

Programming Methodology Notes

Anupama Potluri
School of Computer and Information Sciences
University of Hyderabad

August 20, 2020

Contents

1	Introduction	4
2	Specification	4
3	Design of Algorithms	5
3.1	Algorithmic Primitives	6
4	“Finding the Maximum” problem	7
4.1	Specification 1	7
4.2	Design 1	8
4.3	Specification 2	8
4.4	Design 2	9
4.5	Specification 3	9
4.6	Design 3	10
5	“Finding the Mode” problem	11
5.1	Mode: Specification 1	11
5.2	Mode: Design 1	11
5.3	Mode: Specification 2	11
5.4	Mode: Design 2	11
5.5	Mode: Specification 3	11
5.6	Mode: Design 3	12
5.7	Mode: Specification 4	12
5.8	Mode: Design 4	12
6	Set Operations Problem	12
7	Testing and Debugging	12
7.1	Unit Testing	13
7.2	Integration Testing	13
7.3	Code Coverage	13
7.4	Design of Test Cases	14
8	C language discussion	14
8.1	Declaration versus Definition	15
8.2	Global Variables and Side Effects	16
8.3	Scope and Extent of Variables	17
8.4	Parameter Passing Mechanisms	19
8.5	File Operations	20
8.6	Pointers	21
8.7	Bit Manipulations	24
8.8	Pre-processor directives	26
8.9	Command Line Arguments	28
8.10	typedef and union	29
8.10.1	typedef	30
8.10.2	union	30

9	Some Common Compiler Errors and What They Mean	31
9.1	warning: implicit declaration of function printf	31
9.2	warning: val may be used uninitialized in this function	31
9.3	sqrt_max.c:7:1: error: expected , or ; before int	32
9.4	error: i undeclared	32
9.5	sqrt_max.c:(.text+0x6a): undefined reference to 'sqrt'	33
10	Coding Standards	33
10.1	Meaningful Names	33
10.2	Constants/Macros	34
10.3	Column width of 75-78	34
10.4	Indentation	34
10.5	Comments	34
10.6	Good Parenthesisation	34
10.7	Braces for Functions vs Other Compound statements	35
10.8	Block Coding	35

1 Introduction

Programming is primarily about solving problems. It is about capturing the process of how the human brain solves the problem and writing it in terms that the computer can understand. Unlike human beings, computers have to be told every small detail – not even a small child needs to be told in as much detail as a computer needs to be. That is what makes it challenging because humans make many hidden assumptions that are not easy for them to assess and capture them in explicit statements. But, when programming is approached in this fashion, i.e., as a process of our own brain examining itself, it can be a lot of fun.

Programming starts with somebody asking us to make the computer do a task. An example would be “*Find the maximum of the given set of numbers*”. Sounds simple to a human and most people would do it, including pretty small children. The usual response of humans is to start listening for numbers or see if somebody writes the numbers on the board. They know when the person has stopped giving them numbers by the cues of conversation such as a long enough pause in speech or the person stops writing. Now, the computer, in typical scenarios (if we ignore all the new-fangled AI personal assistants) cannot see or hear. The input is given to them quite differently. And, the problem starts here. How do we give the input to the computer? So, the statement above is not enough to start instructing the computer how to solve the problem. Much more information is needed. The process of extracting all the relevant information is therefore the first step in solving problems by computer. The output of this extraction is a **Specification**.

Once the relevant information is all available, the human captures the logic or functioning of their brain in a series of logical steps. These steps should result in the correct answer under all normal and abnormal conditions. The steps should also be done at some point, i.e., the computer must halt its computation at some point when following these steps. Such a series of steps is called an **Algorithm**. An algorithm is independent of the computer hardware, the operating system and the computer language in which it may finally be executed in the computer. This abstraction of details specific to particular environments make it a powerful tool to solve the problem without getting lost in nitty gritty details. This process is called **Design** and is at the heart of programming.

When designing solutions to problems, users have to think through all possible conditions, especially error conditions and what are called *boundary* conditions. Most solutions fail under these conditions. The design process needs to give emphasis to these conditions.

Once design is done, the programmer needs to come up with test cases that will be used to verify if the algorithm is correct or not. Based on these test cases, the algorithm is traced with different inputs. If the algorithm gives correct results in all the test cases, then, we can proceed to implement the algorithm in a specific language on a specific system. Of course, the tracing may not be possible for complex systems but we will see how even for reasonable systems, these steps can be done without undue strain.

2 Specification

Let us go back to the problem posed in the previous section: “*Find the maximum of the given set of numbers*”. This is called a **Requirement**.

What is missing in this statement for a computer? How do we go about finding the missing information? Are there ambiguities in the question? If so, what are they?

Disambiguating the requirement and ensuring all the necessary information without any hidden assumptions is obtained, as we said earlier is the first step. This is written down in clear terms and is called **Specification**. In other words, a specification consists of unambiguously identifying **all** the input to the computer and **how and in what format** it is given and similarly for the output – **all the output for all possible conditions and in what format is it given**.

Let us see what information is missing in the above requirement. As we noted earlier, a human knows how the numbers are being given and when no more numbers are forthcoming from other cues. For the computer, the numbers can be given through a keyboard or they may be read from a file which is on a specific device such as a hard disk, a CD or a pen drive or it may come over the network etc. We need to inform the computer where from the data needs to be read. If it is a file or a network, there may be ways for the computer to determine that it has reached the end of the input. But, if we are typing the data with the keyboard, there needs to be some signal to the computer that the data has all been given.

In our example, we may say that the input is given from the keyboard, that when a value of -1 is entered, it means end of input. Here, there is a hidden assumption that no negative values (or at least -1) are part of the input data whose maximum we have to find. On the other hand, we may say that first we will give an input of how many numbers will be entered and then enter that many numbers. So, we first enter a value N followed by N values. We return the maximum of the numbers entered at the end.

Examining the problem further, we can ask ourselves if the data consists of duplicates. We can ask if it matters if there are duplicates? It does if the user is expecting, but does not specify, that the output will print the location of the maximum. If the location needs to be printed, then the question is whether to print all locations, only the first occurrence or only the last occurrence. Based on the answer to these questions, the way we write our logic changes. Thus, you can see that a specification has an impact on the design of the algorithm.

3 Design of Algorithms

This is, of course, the heart of problem solving or programming – how to solve the problem. There are five essential qualities for a good program. They are:

1. **Generality:** Make the solution to the problem as general as possible.
2. **Modularity:** This is the process of breaking the given problem into small sub-problems, each of which appears almost trivial but put together, lead to the overall solution to the problem.
3. **Portability:** This relates more to the implementation in a specific computer language than design. This should be written such that the program can be run on any hardware or operating system without modifications.
4. **Readability:** The program should be easily understood when it is read. Therefore, the names used for variables, functions etc. should be meaningful and convey immediately their purpose.
5. **Maintainability:** A readable code with clear comments wherever the logic is complex helps in maintenance of the code. Maintenance is needed whenever some bugs are encountered or new features need to be added to the existing software.

The most important lesson in programming is “*Think First, Code Later*”. There are two methods of problem solving – bottom-up or top-down. Normally, we use a top-down approach, where we take the given problem and break it down into small pieces. Then, we take the small pieces and break them down into even smaller pieces until the piece is easily doable. An e.g. would be finding mode of given set of numbers. We can say that this consists of two steps: find the frequency of each value and then the maximum of the frequencies. Then, each of these sub-problems can be done separately. An example of how to work through the entire problem may be seen in Prof.Chakravarthy’s notes on “How to Program” [4].

3.1 Algorithmic Primitives

The logic of the solution is captured in algorithmic primitives which are independent of any language.

Data stored in the computer can be of two basic types – simple data type or complex data type. A simple data type is either an integer, long integer, float (for real numbers), double (for double precision reals), character. These can be combined in multiple ways to create complex data types. Two such data types are discussed in items 2 and 3 below.

