Title	Introduction to Programming using C
Author	Brian "Beej" Hall
Adaption and Typesetting	**Balam Singh Dafouti** Academic Consultant- School of CS & IT, Uttarakhand Open University, Haldwani
ISBN:	

Acknowledgement

This book has been adapted from "Beej's Guide to C Programming" by Brian "Beej" Hall. The University also acknowledges with thanks to KKHSOU for providing the study material on http://www.kkhsou.in/web_new/index.php portal a Creative Commons Attribution-Non Commercial-Share Alike 4.0
License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Published By: Uttarakhand Open University

Introduction to Programming using C
MCS-101

BLOCK I

Unit 1: Programming Building Blocks: Specification, Implementation **,** Hello, World! Example.
Unit 2: Variables, Expressions, and Statements: Variables, Operators, Expressions, Statements.
Unit 3: Functions: Passing by Value, Function Prototypes.
Unit 4: Variables: Up Scope, Storage Classes

BLOCK II

Unit 5: Pointers: Memory and Variables, Pointer Types, Dereferencing, Passing Pointers as Parameters.
Unit 6: Structures: Pointers to structs, Passing struct pointers to functions.
Unit 7: Arrays: Passing arrays to functions.
Unit 8: Strings
Unit 9: Dynamic Memory: malloc()**,** free(), realloc(), calloc().

BLOCK III

Unit 10 Advance Topics: Pointer Arithmetic, typedef, enum, More struct declarations, Command Line Arguments, Multidimensional Arrays, Casting and promotion, Incomplete types**,** void pointers, NULL pointers, More Static, Typical Multifile Projects**,** The Almighty C Preprocessor, Pointers to pointers, Pointers to Functions, Variable Argument Lists.
Unit 11. Standard I/O Library: fopen(), freopen(), fclose(), printf(), fprintf(), scanf(), fscanf(), gets(), fgets(), getc(), fgetc(), getchar(), puts(), fputs(), putc(), fputc(), putchar(), fseek(), rewind(), ftell(), fgetpos(), fsetpos(), ungetc(), fread(), fwrite(), feof(), ferror(), clearerr(), perror(), remove(), rename(), tmpfile(), tmpnam(), setbuf(), setvbuf(), fflush().
Unit 12. String Manipulation: strlen(), strcmp(), strncmp(), strcat(), strncat(), strchr(), strrchr(), strcpy(), strncpy(), strspn(), strcspn(), strstr(), strtok().
Unit 13: Mathematical Functions: sin(), sinf(), sinl(), cos(), cosf(), cosl(), tan(), tanf(), tanl(), asin(), asinf(), asinl(), acos(), acosf(), acosl(), atan(), atanf(), atanl(), atan2(), atan2f(), atan2l(), sqrt().

Suggested Readings:

1. Let us C-Yashwant Kanetkar.
2. Programming in C- Balguruswamy
3. The C programming Lang., Pearson Ecl – Dennis Ritchie
4. Structured programming approach using C-Forouzah & Ceilberg Thomson learning publication.
5. Pointers in C – Yashwant Kanetkar

Supplementary Course Material available at: http://www.freetechbooks.com/beejs-guide-to-c-programming-t986.html

Block-1

Unit-1

Programming Building Blocks

1.1 Learning Objectives

1.2 Introduction

1.3 Programming Building Block

- **1.3.1** The Specification
- **1.3.2** The implementation

1.4 Answer to check your Progress

1.5 Model Questions

1.1 Learning Objectives

After going through this unit the learner able to learn:

- The Programming building block
- Variable, Operators, Expression and Statements
- Function: Passing by value and function prototypes

1.2 Introduction

The **C programming language** is a computer programming language that was developed to do system programming for the operating system UNIX and is an imperative programming language. C was developed in the early 1970s by Ken Thompson and Dennis Ritchie at Bell Labs. It is a procedural language, which means that people can write their programs as a series of step-by-step instructions. C is a compiled language.

C is available for many different types of computers. This is why C is called a "portable" language. A program that is written in C and that respects certain limitations can be compiled for many different platforms.

1.3 Programming building block

What is programming, anyway? I mean, you're learning how to do it, but what is it? Well, it's, umm, kind of like, well, say you have this multilayered chocolate and vanilla cake sitting on top of an internal combustion engine and the gearbox is connected to the coil with a banana.

Now, if you're eating the cake a la mode, that means... Wait. Scratch that analogy. I'll start again.

What is programming, anyway? It's telling the computer how to perform a task. So you need two things (besides your own self and a computer) to get going. One thing you need is the task the computer is to perform. This is easy to get as a student because the teacher will hand you a sheet of paper with an assignment on it that describes exactly what the computer is to do for you to get a good grade. If you don't want a good grade, the computer can do that without your intervention. But I digress. The second thing you need to get started is the knowledge of how to tell the computer to do these things. It turns out there are lots of ways to get the computer to do a particular task...just like there are lots of ways to ask someone to please obtain for me my fluffy foot covering devices in order to prevent chilliness. Many of these ways are right, and a few of them are best.

What you can do as a programmer, though, is get through the assignments doing something that works, and then look back at it and see how you could have made it better or faster or more concise. This is one thing that seriously differentiates programmers from excellent programmers.

Eventually what you'll find is that the stuff you wrote back in college (e.g. The Internet Pizza Server, or, say, my entire Masters project) is a horridly embarrassing steaming pile of code that was quite possibly the worst thing you've ever written. The only way to go is up.

1.2.1 The Specification

what do you do with this specification? It's a description of what the program is going to do, right? But where to begin? What you need to do is this: break down the design into handy bite-sized pieces that you can implement using techniques you know work in those situations.

As you learn C, those bite-sized pieces will correspond to function calls or statements that you will have learned. As you learn to program in general, those bite-sized pieces will start corresponding to larger algorithms that you know (or can easily look up.) Right now, you might not know any of the pieces that you have at your disposal. That's ok. The fastest way to learn them is to, right now, press the mouse to your forehead and say the password, “K&R2”.

That didn't work? Hmmm. There must be a problem with the system somewhere. Ok, we'll do it the old-school way: learning stuff by hand.

Let's have an example:

Assignment: Implement a program that will calculate the sum of all numbers between 1and the number the user enters. The program shall output the result. Ok, well, that summary is pretty high level and doesn't lend itself to bite-sized pieces, so it's up to us to split it up. There are several places that are good to break up pieces to be more bite-sized. Input is one thing to break out, output is another. If you need to input something, or output something, each of those is a handy bite-sized piece. If you need to calculate something, that can be another bite-sized piece (though the more difficult calculations can be made up of many pieces themselves!)

So, moving forward through a sample run of the program:

1. We need the program to read a number from the keyboard.

2. We need the program to compute a result using that number.

3. We need the program to output the result.

This is good! We've identified the parts of the assignment that need to be worked on. “Wait! Stop!” I hear you. You're wondering how we knew it was broken down into enough bite-sized pieces, and, in fact, how we even know those are bite-sized pieces, anyhow! For all you know, reading a number from the keyboard could be a hugely involved task! The short of it is, well, you caught me trying to pull a fast one on you. I know these are bite-sized because in my head I can correspond them to simple C function calls or statements. Outputting the result, for instance, is one line of code (very bite-sized). But that's me and we're talking about you. In your case, I have a little bit of a chicken-and-egg problem: you need to know what the bite-sized pieces of the program are so you can use the right functions and statements, and you need to know what the functions and statements are in order to know how to split the project up into bite-sized pieces! Hell's bells! So we're compromising a bit. I agree to tell you what the statements and functions are if you agree to keep this stuff about bite-sized pieces in the back of your head while we progress. Ok? ...I said, “Ok?” And you answer... “Ok, I promise to keep this bite-sized pieces stuff in mind.” Excellent!

1.2.2 The implementation

Right! Let's take that example from the previous section and see how we're going to actually implement it. Remember that once you have the specification (or assignment or whatever you're going to call it) broken up into handy bite-sized pieces, then you can start writing the instructions to make that happen. Some bite-sized pieces might only have one statement; others might be pages of code. Now here we're going to cheat a little bit again, and I'm going to tell you what you'll need to call to implement our sample program. I know this because I've done it all before and looked it all up. You, too, will soon know it for the same reasons let's take our steps, except this time, we'll write them with a little more information. Just bear with me through the syntax here and try to make the correlation between this and the bite-sized pieces mentioned earlier. All the weird parentheses and squirrely braces will make sense in later sections of the guide. Right now what are important the steps and the translation of those steps to computer code.

The steps, partially translated:

1. Read the number from the keyboard using **scanf()**.
2. Compute the sum of the numbers between one and the entered number using a **for-**loop and the addition operator.

1. Print the result using **printf()**.

Normally, this partial translation would just take place in your head. You need to output to the console? You know that the **`printf()`** function is one way to do it. And as the partial translation takes place in your head, what better time than that to actually code it up using your favorite editor:

```
#include <stdio.h>
int main(void)
{
int num, result = 0;
scanf("%d", &num); // read the number from the
keyboard
for(i = 1; i <= num; i++) { // compute the result
result += i;
}
printf("%d\n", result); // output the result
return 0;
}
```

Remember how there were multiple ways to do things? Well, I didn't have to use **`scanf()`**, I didn't have to use a **`for`**-loop, and I didn't have to use **`printf()`**. But they were the best for this example.

Example:

This is the canonical example of a C program. Everyone uses it:

```
/* helloworld program */
#include <stdio.h>
int main(void)
{
printf("Hello, World!\n");
return 0;
}
```

We're going to don our long-sleeved heavy-duty rubber gloves, grab a scapel, and rip into this thing to see what makes it tick. So, scrub up, because here we go. Cutting very gently... Let's get the easy thing out of the way: anything between the digraphs `/*` and `*/` is a comment and will be completely ignored by the compiler. This allows you to leave messages to yourself and others, so that when you come back and read your code in the distant future, you'll know what the heck it was you were trying to do. Believe me, you will forget; it happens. (Modern C compilers also treat anything after a `//` as a comment. GCC will obey it, as will VC++. However, if you are using an old compiler like Turbo C, well, the `//` construct was a little bit before its time. So I'll try to keep it happy and use the old-style `/*comments*/` in my code. But everyone uses `//` these days when adding a comment to the end of a line, and you should feel free to, as well.)

Now, what is this `#include`? Well, it tells the C Preprocessor to pull the contents of another file and insert it into the code right *there*.

Wait--what's a C Preprocessor? Good question. There are two stages (well, technically there are more than two, but hey, let's pretend there are two and have a good laugh) to compilation: the preprocessor and the compiler. Anything that starts with pound sign, (`#`) is something the preprocessor operates on before the compiler even gets started. Common *preprocessor directives*, as they're called, are `#include` and `#define`.

After the C preprocessor has finished preprocessing everything, the results are ready for the compiler to take them and produce assembly code, machine code, or whatever it's about to do. Don't worry about the technical details of compilation for now; just know that your source runs through the preprocessor, then the output of that runs through the compiler, then that produces an executable for you to run.

What about the rest of the line? What's `<stdio.h>`? That is what is known as a *header file*. It's the dot-h at the end that gives it away. In fact it's the "Standard IO" (stdio) header file that you will grow to know and love. It contains preprocessor directives and function prototypes (more on that later) for common input and output needs. For our demo program, we're outputting the string "Hello, World!", so we in particular need the function prototype for the **`printf()`** function from this header file.

How did I know I needed to `#include <stdio.h>` for **`printf()`**? Answer: it's in the documentation. If you're on a Unix system, **man printf** and it'll tell you right at the top of the man page what header files are required. That was all to cover the first line! But, let's face it, it has been completely dissected. No mystery shall remain!

The next line is **`main()`**. This is the definition of the function **`main()`**; everything between the squirrely braces (`{` and `}`) is part of the function definition.

A *function*: “a function is a collection of code that is to be executed as a group when the function is called. You can think of it as, “When I call **`main()`**, all the stuff in the squirrley braces will be executed, and not a moment before.” How do you call a function, anyway? The answer lies in the **`printf()`** line, but we'll get to that in a minute.

Now, the main function is a special one in many ways, but one way stands above the rest: it is the function that will be called automatically when your program starts executing. Nothing of yours gets called before **`main()`**. In the case of our example, this works fine since all we want to do is print a line and exit.

that's another thing: once the program executes past the end of **`main()`**, down there at the closing squirrley brace, the program will exit, and you'll be back at your command prompt.

So now we know that that program has brought in a header file, `stdio.h`, and declared a **`main()`** function that will execute when the program is started. What are the goodies in `main()`?

Check Your Progress

Choose the Correct one

Q.1: What is function?

A. Function is a block of statements that perform some specific task.

B. Function is the fundamental modular unit. A function is usually designed to perform a specific task.

C. Function is a block of code that performs a specific task. It has a name and it is reusable

D. All the above

Q.2: C programs are converted into machine language with the help of

A. An Editor

B. A compiler

C. An operating system

D. None of the above

Q.3: Name the loop that executes at least once.

A. For

B. If

C. do-while

D. while

A call to the function **`printf()`**. You can tell this is a function call and not a function definition in a number of ways, but one indicator is the lack of squirrely braces after it. And you end the function call with a semicolon so the compiler knows it's the end of the expression. You'll be putting semicolons after most everything, as you'll see.

You're passing one parameter to the function **`printf()`**: a string to be printed when you call it. Oh, yeah--we're calling a function! We rock! Wait, wait--don't get cocky. What's that crazy `\n` at the end of the string? Well, most characters in the string look just like they are stored. But here are certain characters that you can't print on screen well that are embedded as two-character backslash codes. One of the most popular is `\n` (read "backslash-N") that corresponds to the *newline* character. This is the character that causing further printing to continue on the next line instead of the current. It's like hitting return at the end of the line. So copy that code into a file, build it, and run it--see what happens:

```
Hello, World!
```

It's done and tested!

Check Your Progress

Fill in the blanks

Q.4: Read the number from the keyboard using ……………….

Q.5: Print the result using……………..

1.4 Answer to check your Progress

`Ans to Q.1: D`

`Ans to Q.2: B`

`Ans to Q.3: C`

`Ans to Q.4:` **scanf()**.

`Ans to Q.5:` **printf()**.

1.5 Model Questions

1. What is Variable, Operators, Expression? Explain with the help of example.
2. What is Function and function prototypes?
3. What is Compiler?
4. What is the difference between printf() and scanf()?

Unit-2
Variables, Expressions, and Statements

1.1 Learning Objectives
1.2 Introduction
1.3 Variable in C
    1.3.1 Declaration of Variables
    1.3.2 Initialization of Variables
    1.3.3 Operators
    1.3.4 Expression
    1.3.5 Statements
1.4 Answer to Check Your Progress
1.5 Model Questions

1.1 Learning Objectives

After going through this unit the learner will able to Learn:

- About the variables
- Declaration of Variables
- Initialization of Variables
- Operators
- Expression
- Statements

1.2 Introduction

We have already learned about basic Building Blocks In addition, concepts of Program implementation and specification have also been introduced in the previous unit. In this unit, you will come across the basic building block and their implementation used in C language.

1.3 Variable in C

A variable is defined as name given to the storage location in computer memory. When using a variable, we actually refer to an address of the memory where the data is stored. A variable name can be chosen by the programmer in a meaningful way that reflects what it represents in the program. The naming convention of variable follows the rule of constructing identifiers.

1.3.1 Declaration of Variables

Each variable to be used in the program must be declared. For declaration of variable, we first specify the data type of the variable followed by its name. The data type indicates the kind of data that the variable will store. A variable cannot be of type void. In C, variable declaration always ends with a semicolon. The general syntax of declaration of a variable is :

data_type variablename;

We can also declared more than one variable in a single statement as follows:

data_type variable1, variable2, ... , variableN;

For example,

```
    int roll;
    float salary;
    char grade;
    int m1,m2,m3;
int total_marks;
```

In C, variables can be declared at any place in the program but one thing should be kept in mind. Variables should be declared before using them. By declaring a variable we usually tell three things to the compiler :

- What the variable name is.
- What type of data the variable will hold.
- and the scope of the variable.

1.3.2 Initialization of Variables

While declaring the variables, we can also initialize them with some value.

Following are few examples of initialization statements:

int roll_no= 5;

float average=200.75;

char grade= 'A';

The initializer applies only to the variable defined immediately before it. Therefore, the statement

int number, sum=0;

intializes the variable *sum* and not *number*.

Let us take a simple example for explaining declaration and initialization of variables. Suppose we want to store value 153 to a variable. We first create the name of the variable, suppose A. Since 153 is integer, so we declare the variable A as *integer* type, and then assign 153 to that variable. These can be done as follows :

```
      int A ;
    A = 153 ;
```

The first statement says that A is a container, where we can store only integer type variable. This means that we cannot store value into A other than integer. Therefore, this type of statement is known as *declaration* statement. The second statement says that the value 153 is stored in A. This means variable A is initialized with 153. Therefore this type of statement is known as variable *initialization.* We can store values to a variable in two ways:

- using assignment statement, and
- using a read statement.

