
argv[3] = "-g"
argv[4] = "-x"
argv[5] = "3"
argv[6] = "4"
```

Note that the arguments, even the numeric ones, are all *strings* at this point. It is the programmer's job to decode them and decide what to do with them.

The following program simply prints out its own name and arguments:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int i;
    printf("argc = %d\n", argc);

    for (i = 0; i < argc; i++)
```

```
     printf("argv[%d] = \"%s\"\n", i, argv[i]);

return(0);
}
```

UNIX programmers have certain conventions about how to interpret the argument list. They are by no means mandatory, but it will make your program easier for others to use and understand if you stick to them. First, switches and key terms are always preceded by a “-” character. This makes them easy to recognize as you loop through the argument list. Then, depending on the switch, the next arguments may contain information to be interpreted as integers, floats, or just kept as character strings. With these conventions, the most common way to “parse” the argument list is with a for loop and a switch statement.

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

/* sample usage a.out –a 100 */


int main(int argc, char** argv)
{
    /* Set defaults for all parameters: */


    int a_value = 0;
    float b_value = 0.0;
    char* c_value = NULL;
    int d1_value = 0, d2_value = 0;


    int i;


    /* Start at i = 1 to skip the command name. */
    for (i = 1; i < argc; i++) {
     /* Check for a switch (leading "-"). */
     if (argv[i][0] == '-') {
         /* Use the next character to decide what to do. */
         switch (argv[i][1]) {
case 'a': a_value = atoi(argv[++i]);
          break;
case 'b': b_value = atof(argv[++i]);
          break;
          case 'c': c_value = argv[++i];
                    break;
case 'd': d1_value = atoi(argv[++i]);
```

```
            d2_value = atoi(argv[++i]);
            break;
           }
      }

    }
    printf("a = %d\n", a_value);
    printf("b = %f\n", b_value);
    if (c_value != NULL) printf("c = \"%s\"\n", c_value);
    printf("d1 = %d, d2 = %d\n", d1_value, d2_value);
return(0);
}
```

MONASH University

Task 13 – Practice C Pointers (20%)

C and C++ provide an added feature for accessing the memory location in the computer directly by using pointers. The pointers as their name suggest point to either the address of a variable or the value stored at the address of the variable. The pointers are implemented by the reference and dereference operators.

Reference Operator (&) is used to find the address of a variable. Dereference Operator (*) is used to find the value stored at the address of the variable.

The following C code demonstrates the use of these operators. You may cut and paste this code in a text editor, compile and execute to see the outputs:

```
gcc filename.c - o filename.extension

#include <stdio.h>

int main() {
  int AValue, * BValue;
  /* See what is stored in AValue at present*/
  printf("AValue=%d \n", AValue);
  /* Set the Value of AValue to what we want */
  AValue = 101;

  /* Print the memory address where AValue is stored and the
value*/
  /* by using the "&" Reference operator*/
  printf("Address of AValue %p AValue= %d \n", & AValue, AValue);
  /* Save the address of AValue at a separate location*/
  BValue = & AValue;
  /* Print the BValue , which is the address of AValue */
  /* Dereference BValue with "*" operator and print actual Value*/
  printf("Address of BValue %p BValue= %d \n", BValue, * BValue);
  /* Note output will show BValue=101 (it was never explicitly set
to 101) */
}
```

Sample output:

```
AValue = 0
Address of AValue 0x7fff5fbff6bc AValue = 101
Address of BValue 0x7fff5fbff6bc BValue = 101
```

Why would '*BValue' returns '101', without being set to this value explicitly anywhere within the sample code? Write down your answers and submit as a file (Task13.txt) to Moodle.