The following are the standard primitives in algorithms:

1. **Variables and Constants:** Variables are those which change value in the course of execution of the program. Constants are those which do not change in value. It is a convention that all constants are named in uppercase letters and variables use all lower case or a combination of upper and lower case characters. It is also important to name these such that their function is immediately apparent from the name. So, if there is a variable which holds the maximum value found so far, it should be name *max* and not *x* or *a* etc. Similarly, a constant for the size of a string may be called *MAX_STR*.
2. **Arrays:** Arrays are contiguous memory spaces where each element of the array has the same data type and can be accessed using the location or index operator []. For example, if we have a variable for a table of integers which is an array, say *Table*, each element of the table can be addressed as follows: *Table*[1], *Table*[2] and so on until the last element. The total size of an array is usually declared using a constant as follows: *Table*[*MAXVAL*].
3. **Structures:** Structures can also be called *Records*. The elements of a structure, unlike the elements of an array, have different data types. For e.g., if we want to store student information, we need to store the *Name* which is a string, *Reg.No.* which can be a string or an integer, *CGPA* which is a float. We can then have arrays of such structures to maintain information of multiple students. Such data types are called complex data types.
4. **Assignment Operator:** Whenever a variable needs to be assigned a value, the assignment operator is used as in the following statement:

$$max \leftarrow num$$
5. **Conditional Statements** A conditional statement is executed only if the condition being tested is TRUE. We can also have alternate statements if the condition is FALSE. An example is the following:

if *N* < 0 **then**

```

    Print "Error: N must be greater than or equal to 0"
  end if
  if  $N = 0$  then
    Print "No input is given"
  end if

```

6. **Logical Operators:** These allow one to combine conditions such as the following:

```

  if  $N > 0$  and  $i < N$  then
     $max \leftarrow Table[i]$ 
  end if

```

7. **Arithmetic Operators:** These are standard arithmetic operator for addition, subtraction, multiplication and division represented by the standard symbols. However, the remainder function is represented using the "%" symbol. Thus $a \% b$ would mean the remainder of a divided by b .

8. **Looping Statements:** There are situations in programs where we want to repeat the same operation or set of statements on multiple data points. We use looping statements for such purposes. An e.g. would be to find the sum of the first N integers. We need to repeat the addition of the current value to a variable called *sum*. This is done as follows:

```

  for  $i := 1$  to  $N$  do
     $sum \leftarrow sum + i$ 
  end for

```

When we need to repeat as long as a condition is TRUE, we use the **while** loop construct. Thus if we need to read values and add them up until a negative value is entered, we write it as follows:

```

  Read  $num$ 
   $sum \leftarrow 0$ 
  while  $num > 0$  do
     $sum \leftarrow sum + num$ 
  end while

```

4 “Finding the Maximum” problem

In this section, we will look at different specifications for finding the maximum problem and for each such specification, we will write the algorithm. This shows how the specification impacts design even for such a trivial problem.

4.1 Specification 1

We find maximum of N numbers.

There are no duplicates in the given values.

Only the maximum value needs to be given as output.

If any value other than a number is given as input, it should print an error string “Error in giving input...must be a number”.

If no values are given ($N = 0$), it should print a message “No values have been given”.

If $N < 0$, it should print the message “ N must be > 0 ”.

Based on this specification, we write the algorithm in the next section.

4.2 Design 1

The algorithm for the specification in 4.1 is given in Algorithm 1.

```
Read  $N$ 
if  $N < 0$  then
    Print “ $N$  must be  $> 0$ ”
    return
end if
if  $N = 0$  then
    Print “No input to find maximum”
end if
Read  $max$ 
if  $max$  not an integer then
    Print “Wrong Input”
end if
for  $i \leftarrow 1$  to  $N - 1$  do
    Read  $val$ 
    if  $val$  not an integer then
        Print “Wrong Input”
    end if
    if  $val > max$  then
         $max \leftarrow val$ 
    end if
end for
Print “Maximum of values input = ”  $max$ 
return
```

Algorithm 1: Algorithm for “Finding Maximum” with Specification 1

4.3 Specification 2

We find maximum of numbers input where the input ends if a negative value is encountered.

There are duplicates in the given values.

The maximum value and the last location it occurs in are to be given as output.

If any value other than a number is given as input, it should print an error string “Error in giving input...must be a number” and exit.

If no values are given (i.e., if the first number is negative), it should print a message “No values have been given”.

4.4 Design 2

The algorithm for the specification in 4.3 is given in Algorithm 2.

```
Read val
if val not an integer then
  Print “Wrong Input”
  return
end if
if val < 0 then
  Print “No input to find maximum”
  return
end if
 $max \leftarrow val$ 
 $loc \leftarrow 1$ 
 $i \leftarrow 1$ 
while val ≥ 0 do
  Read val
  if val not an integer then
    Print “Wrong Input”
    return
  end if
   $i \leftarrow i + 1$ 
  if val ≥ max then
     $max \leftarrow val$ 
     $loc \leftarrow i$ 
  end if
end while
Print “Maximum of values input = ” max
Print “and its last occurrence is at ” loc
return
```

Algorithm 2: Algorithm for “Finding Maximum” with Specification 2

4.5 Specification 3

We find maximum of numbers input where the input ends if a negative value is encountered.

There are duplicates in the given values.

The maximum value and all the locations it occurs in are to be given as output.

If any value other than a number is given as input, it should print an error string “Error in giving input...must be a number”.

If no values are given (first value input is negative), it should print a message “No values have been given”.

4.6 Design 3

This is left as an exercise for the user.

5 “Finding the Mode” problem

This is another illustration of how the specification impacts design. Four different specifications are given with hints on how the design changes. The actual design of the problems is left as an exercise for the user.

5.1 Mode: Specification 1

In this the user is given N integers as input in sorted order. The mode of the given numbers has to be printed as output along with its frequency. If there are multiple values with the same frequency, the last value with the highest frequency is to be printed as the mode. Boundary conditions such as all numbers being unique should be taken care of. In this case, all values are modes with frequency=1. Error conditions on the value of N need to be taken care of as illustrated for the *max* problem.

5.2 Mode: Design 1

In the solution to this problem, we read the values one by one and at the end of reading the input, we should be able to print the mode since the values are in sorted order. There is no need to use arrays or any other complex storage mechanisms to solve this problem.

5.3 Mode: Specification 2

In this problem, the numbers are not in sorted order. However, the values are in the fixed closed interval $[1 \cdots R]$. The data is read from a file unlike in the other cases so far. Therefore, the name of the file needs to be taken as input. Error conditions on accessing the file need to be taken care of. Boundary conditions such as an empty file need to be taken care of. The other part of the specification is as in Specification 1 regarding occurrence of multiple modes in the data etc.

5.4 Mode: Design 2

For solving this problem, we need to maintain an array of dimension R as this is the range of values that can occur. It is a question of finding the frequency of each value and determining the highest frequency. So, it can be thought of as two subproblems – finding the frequency of occurrence of each value and then finding the maximum of the frequencies. However, as in the previous case, the solution can be found in a single pass over the input values. In other words, as the values are read, the mode can be calculated. We don't need to go back and do any additional operations to find the mode.

5.5 Mode: Specification 3

In this problem, we do away with both the sorted values as well as the limited range of values restrictions. However, there is a restriction on how many unique values are present in the input. For e.g., the input may consist of any number of occurrences of only three unique values – 10, 100, 10000 – in any order in the file. We need to print the mode as in the previous examples. The last value encountered which has the highest frequency is the mode. So, if 10 and 10000 have the same frequency but the very last value in the file is 10, then, 10

is the mode as the requirement is that the last occurring value in the file is the mode, not the maximum of equal frequency values. All error and boundary conditions need to be taken care of as usual.

5.6 Mode: Design 3

To solve this problem, we will need to maintain an array of structures. The dimension of the array will be the number of unique values in the input. The structure will consist of the value and its frequency. When a value is read, the structure array has to be walked to find the value and increment the frequency. Once again, this does not require the user to do more than one pass over the input to find the solution.

5.7 Mode: Specification 4

The final specification has no restrictions on input. The value can be in any order, in any range and no restriction on the number of unique values.