The use of assignment statement is already shown. In the second approach you can make a call of C standard input function like *scanf*, *getch*, *getc*, *gets* etc. to store value to a variable. For example, the
above initialization statement can be written as

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

This statement will take an integer type input from standard input device (that is keyword) and store it to A.

A few examples of variable declaration and initialization are shown below:

Variable declaration	**Remarks**
int i = 0, j = 1;	*i* and *j* are declared as integer variables. The variables i and j are initialized with value as 0 and 1 respectively.
float basic_pay;	*basic_pay* is a floating point variable with a real value or values containing decimal point.
char a;	*a* is a character variable that stores a single character.
double theta;	*theta* is a double precision variable that stores a double precision floating point number.

1.3.3 Operators

I've snuck a few quick ones past you already when it comes to expressions. You've already seen things like:

```
result += i;
i = 2;
```

Those are both expressions. In fact, you'll come to find that most everything in C is an expression of one form or another. But here for the start I'll just tell you about a few common
types of operators that you probably want to know about.

```
i = i + 3; /* addition (+) and assignment (=)
operators */
i = i - 8; /* subtraction, subtract 8 from i */
i = i / 2; /* division */
i = i * 9; /* multiplication */
i++; /* add one to i ("post-increment"; more
later) */
++i; /* add one to i ("pre-increment") */
i--; /* subtract one from i ("post-decrement") */
```

Looks pretty weird, that `i = i + 3` expression, huh. I mean, it makes no sense algebraically, right? That's true, but it's because it's not really algebra. That's not an equivalency statement--it's an assignment. Basically it's saying whatever variable is on the left hand side of the assignment (=) is going to be assigned the value of the expression on the right.

1.3.4 Expression

An expression in C consists of other expressions optionally put together with operators. The basic building block expressions that you put together with operators are variables, constant numbers (like 10 or 12.34), and functions and so when you chain these together with operators, the result is an expression, as well. All of the following are valid C expressions:

```
i = 3
i++
i = i + 12
i + 12
2
f += 3.14
```

Now where can you use these expressions? Well, you can use them in a function call (I know, I know--I'll get to function real soon now), or as the right hand side of an assignment. You have to be more careful with the left side of an assignment; you can only use certain things there, but for now suffice it to say it must be a single variable on the left side of the assignment, and not a complicated expression:

```
radius = circumference / (2.0 * 3.14159); /* valid */
diameter / 2 = circumference / (2.0 * 3.14159); /*
INVALID */
```

(I also slipped more operator stuff in there--the parentheses. These cause part of an expression (yes, any old expression) to be evaluated first by the compiler, before more of the expression is computed.

Check Your Progress

Fill in the Blanks

Q.1: A ………is simply a name for a number.

Q.2: A variable is defined as name given to the storage location in……………….

Q.3: Each……………. to be used in the program must be declared

Q.4: An expression in C consists of other expressions optionally put together with…………..

1.3.5 Statements

A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer program is made up of a series of statements.

What are these pesky statements? Let's say, completely hypothetically, you want to do something more than the already amazingly grand example program of assigning a value to a variable and printing it. What if you only want to print it if the number is less than 10? What if you want to print all numbers between it and 10? What if you want to only print the number on a Thursday? All these incredible things and more are available to you through the magic of various statements.

The if statement

The easiest one to wrap your head around is the conditional statement, if. It can be used to do (or not do) something based on a condition.

Like what kind of condition? Well, like is a number greater than 10?

```
int i = 10;
if (i > 10) {
printf("Yes, i is greater than 10.\n");
     printf("And this will also print if i is greater
than 10.\n");
}
if (i <= 10) print ("i is less than or equal to
10.\n");
```

In the example code, the message will print if i is greater than 10, otherwise execution continues to the next line. Notice the squirrley braces after the if statement; if the condition is true, either the first statement or expression right after the if will be executed, or else the collection of code in the squirlley braces after the if will be executed. This sort of code block behavior is common to all statements.

What are the conditions?

```
i == 10; /* true if i is equal to 10 */
i != 10; /* true if i is not equal to 10 */
i > 10; /* true if i greater than 10 */
i < 10; /* true if i less than 10 */
i >= 10; /* true if i greater than or equal to 10 */
i <= 10; /* true if i less than or equal to 10 */
i <= 10; /* true if i less than or equal to 10 */
```

Guess what these all are? No really, guess. They're expressions! Just like before! So statements take an expression (some statements take multiple expressions) and evaluate them. The if statement evaluates to see if the expression is true, and then executes the following code if it is.

What is “true” anyway? C doesn't have a “true” keyword like C++ does. In C, any non-zero value is true, and a zero value is false. For instance:

```
if (1) printf("This will always print.\n");
if (-3490) printf("This will always print.\n");
if (0) printf("This will never print. Ever.\n");
```

And the following will print `1` followed by `0`:

```
int i = 10;
printf("%d\n", i == 10); /* i == 10 is true, so it's
1 */
printf("%d\n", i > 20); /* i is not > 20, so this is
false, 0 */
```

We just passed those expressions as arguments to the function printf()! Just like we said we were going to do before!

Now, one common pitfall here with conditionals is that you end up confusing the assignment operator (=) with the comparison operator (==). Note that the results of both operators is an expression, so both are valid to put inside the if statement. Except one assigns and the other compares! You most likely want to compare. If weird stuff is happening, make sure you have the two equal signs in your comparison operator.

The while statement

Let's have another statement. Let's say you want to repeatly perform a task until a condition is true. This sounds like a job for the while loop. This works just like the if statement, except that it will repeately execute the following block of code until the statement is false, much like an insane android bent on killing its innocent masters. Or something.

Here's an example of a while loop that should clarify this up a bit and help cleanse your mind of the killing android image:

```
// print the following output:
//
// i is now 0!
// i is now 1!
// [ more of the same between 2 and 7 ]
// i is now 8!
// i is now 9!
i = 0;
while (i < 10) {
printf("i is now %d!\n", i);
i++;
}
printf("All done!\n");
```

The easiest way to see what happens here is to mentally step through the code a line at a time.

1. First, `i` is set to zero. It's good to have these things initialized.
2. Secondly, we hit the while statement. It checks to see if the *continuation condition* is true, and continues to run the following block if it is. (Remember, true is `1`, and so when `i` is zero, the expression `i < 10` is `1` (true).
3. Since the continuation condition was true, we get into the block of code. The printf() function executes and outputs "i is now 0!".
4. Next, we get that post-increment operator! Remember what it does? It adds one to `i`

   in this case. (I'm going to tell you a little secret about post-increment: the increment

   happens *AFTER all of the rest of the expression has been evaluated*. That's why it's

   called "post", of course! In this case, the entire expression consists of simply `i`, so the

   result here is to simply increment `i`.
5. Ok, now we're at the end of the basic block. Since it started with a while statement, we're going to loop back up to the while and then:
6. We have arrived back at the start of the while statement. It seems like such a long time

ago we were once here, doesn't it? Only this time things seem slightly different...what

could it be? The variable *i* is equal to 1 this time instead of 0! So we have to check the continuation condition again. Sure enough, 1 < 10 last

time I checked, so we enter the block of code again.

7. We **printf()** "i is now 1!".

8. We increment *i* and it goes to 2.

9. We loop back up to the while statement and check to see if the continuation condition
is true.

10. And where the variable *i* has finally been incremented so it's value is 10. Meanwhile, while we've slept in cryogenic hybernation, our program has been dutifully fulfilling its thousand-year mission to print things like "i is now 4!", "i is now 5!", and finally, "i is now 9!"

11. So *i* has finally been incremented to 10, and we check the continuation condition. It 10 < 10? Nope, that'll be false and zero, so the while statement is finally completed and we continue to the next line.

12. And lastly **printf** is called, and we get our parting message: "All done!".

That was a lot of tracing, there, wasn't it? This kind of mentally running through a program is commonly called *desk-checking* your code, because historically you do it sitting at your desk. It's a powerful debugging technique you have at your disposal, as well.

The do-while statement

So now that we've gotten the while statement under control, let's take a look at its closely related cousin, do-while.

They are basically the same, except if the continuation condition is false on the first pass, do-while will execute once, but while won't execute at all. Let's see by example:

```
/* using a while statement: */
```

```
i = 10;
// this is not executed because i is not less than
10:
while(i < 10) {
printf("while: i is %d\n", i);
i++;
}
/* using a do-while statement: */
i = 10;
// this is executed once, because the continuation
condition is
// not checked until after the body of the loop runs:
do {
printf("do-while: i is %d\n", i);
i++;
} while (i < 10);
printf("All done!\n");
```

Notice that in both cases, the continuation condition is false right away. So in the while, the condition fails, and the following block of code is never executed. With the do-while, however,

the condition is checked *after* the block of code executes, so it always executes at least once. In

this case, it prints the message, increments i, then fails the condition, and continues to the "All

done!" output.

The moral of the story is this: if you want the loop to execute at least once, no matter what

the continuation condition, use do-while.

The for statement

Now you're starting to feel more comfortable with these looping statements, Well, listen up! It's time for something a little more complicated: the for statement. This is another looping construct that gives you a cleaner syntax than while in many cases, but does basically the same thing. Here are two pieces of equivalent code:

```
// using a while statement:
// print numbers between 0 and 9, inclusive:
i = 0;
while (i < 10) {
```

```
printf("i is %d\n");
i++;
}
// do the same thing with a for-loop:
for (i = 0; i < 10; i++) {
printf("i is %d\n");
}
```

But you can see how the for statement is a little more compact and easy on the eyes. It's split into three parts, separated by semicolons. The first is the initialization, the second is the continuation condition, and the third is what should happen at the end of the block if the contination condition is true. All three of these parts are optional. And empty for will run forever:

```
for(;;) {
printf("I will print this again and again and
again\n" );
printf("for all eternity until the cold-death of the
universe.\n");
}
```

Before we start with functions in the next section, we're going to quickly tie this in with that very important thing to remember back at the beginning of the guide. Now what was it...oh, well, I guess I gave it away with this section title, but let's keep talking as if that didn't happen. Yes, it was basic building blocks, and how you take a specification and turn it into little bite-sized pieces that you can easily translate into blocks of code. I told you to take it on faith that I'd tell you some of the basic pieces, and I'm just reminding you here, in case you didn't notice, that all those statements back there are little basic building blocks that you can use in your programs.

Check Your Progress

Q.5: A ……………..is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input.

Q.6 A ……loop in C programming repeatedly executes a target statement as long as a given condition is true.

Q.7: An…………… is a symbol that tells the compiler to perform a certain mathematical or logical manipulation.

1.4 Answer to check your progress

Ans to Q.1: Variable

Ans to Q.2: computer memory.

Ans to Q.3: Variable

Ans to Q.4: Operators

Ans to Q.5: Statement

Ans to Q.6: While

Ans to Q.7: operator

1.5 Model Questions

1. What is the difference between a keyword and an identifier ?
2. List the rules of naming an identifier in C ?
3. Differentiate between a variable and a constant.
4. Define constants in C.What are the different types of constants in C? Give suitable examples.
5. How to declare variable? Explain with the help of example.

Unit-3

Functions

1.1 Learning Objectives

1.2 Introduction

1.3 Functions

- **1.3.1** Use of Function
- **1.3.2** Function declaration

1.4 Function call

- **1.4.1** Call by Value
- **1.4.2** Call by Reference

1.5 Nesting of Functions

1.6 Function Parameters

1.7 Categories of Function

1.8 Recursive Function

1.9 Answer to check your progress

1.10 Model Questions

1.1 Learning Objectives

After going through this unit the learner will able to learn about:

- learn about function and its use in programs
- declare a function
- define a function
- describe function call
- learn about nesting of function
- learn about function parameters
- describe function categories
- illustrate recursive function
- Function Prototype

1.2 Introduction

In C programming, all executable code resides within a function. A function is a named block of code that performs a task and then returns control to a caller. Note that other programming languages may distinguish between a "function", "subroutine", "subprogram", "procedure", or "method" -- in C, these are all functions.

A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a function, the program will branch back (return) to the point after the call. Functions are a powerful programming tool.

However, C language also allows the users to define their own functions for carrying out various tasks. This unit concentrates on the creation and utilization of such ***user-defined*** functions.With the proper use of such user-defined functions, a large program can be broken down into a number of smaller, self-contained components, each of which has some unique purpose. This unit will help you in writing user-defined functions.

1.3 Functions

With the previous section on building blocks fresh in your head, let's imagine a freaky world where a program is so complicated, so insidiously large, that once you shove it all into your **`main()`**, it becomes rather unwieldy.

What do I mean by that? The best analogy I can think of is that programs are best read, modified, and understood by humans when they are split into convenient pieces, like a book is most conveniently read when it is split into paragraphs.

Ever try to read a book with no paragraph breaks? It's tough, man, believe me. I once read through *Captain Singleton* by Daniel Defoe since I was a fan of his, but Lord help me, the man
didn't put a single paragraph break in there. It was a brutal novel.

But I digress. What we're going to do to help us along is to put some of those building blocks in their own functions when they become too large, or when they do a different thing than the rest of the code. For instance, the assignment might call for a number to be read, then the sum of all number between 1 and it calculated and printed. It would make sense to put the code for calculating the sum into a separate function so that the main code a) looks cleaner, and b) the function can be *reused* elsewhere.

Reuse is one of the main reasons for having a function. Take the **`printf()`** for instance. It's pretty complicated down there, parsing the format string and knowing how to actually output
characters to a device and all that. Imagine if you have to rewrite all that code every single time
you wanted to output a measly string to the console? No, no--far better to put the whole thing in
a function and let you just call it repeatedly.

You've already seen a few functions called, and you've even seen one *defined*, namely the almighty **main()** (the definition is where you actually put the code that does the work of the function.) But the **main()** is a little bit incomplete in terms of how it is defined, and this is allowed for purely historical reasons. More on that later. Here we'll define and call a normal function called **plus_one()** that take an integer parameter and returns the value plus one:

```
int plus_one(int n) /* THE DEFINITION */
{
return n + 1;
}
int main(void)
{
int i = 10, j;
j = plus_one(i); /* THE CALL */
printf("i + 1 is %d\n", j);
return 0;
}
```

(Before I forget, notice that I defined the function before I used it. If hadn't done that, the compiler wouldn't know about it yet when it compiles **main()** and it would have given an unknown function call error. There is a more proper way to do the above code with function prototypes, but we'll talk about that later.) So here we have a function definition for **plus_one()**. The first word, `int`, is the return type of the function. That is, when we come back to use it in **main()**, the value of the expression (in the case of the call, the expression is merely the call itself) will be of this type.

By wonderful coincidence, you'll notice that the type of *j*, the variable that is to hold the return

value of the function, is of the same type, `int`. This is completely on purpose. Then we have the function name, followed by a *parameter list* in parenthesis. These correspond to the values in parenthesis in the call to the function...but *they don't have to have the same names*. Notice we call it with *i*, but the variable in the function definition is named *n*.

This is ok, since the compiler will keep track of it for you.

Inside the **`plus_one()`** itself, we're doing a couple things on one line here. We have an expression `n + 1` that is evaluated before the return statement. So if we pass the value 10 into
the function, it will evaluate `10 + 1`, which, in this universe, comes to 11, and it will return
that.

Once returned to the call back in **`main()`**, we do the assignment into *`j`*, and it takes on the return value, which was 11.

I mentioned that the names in the parameter list of the function definition correspond to the *values* passed into the function. In the case of **`plus_one()`**, you can call it any way you like, as long as you call it with an `int`-type parameter. For example, all these calls are valid:

```
int a = 5, b = 10;
plus_one(a); /* the type of a is int */
plus_one(10); /* the type of 10 is int */
plus_one(1+10);  /*  the  type  of  the  whole  expression  is
still int */
plus_one(a+10);  /*  the  type  of  the  whole  expression  is
still int */
plus_one(a+b);  /*  the  type  of  the  whole  expression  is
still int */
plus_one(plus_one(a));  /*  oooo!  return  value  is  int,  so
it's ok! */
```

If you're having trouble wrapping your head around that last line there, just take it one expression at a time, starting at the innermost parentheses (because the innermost parentheses are evaluated first, rememeber?) So you start at *`a`* and think, that's a valid `int` to call the function **`plus_one()`** with, so we call it, and that returns an `int`, and that's a valid type to call the next outer **`plus_one()`**.

What about the return value from all of these? We're not assigning it into anything! Where is it going? Well, on the last line, the innermost call to **`plus_one()`** is being used to call **`plus_one()`** again, but aside from that, you're right--they are being discarded completely. This
is legal, but rather pointless unless you're just writing sample code for demonstration purposes.

It's like we wrote “5” down on a slip of paper and passed it to the **`plus_one()`** function, and it went through the trouble of adding one, and writing “6” on a slip of paper and passing it back to us, and then we merely just throw it in the trash without looking at it. We're such bastards.

I have said the word “value” a number of times, and there's a good reason for that. When we pass parameters to functions, we're doing something commonly referred to as *passing by value*. This warrants its own subsection.

1.3.1 Use of Function

A function is a set of statements that carries out some specific task in a program and it can be processed independently. Every C program consists of one or more functions. One of these functions must be called ***main***. Program execution will always begin by carrying out the instructions in *main*.

There are many advantages in using functions in a program. They are:

- Many programs require that a specific function is repeated many times. Instead of writing the function code as many times as it is required, we can write it as a single function and access the same function again and again as many times as it is required.

- The length of the source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function instead of modifying the whole program.
- If the whole large program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.
- A single function written in a program can be used in other programs also.

1.3.2 Function Declaration

A function declaration is also known as ***function prototype.*** Function prototypes are usually written at the beginning of the program, ahead of any user defined functions including *main*. It hints to the *compiler* that the ***main()*** function is going to call the function which is declared, later in the program. The general format of a function prototype is as follows:

```
return_type function_name( type1 arg1, type2
arg2,..,typeN argN);
```

where, ***return_type*** represents the data type of the item that is returned by the function, ***function_name*** represents the name of the function, ***type1, type2,...,typeN*** represent the data types of the arguments ***arg1, arg2, , , , ,argN.*** It is necessary to use a semicolon at the end of function prototype. Arguments name *arg1*, *arg2* etc. can be omitted. However, the argument data types are essential. For example:

int add(int, int);

In the above prototype declaration, *int* is the data type of the item returned by the function, *add* is the name of the function and *int* within the brackets '(' and ')' are the data types of the arguments.

Program: Program to find the sum of two numbers.

Solution:

```
#include<stdio.h>
#include<conio.h>
int add(int,int); //function prototype or
declaration
void main()
{
int a,b,s; // integer variable a,b,s are
declared
clrscr();
printf("Enter two integer number \n"); / /
display statement
scanf("%d%d", &a,&b);
s=add(a,b); //function call
printf("\nThe summation is %d", s);
getch();
}
int add(int a, int b)
{
int sum=0; // local variable sum and it is
intialised to zero
sum=a+b;
return sum; // value of sum is returned
}
```

1.4 Function call

Once a function has been declared and defined, it can be called from anywhere within the program: from within the *main()* function, from another function, and even from itself. We can call a function by simply using the function name followed by a list of parameters(or arguments) if any, enclosed in parentheses. For example,

s = add (a,b) ; //Function call

In the above statement **add(a,b)** function is called and value returned by it is stored in the variable **s**. When the compiler encounters a function call, the control is transferred to **int add(int x, int y)**. This function is then executed line by line as described and a value is returned when a return statement is encountered. In our example, this value is assigned to **s**. This is illustrated below:

Program: Program to find the summation of two numbers using function

```
#include<stdio.h>
#include<conio.h>
int add(int, int); // function declaration
void main()
{ int a,b,s;
    clrscr();
printf("Enter two integer number \n");
scanf("%d%d", &a,&b);
s=add(a,b); // function call
printf("\nThe summation is %d", s);
getch();
}
int add(int x, int y) //function header
{
int sum=0; //local variable sum and it is
intialised to zero
sum=x+y;
return sum; //value stored in sum is returned
to the calling function
    }
```

Parameter passing is a method for communication of data between the *calling function* and *called function*. These can be achieved by two ways:

- Call-by-value
- Call-by-reference

1.4.1 Call-by-value

In case of *call-by-value*, the compiler copies the value of an argument in a *calling function* to a corresponding parameter in the *called function* definition. The parameter in the called function is initialized with the value of the passed argument. As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are performed only within the scope of the *called function*; they have no effect on the value of the argument in the *calling function*.

In the following example, the *calling function* **main()** passes two values 5 and 10 to the *called function* **func()**. The function **func()** receives copies of these values and accesses them by the identifiers **a** and **b**. The function **func()** changes the value of **a**. When control passes back to **main()**, the actual values of **x** and **y** are not changed.

Program: Program to illustrate calling a function by value.

```
#include<stdio.h>
void func(int, int);
void main(void)
{
int x = 5, y = 10;
clrscr();
func(x, y);
printf("In main, x = %d y = %d\n", x, y);
}
void func(int a, int b)
{
a = a + b;
printf("In func, a = %d b = %d\n", a, b);
}
```

Output:
```
In func, a = 15 b = 10
In main, x = 5 y = 10
```

1.4.2 Call-by-reference

Call-by-reference refers to a method of calling a function by passing the address of an argument in the *calling function* to a corresponding parameter in the *called function*.

We have used an example below to illustrate the concept of *call-by-value* and *call-by-reference*. The aim of the two programs listed below is to perform swapping (interchange) of two values.

Program: Example to illustrate calling a function by value.

```
#include<stdio.h>
#include<conio.h>
void   swap(int,   int);   //function   prototype   or
declaration
void main()
{
int a,b;
a=5;
b=10;
printf("a and b before interchange: %d %d", a,
b);
swap(a,b); //function call
printf("\na and b after interchange: %d %d",
a, b);
getch();
}
void swap(int i, int j) //function definition
{
int t;
t = i;
i = j;
j = t;
}
```

Here, the value to function *swap( )* is passed by value.When we execute this program, we will find that no swapping takes place.

The values of **a** and **b** are passed to swap, and the swap function does swap them, but when the function returns to main() nothing happens. The values of **a** and **b** are still the same. The output will
be:

a and b before interchange: 5 10

a and b after interchange: 5 10

In the next program we use pointers to perform call-by-reference for swapping of two values. Our next unit will help you in understanding ***pointers***

Program: Example to illustrate calling a function by reference

```
#include <stdio.h>
#include<conio.h>
void swap(int *, int *); //function declaration
void main( )
{
int a,b;
a=5;
b=10;
clrscr( ); // clearing the screen
printf("a and b before interchange: %d
%d\n",a,b);
swap(&a,&b); //function call
printf("a and b after interchange: %d
%d\n",a,b);
}
void swap(int *i, int *j)
{
int t;
t = *i;
*i = *j;
*j = t;
}
```

Here, the function uses called-by-reference. In other words, address is passed by using the symbol **"&"** and the value is accessed by using the symbol **"*"**.When **swap( )** function is called, the addresses of **a** and **b** are passed to the function. Thus, **i** points to **a** and **j** points to **b**. Once the pointers are initialized by the function call, ***i** is another name for **a**, and

***j** is another name for **b**. When the code uses ***i** and ***j**, it really means **a** and **b**. So, when we interchange the values of **i** and **j** in function **swap()**, we interchange the values of **a** and **b**. Hence, when the function is complete, **a** and **b** have been interchanged. The output will be:

```
a and b before interchange: 5 10
a and b after interchange: 10  5
```

1.5 Nesting of Functions

C permits nesting of functions freely. There is no limit to how deeply functions can be nested. A *nested function* is encapsulated within another function. Suppose a function **a** can call function **b** and function **b** can call function **c** and so on. We have taken the following example to illustrate nesting of function.

Program: Program to illustrate the concept of nested function

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
      float r;
clrscr();
float ratio(int,int,int); // function
ratio( ) declared
printf("Enter a,b and c :");
scanf("%d%d%d",&a,&b,&c);
r=ratio(a,b,c); // ratio( ) function called
printf("%f\n",r);
getch();
}
float ratio(int x, int y, int z)
{
```

```
int difference(int,int); // function
difference( ) declared
if(difference(y,z))
return(x/(y-z));
else
return(0.0);
}
int difference(int p, int q)
{
if(p!=q)
return(1);
else
return(0);
      }
```

The above program calculates the ratio **a / b - c** and prints the result. We have the following three functions:

main( )

ratio( )

difference( )

The **main( )** function reads the value of a,b,c and calls the function **ratio( )** to calculate the value a / (b-c). This ratio cannot be evaluated if (b - c) =0. Therefore, **ratio( )** calls another function **difference( )** to test whether the **difference(b-c)** is zero or not.

1.6 Function Parameters

C functions exchange information by means of parameters. The term ***parameter*** refers to any declaration within the parentheses following the function name in a function declaration, definition or function call.

Formal Parameters: The parameters which appear in the first line of the function definition are referred to as *formal parameter*. Formal parameters are written in the function prototype and function header of the definition. Formal parameters are local variables which are assigned values from the arguments when the function is called.

Actual Parameters: When a function is called, the values (expressions) that are passed in the call are called the *actual parameters*. At the time of the call each actual parameter is assigned to the corresponding formal parameter in the function definition. It may be expressed in constants, single variables, or more complex expressions. However, each actual parameter must be of the same data type as its corresponding formal parameter.

The following rules apply to parameters of C functions:

- Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.
- Arguments are separated by commas.
- The scope of function parameters is the function itself. Therefore, parameters of the same name in different functions are unrelated. Let us consider the following example to illustrate the concept of formal and actual parameters:

Program: Program to illustrate the concept of formal and actual parameters

```
   #include<stdio.h>
   #include<conio.h>
void display(int,int);
   void main()
   { int a,b;
   display(a,b);
   getch();
   }
   void display(int x, int y)
   {
   printf("%d%d",x,y);
}
```

Here, **x** and **y** are formal parameters and take the value (**a,b**) from the calling function **display(a,b)**.

Check Your Progress

Q.1: Fill in the blacks:

i) When a function returns nothing then the return type is__________.

ii) If a C program has only one function then that function is__________.

iii) The parameters used in a function call are __________.

iv) When a variable is passed to a function by value, its value remains __________ in the calling program.

v) A function can be called either by __________ or__________ or both.

1.7 Categories of Function

Depending on whether arguments are present or not and whether a value is returned or not, functions are categorized as follows:

- Functions with no arguments and no return values
- Functions with arguments and no return values
- Functions with arguments and one return value
- Functions with no arguments but a return value
- Functions that return multiple values

We have illustrated the above categories of functions by using different programs. The concept of different categories of functions is explained using an example to “Multiply of two integer numbers”.

- **Functions with no arguments and no return values**

  Program: Program to illustrate the concept of a function with no arguments and no return values

```
#include<stdio.h>
#include<conio.h>
void multi(void); //function declaration with
no argument
void main( )
{
clrscr( );
multi( );
getch( );
}
void multi(void)
{
int a,b,m;
printf("Enter two integers:");
scanf("%d%d", &a,&b);
m=a*b;
printf("\nThe product is: %d",m);
}
```

- **Functions with arguments and no return values**

  Program: Program to illustrate the concept of a function with arguments and no return values

```
#include<stdio.h>
#include<conio.h>
void multi(int,int); //function declaration with
two argument
void main( )
{
```

```
int a,b;
clrscr( );
printf("Enter two integers:");
scanf("%d%d", &a,&b);
multi(a,b);
getch( );
}
void multi(int a,int b )
{ int m;
m=a*b;
printf("\nThe product is: %d",m);
}
```

- Functions with arguments and one return value

  Program: Program to illustrate the concept of a function with arguments and one return values

```
#include<stdio.h>
#include<conio.h>
int multi(int,int); //function declaration with
two argument
void main( )
{
int a,b,m;
clrscr( );
printf("Enter two integers:");
scanf("%d%d", &a,&b);
m=multi(a,b);
printf("\nThe product is: %d",m);
getch( );
}
int multi(int a,int b )
{
int z;
z=a*b;
return z; /*return statement. the value of
z is returned to the calling
function*/
}
```

- **Functions with no arguments but a return value**

  Program: Program to illustrate the concept of a function with no arguments but a return value

```
#include<stdio.h>
#include<conio.h>
int multi(void); //function declaration with
no argument
void main( )
{
int m;
clrscr( );
m=multi( );
printf("\nThe product is: %d",m);
getch( );
}
int multi(void)
{
int a,b,p;
printf("Enter two integers:");
scanf("%d%d", &a,&b);
p=a*b;
return p;
}
```

Return statement can return only one value. In C, the mechanism of sending back information through arguments is achieved by two operators known as the *address operator* (&) and *indirection operator* (*).

Let us consider an example to illustrate this.

- **Functions returning multiple values**

Program: Program to illustrate the concept of functions returning multiple values

```
#include<iostream.h>
#include<conio.h>
void calculate(int, int, int *, int *);
void main( )
{
int a,b,s,d;
clrscr( );
printf("\nEnter two integer:");
scanf("%d%d",&a,&b);
calculate(a,b,&s,&d);
printf("\nSummation is:%d \n Difference is:%d",
s,d);
getch();
}
void calculate(int x,int y, int *sum, int *diff)
{
*sum=x+y;
*diff=x-y;
}
```

In the fuction call, while we pass the actual values of a and b to the function calculate(), we pass the address of locations where the values of s and d are stored in the memory.

When the function is called, the value of **a** and **b** are assigned to **x** and **y** respectively. Address of **s** and **d** are assigned to **sum** and **diff** respectively. The variables ***sum** and ***diff** are known as pointers and **sum** and **diff** pointer variables. Since they are declared as **int**, they can point to locations of **int** type data.

1.8 Recursive Function

When a function calls itself it is called a ***recursive function***. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. A very simple example is
presented below:

Program: Program to illustrate the concept of recursive function

```
#include<stdio.h>
#include<conio.h>
void main( )
{
printf("Recursive function\n");
main( );
getch( );
}
```

The output of the above programme will be like this:

```
Recursive function
Recursive function
Recursive function
Recursive function
............................
............................
```

In the above case, we will have to terminate the execution abruptly; otherwise the program will execute indfinitely.

The factorial of a number can also be determined using recursion. The factorial of a number **n** is expressed as a series of repeatitive multiplications as shown below:

Factorial of n = n(n-1)(n-2)(n-3).....1

For example, factorial of 5= 5x4x3x2x1 =120

Program: Program to find factorial of an integer number

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

```
#include<conio.h>
long int factorial(int);
void main( )
{
int n ;
long int f ;
clrscr( ) ;
printf("\nEnter an integer number:") ;
scanf("%d", &n) ;
f=factorial(n) ;
printf("\nThe factorial of %d is : %ld",n,f) ;
getch() ;
}
long int factorial(int n)
{
long int fact ;
if(n<=1)
return(1);
else
fact=n*factorial(n-1);
return(fact);
}
```

Let us see how recursion works assuming n = 5. If we assume n=1 then the factorial( ) function will return 1 to the calling function. Since n ¹1, the statement

fact = n * factorial (n-1);

will be executed with n=5. That is,

fact = 5 * factorial (4);

will be evaluated. The expression on the right-hand side includes a call to factorial with n = 4 .This call will return the following value:

4 * factorial(3)

In this way factorial(3), factorial(2), factorial(1) will be returned. The sequence of operations can be summarized as follows:

fact = 5 * factorial (4)

= 5 * 4 * factorial (3)

= 5 * 4 * 3 * factorial (2)

= 5 * 4 * 3 * 2 * factorial (1)

= 5 * 4 * 3 * 2 * 1

=120

When we write recursive functions, we must have an *if* statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

Program: Program to find the sum of digits of a number using recursion.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int sum(int); //function prototype
int n,s;
clrscr();
printf("\n Enter a positive integer:");
scanf("%d",&n);
s=sum(n);
printf("\n Sum of digits of %d is %d ", n,s);
getch();
}
int sum(int n)
{
if(n<=9)
return(n);
else
return(n%10+sum(n/10)); // recursive
call of sum()
}
```

Output: Enter a positive integer: 125

Sum of digits of 125 is 8

Check Your Progress

Q.2: Write down the syntax of function definition.

Q.3: Write the first line of the function definition, including the formal argument declarations, for each

of the situations described below:

i) A function called ***average*** accepts two integer arguments and returns a floating-point result.

ii) A function called ***convert*** accepts a character and returns another character.

1.9 Answer to check your progress

Ans. to Q. No. 1: i) void, ii) main( ), iii) actual parameters, iv) unchanged, v) call by value, call byreference

Ans. to Q. No. 2: General format of function definition is given below:

```
return_type function_name(parameter list)
{
local variable declaration;
executable statement1 ;
executable statement2 ;
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
return statement ;
}
```

Ans. to Q. No. 3: i) float average(int a, int b) ii) char convert(char a)

1.10 Model Questions

1. What is a main() function?
2. What is recursion? Explain it with example.
3. What is the purpose of return statement?
4. What are formal and actual arguments? What is the relationship between formal and actual argument?
5. What is meant by a function call?

Unit-4
Variables

1.1 Learning Objectives
1.2 Introduction
1.3 Variables, The Sequel
1.4 UP Scope
1.5 Storage Classes
1.6 Automatic variable
1.7 External Variable
1.8 Static Variable
1.9 Register Variable
1.10 Answer to Check Your Progress

1.11 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will able to:

- Learn about, up scope, global variable
- Learn about Storage Class
- Describe Automatic, External, Static and Register Variable
- Describe the Scope of Variables
- Define lifetime of a Variable

1.2 Introduction

We already have some basic idea about variables. Variables can be defined as the memory location where we can store the values of a particular data type. The value stored in the variable may be changed during the program executions.