5.8 Mode: Design 4

To solve this problem, we will need to maintain a linked list of structures at the very least because now the dimension of an array will be unknown. Therefore, this requires knowledge of data structures to solve the problem.

These two examples should amply demonstrate that specification has an immense impact on the design and choice of data structures for a problem.

6 Set Operations Problem

In this problem, we need to read integers from one or more input files and do various set operations such as find if a value is a member of the set, Union, Intersection, Difference etc. It involves reading integers from files into arrays before set operations are done. Since this is repeated more than once, it is a classic example of how modularity helps.

We can create a separate module for opening and reading from the file including handling all error conditions related to file operations. This module is completely independent of the set operations problem. It can be reused for all other problems that require reading integers from a file into an array. We can then build a module for set operations which take arrays of integers as input for the sets to be operated on.

Each of the modules above can become part of a library we construct for ourselves just as we have standard C library functions. These modules can be reused as long as the prototypes do not change from problem to problem. Therefore, design of the prototypes for such functions can be carefully done to be generalised such that they can be used in multiple problems.

7 Testing and Debugging

Testing and Debugging of programs is an art as well as a science. It is not just giving some random inputs to the program. There are some well-defined rules for how we proceed with testing.

Testing consists of many levels. At the beginners stage that we are discussing there are at least two stages of testing, namely, **unit testing** and **integration testing**. Modularity plays a big role in reducing the complexity and time taken for testing.

7.1 Unit Testing

Taking the example of set operations in Section 6, we pointed out there are two modules – file operations module and a set operations module. Unit testing will mean that we implement the file operations module and then test it by printing out the values of the arrays to ensure that the values match those found in the file(s). Once this is confirmed, we can be pretty sure we have a correct file operations module.

We can then proceed to implementing the set operations. Even in this, we implement only one set operation at a time and test it before proceeding to the other. Thus, at every stage we are ensuring that what we have done so far is correct. At the end, putting it all together is trivial.

This logic of dividing a large problem into smaller sub-problems, design, implement and test each module separately helps in reducing the probability of bugs even in highly complex softwares.

Typically, when doing unit testing, we will need either input from some other module or we will produce output for some other module. However, that module may not yet be ready. In such cases, where we are testing a module which needs input from some other module, we build a dummy function which passes the needed input in the needed format as part of testing. Such a function is called a **driver** software. In other cases, we need a function to which we are passing input and which returns output to us. But, this function is not yet ready. We then build a dummy function which returns meaningful return value/output parameter values in expected format. Such a dummy function is called **stub** software.

7.2 Integration Testing

For Integration testing, we have all modules which have been developed and unit tested using driver and stub software. Therefore, we are confident that individually each module works fine. We put together all modules and test them together as part of integration testing. The functions which belong to a particular module and are called by other modules are called *interfaces* between the modules.

As long as the *interfaces* have been well designed with the input and output parameters clear and return values that convey clearly the success or failure and which failure as needed, integration can be trivial.

However, it must be noted that most of the software fails during integration because these interfaces are usually not well defined. In fact, the very first step of design must be to not only identify the modules but the interfaces between modules. After that, each module can be independently developed by different programmers also without any loss of coherence.

7.3 Code Coverage

When testing the code, it is best that every statement in the program is actually executed as part of testing. The % of statements of a program that are executed when all test cases are combined together is called **code coverage** of the test cases. It seems a trivial statement to make that 100% code coverage will mean that bugs in the program will be minimized. But,

for highly complex software, it is not easy to come up with nor run all possible test cases. Hence, in many cases there remain hidden bugs in software. The duty of a good programmer is to design test cases such that the code coverage is as close to 100% as possible.

7.4 Design of Test Cases

When designing test cases, just as in the case of design, we need to look at the boundary and error conditions. Let us take the “find max” problem in Section 4.3 and see how do you design test cases to understand the process better.

In this specification, there are two exceptions listed – one is that there is no input and the other is that the input value is not what is expected. So, we **MUST** have two test cases for each of these exceptions.

Now, to test the regular condition of no errors, we need to look at the problem. It is a question of finding maximum of given values.

1. A boundary condition would be that only one value is entered. Did we design our code correctly to handle this? Many times such boundary conditions are missed in the design. Such errors are called *off-by-one errors* and are some of the most common errors in software.
2. If we have more than one number, another boundary condition for this problem would be two or more values are input and all of them have the same value since duplicates are allowed. The output in this case should print the last location for the occurrence of the value.
3. Another test case would be more than one number and all values are unique. In this case, there are sub-testcases – the maximum is the first value or some middle value or last value.
4. Another test case is where more than one value is input and there are duplicates of the maximum value. The proper location is printed for the maximum in this case.

As can be seen above, there are many test cases for as simple a problem as “finding the maximum value”. For each such test case, we need to give the proper input which satisfies the conditions specified in the test case. We know what is the expected output and verify it against the output from the program. If the output from the program does not match the expected output then, there is a bug in the program corresponding to that test case.

Designing test cases such that there is maximum code coverage is very important for reliability of software. Once again, as pointed above, modularity comes to the rescue of the programmer. Unit testing is done per module and test case design per module will ensure a much better code coverage than if someone were to test complex software without doing unit testing. In fact, that is the reason why there are many levels of testing in software – to ensure that all test cases are taken care of in a methodical and scientific manner.

8 C language discussion

In this section, we will discuss some of the interesting aspects of the C programming language. For a comprehensive and detailed exposition, the user is directed to read “**The C Programming Language**” by Kernighan and Ritchie.

8.1 Declaration versus Definition

Function Declaration versus Definition Let us first discuss the difference between declaration and definition of functions. A function declaration or prototype specifies only the name of the function, the parameters passed to it in terms of data types and the return value data type. It does NOT include the actual logic of the function. The function definition is where the logic of the function is specified. Declarations may be needed if the definition follows a call by another function or the function is defined in some .c file and called by a function in another .c file. Declarations of functions which are used across .c files are given in .h files. Such .h files are included in all .c files where these functions are called or defined.

Global versus Local Variables A variable which is declared within a function definition is called a **local** variable. A variable which is defined outside the definitions of functions is called a **global** variable. In the example code in Program 1, *max* is a global variable and *i*, *N*, *val* are local variables.

Program 1: Finding Maximum (in a single .c file)

```
#include <stdio.h>

int max = -1;

int main(void)
{
    int      i , val , N;

    /* Read N from the user */
    for (i = 0; i < N; i++) {
        /* Read value from user */
        if (val > max)
            max = val;
    }

    printf("Max = %d\n", max);
    exit(0);
}
```

Variable Declaration versus Definition For a variable, the *Declaration* is specification of the data type of the variable. *Definition* is allocation of storage for the variable.

Program 2: Finding Maximum (main.c file)

```
#include <stdio.h>

#define MAX      1000
#define MAXSTR   128

int i = 0;
int Table[MAX];
```

```

int main(void)
{
    char    filename[MAXSTR];

    /* Open File and check errors */
    while (fread(&Table[i], sizeof(int), 1, fp) != 0) {
        i++;
    }

    printf("Max = %d\n", findmax(i));
    exit(0);
}

```

The same code for finding maximum can be written in two files – *main.c* for reading data from a file into an array as shown in Program 2 and *max.c* to find the maximum of the values in the array shown in Program 3.

The variable *i* is declared and defined as a global variable in *main.c* However, it is only a declaration in *max.c* which means that it informs the compiler that the variable is an integer data type but does not allocate any memory for it. The *extern* keyword also informs the compiler that the definition is external to this file.

Program 3: Finding Maximum (max .c file)

```

#include <stdio.h>

extern int i;
int findmax(int N)
{
    int      max = -MAXINT;

    while ( i < N) {
        if (Table[i] > max)
            max = Table[i];
        i++;
    }

    printf("Max = %d\n", max);
    return max;
}

```

8.2 Global Variables and Side Effects

An observant reader would have noticed a bug in the code given in Programs 3 and 2. The variable *i* is used in *main.c* to read data from the file into the array. So, when the function *max()* is called, the value of $i = N$, where N is the total number of values read from the file. When we start using *i* in *max.c*, we are actually starting with a value of N and not 0! So, the value of *max* that is returned is the highest negative value $-MAXINT$!