Every C variable has a storage class and a scope. This storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist during the execution of program. The storage class also determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name. In this unit, we will discuss the various storage classes.

1.3 Variables, The Sequel

We're going to talk about a couple things in this section that increase the power you have over variables *TO THE EXTREME*. Yes, by now you realize that melodrama is a well-respected part of this guide, so you probably weren't even taken off-guard by that one, ironically. Let's talk about variable *scope* and *storage classes*.

1.4. Up Scope

You recall how in some of those functions that we previously defined there were variables that were visible from some parts of the program but not from others? Well, if you can use a variable from a certain part of a program, it's said to be *in scope* (as opposed to *out of scope*.) A variable will be in scope if it is declared inside the block (that is, enclosed by squirrley braces) that is currently executing.

Take a look at this example:

```
int frotz(int a)
{
int b;
b = 10; /* in scope (from the local definition) */
a = 20; /* in scope (from the parameter list) */
c = 30; /* ERROR, out of scope (declared in another
block, in main()) */
}
int main(void)
{
int c;
c = 20; /* in scope */
b = 30; /* ERROR, out of scope (declared above in
frotz()) */
return 0;
}
```

So you can see that you have to have variables declared locally for them to be in scope. Also note that the parameter `a` is also in scope for the function **`frotz()`** What do I mean by *local variables*, anyway? These are variable that exist and are visible only in a single basic block of code (that is, code that is surrounded by squirrley braces) and, basic blocks of code within them. For instance:

```
int main(void)
{ /* start of basic block */
int a = 5; /* local to main() */
if (a != 0) {
int b = 10; /* local to if basic block */
a = b; /* perfectly legal--both a and b are visible
here */
}
b = 12; /* ERROR -- b is not visible out here--only
in the if */
{ /* notice I started a basic block with no
statement--this is legal */
int c = 12;
int a; /* Hey! Wait! There was already an "a" out in
main! */
/* the a that is local to this block hides the a from
main */
a = c; /* this modified the a local to this block to
be 12 */
}
/* but this a back in main is still 10 (since we set
it in the if): */
printf("%d\n", a);
return 0;
}
```

There's a lot of stuff in that example, but all of it is basically a presentation of a simple rule:

when it comes to local variables, you can only use them in the basic block in which they are declared, or in basic blocks within that. Look at the “ERROR” line in the example to see exactly

what *won't* work. Let's digress for a second and take into account the special case of parameters passed to functions. These are in scope for the entire function and you are free to modify them to your heart's content. They are just like local variables to the function, except that they have copies of the data you passed in, obviously.

```
void foo(int a)
{
int b;
a = b; /* totally legal */
}
```

Global variables

There are other types of variables besides locals. There are *global variables.* A global variable is visible thoughout the entire file that it is defined in (or declared in more on that later). So it's just like a local, except you can use it from anyplace. I guess, then, it's rather not like a local at all. But here's an example:

```
#include <stdio.h>
/* this is a global variable. We know it's global,
because it's */
/* been declared in "global scope", and not in a
basic block somewhere */
```

```
int g = 10;
void afunc(int x)
{
g = x; /* this sets the global to whatever x is */
}
int main(void)
{
g = 10; /* global g is now 10 */
afunc(20); /* but this function will set it to 20 */
printf("%d\n", g); /* so this will print "20" */
return 0;
}
```

Remember how local variables go on the stack? Well, globals go on *the heap*, another chunk of memory. And never the twain shall meet. You can think of the heap as being more "permanent" than the stack, in many ways.

Now, I mentioned that globals can be dangerous. How is that? Well, one thing you could imagine is a large-scale project in which there were a bazillion globals declared by a bazillion
different programmers. What if they named them the same thing? What if you thought you were
using a local, but you had forgotten to declare it, so you were using the global instead? (That's a good side note: if you declare a local with the same name as a global, it hides the global and all operations in the scope will take place on the local variable.) What else can go wrong? Sometimes using global variables encourages people to not structure their code as well as they might have otherwise. It's a good idea to not use them until there simply is no other reasonable way to move data around.

Another thing to consider is this: does it actually make sense to have this data stored globally for all to see? For example, if you have a game where you use "the temperature of the world" in a lot of different places that might be a good candidate for a global varible. Why? Because it's a pain to pass it around, and everyone has the same need for it.

1.5 Storage Classes

Storage class is related to the declaration of variable, function, and parameters that we use in C programs. The storage class in the function is used when returning a value of a particular data type from the function. The storage class specifier used within the declaration determines whether:

- the object is to be stored in memory or in a register.
- the object receives the default initial value or an indeterminate default initial value.
- the object can be referenced throughout a program or only within the function, block, or source file where the variable is defined.

Here, the term 'object' refers to variable, function and parameters in which storage class is going to be used. Depending upon the above, storage class can be classified into four categories:

I. Automatic
II. Register
III. Static
IV. External

Now, let us try to understand each storage class in the next section using examples.

1.6 Automatic variable

We already know how variables are used in C program. Now, we are going to use storage class in the variable declarations. Actually, we have already used automatic storage class in the programs in the previous units. Let us take an example as shown below:

Program: Program to illustrate the use of automatic variable.

```
void main()
{
int a,b,s; // or we can write here as auto
int a,b,s;
scanf("%d %d", &a,&b);
s=a+b;
printf("Sum is %d",s);
}
```

By default all the variables declared are automatic. We can also explicitly define the variables using **auto** keyword.

Let us look at the special properties of automatic storage that make it different from other storage class. The automatic variable has the following characteristics:

a. **Storage:** The value of the variable is stored in the memory of the computer (not in register).

b. **Default intial value:** The default initial value for automatic variables is garbage value i.e. any unpredictable value. It means that if the variable is not initialized then the variable contains some useless value.

c. **Scope:** The scope means the availability of the variable. Automatic variable is local to the block in which the variable is declared. Outside this block the variable cannot be accessed.

d. **Life time:** The life time of this variable is within the block it is declared.

Program: Program to illustrate the use of automatic variables.

```
void main ()
{
auto int i,j;
i=10;
printf("i= %d \n j= %d",i,j);
}
```

Output: i=10

j=8214 (or some other garbage value)

Since here we initialized **i** to 10 explicitly, therefore the value of **i** is displayed as 10; But in case of **j,** as we do not assign any value so garbage value is displayed at output. This example describes about the default initial value of auto variables. Now, let us understand about the life time and scope of automatic variables using another example.

Program: Program to show lifetime and scope of automatic variables.

```
void main ()
{
auto int i=1;
{
auto int i=2;
{
auto int i=3;
printf("%d",i);
}
printf("%d",i);
}
printf("%d",i);
}
```

Output: 3

2

1

The program has three blocks and each block initializes the value of **i**. Note that variable **i** allocates extra memory for each declarations and each **i** is different from one another. The first inner block is executed and therefore 3 is displayed as the first output as in this block **i=3.**After that the second inner block is executed and 2 is displayed as in the second inner block the value of **i** is 2**(i=2)**. Next, we come to the outer block where the value of **i** is 1**(i=1)** so 1 is displayed as the last output. We have seen that the value of **i** is different in each block. Whatever value we have initialized **i** with, it remains valid only within the block where we have declared it. This is known as **scope** of the variable.

1.7 External Variable

We have now learnt about automatic variable which is local to the block in which it is declared. But sometimes we need a variable which should be available to all the functions and blocks within the program. External storage class fits this need. The properties of the external variable are:

a. **Storage:** The external storage variables are stored in memory.
b. **Default initial value:** The default initial value for external variables is zero.
c. **Scope:** The scope for external variables is global i.e., the variable is available to all functions and
blocks within the program.
d. **Life time:** The lifetime of the variables is until the program's execution stops.

The main difference between automatic and external variable is in the scope and life time of the variables. They have similarities in storage and default initial value. External variables are declared outside all functions. Let us understand the scope and life time of external variables using examples.

Program: Program to illustrate the concept of external variables.

```
int var; // external variable
void main()
{
printf("%d",var); // just print the value of
'var'
var_add_two(); // add 2 to var and display
it
var_sub_one();
}
void var_add_two()
{
var=var+2;
printf("\n%d",var);
}
void var_sub_one()
{
var=var-1;
printf("\n%d",var);
}
```

Output: 0

2

1

In the above pogram, the first output is 0 since default initial value of external variable is 0.Next we increment 'var' by 2, so the next output is 2 and after that we decrement the value of 'var' by one so the output is 2-1 i.e. 1. Note that, here the value of 'var' is visible to the funcitons var_add_two() and var_sub_one() each of which modifies value of 'var'.

Check Your Progress

Q.1: Write true or false:

a. The default initial value of external variable is same as with automatic variable.
b. By default the variables declared are automatic.
c. The storage for the automatic variable is in the registers.

Q.2: Write down one similarity between automatic and external variables.

1.8 Static Variable

Static variable is similar to the automatic variable. Like the automatic variables static variables are local to the block in which they are declared. The difference between them is that for static variables the value does not disappear when the function is no longer active. The last updated value for static variable always persists. That is, when the control comes back to the same function again, the static variables have the same value as they left at the last time. Properties of static variables are:

- **a. Storage:** The static variables are stored in memory.
- **b. Default initial value:** The default initial value for static variables is zero.
- **c. Scope:** The scope of static variables is global.
- **d. Life time:** The life time of static variables is untill the program's execution does not stop.

Program: Program to illustrate the concept of static variable.

```
void add_one();
void main()
{
add_one();
add_one();
}
void add_one()
{
static int var=3;
var=var+1
printf("\n %d",var);
}
```

Output: 4

```
      5
```

It can be observed that the output of the above program is 4 and 5. Since we have initialized 'var' with 3 and then increment 'var' by 1 so the first output is 4 during the first call of the add_one() function. The variable 'var' retains its previous value 4 and thus in the second call of the add_one() function, increments 4 by 1; thus the output is 5. If we write the above function definition as:

```
void add_one()
{
int var=3; //or auto int var=3;
var=var+1
printf("\n %d",var)
}
```

The output of the program after the modifications should be: 4

```
            4
```

The reason behind this is that the automatic storage class variable does not retain its previous value. Whenever the add_one() function is called 'var' has always initialized value 3 and then increment by 1; so the output is always 4.

1.9 Register Variable

We have discussed about the automatic, static, external storage classes in earlier sections. Each class stores the variables in the memory of the computer. We know that there are mainly two areas for storing data in the computer– Memory and CPU registers. Accessing data from the register is faster than from memory. This makes the program to run faster. Register storage class makes it possible for the program to run faster. We use the register class with variables which accessed frequently like loops (such as for, do-while etc.).

The characteristics of register variables in terms of scope, life time etc are:

a) Storage: The register variables are stored in CPU registers.

b) Default initial value: The register variables take garbage values as the default values.

c) Scope: The scope of register variables is local to the block in which it is declared.

d) Life time: The life time of register variables is till the control remains within the particular block where it is declared.

Program: Program to illustrate the concept of register storage class variables.

```
void main()
{
register int var;
for(var=1; var<=10; var++)
printf("%d",var);
}
```

A question that may arise in your mind here is that if we declare most of the variables as register then every program should run faster. But then why do we not always use this concept. This is because the number of registers is limited in a computer system and so we cannot use register class for all variables. If in a program the total number of register variables exceeds the system register quantity; then all the variables that exceed are by default declared as automatic.

For example, if we write a program with a loop that uses 20 register variables (assume) and the computer has only 16 CPU registers, then the rest of the 4 variables are automatically transformed to automatic storage class variables. A important point to be noted here, is that register storage cannot be used for float and double data type since CPU registers usual capacity is 16 bit and both float and double data types require 4 byte (4 x 8 = 32 bit) and 8 byte (8 x 8 = 64 bit) storage respectively.

Check Your Progress

Q.3: Identify from the following which statements are true:

a. Default initial value of register storage class variable is zero.

b. Scope of external and register variables are not same.

c. There is no limit of using register variable.

Q.4: Write one difference between register and automatic storage class variable.

1.10 Answer to Check Your Progress

Ans. to Q. No. 1: a) False, b) True, c) False

Ans. to Q. No. 2: One similarity between automatic and external variables is that the storage location for both class variables are in the memory.

Ans. to Q. No. 3: a) False, b) True, c) False.

Ans. to Q. No. 4: Register class variable stores value in CPU registers whereas automatic class variable use memory for storing data.

1.11Model Questions

1. Define storage class of a variable. What are the different types of storage class?
2. Compare external and automatic storage class variables.
3. Explain the static and automatic storage class variable with examples.
4. Why are register storage class variables used with loop counter variables?
5. Mention the common factor among automatic, static and external variables.

Block-II

Unit-5

Pointers

1.1 Learning Objectives

1.2 Introduction

1.3 Pointers

- **1.3.1** Memory and Variables
- **1.3.2** Pointer Types
- **1.3.3** Dereferencing

1.3.4 Passing Pointers as Parameters
1.4 Pointer Arithmetic
1.5 Pointers and One Dimensional Arrays
1.6 Pointers and Character Arrays
1.7 Passing Pointers to Functions as Arguments
1.8 Dynamic Memory Allocation
1.8.1 Library Function for Dynamic Memory Allocation
1.9 Answer to Check Your Progress
1.10 Model Questions

1.1 Learning Objectives

After going through this unit the learner will be able to:

- Learn the basic concept of pointer
- Learn about memory and variable
- Learn about types of pointer
- Declaring the pointer variable
- Access array elements using pointer
- Relate pointers to functions

1.2 Introduction

A pointer is a variable that represents the location of a data item or memory area. In other words, pointer variable holds address of other memory location rather than a value. Pointers make C and C++more reliable and flexible. One can access the memory location directly using pointer. Within the computer memory, every data item occupies one or more contiguous memory locations. The number of required memory location depends on the data type used. For example, a character type variable occupies 1 byte (8 bits) of memory, an integer usually requires two contiguous bytes (16bits), a floating-point number requires4 contiguous bytes (32 bits) and soon. Each and every memory location has a unique memory address or location number.

1.3 Pointers

Pointers are one of the most feared things in the C language. In fact, they are the one thing that makes this language challenging at all. But why? Because they, quite honestly, can cause electric shocks to come up through the keyboard and physically *weld* your arms permantly in place, cursing you to a life at the keyboard. Well, not really. But they can cause huge headaches if you don't know what you're doing when you try to mess with them.

1.3.1 Memory and Variables

Computer memory holds data of all kinds, right? It'll hold `float`s, `int`s, or whatever you have. To make memory easy to cope with, each byte of memory is identified by an integer. These integers increase sequentially as you move up through memory. You can think of it as a bunch of numbered boxes, where each box holds a byte of data. The number that represents each box is called its *address*.

Now, not all data types use just a byte. For instance, a `long` is often four bytes, but it really depends on the system. You can use the **`sizeof()`** operator to determine how many bytes of

memory a certain type uses. (I know, **`sizeof()`** looks more like a function than an operator, but there we are.)

```
        printf("a long uses %d bytes of memory\n",
sizeof(long));
```

When you have a data type that uses more than a byte of memory, the bytes that make up the data are always adjacent to one another in memory. Sometimes they're in order, and sometimes they're not, but that's platform-dependent, and often taken care of for you without you needing to worry about pesky byte orderings.

So *anyway*, if we can get on with it and get a drum roll and some for boding music playing for the definition of a pointer, *a pointer is the address of some data in memory*

Pointer is the address of data. Just like an `int` can be `12`, a pointer can be the address of data. Often, we like to make a pointer to some data that we have stored in a variable, as opposed to any old random data out in memory wherever. Having a pointer to a variable is often more useful.

So if we have an `int`, say, and we want a pointer to it, what we want is some way to get the address of that `int`, right? After all, the pointer is just the *address of* the data. What operator do you suppose we'd use to find the *address of* the `int`?

Well, by a shocking suprise that must come as something of a shock to you, gentle reader, we use the **`address-of`** operator (which happens to be an ampersand: "**`&`**") to find the address of the data. Ampersand.

So for a quick example, we'll introduce a new *format specifier* for **`printf()`** so you can print a pointer. You know already how `%d` prints a decimal integer, yes? Well, `%p` prints a pointer. Now, this pointer is going to look like a garbage number (and it might be printed in hexidecimal instead of decimal), but it is merely the number of the box the data is stored in. (Or the number of the box that the first byte of data is stored in, if the data is multi-byte.) In virtually all circumstances, including this one, the actual value of the number printed is unimportant to

you, and I show it here only for demonstration of the **`address-of`** operator.

```
#include <stdio.h>
int main(void)
{
int i = 10;
printf("The value of i is %d, and its address is
%p\n", i, &i);
return 0;
}
```

Output: **`The value of i is 10, and its address is 0xbffff964`**

1.3.2 Pointer Types

You can now successfully take the address of a variable and print it on the screen. When we met last we were talking about how to make use of pointers. Well, what we're going to do is store a pointer off in a variable so that we can use it later. You can identify the *pointer type* because there's an asterisk (`*`) before the variable name and after its type:

```
int main(void)
{
int i; /* i's type is "int" */
int *p; /* p's type is "pointer to an int", or "int-
pointer" */
return 0;
}
```

so we have here a variable that is a pointer itself, and it can point to other `int`s. We know it points to `int`s, since it's of type `int*` (read "int-pointer").

When you do an assignment into a pointer variable, the type of the right hand side of the assignment has to be the same type as the pointer variable. Fortunately for us, when you take the

`address-of` a variable, the resultant type is a pointer to that variable type, so assignments like

the following are perfect:

```
int i;
int *p; /* p is a pointer, but is uninitialized and
points to garbage */
p = &i; /* p now "points to" i */
```

I know is still doesn't quite make much sense since you haven't seen an actual use for the pointer variable, but we're taking small steps here so that no one gets lost. So now, let's introduce you to the anti-address-of, operator.

1.3.3 Dereferencing

A pointer, also known as an address, is sometimes also called a *reference*. How in the name of all that is holy can there be so many terms for exactly the same thing? I don't know the answer to that one, but these things are all equivalent, and can be used interchangeably.

The only reason I'm telling you this is so that the name of this operator will make any sense to you whatsoever. When you have a pointer to a variable (AKA "a reference to a variable"), you can use the original variable through the pointer by *dereferencing* the pointer. (You can think of this as "de-pointering" the pointer, but no one ever says "de-pointering".)

What do I mean by "get access to the original variable"? Well, if you have a variable called *`i`*, and you have a pointer to *`i`* called *`p`*, you can use the dereferenced pointer `p` *exactly as if it were the original variable* `i`.

You almost have enough knowledge to handle an example. The last tidbit you need to know is actually this: what is the dereference operator? It is the asterisk, again: **`*`**. Now, don't get this confused with the asterisk you used in the pointer declaration, earlier. They are the same character, but they have different meanings in different contexts.

Example:

```
#include <stdio.h>
int main(void)
{
int i;
int *p; // this is NOT a dereference--this is a type
"int*"
p = &i; // p now points to i
i = 10; // i is now 10
*p = 20; // i (yes i!) is now 20!!
printf("i is %d\n", i); // prints "20"
printf("i is %d\n", *p); // "20"! dereference-p is
the same as i!
return 0;
}
```

Remember that `p` holds the address of `i`, as you can see where we did the assignment to `p`. What the dereference operator does is tells the computer to *use the variable the pointer points to* instead of using the pointer itself. In this way, we have turned `*p` into an alias of sorts for `i`.

1.3.4 Passing Pointers as Parameters

Right about now, you're thinking that you have an awful lot of knowledge about pointers, but absolutely zero application, right? I mean, what use is `*p` if you could just simply say `i` instead?

Well, the real power of pointers comes into play when you start passing them to functions. Why is this a big deal? You might recall from before that you could pass all kinds of parameters to functions and they'd be dutifully copied onto the stack, and then you could manipulate local copies of those variables from within the function, and then you could return a single value.

What if you wanted to bring back more than one single piece of data from the function? What if I answered that question with another question, like this:

What happens when you pass a pointer as a parameter to a function? Does a copy of the pointer get put on the stack? *You bet your sweet peas it does.* Remember how earlier I rambled on and on about how *EVERY SINGLE PARAMETER* gets copied onto the stack and the function uses a copy of the parameter? Well, the same is true here. The function will get a copy of the pointer.

But, and this is the clever part: we will have set up the pointer in advance to point at a variable...and then the function can dereference its copy of the pointer to get back to the original

variable! The function can't see the variable itself, but it can certainly dereference a pointer to that variable! Example!

```
#include <stdio.h>
void increment(int *p) /* note that it accepts a pointer
to an int */
{
*p = *p + 1; /* add one to p */
}
int main(void)
{
int i = 10;
printf("i is %d\n", i); /* prints "10" */
increment(&i); /* note the address-of; turns it into a
pointer */
printf("i is %d\n", i); /* prints "11"! */
return 0;
}
```

There are a couple things to see here...not the least of which is that the **`increment()`** function takes an `int*` as a parameter. We pass it an `int*` in the call by changing the `int` variable *`i`* to an `int*` using the **`address-of`** operator. (Remember, a pointer is an address, so we make pointers out of variables by running them through the **`address-of`** operator.) The **`increment()`** function gets a copy of the pointer on the stack. Both the original pointer *`&i`* (in **`main()`**) and the copy of the pointer *`p`* (in **`increment()`**) point to the same address. So dereferencing either will allow you to modify the original variable *`i`*! The function can modify a variable in another scope! Rock on! Pointer enthusiasts will recall from early on in the guide, we used a function to read from the keyboard, **`scanf()`**...and, although you might not have recognized it at the time, we used the **`address-of`** to pass a pointer to a value to **`scanf()`**. We had to pass a pointer, see, because **`scanf()`** reads from the keyboard and stores the result in a variable. The only way it can see that variable that is local to that calling function is if we pass a pointer to that variable:

```
int i = 0;
scanf("%d", &i); /* pretend you typed "12" */
printf("i is %d\n", i); /* prints "i is 12" */
```

See, **`scanf()`** dereferences the pointer we pass it in order to modify the variable it points to. And now you know why you have to put that pesky ampersand in there!

1.4 Pointer Arithmetic

The following arithmetic operations can be performed with pointer variables in C and C++:

Subtraction –

Incrementation ++

Decrementation – –

Pointer arithmetic follows data type size i.e., it causes the pointer to be incremented or decremented by the number of bytes occupied by a particular data type. For example, an integer pointer variable when incremented by 1, will increase by 2, as the size of an integer variable in windows environment is 2 bytes. This could be explained with the help of the following example.

Program: Program showing pointer arithmetic.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int *p, x=10;
clrscr();
p=&x;
printf("p= %d\n", p);
p=p+1;
printf("p= %d\n",p);
p=p+1;
printf("p=%d ", p);
getch();
}
```

Output: p= –12

p= –10

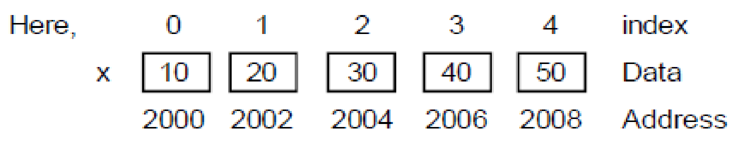
p= –8

Here, the value of p increases by 2 rather than by 1. It is because of the data type integer. One can change the data type from integer to float. In such situation p will increase by 4.

1.5 Pointers and One Dimensional Arrays

As discussed earlier, an array is a variable to represent multiple memory locations with the same name. Each and every element of an array has their unique memory addresses. These memory addresses can store into pointer variables. Therefore, if x is a one dimensional array, then the address of the first array element can be expressed as either &x [0] or simply as x. Moreover, the address of second array element can be written as either &x [1] or (x+1), and so on. This can be explained with one example.

Let int x[5] = {10,20,30,40,50};



Here, we assume that address of the first array element i.e., x [0] is 2000, so the address of second array element will be 2002, since one integer variable occupies 2 bytes in memory.

Program: Display the content of an array using pointer

```
#include<stdio.h>
#include<conio.h>
void main()
{
int *p, x[5]={10,20,30,40,50};
clrscr();
p=&x[0];
printf("First element=%d\t",*p);
p=p+1;
printf("Second element= %d\t",*p);
p=p+1;
printf("Third element= %d\t",*p);
p=p+1;
printf("Fourth element= %d\t",*p);
p=p+1;
printf("Fifth element= %d\t",*p);
getch();
}
```

The above program can also be written using loop construct, which reduces the number of statements.

Program: Program to display the array content using pointer and loop

```
#include<stdio.h>
#include<conio.h>
void main()
{
int *p,i, x[5]={10,20,30,40,50};
clrscr();
p=&x[0];
printf("Elements are : ");
for(i=0;i<=4;i=i+1)
{
printf("%d",*p);
p=p+1;
}
getch();
}
```

Here, p is a pointer variable of type integer and initially assigned the address of the first array element x [0]. In subsequent iteration or repetition the value of p is increased by 1. i.e., by 2 bytes, since p is an integer variable and displays the value of that address.

Check Your Progress

1. A pointer, also known as an address, is sometimes also called a…………
2. You can identify the *pointer type* because there's an……………… before the variable name.
3. You can use the………… operator to determine how many bytes of memory a certain type uses.
4. ……………which tells the compiler that the definition of the variable is in a different file.
5. A………… is a programming language construct, a variable type that is declared outside any function and is accessible to all functions throughout the program.
6. ……………is a named block of code that performs a task and then returns control to a caller.

1.6 Pointers and Character Arrays

A character array forms a string. In other words, a string may be considered as a string of characters. So, string can be processed using pointer variables. Strings are always terminated with null character i.e. '\0'.

Program: Program to display characters of a string using pointer.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char name[ ]=”KKHO UNIVERSITY”;
char *p;
clrscr();
p=&name[0];
while(*p!=’\0')
{
printf(“%d”,*p);
p++;
}
getch();
}
```

Here, the base address of the character array “name” is stored into the pointer variable p. In each iteration, value at address of p is displayed, then p is incremented by 1 until ’\0’ is found. One can increment the pointer by using “++” operator and can decrement the pointer by using “--” operator.

Program: Program to display characters of a string in reverse order using pointer.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char name[]=”KKHSO UNIVERSITY”;
char *p,*q;
clrscr();
p=&name[0];
q=&name[0];
while(*p!=’\0')
{
p++;
}
p—;
while(p!=q)
{
Printf("%d\t",*p);
p-;
}
getch();
}
```

1.7 **Passing Pointers to Functions as Arguments**

Pointers can pass to a function as arguments. Arguments can be passed to a function in two ways i.e. **by value** and **by reference**. When an argument is passed by value, the data item is copied to the function i.e. it makes a duplicate copy of the original variable. In such situation, any alteration or modification made in the variables of the function does not affect the value of the original variable. Some situation may arise where we want to change the value of variable in the original program with the help of a function. When calling a function using pointers, it copies the address of a variable to the function, not the value of the variable. The contents of that address can be accessed directly either within the called function or calling function. In other words, pointer variables are the direct means of Communication among the function variables. Let us explain the use of a pointer variable with the help of the following example.

Program: Program to show passing address of a variable to a function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
void passdata(int *p);
int x=10;
clrscr();
passdata(&x);
getch();
}
void passdata(int *p)
{
printf("Address of x : %d\n",p);
printf("Value of x : \n", *p)";
}
```

Here, there are two functions viz. main() and passdata(). The main() function calls the passdata() function and passed the address of variable x to the function. In passdata(), p is a pointer variable to accept the address of variable x. Here, p is a formal parameter and x is actual parameter.

Program: Program to show call by value and call by reference.

```
#include<stdio.h>
#include<conio.h>
void main()
{
void passdata1(int *p);
void passdata2(int p);
int x=10;
clrscr();
printf("Value of x before passdata1 call %d\n",
x);
passdata2(x);
printf("Value of x after passdata1 call
%d\n",x|);
passdata1(&x);
printf("Value of x after passdata2 call %d\n",
x);
getch();
}
void passdata1(int *p)
{
```

```
*p=20;
}
void passdata2(int p)
{
p=20;
}
```

This program contains two functions, called passdata1() and passdata2(). The function passdata1() receives one pointer to integer variable as its argument and passdata2() receives one integer variable as argument. This variable originally assigned a value 10. The value is then changed to 20 within passdata2() function. The new value is not reflected within main, because the argument x was passed by value. As we have discussed earlier, when a function is called by value, then a duplicate copy of the original variable is copied to the called function. Any changes to the variables of called function are local to that function only. In passdata1(), the statement *p=20 indicates that the value 20 is assigned to the contents of the pointer address, which is address of x. Since the address is recognized in both functions i.e. passdata1() and main(), the reassigned value will be recognized within main after call to passdata1().

Note : *Function declaration, call and definition need not to be in order*.

Program: Program to exchange value of two variable using function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
void exchange(int *p, int *q);
int x=10, y=20;
clrscr();
printf("Value of x & y before function call");
printf("x=%d y=%d\n",x,y);
exchange(&x, &y); //passing address of x and
y
printf("Value of x & y after function call");
printf("x=%d y=%d\n",x,y);
getch();
}
void exchange(int *p, int *q)
{
int temp;
temp=*p;
*p=*q;
*q=temp;
}
```

1.8 Dynamic Memory Allocation

Memory allocation refers to the reservation of memory for storing data. Memory allocation is done in C language in two ways (i) Static allocation and (ii) Dynamic allocation. Static allocation is done by using array. The main disadvantage of static allocation is that the programmer must know the size of the array or data while writing the program. Generally, it is not possible to know the required memory in advance. To overcome this problem dynamic memory allocation is done. Dynamic memory allocation refers to
the allocation of memory during program execution. The basic difference between static and dynamic memory allocation is that in static allocation memory is allocated during the compile time, whereas in dynamic allocation memory is allocated during the program execution time.

1.8.1 Library Function for Dynamic Memory Allocation

C language provides a set of library functions for dynamic memory allocation and de-allocation. There are basically four functions used for this purpose. They are: ***malloc( )***, ***calloc( )***, ***realloc( )*** and ***free( )***

- **malloc():** This function is used to allocate a block of memory.

  After allocating the memory it returns a pointer of type *void*. This means that one can assign it to any type of pointer. The syntax for using *malloc()* is as follows:

  ptr =(cast-type *)malloc(no. of bytes);

  Here, ptr is a pointer of type cast-type. For example,

  int x;

  x=(int *)malloc(100);

  If it is unable to find the requested amount of memory, malloc() function returns NULL.

- **calloc():** It is a library function to allocate memory. The difference between *calloc()* and *malloc()* is that *calloc()* initialize the allocated memory to zero where as *malloc()* does not initialize allocated memory to zero.

  Declaration Syntax: Following is the declaration for *calloc()* function.

  void (cast-type*)calloc(no. of elements to be allocated, size of each element)

  For example:

  ptr = (int*) calloc(10, sizeof(int));

  This statement allocates contiguous space in memory for an array of 10 elements each of size of

  int, i.e., 2 bytes.

Program: Program to show the usage of calloc() function.

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

```
#include <stdlib.h>
int main()
{
int i,no;
int *p;
printf("Enter number of elements :");
scanf("%d",&no);
p = (int*)calloc(no, sizeof(int));
printf("Enter %d numbers:\n",no);
for( i=0 ; i < no ; i++ )
{
scanf("%d",&p[i]);
}
printf("The numbers are: ");
for( i=0 ; i < n ; i++ )
{
printf("%d ",p[i]);
}
free( p );
return(0);
}
```

- **free( ):** A memory area that is dynamically allocated using either calloc() or malloc() doesn't get freed automatically when the execution terminates. You must explicitly use free() library function to release the memory space.

  Syntax: free(ptr);

  This statement frees the space allocated in the memory pointed by ptr.

- **realloc( ):** The size of dynamically allocated memory can be changed by using realloc() library function.

  Syntax: void *realloc(void *ptr, size_t size);

Example: int *ptr = (int *)malloc(sizeof(int)*2);//dynamic allocation for two integer value

int *ptr_new;

ptr_new = (int *)realloc(ptr, sizeof(int)*3);// dynamic reallocation

Check Your Progress

Q.7: Choose the appropriate option:

I. Which one of the following is valid pointer declaration:

- **a.** int a*;
- **b.** int *a,
- **c.** int *a
- **d.** int *a;

II. What will be the output of the following statements if the variable ‘a’ located at address 50 [Assume no syntax error and working platform is MS Windows]