This illustrates one of the major issues with using global variables – the problem of *side effects*. Modifying the value of a variable in one location has an impact on the logic in another location because the changes are carried over to multiple locations.

Therefore, a good programming principle is to minimize the use of global variables and limit them to only those cases where it makes absolute sense to use them.

8.3 Scope and Extent of Variables

Scope of a variable is the region of code that it is visible, i.e., its spatial visibility. **Extent** defines the lifetime of the variable, i.e., how long does a variable live during the execution of the program. The scope and extent can be modified through the use of the keyword **static** in C.

Let us look at the example Programs 4 and 5. In Program 4, two variables are declared (and defined) outside the main function. As we saw earlier these are global variables. However, the difference is that the variable *N* now has the keyword *static* before it. The scope of the variable *sum* is the entire program which consists of two source files, i.e., both in Programs 4 and 5. On the other hand, the scope of *N* is only Program 4. All variables declared inside a function such as *i*, *sum1* in all functions have a scope limited to that function. Hence, *i* in main is not seen anywhere else and vice versa. The variable *N* referred to in Program 5 is the formal parameter passed and not the *N* in Program 4.

When we have the word *static* in front of a function, the meaning is exactly the same as for a variable – this function is visible only in that .c file and not from any of the other .c files that make up the final executable file. Hence, we cannot call `read_val()` from Program 5.

Now, as for the extent of the variables – all global variables live as long as the program is executing. Hence, the extent of *N* and *sum* is the entire life of the program. All local variables of functions have a lifetime of the function, i.e., they are instantiated when the function is entered and die when the function exits. What this actually means is that the memory location containing these values is no longer valid once the function exits. It, however, does not mean that the memory is erased. The memory location will be overwritten whenever it is reused at a later point of time by the operating system for whatever purpose.

Program 4: Main Program for Illustrating Scope and Extent

```
#include <stdio.h>
int          sum;
static int   N;

int main(void)
{
    int          val[N] , i;

    scanf("%d", &N);
    read_val(val);

    sum_odd(val , N);
    printf("Sum of odd values = %d\n", sum);
}
```

```

    sum_odd(val, N);
    printf("Sum of odd values = %d\n", sum);

    exit(0);
}

static void read_val(int *val)
{
    int i;

    for (i = 0; i < N; i++)
        scanf("%d", &val[i]);
}

```

Program 5: Program containing Module logic for Illustrating Scope and Extent

```
#include <stdio.h>
sum_odd(int *table, int N)
{
    int i;
    static int sum1 = 0;

    for (i = 0; i < N; i++) {
        if (table[i] % 3 == 0)
            sum += table[i];
        else
            sum1 += table[i];
    }

    printf("Sum of values that are not odd = %d\n", sum1);
    return;
}
```

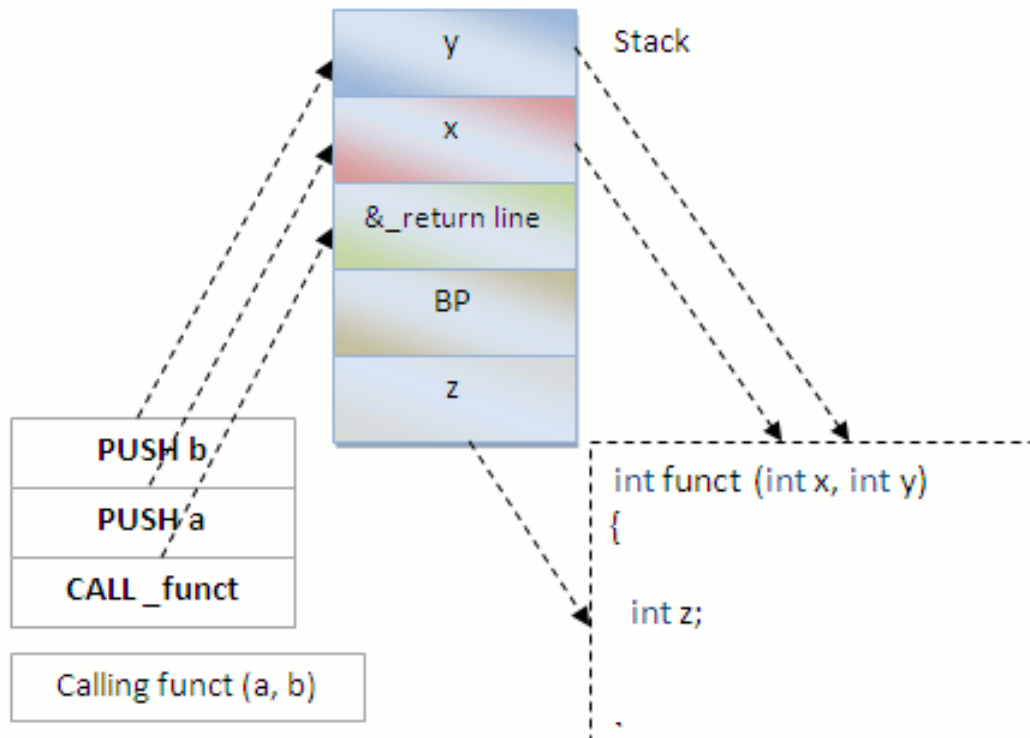
One final point is regarding the extent of the variable *sum1* in Program 5. When the keyword *static* is used before a variable within a function, it means that its extent is the entire program even though the scope is only the function. Hence, we will not be able to print the value of *sum1* from *main()* function. But, the variable's value will be preserved across multiple invocations of the function. Since we are calling the *sum_odd()* function twice in the *main()*, the first time it will print the sum of all values that are not odd. This value will be retained in this variable. So, the next time we call the function *sum_odd()*, the initial value of this variable is the sum calculated in the previous call. When we print the value the second time, therefore, the value will be double the previous value.

8.4 Parameter Passing Mechanisms

When a function is called, we pass parameters to it. A parameter whose value is filled in when calling the function and which is used inside the function for computing the output is called an *input parameter*. A parameter which is a place holder for passing back the output to the calling function is called an *output parameter*. A parameter that is used to pass data to the function and also back from the function is called an *I/O parameter*.

Parameters are stored in a special memory called **stack** in the process. A stack has the property that the last element inserted is the first one removed. Whenever a function calls another function, we need to return to the calling function. Thus, we need to store the address of the memory location which has the next instruction in the calling function. We push this into the stack first. Then, we push the parameters into the stack. The called function pops the parameters, uses them for computation and finally pops the return address from the stack to return to the calling function.

There are different parameter passing mechanisms in programming languages. One is called *Call by Value* where the parameter values are copied into a different location on the stack. The C programming language uses this mechanism only for all of its parameter passing. This is illustrated in Fig. 1 [1].



Disassembly and Call stack [x86]

Note: BP = Base Pointer

Figure 1: Parameter Passing: Call by Value [1]

8.5 File Operations

There are many C library functions to access data in files. These include *fgetc*, *fputc*, *fgets*, *fputs*, *fread*, *fwrite*, *fprintf* and *fscanf*. Each of them has a distinct reason for its existence.

The functions *fprintf*() and *fscanf*() are for dealing with files that have formatted data where the formatting of the files is clearly specified.

The functions *fread*() and *fwrite*() deal with binary files. Therefore, their contents cannot be read by using regular text editors unlike all the other file operations which deal with text files.

Functions *fgetc*() and *fputc* access one character at a time from the file. It should be used only in exceptional circumstances because file operations that fetch one character at a time are highly inefficient. Of course, the operating system takes care of this by buffering data in kernel memory but still it is NOT good practice to read one character at a time in the typical case.