```
int *p, a=50;
p=&a
a++, p++;
a++, p++;
printf(“%d%d”, a,p);
```

- **a.** 52, 54
- **b.** 54, 52
- **c.** 52, 52
- **d.** 54, 54

III. In a string ‘\0’ is known as:

- **a.** Back zero
- **b.** Slash zero
- **c.** Null character
- **d.** End character

IV. The meaning of the following statements is: void sum(int *p, int *q);

- a. Function declaration
- b. Call by address
- c. Function does not return value
- d. All of the above

V. malloc() is:

- a. used to allocate memory statically
- b. used to allocate memory dynamically

c. a user defined function
d. does not return value

1.10 Answer to check your progress

Ans. to Q. No.1: *reference*

Ans. to Q. No.2: asterisk (*)

Ans. to Q. No.3: *sizeof()*

Ans. to Q. No.4: *extern*

Ans. to Q. No.5: global variable

Ans. to Q. No.6: A function

Ans. to Q. No. 7: i. (d) int *a, ii. (a) 52, 54, iii. (c) Null character, iv. (d)None of the above, v. (b) used to allocate memory dynamically

1.11 Model Questions

1. What are the difference between 'pass by value' and 'pass by reference'?
2. What is dynamic memory allocation? What are the functions used to allocate memory dynamically.
3. Explain how pointers are passed to a function.
4. Write a program to input your name and display it using pointer
5. What is pointer arithmetic? What are the operators used in pointer arithmetic.
6. Explain the mechanism to access one dimensional array using pointer.
7. What is pointer? How pointer variables are declared?

Unit-6

Structure and Union

1.1 Learning Objectives

1.2 Introduction

1.3 Structure

- 1.3.1 Structure Declaration
- 1.3.2 Initialization of Structure
- 1.3.3 Accessing the Members of a Structure

1.4 Array of Structures

1.5 Structure within a Structure

1.6 Passing Structures to Functions

1.7 Pointer to Structure

1.8 Union

1.9 Enumerated Data Types

1.10 Defining Your Own Types (Typedef)

1.11 Answers To Check Your Progress

1.12 Model Questions

1.1 Learning Objectives

After going through this unit, you will be able to:

- write program using a structure rather than several arrays
- learn how structures are defined and how their individual members are accessed and processed within a program
- declare structure variables
- learn about array of structures
- declare and use pointer to structure
- learn about union
- describe enumerated data types
- learn about typedef

1.2 Introduction

We have already been acquainted with array which is a linear data structure. Array takes basic data types like *int*, *char*, *float* or *double* and organises them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of which are of the same type. If we need to use a collection of different data type items it is not possible by using an array. When we require using a collection of different data items of different data types we can use a *structure*.

In this unit we will learn about structure and union. Here we will see how a structure and union are defined, declared and accessed in C programming language.

1.3 Structure

A structure is similar to records. It stores related information about an entity. With the use of structures, programmers can conveniently handle a group of related data items of different data types.

In C language, structure is basically a user defined heterogeneous data type. The main difference between a structure and an array is that an array contains related information of the same data type.

1.3.1 Structure Declaration

A structure is declared using the keyword ***struct*** followed by a structure name. All the variables of the structure are declared within the structure. The data types of all these variables within a structure can be of different types. A structure is generally declared with the following syntax:

```
struct struct_name
{
data_type variable_name; data_type variable_name;

.........................................
.........................................
        };
```

For example, to keep the details of a book, we have to declare a structure for the book containing variables like title, author, pages, price, publisher etc. This book structure can be declared as:

```
struct book
{
char title[20]; char author[15];
char publisher[25]; int pages;
float price;
};
```

In the above declaration, the book name (i.e., title), author name and publisher name would have to be stored as **string**, and the page and price could be **int** and **float** respectively. The keyword *struct* declares a structure to hold the details of five fields namely title, author, publisher, pages and price. These are members of the structures. Each member may belong to different or same data type. It is not always necessary to define the structure within the *main()* function.

Structure declaration acts as a template which conveys structure information and member names to the compiler. Structure is a user-defined data type. Now, let us discuss how to declare structure variables.

We can declare structure variables using the structure name (tag name) any where in the program. For example, the statement,

struct book book1, book2, book3;

declares *book1, book2, book3* as variables of type *struct book*. Each declaration has five elements of the structure *book*. The complete structure declaration might look like this:

```
struct book
{
char title[20], author[15], publisher[25]; int pages;

float price;
};
struct book

{

char title[20], author[15], publisher[25]; int pages;

float price;
} book1,book2,book3;
```

The use of tag name is optional. In the declaration, ***book1***, ***book2***, ***book3*** are structure variables representing three books. Tag_name is not included in this declation. A structure is usually defined before ***main()***. In such cases, the structure assumes global status and all the functions can access the structure.

Again, let us consider another structure declaration "employee".

```
struct employee
```

```
{
char fname[15]; char lname[15]; int id_no;
int month; int day; int year;
} emp1;
```

Here we have declared one variable, ***emp1***, to be structure with six fields, some integers, some strings. Right after the declaration, a portion of the main memory is reserved for the variable ***emp1***. This variable takes a size of 38 bytes for different members of struct ***employee***: 15 bytes for fname, 15 bytes for lname, 2 bytes for id_no, 2 bytes for month, 2 bytes for day, 2 bytes for year.

1.3.2 Initialization of Structures

Initialization of structure means assigning some constants to the members of the structure. A structure can be initialized in the same way as other data types are initialized. The general syntax to initialize a structure variable is as follows:

```
struct struct_name
{
data_type member_name1; data_type member_name1; data_type
member_name1;

..........................................
..........................................
}............................................... struct_var = {constant1, constant2,     };
```

or,

```
struct struct_name
{
data_type member_name1; data_type member_name1; data_type
member_name1;

..........................................
```

..

}; struct struct_name struct_var = {constant1, constant2, };

For example, let us initialize an employee structure as follows:

```
struct employee
{
int empid;
char name[20]; char address[30]; float salary;
}
emp1 = {01,"Ranjan","Guwahati", 42000.00}; or by writing
struct employee emp1 = {01,"Ranjan","Guwahati", 42000.00};
```

C language automatically initializes the structure members if the user does not explicitly initialize all the members. This is known as *partial initialization*. ***Integer*** and ***float*** members are initialized to ***zero*** and ***character arrays*** are initialized to '**\0**' (null value) by default. Pictorially we can represent it as follows:

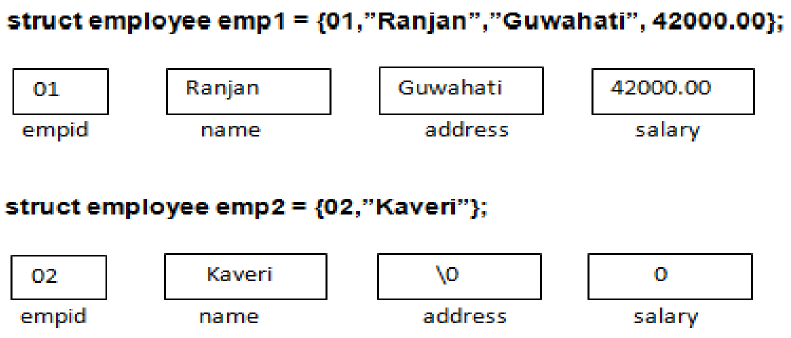

struct employee emp1 = {01,"Ranjan","Guwahati", 42000.00};
01
empid
Ranjan
name
Guwahati
address
42000.00
salary
struct employee emp2 = {02,"Kaveri"};
02
empid
Kaveri
name
\0
address
0
salary

Fig. Assigning values to a structure element

1.3.3 Accessing the Members of a Structure

The members of structure themselves are not variables. They should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' which is known as ***dot operator*** or ***period operator***. A structure member variable is generally accessed using the '.' dot operator. The syntax is:

struct_var . member_name;

For example, book1**.**price;

book1**.**pages; book2**.**price;

book1.price is the variable representing the price of *book1* and can be treated like any other ordinary variable.

To assign value to the individual data members of the structure variable *book1*, we may write,

book1.price = 520.00; book1.name = “Java”; book1.author=”Kumar”;

We can use *scanf()* function to input values for data members of the structure variable *book1* like this :

scanf(“%f”,&book1.price); scanf(“%d”,&book1.pages);

For displaying the values of structure variable book1, we can use *printf()* function like this:

printf(“%f”, & book1.price); printf(“%s”, &book1.author);

Example: Program to enter information of one student and to display that information.*/

```
#include<stdio.h> #include<conio.h> void main()
{
struct studentinfo
{
int roll;
char name[20]; char address[30]; int age;
}       s1;
clrscr();
printf("Enter the student information:"); printf("\nEnter the student roll
no.:"); scanf("%d",&s1.roll);
printf("\nEnter the name of the student:"); scanf("%s",&s1.name);
printf("\nEnter the address of the student:"); scanf("%s",&s1.address);
printf("\nEnter the age of the student:"); scanf("%d",&s1.age);
printf("\n\nStudent information:"); printf("\nRoll no.:%d",s1.roll);

            printf("\nName:%s",s1.name);
printf("\nAddress:%s",s1.address);

            printf("\nAge of student:%d",s1.age); getch();
            }
```

1.4 Array of Structures

In programming we may often need to handle many records. For example, in a class, there may be many numbers of students, say 50 students. So, to keep the records of 50 students we need an array of structures. This can be written as

```
struct student
{
int rollno;
char name[20]; char course[15]; float fees;

};
struct student s[50];                 //s is an array of structure
```

Here, s is an array of structure of 50 students, each of which is of type *struct student*.

Example: **Program for storing records of 50 students and displaying those records*/**

```
                #include<stdio.h>
                #include<conio.h>
                void main()
                 {
                 struct student
                     {
                       int roll;
```

```
char name[20]; char address[30]; int age;
        };
struct student s[50]; clrscr();
int n, i;
printf("\nHow many students information do you want to enter?");
scanf("%d",&n);
printf("Enter Student Information:"); for(i=1;i<=n;i++)
{
printf("\nEnter Roll no.:"); scanf("%d",&s[i].rollno);
printf("\nEnter the name of the student:"); scanf("%s",&s[i].name);
printf("\nEnter the course of the student:"); scanf("%s",&s[i].course);
printf("\nEnter the dues of the student:"); scanf("%f",&s[i].fees);
}
printf("\n\nInformation of all students:"); for(i=1;i<=n;i++)
{
printf("\nRoll no.:%d",s[i].rollno);
printf("\nName:%s",s[i].name);          printf("\nCourse:%s",s[i].course);
printf("\nSchool dues:%f\n\n",s[i].fees);
}
getch();
}
```

Check Your Progress

Q.1: Define a structure consisting of two floating point members, called *real* and *imaginary*.
Include the tag *complex* within the definition. Declare the variables c1,c2 and c3 to be structure of type *complex*.

Q.2: Declare a variable ***“a”*** to be a structure variable of the following structure type:

struct account

{

```
int ac_no; char ac_type;
char name[30]; float balance;
};
```

initiaze ***a*** as follows: ac_no : 12437 ac_type: Saving name: Rahul Anand balance: 35000.00

1.5 Structure within a Structure

A structure may be defined as a member of another structure. In such structures, the declaration of the embedded structure must appear before the declarations of other structures. For example,

```
struct date
{
int day; int month; int year;
};
struct student
{
int roll;
char name[20];
char combination[3]; int age;
struct date dob;        //structure within structure
}       student1,student2;
```

the structure ***student*** contains another structure ***date*** as one of its members.

1.6 Passing Structures to Functions

Structure variables may be passed as arguments and returned from functions just like other variables. A structure may be passed into a function as individual member or a separate variable.

Passing individual members of structure to a function: To pass any individual member of the structure to a function as argument, we have to use the dot operator to refer to the particular member of the structure.

For example, a program to display the contents of a structure passing the individual elements to a function is shown below:

***Example:* Program to illustrate passing individual structure .elements to a function.**

```
#include<stdio.h>
 #include<conio.h>
void display(int, float);
void main()
{
struct employee
{
int emp_id; char name[25];
char department[15]; float salary;
};
static struct employee e1={15, "Rahul","IT",8000.00}; clrscr();
/* only emp_id and salary are passed to the display  function*/
display(e1.emp_id,e1.salary);                    //function     call
getch();

}
void display(int eid, float s)
{
printf("\n%d\t%5.2f",eid,s);
}
```

Output: 15 8000.00

When we call the display function using *display(e1.emp_id,e1.salary);* we are sending the ***emp_id*** and ***name*** to function ***display( )***. It can be immediately realized that, passing individual elements would become more tedious as the number of structure elements increases. A better way would be to pass the entire structure variable at a time.

Passing entire structure to a function: There may be numerous structure members (elements) in a structure. Passing these individual elements as argument to a function would be a tedious task. Just like any other variable, we can pass an entire structure as a function argument. A structure is passed as an argument using the call by value method. This means a copy of each member of the structure is made. This method is very inefficient, especially when the structure is very big or the function is called frequently. Use of pointers in such sitiation is more suitable.

In the following program, we are passing a whole structure to a function.

Example: Program to illustrate passing a whole structure to a function.

```
#include<stdio.h> #include<conio.h> struct employee
{
int emp_id; char name[25];
char department[10]; float salary;
};
static struct employee  e1={10,"Palash","Sales",26000.00}; void display(struct
employee e);                    //prototype decleration void main()

{
clrscr();
display(e1);    /*sending entire employee structure*/ getch();
}
void display(struct employee e)
{
printf("%d\t%s\t%s\t%5.2f", e.emp_id,e.name,e.department,e.salary);
}
```

Output:

```
10          Palash Sales     26000.00
```

Example: Program to illustrate structure working within a function

```
#include<stdio.h> #include<conio.h> struct item

{
int code; float price;
};
struct item a;
void display(struct item i); //prototype decleration void main()
{
clrscr();
display(a); /*sending entire item structure*/ getch();
}
void display(struct item i)
{
i.code=20; i.price=299.99;
printf("Item Code and Price of the item:%d\t%5.2f", i.code,i.price);
}
```

Output : 20 299.99

1.7 Pointer to Structure

Instead of passing a copy of the whole structure to the function, we can pass only the address of the structure in the memory to the function. Then, the program will get access to every member in the function. This can be achieved by creating a pointer to the address of a structure using the indirection operator “*”.

To write a program that can create and use pointer to structures, first, let us define a structure:

```
struct item
{
int code; float price;
};
```

Now let us declare a pointer to struct type *item*.

```
struct item *ptr;
```

Because a pointer needs a memory address to point to, we must declare an instance of type *item*.

```
struct item p;
```

The following program shows the relationship between a structure and a pointer.

Example: Program to demonstrate pointers to structure.

```
#include<stdio.h>

#include<conio.h> void main()
{
struct item
{
int code; float price;
};
struct item i; clrscr();
struct item *ptr;       //declare pointer to ptr structure ptr=&i;       //
assign address of struct to ptr ptr->code=20;
ptr->price=345.00;
printf("\nItem Code: %d",ptr->code); printf("\tPrice: %5.2f",ptr->price);
getch();
}
```

Output:

```
Item Code: 20       Price: 345.00
```

1.8 Union

In some situations we may wish to store information about a person. The person may be identified either by name or by an identification number, but never both at the same time. We could define a structure which has both an integer field and a string field; however, it seems wasteful to allocate memory for both fields. This is particularly important if we are maintaining a very large list of persons, such as payroll information for a large company. In addition, we wish to use the same member name to access the information for a person.

C provides a data structure which fits our needs for the above scenario, called a ***union*** data type. A union type variable can store objects of different types at different times; however, at any given moment, it stores an object of only one of the specified types. Unions are also similar to structure data type except that members are overlaid one on top of another, so members of union data type share the same memory.

The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure data type. For example, we can declare a union variable, person, with two members, a string and an integer. Here is the union declaration:

```
union human

{

int id;

char name[30];

} person;
```

This declaration differs from a structure in that, when memory is allocated for the variable ***person***, only enough memory is allocated to accommodate the largest of the specified types. The memory allocated for person will be large enough to store the larger of an integer or an 30 character array. Like structures, we can define a tag for the union, so the union template may be later referenced by the tag name.

Unions obey the same syntactic rules as structures. We can access elements with either the dot operator (.) or the right arrow operator (->). There are two basic applications for union. They are:

i) Interpreting the same memory in different ways.

ii) Creating flexible data structure that can hold different types of data.

Example: Program demonstrating initializing union members and displaying the contents.

```
#include<stdio.h> #include<conio.h> void main()
{
union data
{
int a; float b;
};
union data d;
        d.a=20;
             d.b= 195.25;
            printf("\nFirst member is %d",d.a); printf("\nSecond member
is
           %5.2f",d.b); getch();
        }
```

Output: First member is 16384

Second member is 195.25

Here only the float values are stored and displayed correctly and the integer values are displayed wrongly as the union only holds one value for one data type.

1.9 Enumerated Data Types

In addition to the predefined types such as int, char, float etc., C allows us to define our own special data types, called enumerated data types.

An enumeration type is an integral type that is defined by the user.

The syntax is:

enum typename {enumeration_list};

Here, ***enum*** is keyword, ***type*** stands for the identifier that names the type being defined and ***enumeration list*** stands for a list of identifiers that define integer constants. For example:

```
enum color {yellow, green, red, blue, pink};
```

defines the type *color* which can then be used to declare variables like this:

```
color flower=pink;

                    color car[ ]={green, blue, red};
```

Here, *flower* is a variable whose value can be any one of the 5 values of the type *color* and is initialialized to have the value pink.

Example: Program to illustrate the concept of enumerated data type

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

    void main()
    {
    enum month { jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
    month m; clrscr();
for(m=jan;m<=dec;m++) printf("%d\t", m+1);
    getch();
    }
```

Output : 1 2 3 4 5 6 7 8 9 10 11 12

In the above declaration, *month* is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable *m* is declared to be of the same type as month, m cannot be assigned any values outside those specified in the initialization list for the declaration of month.