Functions *fgets*() and *fputs*() are highly recommended for use with text files, even for formatted text files. It is better to use *fgets*() followed by function *sscanf*() than *fscanf*() to ensure that any formatting errors do not result in the program being stuck in an infinite loop. It is also good to use *fgets*() to ensure that any input accepted from the user does not result in buffer overflow which can happen in other cases. Buffer overflow problem is a major weakness that is exploited by viruses etc. and compromises system security.

8.6 Pointers

Pointers are addresses of memory locations where data is located. We can understand pointers through a simple real-life example. Let us say that there is a room called *visiting_faculty_room* in a building. The address of this can be S101 within the particular building. The person who is currently sitting in the room may be Prof. A. How does this map to C concepts?

When we declare a variable such as *int i*, *i* is equivalent to *visiting_faculty_room*. The address of *i*, i.e., *&i* is equivalent to S101 and the value stored in *i* when we initialise it as in *i = 1* is equivalent to saying Prof. A is currently in *visiting_faculty_room*. If Prof. A leaves and Prof. B starts using the room S101, then it is equivalent to saying *i = 2*, i.e., the value has changed in that memory location. The address has not changed however. Now, supposing we build an extension to the building and decide that the *visiting_faculty_room* will in future be the room N105, then, this is saying that the variable has been moved to a new location in memory. In other words, it is like saying

```
int      vfac_room1, vfac_room2, *vfac_room_addr;

vfac_room_addr = &vfac_room1;
/* A is currently in the room */
*vfac_room_addr = A;
...
/* B is currently in the room */
*vfac_room_addr = B;
/* Now, change the address visiting_faculty_room is pointing to */
vfac_room_addr = &vfac_room2;
/* where vfac_room1_addr=S101 and vfac_room2_addr=N105 for the */
/* room example above */
```

Given a pointer, the value stored in the address pointed to can be obtained by *dereferencing the pointer*. Given that *ptr* is a pointer, the value stored in that location is obtained by the expression **ptr*. Any variable in C consists of four parameters that define it: the variable name, data type, address and value. The data type will determine the amount of memory occupied by that variable in bytes. So, typically, an *int* variable occupies 4B whereas a *char* occupies 1B and so on.

Why do we need pointers? C is the only language to support them other than C++ which is derived from C, of course! We should remember that C is a systems programming language. Operating systems are written in C. Typically, an OS needs to access specific memory locations to store data in those locations. So, C, which was invented to write the Unix operating system comes with this powerful mechanism.

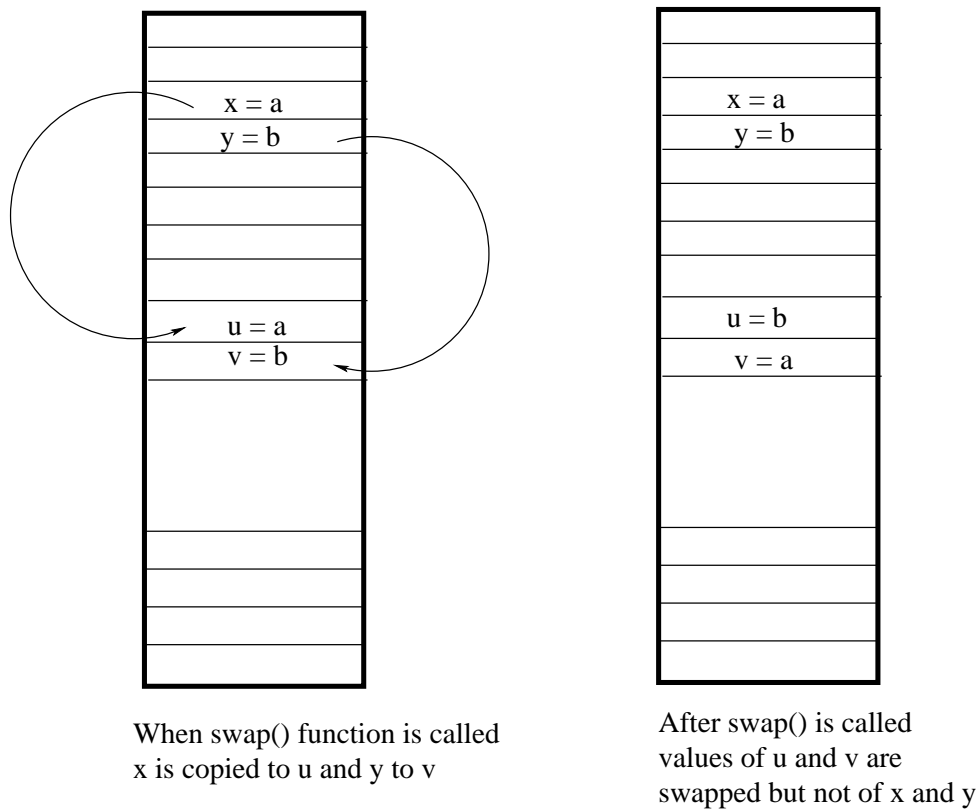


Figure 2: Memory snapshot when *swap* function is called and after *swap* is executed. *x, y* are actual parameters and *u, v* are formal parameters.

Let us look at the first version of *swap* function as given in K&R's book on C [5].

Program 6: Function to swap two variables

```
void swap(char u, char v)
{
    char        temp;

    temp = u;
    u      = v;
    v      = temp;

    printf("swap: U = %c, V = %c\n", u, v);
    return;
}
```

Now, the function *swap* is called by the *main* function with actual parameters *x, y* as shown in Program 7.

Program 7: Main function calling swap function

```
int main(void)
{
    char          x = 'a', y = 'b';

    swap(x, y);
    printf("main: X = %c, Y = %c\n", x, y);
    exit(0);
}
```

We will find that the values are NOT swapped in the *main* function. However, the values are found to be swapped in the *swap* function. This is because the parameters are passed using Call-by-Value as discussed earlier in Section 8.4. The memory in the system can be represented as shown in Fig. 2. Each slot shown represents one byte each. The memory locations *x* and *y* are different from the locations *u* and *v*. So, when *swap* returns to *main* the values in *x, y* do not change. At the same time since the extent of the variables *u, v* is the function *swap*, these memory locations are no longer available to the program.

Program 8: Program to swap two variables whose pointers are passed to the swap function

```
void swap(char *u, char *v)
{
    char          temp;

    temp = *u;
    *u    = *v;
    *v    = temp;

    printf("swap: U = %c, V = %c\n", *u, *v);
    return;
}

int main(void)
{
    char          x = 'a', y = 'b';

    swap(&x, &y);
    printf("X = %c, Y = %c\n", x, y);

    exit(0);
}
```

To achieve swapping of variables, we will have to pass the pointers of the actual variables to the *swap()* function as shown in Program 8. When we pass pointers, the memory of the system is as shown in Fig. 3. Here, the actual parameters are the addresses of variables *x, y*, i.e., *&x, &y* as seen in *main()* function in Program 8. In other words, the parameters *u* and *v* contain the addresses of variables *x* and *y* as shown in Fig. 3. Therefore, **u* refers to the value in location *pa* or in other words **u* is *a* and similarly **v* is *b*. When we swap **u* and **v* as shown in Program 8, we are swapping values in locations *pa* and *pb*. Therefore, at the

end of the swap function, the values of x, y are swapped but the values of u, v do not change in the function. Of course, as stated earlier, u, v are no longer accessible once we exit the function.