1.10 Defining Your Own Types (typedef)

Using the keyword ***typedef*** we can rename basic or derived data types giving them names that may suit our program. A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. We can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that we have declared. Typedef names allow us to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

For example: **typedef unsigned long int ulong;**

The new type (***ulong***) becomes known to the compiler and is treated the same as ***unsigned long int****.* If we want to declare some more variables of type unsigned long int, we can use the newly defined type as:

ulong distance;

We can use the typedef keyword to define a structure as folllows:

```
typedef struct
{
type member1; type member2;

....
}type_name;
```

type_name can be used to declare structure variables as follows:

type_name variable1,variable2,...;

Check Your Progress

Q.3: State whether the following statements are True (T) or False (F)

Collection of different data types can be used to form a structure.

Structure variables can be declared using the tag name anywhere in the program.

Tag-name is mandatory while defining a structure.

A program may not contain more than one structure.

We cannot assign values to the members of a structure.

It is always necessary to define the structure variable within the main() function.

Q.4: State whether the following statements are True (T) or False (F)

i) It is possible to pass a structure to a function in the same way a variable is passed.

ii) When one of the fields of a structure is itself a structure, it is called nested structure.

iii) We cannot create structures within structure in C.

iv) It is illegal for a structure to contain itself as a member.

v) A sstructure can include one or more pointers as members.

Q.5: Fill in the blanks:

i) __________ can be used to access the members of structure variables.

ii) The name of a structure is referred to as __________________ _.

Q.6: Mentions the features of union data type.

Q.7: Mention the advantages of structure type over the union type.

1.11 Answer to Check Your Progress

Ans. to Q. No. 1: struct complex

```
{
float real, imaginary;
};
struct complex c1,c2,c3;
```

Ans. to Q. No. 2: Static struct account a={12437, “Saving”, “Rahul Anand”, 35000.00};

(*a* is a static structure variable of type account, whose members are assigned initial values.)

Ans. to Q. No. 3: i) True, ii) True, iii) False, iv) False, v) False, vi) False

Ans. to Q. No. 4: i) True, ii) True, iii) False, iv) True, v) True

Ans. to Q. No. 5: i) Pointers, ii) tag name

Ans. to Q. No. 6: The main characteristics of Union data type are:

a. The size of union is equal to the size of number of bytes occupied by the largest data member in it.
b. only one data member in union is active at a time.

Ans. to Q. No. 7: The structure data type can hold many data related to an entity like person, book, student etc., but a union type holds only one data active at a time.

1.12 Model Questions

1. What is a structure? How is a structure different from an array?
2. How is structure declared? Define a structure to represent a date.
3. What is meant by array of structure?
4. How are the data elements of a structure accessed and processed?
5. What ismeant by union? Differentiate between structure and union.
6. What is the purpose of typedef feature? How is this feature used with structure?

Unit-7
Arrays

1.1 Learning Objectives

1.2 Introduction

1.3 Array

- 1.3.1 Terminologies Associated with Array

1.4 Array Declaration and Initialization

1.5 One-Dimensional Array (1-D Array)

- 1.5.1 Entering Data Values in 1-D Array
- 1.5.2 Accessing Values from 1-D Array

1.6 Two-Dimensional Array (2-D Array)

- 1.6.1 Storage Representation of 2-D Arrays
- 1.6.2 Entering Data Values in 2-D Array
- 1.6.3 Accessing Values From 2-D Array

1.7 Check Your Progress

1.8 Model Questions

1.1 Learning Objectives

After going through this unit, you will be able to:

- define an array
- declare and initialize array
- create and access one dimensional array
- create and access two dimensional array.

1.2 Introduction

In the previous unit, we have learnt about the different storage class. We have also learnt about conditional statements and loop control structures in the earlier units.

You must have come across the term ***array*** many times and wondered what it is and where it can be applied.

In this unit we will learn to define an array. We will also learn to declare an array and to initialize an array. In addition to these, different types of arrays like one dimensional and two dimensional arrays will also be covered in this unit.

1.3 Array

Array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations. In this definition by ‘finite’, we mean that the array contains a fixed number of elements. By ‘ordered’ we mean that all the elements are stored in contiguous locations of the computer memory in a linear way.

By ‘homogeneous’ we mean that all the elements in the array must belong to the same data type.

1.3.1 Terminology

Let us look at some of the terminologies associated with array.

- **Size:** The number of elements in the array.
- **Type:** The type indicates the data type of the array. It can be integer, floating point or character.
- **Base:** The base of the array is the address of the first element of the array.
- **Index:** Elements in an array are referred using a subscript or index value. The index is an integer value which gives the position of the element in an array. It is denoted by $\mathbf{A_i}$ or **A [i]**
  where **A** is the name of the array and **“i”** is the subscript or
  index. Since array elements are identified by using index or subscripts, the array is also called an indexed or subscripted variable.

1.4 Array Declaration and Initialization

Declaration of Array: An array can be declared just like we declare any other variable. We give the data type of the variable followed with the name of the variable. In case of an array also we give the data type followed with the name of the array but we also include an additional component that is the size of the array. The number of elements that the array can contain needs to be declared at the beginning of the array. This is because according to the definition of an array, it contains a fixed number of elements. We can declare an array in the following way:

data_type name_of_array [size_of_array];

In the above declaration syntax, data type can be any of the valid data types for variables. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the size of the array that is given inside square brackets. For example:

int array [5] ; char arr [10] ; float balance [3] ;

Need for arrays: To stress the need for arrays let us look at the following example.

A cricket match has been organized between two teams A and B. Suppose we do not have the knowledge of arrays and we need to keep the batting scores of all the players for the team batting first. If we want to keep the batting score for one player we can declare a variable like the one below:

```
int bat_score1;
```

where we store the score of player 1. To store the score of rest of the 10 players we will need to declare 10 more variables of data type integer as given below:

```
int bat_score2, bat_score3,                    …. bat_score11;
```

Now if, instead of keeping the scores of just 11 players for one match suppose we are required to keep score of 3 teams. How many variables do we create using the above approach? Do we create 33 integer variables? An easier way to keep the scores of the team would be to use an array. Instead of declaring 11 variables for keeping the score of 11 players in a team wouldn't it be easier to store all the batting values of one team for one match in one array variable.

```
int     team_a [ 11];
int     team_b [ 11];
int     team_c [ 11];
```

Definition of array: When we declare an array, a contiguous memory location is allocated to that array. Let us look at an example of the physical representation of array of size 10 in computer's memory.

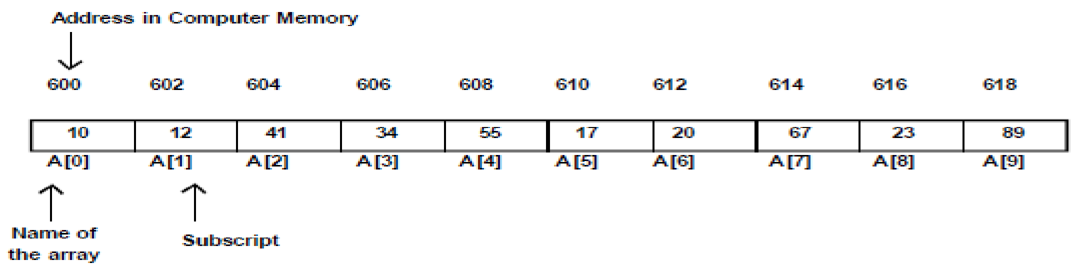


Figure: Physical representation of array in memory

In the above figure we can see how array is actually stored in the memory. Let us consider an integer array of ten elements. Here, the name of the array is A. The smallest index of an array is called a lower bound and the highest index is called an upper bound. In case of C, the lower bound of an array is 0 and the number of elements can be calculated as the difference between upper and lower bound plus 1.

No of elements = Upper – Lower + 1

The number of elements in the above case will be (9-0+1=10) ten elements. The base address of the array in this case is 600, since that is the address of the first element in the array. If we assume that the compiler to store an integer value needs two bytes of storage then the address of the second element is 602. Similarly, the rest of the elements are also stored continuously taking two bytes of storage per integer number.

Array initialization: We can initialize the elements of the array when we declare the array. Initialization is generally done when we already know the values of the elements of an array. Just as in declaration, in initialization also we give the data type of the array, followed by the name and size of the array. But in addition to these we also provide the values of the data elements of the array within braces **{ }** separated by commas. Let us look at how initialization is done using the following example:

```
int A [5] = { 1, 2, 3, 4, 5 };
```

In the above example, the array A is declared and initialized. The array A has a size of 5 elements and the values of the five elements have been initialized as follows:

A [0] = 1

A [1] = 2

A [2] = 3

A [3] = 4

A [4] = 5

The first value that is given in braces is put into the first array location with subscript value 0. So 1 is kept at location A[0]. Similarly, the second value inside braces is put into the second array location with subscript value A[1] and so on till all the given values has been put in the array. When initialization of values is done, a possibility of not mentioning the array size in the square brackets [ ] beforehand is also provided. That is, we can leave the square brackets empty. For this case the compiler assumes the array size equal to the number of values provided inside braces { }. For example, we can write the following statement:

int A [ ] = { 1, 2, 3, 4, 5, 6 };

In the above statement, the compiler will assume that the size of the array is 6 since six data values have been given inside the braces.

Check Your Progress

Q.1: What is an array?

Q.2: Give the syntax for declaring an array.

Q.3: Arrays cannot store elements of __________ data types.

Q.4: Can we declare an array without mentioning the size of an array?

1.5 One Dimensional Array (1-D Array)

One dimensional array is a collection of homogeneous data elements with only one row. It is the simplest form of array. The declaration and definition that we have learnt so far has been for a single dimensional array.

Address calculation: The elements of an array are stored in contiguous memory locations. This means that if we know the base address then we can calculate the address of the other array elements.

Let ‘b’ be the memory location of the first element of the array and each element requires ‘w’ words of memory space. Then, the address or location of element A[i] will be the summation of the base address and the product value of ‘i’ and ‘w’.

Address of element A[i] = b + i X w

For example if we consider the figure 9.1, then the base address, b = 600. Now if we consider an integer array that requires word size 2, the address of the 10th element should be 618. If we calculate the address according to the formula given above we get the same answer.

Address of element A[9] = 600 + (9 X 2) = 600 + 18 = 618

1.5.1 Entering Data Values in 1-D Array

We can insert values into an array at the time of initialization. But apart from that, we can also insert values into array elements by accessing them individually. For example, if we have an integer array A of 5 elements, we can insert the values for each of the five array positions as follows:

```
A [0] = 11;
A [1] = 22;
A [2] = 33;
A [3] = 44;
A [4] = 55;
```

In the above statements, the value for each array position has been filled individually instead of giving the values at the time of initialization and declaration.

Let us now suppose that we have an array of 50 elements. Also we can write statements like above only when we have the knowledge of the element values in advance. Otherwise when we have to take the values from the user at run time these types of statements do not provide a suitable solution.

An easier solution to the above problem is to use loop control structure statements like ***while***, ***do_while*** and ***for*** loop. Let us look at a code where with the help of for loop we can easily enter any number of values at the run time. This can be implemented as:

```
for ( i = 0; i < 5; i++ )
{
scanf("%d", &A[ i ]);
}
```

In the code above a 'for loop' is used to traverse from the first element of the array to the last element. The above 'scanf' statement accepts an integer value and stores it in the address of the given array location. The array location is moved from the first element position to the last element position using for loop. "&A[i]" gives the address of the 'ith' element of the array where the current data value needs to be stored.

1.5.2 Accessing Values from 1-D Array

We have learnt in the previous part about entering values in a one dimensional array. Once the values have been inserted we may need to access the array for various purposes. The procedure for accessing the values in a one dimensional array is similar to entering the values in one. This can be implemented as:

```
for ( i = 0; i < size; i++ )
{
printf(“%d”, A[ i ]);
}
```

In the above code, a ‘for loop’ is used to traverse from the first element of the array to its last element. The ‘printf’ statement prints the integer value which is stored in the address of the given array location. The array location is moved from the first element position to the last element position using for loop. “**A[i]**” gives the value of the element at the **ith position** of the array.

Example: **Program to enter and access data elements in a 1-D array**

```
# include<stdio.h> # include<conio.h> int main()
{
// declaring integer array of size 10 int array[10];
int i;
// Entering data values in array printf(“Enter any 10 integer values:\n”);
for(i=0;i<10;i++)
{
scanf(“%d”,&array[i]);
}
// Traversing and printing data values from array for(i=0;i<10;i++)
{
printf(“array[%d] element is %d\n”,i,array[i]);
}
getch(); return 0;
}
```

1.6 Two Dimensional Array (2-D Array)

Two- dimensional arrays are collection of homogenous data elements where the elements are ordered in number of rows and columns. A two dimensional array can be declared just as we declare a one dimensional array variable. We give the data type of the array variable followed with the name of the array variable and the size of the array in square brackets. In case of a two dimensional array also we give the data type followed with name and size of the array but we add another value in brackets to give the second size of the array. The first size represents the row size and the second size represents the column size of the array. This is because according to the definition of a two dimensional array the array is organized in rows and columns.

Declaration of 2-D Array: We can declare a two dimensional array in the following way:

```
data_type name_of_array [ row_size ] [ column_size];
```

In the above declaration syntax, data type can be any of the valid data types for 1-D arrays. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the row size and column size of the array that are given inside different square brackets. For example:

```
int array [ 5 ] [ 5 ];
char   arr [ 10 ] [ 3] ; float balance [ 2 ] [ 4 ] ;
```

When we declare an array, a contiguous memory location is allocated to that array. Let us look at an example of the physical representation of a two dimensional array of size [3][4] in computer’s memory.

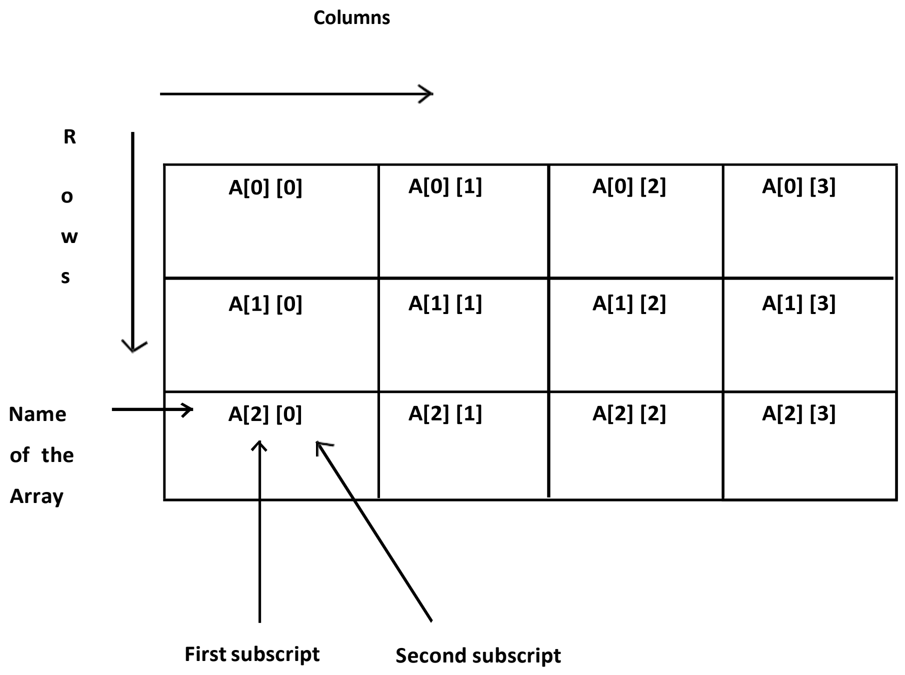


Figure: Representation of two-dimensional array A of size [3][4]

In the above figure, we can see that array A is a two dimensional matrix with three rows and four columns. Two subscripts are needed to represent an element in a two dimensional array. The first subscript represents the row value while the second subscript represents the column value of the array. In C language, similar to single dimensional arrays, the subscripts value for both rows and column values starts from 0. If we want to represent the first element in the array we denote it by **A[0][0],** where the first subscript value 0 means that the element is located in the first row and the second subscript value 0 means that the element is located in the first column. Since we need two subscripts to uniquely represent any element in this type of array, hence it is known as two dimensional arrays.

1.6.1 Storage Representation of 2-D Arrays

Two dimensional arrays can be stored in two different ways. They can be stored in either row-major order or in column-major order.

Row-major order: In row-major order, the elements are stored on a row-by-row basis. Here, the first row is filled first followed by second row and so on. This is the common way of storing elements generally for 2-D array. Let us try to understand this concept using an example. Suppose we need to fill the following 9 element values

{ 1, 2, 3 4, 5, 6, 7, 8, 9 } and in an array B[3][3].