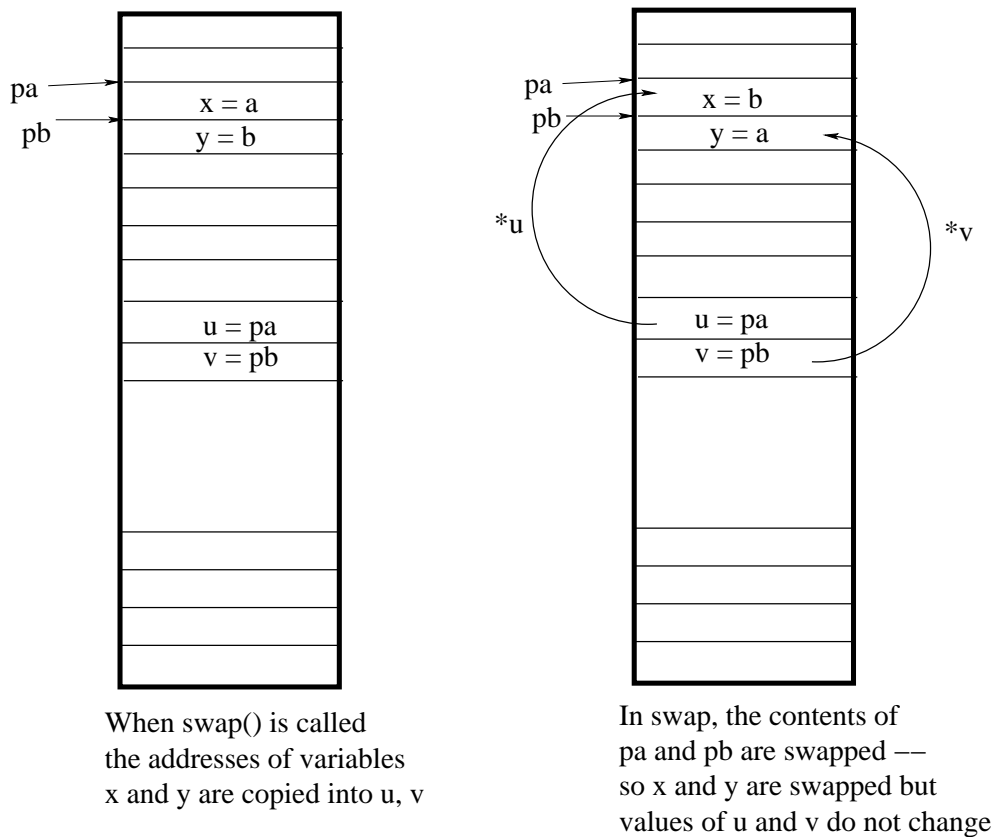


Figure 3: Memory snapshot when *swap* function is called with pointers and after *swap* is executed.

Some Important Dos and Don'ts with Pointers Whenever pointers are used, the most important thing to keep in mind is to initialize it to NULL. If the pointer is not initialized, like any other variable, it is occupying a memory location which may contain any value. The problem with this is that this is considered to be an address by the program when using it. So, if one is lucky, the program segment faults and exits. Otherwise, it is possible that the memory location is valid for the program which may result in corruption of data or even instructions. This is the memory corruption problem, one of the worst bugs anyone can be called upon to debug. One can make life much easier on themselves by initializing pointers to NULL.

8.7 Bit Manipulations

One more interesting feature in C is bit manipulation. Once again, the use stems from systems programming. Operating systems maintain information about free memory etc. as a series of 1s and 0s where a 1 can mean the memory is free and 0 that it is occupied. It is also

extensively used in Computer Networks where different bits represent different functionality. Bits need to be set or checked if they are set and so on.

The standard operations with bits are AND (&), OR (|), XOR (^), NOT (~), left shift (<<) and right shift (>>) with the symbols used by C for these operations in parentheses.

1	1	0	0	1	1	0	1
---	---	---	---	---	---	---	---

Original Data item that is
to be shifted

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

Data after shifting it left
by two bits

0	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

Data after shifting it right
by two bits

Figure 4: Illustrating the left and right shift operations

The result of these operations for different combinations of 1 and 0 is:

1. **AND:** When 1 and 0 are ANDed, a 0 is the result whereas 1 is the result when 1 is ANDed with 1.
2. **OR:** When 1 is ORed with either 0 or 1, the result is 1.
3. **XOR:** When 0 is XORed with 0 or 1 with 1, the result is 0 whereas 0 XORed with 1 results in 1.
4. **NOT:** NOT is the complement of the bit – so, 1 becomes 0 and vice versa.
5. **LEFT/RIGHT SHIFT:** When a data item is left shifted, the rightmost bits become zero. It is the reverse for right shift; the leftmost bits become zero. Thus, if the data item is 11001101, if it is left shifted by 2 bits, the value becomes 00110100. The two leftmost bits are shifted out and the two rightmost bits become 0. If we do right shift of the same data, the result will be 00110011. This is shown in Fig. 4.

Is a bit set? Now, if the problem is to determine if a bit is set, we need to do the following: let us say that we are dealing only with a single byte. So, there are only 8 bits (or locations). We want to verify if bit 3 is set or not where we are starting bit positions from 0. The easiest way to do this is to shift the data so that bit 3 now becomes bit 0 (or the rightmost/least significant bit) and AND it with 1. If the bit is 1, the result will be 1; otherwise it will be 0. This is shown in Fig. 5.

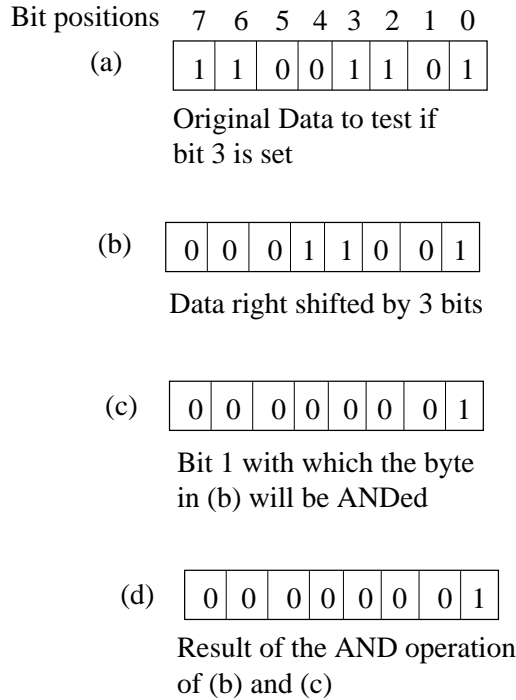


Figure 5: Illustration of steps in checking whether bit 3 is set or not

Set a bit To set a bit, operation OR is used. So, if we need to set bit 3 as in the previous example, we take bit 1, left shift it by 3 bits and OR it with the original data. Since all the other bits in the value 1 are all 0s, the original contents will not be modified. When we OR with 1, irrespective of whether the original data had 0 or 1, the result will be 1. Thus, the bit is set.

Clear a bit Clearing a bit means setting it to 0 always. To clear a bit, we do OR with a 0. The operation is exactly the same as in Set a bit.

Extracting the value in a set of bits In computer networks, we usually need to do an operation such as determining the value in a particular set of bits, say from 4-6 positions. The way this can be achieved is to use the operations we have described so far. So, for this example, we right shift such that position 4 bit is now in bit 0 position. Then, we take a byte which has 1 in all bit positions. This is achieved by taking a 0 and doing a NOT operation on it. Then, left shift this by 3 bits (since we want to find the value in 3 bits, i.e., bit positions 4, 5 and 6). This will result in the rightmost 3 bits being 0 and all other bits being 1. Now, NOT the value such that the bits are reversed. Now, an AND of the original data which was rightshifted with this value will give the value in the bit positions we are interested in. This operation is shown in Fig. 6.

8.8 Pre-processor directives

Preprocessing is the first step in C where the pre-processor directives are processed. The most commonly used pre-processor directives are `#include` and `#define`. Any directive with

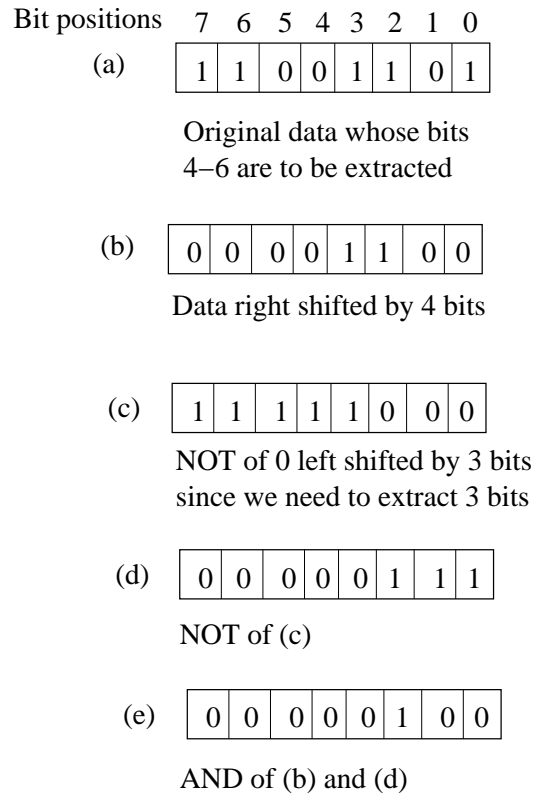


Figure 6: Illustration of steps for extracting bits 4-6 from the given data byte

a “#” in front of it is a pre-processor directive.

The `#include` directive essentially copies the contents of the specified file into the file where this directive is included. Normally, this is used with `.h` files in which the function prototypes are declared as well as any possible global variable declarations. The `.h` files may also contain `#define` primitives which define constants and macros.

In addition, the other important pre-processor directive is `#ifndef`. This has to be used to ensure that no data is multiply defined if the same file is ‘included’ in multiple `.c` files. If, say, we define a global variable in a `.h` file which is included in multiple `.c` files, then, there is an attempt to **define** that variable multiple times which is an error. This can be handled by using the `#ifndef` directive around the contents of the `.h` file as shown in Program 9.

`#ifndef` is checking if the macro `__LINK_H` has not been defined earlier. If that is true, it defines it in the next line. So, if multiple `.c` files include this file, the first file which includes it will find the variable not defined and will include the file in the `.c` file. The next file which attempts to do so will find this variable defined and so the file will not be included again thus avoiding multiple definitions of the global variable **head**.

Program 9: `.h` file showing the use of the `#ifndef` pre-processor directive

```
#ifndef __LINK_H
#define __LINK_H

typedef struct node_s {
    int                value;
```

```

        struct node_s    *next;
    } node_t;

    node_t                head = NULL;

#endif

```

8.9 Command Line Arguments

When we run a command on Unix/Linux systems, we give arguments to them as in the following example:

```
cp a.c b.c
```

where “a.c” and “b.c” are arguments to the command *cp*.

Similarly, whenever we write a program, the input we need can be taken as command line arguments.

Many students have the bad habit of fixing file names by hardcoding them inside their programs. The issue with this is that if the program has to be tested with another filename, either the file has to be renamed to be what the program is expecting (just think how absurd this is!) or the program has to be modified and recompiled before it can be used for this file (an even more absurd solution!).

The next most popular method used by students is to prompt the user to enter the input through keyboard. This is alright but different programmers may use different ways of accepting input which is not standardised. Further, this is difficult from the perspective of automating tasks through scripts.

Command line arguments are an elegant way of giving input. Normally, we do not use command line arguments if the input is large. Instead, what we do is construct a file which consists of all the data that needs to be input and give the name of this file as a command line argument. Thus, by simply giving a different file name which consists of different input, the program can be tested under different conditions very easily. **This is the most preferred form of accepting input from the user.**

The command line arguments are passed through the parameters *argc* and *argv* of the *main()* function.

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

argc gives the total number of arguments including the name of the executable file. The array of character pointers *argv* contains the addresses of the strings which are the actual arguments including the name of the executable file. Thus, from the *cp* command example above, *argv[0]* = *cp*, *argv[1]* = *a.c*, *argv[2]* = *b.c* and *argc* = 3.

The first thing we need to do when writing programs with command line arguments is to verify if the number of arguments is equal to what we are expecting. *cp* command expects a minimum of two arguments to it – the source and destination files. So, *argc* has to be 3. Any value other than that would be wrong. We need to check this before we proceed with the program. If we do not do it and proceed to use *argv*, we will hit a NULL pointer and segment fault.

So, the *cp* program would start as follows:

Program 10: Program illustrating use of command line arguments

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: cp <src> <dest>\n");
        exit(1);
    }

    /* call the function that copies from file1 to file2 */
    copy(argv[1], argv[2]);
    exit(0);
}
```

Program 10 illustrates many good coding practices. When we find that the *argc* value does not match the expected value, we need to print an error. We use *fprintf* with *stderr* instead of *printf* because the error message will be printed on the terminal even if *stdout* is redirected to a file. It is always good to differentiate between the normal print messages and error messages. Error messages should always be printed to *stderr*.

Secondly, look at the use of the function/system call *exit*. The parameter passed to this function is the return status of the program. By convention, Unix/Linux use 0 to indicate success and any other value to indicate failure. So, whenever there is a failure to execute and the program is exiting, it is useful to give a status value. In this case, we gave the value 1. If there are multiple errors, for each of which, the program exits, then, the status value has to be different for each of them. We can check the status of the program by running the command “\$?” in Linux to get the status which will tell us where the program failed.

8.10 typedef and union

A structure is a collection of data which are not of the same type. Let us say that we define a structure called *std_record* as follows:

```
struct std_record {
    char        name[MAX_NAME];
    char        programme:4;
    char        year:4;
    float       cgpa;
}
```

The declarations for *programme* and *year* are stating that the number of bits allocated for these fields is 4 each so that, together, they occupy one byte. (This is one of the interesting facilities provided by C to reduce memory footprint. We have to remember that C is used primarily for systems programming such as operating systems and every byte saved is useful in that context. This is even more so in the context of embedded operating systems.)

8.10.1 typedef

Using the keyword *typedef*, a new data type can be defined in C. We can define a new data type called `std_rec_t` as follows:

```
typedef struct std_record std_rec_t;
```

All future references can declare variables of this structure by using this new data type instead of using the struct:

```
std_rec_t      rec;
```

instead of

```
struct std_record  rec;
```

We can also define new data types using basic data types themselves, e.g., we can define a new data type called `uint16_t` which is an unsigned short in almost all the systems.

```
typedef unsigned short uint16_t;
```

```
uint16_t      ip_protocol;
```

However, since the data type's storage is not defined by the C language and it varies from system to system, such a data type is useful to ensure that exactly the required storage is allocated. This is extremely important in computer networking where systems of different architectures and operating systems are connected together and all of them have to interpret the messages exactly as per the protocol specifications.

8.10.2 union

The declaration of a *union* looks very similar to a structure declaration. Given below is one such declaration:

```
union {  
    char      data[4];  
    float     f;  
}
```

If the above were a structure, the storage space allocated to it would be the sum of the space needed for both the variables – typically 8B. In a union, however, the storage allocated is equal to the storage of the largest data item within the union definition. In the above example, both the fields within the union have the same size of 4B and so the union will have a storage space of 4B. But, if we had only a *char* along with a *float*, it would still be 4B as normally *float* occupies 4B and *char* occupies 1B.

(**Note:** *union* can be used to define data types in such a way that it can lead to polymorphism in the context of object-oriented design.)

9 Some Common Compiler Errors and What They Mean

It is recommended to use *gcc* for compilation which is a default in Linux/Unix systems. Compilation consists of two steps that we are interested in as a beginner of programming – compiling and linking. The first step converts C language statements into machine language which is nothing but a series of 0s and 1s. Linking is needed to get the C library code to be linked with the source code we write where we use these library functions. Without this step, the computer will not know what to do when a C library function is called.

When compiling, it is good to enable all warnings and take care of them. We will discuss why this is important when we come to the specific warning. The command to compile a C program is given below:

```
gcc -Wall hello.c -o hello
```

The option “-Wall” enables *all* warnings. As everyone knows, the C compiler creates an executable file with the name *a.out* by default. However, you are *strongly discouraged* to do this! We observe that if you have many programs, at any point of time, you can have exactly **one(!)** executable program because every time you recompile some source code, it overwrites the existing executable file!

Instead, use the option “-o” to name the executable file. Please note that good programming practice consists of naming files properly. A typical naming strategy is to have the executable file name to be the same as the name of the source file containing the *main* function but without the *.c* extension. Please also note that the extension “.o” is for object files – i.e., for files which contain the machine language equivalent of the C code, but which are not executable. An executable file consists of such object files and also is *linked* with library files which provide the code for the C library functions we use.

We now look at some of the common C compilation errors and how to fix them:

9.1 warning: implicit declaration of function printf

This error is seen when we forget to include the *.h* file containing the prototype of the function. Hence, the compiler does not know what the data types of the arguments passed to the function should be and cannot verify them. This results in this warning. Always make sure you remove this warning by including the right *.h* file. To know which *.h* file to include, execute the command *man jfunction*. In our example warning above, we execute the command *man printf*. The man page of *printf* will be displayed and under the SYNOPSIS, the needed include file(s) information will be given. Include these into the source file where the function is used to solve this warning.

9.2 warning: val may be used uninitialized in this function

The warning also gives the line where this is encountered. This is from the Program 1 and shows the following statement:

```
if (val > max)
```

because I just had the comment “read val” but no valid statement initializing *val*. Same warning will be noticed for *N* too. This is a very important warning because any uninitialized variable has unpredictable value and can lead to wrong output from the program. Hence,

make sure all variables are properly initialized. Fixing this warning at compile time can save hours of debugging time trying to figure out why an error in output occurs at run time. This is especially true if we assume $val = 0$ and say, most of the time the computer does find a memory location with 0 in it. However, once in a while, the program may allocate a memory location which does not have 0 in it. Only under such conditions will the program fail. So, the program will run fine for some runs and maybe even for years and then suddenly fail one day. Such errors can be a nightmare for debugging. And, all it needed was to ensure there was no warning during compilation to avoid this nightmare!

Program 11: Program illustrating the linking error “Undefined Reference”

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

int max = -1;

int main(void)
{
    int i, val, N;

    /* Read N from the user */
    scanf("%d", &N);
    for (i = 0; i < N; i++) {
        /* Read value from user */
        scanf("%d", &val);
        if (val > max)
            max = val;
    }

    printf("sqrt(max) = %d\n", sqrt(max));
    exit(0);
}
```

9.3 sqrt_max.c:7:1: error: expected , or ; before int

If in Program 11, we forget to include “;” after definition of the variable *max*, we get the error that is the title of this subsection. As shown, typically, the compiler will give the number of the line in the program where this error has occurred. We can look at that line (7 here) and do the necessary changes to fix the error.

9.4 error: i undeclared

If we forget to declare the variable *i* in Program 11, we get the error of this subsection. In addition, the error messages will be as follows:

```
sqrt_max.c: In function main:
sqrt_max.c:13:8: error: i undeclared (first use in this function)
```



```

    for (i = 0; i < N; i++) {
        ^
sqrt_max.c:13:8: note: each undeclared identifier is reported only once
for each function it appears in

```

It shows that the error occurs in Line 13. In fact, every time i is referred to in the program, this error occurs. However, as given in the error message, the compiler gives it only for the first occurrence. Once the proper declaration of *int i* is given, this error disappears from all occurrences.

9.5 sqrt_max.c:(.text+0x6a): undefined reference to ‘sqrt’

Here is a program (Program 11) that computes maximum of given positive numbers and then its square root. Note that we need to include the file “math.h” to use the built-in function *sqrt* of the C library. When we compile this program with our usual *gcc* command, we get the error (please note this is an error and not a warning) as given in the paragraph title above. The issue here is not that the prototype was not found because we have included “math.h” in our source file. The issue is the object code of the C library which contains the function *sqrt* is not linked with our object code and so we get this error. The library for all math functions is *libm.so* in Linux. To include this library into the compilation, we need to modify our compilation command as follows:

```
gcc -Wall -lm sqrt_max.c -o sqrt_max
```

The “-lm” option means include “library” “m”. Similarly, there may be other programs we write which may need specialized libraries, e.g., if we write multithreaded programs we need to include the option “-lpthread” when compiling the code to access the *pthread* library functions.

10 Coding Standards

Some good resources for coding standards are [2] and [3].

10.1 Meaningful Names

Names of files, functions and variables should all be meaningful so that the code is highly readable. E.g., if we use the variables x, y instead of *sum, mean*, the purpose of these variables is not immediately apparent in the case of the former whereas it is obvious with the use of latter names. Array indices or total number of items etc. can use variables such as i, j, N but all other names should be meaningful. At the same time, it is important to not take it to extreme and make the names as long as a paragraph! The naming convention for variables uses either “_” between different words that make up the name such as *visit_fac_room* or they capitalise the different words as in *visitFacRoom*. One of the two conventions may be chosen and used consistently throughout the program. Do NOT mix the two conventions!!

10.2 Constants/Macros

Constants or Macros in programming are by convention always all in capital letters. Thus, we define PI and not “pi” as a constant. As soon as any term is seen that is all in capital letters, it should be obvious that this is either a constant or a macro. An example of a macro is:

```
#define MAX(a, b)    a > b ? a : b
```

MAX can then be used in the program. Both constants and macros will be expanded in place *before compilation* as they are part of the pre-processor directives.

10.3 Column width of 75-78

When coding, ensure that no line in your code exceeds 75-78 columns. If you plan to debug, it may sometimes be needed to print out the code to review it. If so, the lines will be truncated. Even otherwise, the lines may wrap around the window of the editor leading to issues of readability. Hence, it is recommended that all code be within the limit specified.

10.4 Indentation

Indentation is a very important part of programming for readability. While there are tools that help to indent automatically and/or after the fact, it is more useful to develop the instinct to program with indentation as almost a reflex action. Many a time, people try to indent the code *after* finishing the testing, which is a superb waste of time. If we learn to indent as we code, there is no need to revisit the code for indentation.

Indentation with tabs is easy but ends up possibly creating issues with column limits if there are more than two levels of indentation. It is better to use a 4 space indentation. However, whether you choose space or tab as your indentation strategy, be consistent! Do NOT mix up spaces and tabs since then the indentation will be quite ruined based on tab definition in different systems.

10.5 Comments

Whenever we write complicated programs, it is good to comment the parts of the code that are difficult to understand. Explain the logic in English in terms easily understood that will enhance the readability and maintainability of the program. In fact, a well commented code can then be used to generate a design document using tools.

10.6 Good Parenthesisation

When using complicated arithmetic or logical expressions, it is useful if proper parenthesisation is done. If not, even the programmer may not be completely sure of the precedence of the operators and it may lead to a wrong expression. It will certainly be difficult for a maintainer of the code to decode it! On the other hand, do not go to the extreme of total parenthesisation which also leads to readability issues.

10.7 Braces for Functions vs Other Compound statements

Braces can be used either in K&R style or they can be on standalone lines when used with looping or conditional statements. However, as cautioned in other cases, stick to one convention throughout and do NOT mix up the styles.

However, when starting and ending functions, it is advised to have the braces on separate lines. This helps to navigate functions very fast in *vi* by using the “[[” and “]]” commands.

10.8 Block Coding

One good programming practice I would like you to include is block coding. Whenever a left brace { is typed, type immediately the corresponding right brace }. This helps in incremental compilation which is very essential for easy debugging. (**Courtesy:** Dr.Anjeneya Swami)

References

- [1] <http://www.equestionanswers.com/c/parameters-are-passed-call-by-value.php>
- [2] Rob Pike, *Notes on Programming in C*
<http://www.lysator.liu.se/c/pikestyle.html>
- [3] Eric Laroche, *C programming language coding guidelines*
<http://www.lrdev.com/lr/c/ccgl.html>
- [4] Chakravarthy Bhagvati, *How to Program*
<http://scis.uohyd.ac.in/~chakcs/howtoprogram.pdf>
- [5] Brian W Kernighan and Dennis M Ritchie, *The C Programming Language* 2nd edition, Prentice Hall, 1988.

Acknowledgements

I wish to thank Dr.Anjeneya Swami for his comments and catching the errors in the earlier drafts. Any remaining errors are obviously my responsibility.