Then, we will first fill up the first row of the array using the first three element values 1, 2, and 3. Once the first row is filled, we then proceed to fill up the rest of the rows one at a time using the rest of the data values. The filled array will have the following structure:

$$B = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}$$

Column-major order: In column-major order, the elements are stored in a column-by-column basis. Here, the first column is filled first followed by second column and so on. Let us use the earlier example itself to understand this concept. We need to fill the following 9 element values

{1, 2, 3 4, 5, 6, 7, 8, 9} and in an array B[3][3].

We will first fill up the first column of the array using the first three element values 1, 2, and 3. Once the first column is filled, we will then proceed to fill up the rest of the columns one at a time using the rest of the data values. The filled array will have the following structure:

$$B = \begin{array}{|c|c|c|}\hline 1 & 4 & 7 \\ \hline 2 & 5 & 8 \\ \hline 3 & 6 & 9 \\ \hline \end{array}$$

Address calculation: The address calculation for elements in two-dimensional arrays will depend on whether the elements are stored in row-major order or in column-major order. Let ‘**b**’ be the base address or the address of the first element in the array and **‘w’** be the word size. Then the address of an element **B[i] [j]** when the array has a maximum row and column size **B [m] [n]** for both the cases can be calculated using the formulas below.

- For row-major, the address of element

  B [i] [j] = b + (i × n + j) × w

  where, i represents the row value,

  j represents the column value and n represents the column size.

- For column-major, the address of element

  B [i] [j] = b + (j × m + i) × w

  where, i represents the row value, j represents the column value and n represents the column size.

1.6.2 Entering Data Values in 2-D Array

We can insert values into a 2-D array at the time of initialization. Insertion of data elements in a 2-D array is similar to insertion of element in 1-D arrays. Let us look at the example below.

```
int  B [3] [4] =   {   { 1, 2, 3, 4 },
                       { 5, 6, 7, 8 },
                       { 9, 10, 11, 12 }
                   };
```

In the above initialization, the data elements of the first row are put together inside braces, followed by data elements of second row inside second braces and so on till the last row. All these individual braces representing each row are then put inside one common brace to indicate that all rows belong to the same array.

We can also insert values into array elements by accessing them individually. For example, if we have an integer array B [2][3] of 6 elements, we can insert the values for each of the 6 data elements using their array positions as follows:

```
B [0] [0] = 11;
B [0] [1] = 22;
B [0] [2] = 33;
B [1] [0] = 44;
B [1] [1] = 55;
B [1] [2] = 66;
```

In the above statements, the value for each array position in the 2-D array has been filled individually instead of giving the values at the time of initialization.

Another easier way to insert data values into elements is to use loop control structure statements like *while*, *do while* and *for* loop. Let us look at a code where with the help of nested for loop we can easily enter the data values for the array B [2] [3] without initialization. This can be implemented as:

```
for ( i = 0; i < 2; i++ )
{
    for ( j = 0; j < 3; j++ )
    {
        scanf("%d", &B[ i ] [ j ]);
    }
}
```

In the above code the first '*for loop'* is used to traverse through the rows and the second for loop is used to traverse through the columns. For each iteration of the first *for loop*, the second *for loop* traverses from the first to the last column in the array. "**B[i][j]"** gives the address of the element at the "**i^{th}" row** and **"j^{th}" column** of the array where the current data value needs to be stored.

1.6.3 Accessing Values from 2-D Array

We have learnt the different ways of inserting values in a two dimensional array. Once the values have been inserted we may need to access the array for various purposes. The procedure for accessing the values in a two dimensional array is similar to entering the values. This can be implemented as:

```
for( i = 0; i < row_size; i++ )
{
for ( j=0; j< column_size; j++ )
{
printf("%d", B[ i ][ j ]);
}
}
```

In the above code we have used a nested '*for loop*' to traverse the two dimensional array. The first '*for loop*' is used to traverse through the rows and the second *for loop* is used to traverse through the columns. For each iteration of the first *for loop*, the second for loop traverses from the first to the last column in the array. The 'printf' statement prints the integer value which is stored in the address of the given array location. "**B[i][j]"** gives the address of the element at the "**i^{th}" row** and **"j^{th}" column** of the array where the current data value needs is stored.

Example: **Program to enter and access data elements in a 2-D array**

```
# include<stdio.h>
 # include<conio.h>
 void main()
{
```

```
int A[10][10], row_size, col_size, i, j; clrscr();

printf("Give the number of rows you want : "); scanf("%d",&row_size);
printf("Give the number of columns you want : ");
scanf("%d",&col_size);
// Entering elements in 2D array printf("\n Enter elemnts in an array\n");
for(i=0;i<row_size;i++)
{
for(j=0;j<col_size;j++)
{
scanf("%d",&A[i][j]);
}
}
// Accessing and printing elements in 2D array printf("\n The elements in
the array A are:\n"); for(i=0;i<row_size;i++)
{
for(j=0;j<col_size;j++)
{
printf("A[%d][%d] = %d\n",i,j,A[i][j]);
}
printf("\n");
}
getch();
}
```

Check Your Progress

Q.5: The base address of a char array C[15] is 210.What is the memory address of the C[14]?

Q.6: Find the memory address of D[4] [8] for an array D[10][15] and when the word size is 4 and base address is 100.

1.7 Answer to Check Your Progress

Ans. to Q. No. 1: An array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations.

Ans. to Q. No. 2: The syntax for declaring an array is:

data_type name_of_array [size_of_array];

Ans. to Q. No. 3: Arrays cannot store elements of different data types.

Ans. to Q. No. 4: Yes, arrays can be declared without mentioning the size provided we initialize the array with the value of the elements.

Ans. to Q. No. 5: In this case, b=210, i=14 and w=1(since character data types takes a byte size of 1

for storage in C).

So, the address of C [14] = b + (i × w)

= 210 + (14×1)

= 224

Ans. to Q. No. 6: Let us assume that row-major storage representation in used in this case. Then in this

case, b =100, i= 4, j=8, n= 15, w=4

So the address for D [4] [8] = b + (i × n + j)

= 100 + (4×15 + 8) × 4

= 100 + 68 × 4

= 100 + 272

= 372

1.8 Model Questions

1. Define array.
2. What is rowmajor order and column major order of representation?
3. Write a program to input a one dimensional array of 10 elements. Also write a function to print the elements on computer screen.
4. How is address translation done in case of 1-D arrays?
5. Write a program to input a two dimensional array A[5][3].Also write a function to print the elements on computer screen.
6. How is address translation done in case of 2-D arrays?
7. What do youmean bymulti-dimensional arrays? Can there be arrays of more than two dimensions?
8. Why do we need arrays?

Unit-8

Strings

1.1 Learning Objectives

1.2 Introduction.

1.3 String

1.3.1 String Declaration and Initialization

1.3.2 Entering Values in String

1.3.3 Accessing Values in String

1.4 Array of Strings

1.5 String Handling Functions

1.5.1 *strlen()* Function

1.5.2 *strcpy()* Function

1.5.3 *strcmp()* Function

1.5.4 *strrev()* Function

1.5.5 *strcat()* Function

1.6 Check Your Progress

1.7 Answer to Check Your Progress

1.1 Learning Objectives

After going through this unit, the learner will be able to learn:

- define a string
- declare and initialize a string
- use string handling functions like *strlen(), strcmp(), strcpy()*
- use string handling functions like *strrev()* and *strcat()*

1.2 Introduction

In the previous unit, we have learnt about arrays and its types. In this unit we will learn about array of characters i.e., string. We will learn to define, declare and initialize string. In addition to this different string handling functions will also be discussed in this unit.

1.3 Strings

An array is a collection of homogeneous elements that are finite and ordered. Strings are also arrays but of character data type. Let us now discuss the concept of strings.

An array of characters is called a ***string***. In other words, strings are arrays where the data type is character. Arrays can be one dimensional or multidimensional. But strings are one-dimensional array of characters which are terminated by a **null character** represented by **'\0'**. Every string contains one or more characters that comprise the string followed by a null character '\0' that indicates the end of the string.

1.3.1 String Declaration and Initialization

Declaration of String: A string can be declared just as we declare any other array variable. We give the data type of the variable followed with the name and size of the variable. In case of a string, we give the data type as character followed with the name of the string and the size of the string. The size of the string should be the sum of the size of a number of elements that the string can contain plus the size for the null character that designates the end of the string. So, to hold the null character at the end of the string, the size of the string should be one more than the number of characters intended to enter in the string.

We can declare a string in the following way:

char name_of_string [size_of_string];

In the above declaration syntax, data type has to be a
character. It is followed by the name of the string variable and the
size of the string that is given inside square brackets. For example:

char array [5] ;

char name [10] ;

char book [30] ;

Need for Strings: To stress the need for strings let us look at the following example:

A cricket match has been organized between two teams A and B. Suppose we do not have the knowledge of string arrays and we need to keep the name of the best player. If we use a character variable then we will be able to store only one character of the player's name.

char name1;

Let us assume that name of the player is "Rahul". To store this name do we declare five char variables to store the five characters in the name? This as shown below does not solve our
problem.

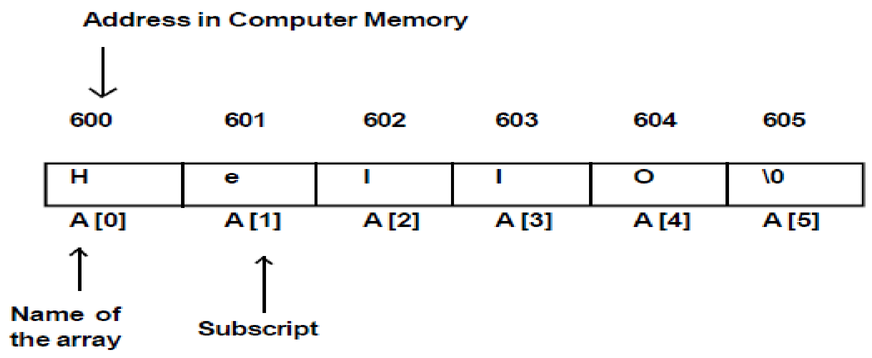
char name1, name2, name3, name4, name5;

An easier way to keep the scores of the team would be to use a character array or a string. Instead of declaring many variables for keeping the name of one individual we can store the name in one string variable.

char best_player [6];

Here, we declare a character array of size six since the first five locations will store the name “Rahul” and the 6th location of the array will store the null character.

Definition of Strings: When we declare a string, a contiguous memory location is allocated to that string. Let us look at an example of the physical representation of string of size 6 which contains the string “Hello” in computers memory.



Physical representation of string in memory

In the above figure we can see how array is actually stored in the memory. Each string element is identified with the help of the index or subscript value. The starting index value for string in C is always 0. In the above example, the first element of the string is stored in the memory address 600. For most C compilers char variable requires one byte of storage. Since strings are stored in contagious memory, so the second element will be stored next to the first element in location 601.

String Initialization: We can initialize the elements of the string when we declare the string. Initialization is generally done when we already know the character values of the string. Just like in declaration, in initialization also we give the data type of the string, followed by the name and size of the array. But in addition to these, we also provide the values of the character elements of the string within braces **{}** separated by commas where the values of the elements need to be put inside single quotes. Let us look at how initialization is done using the following example.

char A [6] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’ };

In the above example, the array A is declared and initialized. The character array A has a size of 6 and the values of the elements in the string have been initialized as follows:

A [0] = H

A [1] = e

A [2] = l

A [3] = l

A [4] = o A [5] = ‘\0’

The first value that is given in braces is put into the first string location with subscript value 0. So ‘H’ is kept at location A [0]. Similarly the second value ‘e’ inside braces is put into the second string location with subscript value A [1] and so on till all the given values have been put in the string.

We can also initialize the values of string without mentioning the string size in the square brackets. That is, we can leave the square brackets empty. For this case the compiler assumes the string size to be equal to the number of char values provided inside braces { }. For example, we can write the following statement:

char A [ ] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’ };

In the above statement, the compiler will assume that the size of the array is 6 since six data values have been given inside the braces. There is another much simpler way to initialize the string. We can also give the values of the character elements together in double quotes “” instead to providing them individually. For example, we can write the following statement:

char A [ ] = “Hello”;

For this case the compiler assumes the string size to be equal to the number of char values provided inside braces {} plus one more for the null character. In this case, we do not need to explicitly provide the null character at the end of the string. The compiler itself will add the null character at the end of the string.

1.3.2 Entering Values in String

We can insert values into a string at the time of initialization. But apart from that, we can also insert values into strings by accessing them individually. For example, if we have a string A of 6 elements, we can insert the values for each of the six string positions as follows:

```
A [0] = ‘H’;
A [1] = ‘e’;
A [2] = ‘l’;
A [3] = ‘l’;
A [4] = ‘o’;
A [5] = ‘\0’;
```

In the above statements, the value for each string position has been filled individually instead of giving the values at the time of initialization and declaration. Let us now suppose that we have a string of 50 elements which consists in the name of a book.

A solution to the above problem is to use loop control structure statements like *while*, *do while* and *for* loop. Let us look at a code where with the help of *for* loop we can easily enter any number of character values. This can be implemented as:

```
for ( i = 0; i < 50; i++ )
{
scanf(“%c”, &A[ i ]);
}
```

In the code above, a ‘for loop’ is used to traverse from the first element of the string to the last element. The above ‘scanf’ statement accepts a character value and stores it in the address of the given string location. The location in the string is moved from the first element position to the last element position using for loop. “&A[i]” gives the address of the ‘ith’ element of the string where the current character data value needs to be stored.

An easier solution to this is that instead of providing the data values individually we can provide them as together as a string. Let us look at the code below which has implemented this solution.

```
scanf(“%s”, &A);
```

The above ‘scanf’ statement accepts a string instead of a single character and stores it in the address of the given string location. For our case, the above statement accepts the string and stores it the string variable A.

1.3.3 Accessing Values from Strings

We have learnt in the previous part about entering values in a string. Once the values have been inserted we may need to access the string for various purposes. The procedure for accessing the values of a string is similar to entering the values. This can be implemented in two ways. In the first way we access the values of the string individually as follows:

```
for ( i = 0; i < 6; i ++ )
{
printf(“c”, A[ i ]);
}
```

In the above code a ‘for loop’ is used to traverse from the first element of the string to its last element. The ‘printf’ statement prints the character value which is stored in the address of the given string location. The array location is moved from the first element position to the last element position using “ *for loop”*. “**A[i]”** gives the value of the element at the **ith position** of the string. In the second way we access the values of the string together. This can be implemented as:

printf(“%s”, A);

The above ‘printf’ statement prints a string instead of a single character from the address of the given string location.

Eample: Program to enter and access elements in strings #

```
include<stdio.h>
# include<conio.h> void main()
{
// Program to enter and access elements in strings char str1[10], str2[10];
int i; clrscr();
// Entering values individually printf(“Enter string str1:\n”);
for(i=0;i<9;i++)
{
scanf(“%c”,&str1[i]);
}
// Displaying entered values individually printf(“The values entered in
string1 are:\n”); for(i=0;i<9;i++)
        {

                printf(“Value in string 1 str1[%d] are %c\n”,i,str1[i]);
               }
// Entering values together as a string printf(“Enter string str2:\n”);
scanf(“%s”,&str2);
// Displaying entered values together as a string printf(“The values
entered in string2 are:\n”); printf(“%s\n”,str2);
        getch();
```

}

1.4 Array of Strings

Array of strings is a two dimensional character array. Similar to two dimensional arrays, strings can also be of two or more dimensions.

We can declare a two dimensional string in the following way:

char name_of_string [row_size] [column_size] ;

In the above declaration syntax, data type of the string has to be of character type. It is followed by the name of the string variable and the row size and column size of the string are given inside different square brackets. For example:

char array [5] [5];

char student_name [3] [10] ; char book_name [2] [4] ;

Inserting and accessing of values in a two dimensional string is similar to insertion and access of data elements in 2-D arrays. For example, suppose we need to store the name of 5 best players in the team. We can declare a string **name [5][30]** to keep the names of the persons and this can be implemented as:

char name [5] [30] ;

In the above declaration, the string variable **name** has memory to store five names where each name can have a maximum of 29 characters. A *for loop* can be used for traversing from the first row to the 5th row of the string variable to insert and access the names of the five individuals. This can be implemented as:

```
for ( i=0 ; i < 5; i++ )
{
scanf("%s",name[i]);
}
for ( i=0 ; i < 5; i++ )
```

{

printf("%s",name[i]);

}

Example: Program to enter and access elements in two dimensional strings */

```
# include<stdio.h> # include<conio.h> void main()
{
// Program to enter and access elements in 2-D strings char name[5][30];
int i; clrscr();
// Entering values in 2D string printf("Enter first name of five persons:");
for(i=0;i<5;i++)
{
printf("\nEnter name of %d person:",i+1); scanf("%s",&name[i]);
}
// Displaying values in 2D string for(i=0;i<5;i++)
{
printf("\nName of %d person is :",i+1); printf("%s",name[i]);
}
        getch();

            }
```

Check Your Progress

Q.1 What is a string?

Q.2: Give the syntax for declaring a string.

1.5 String Handling Functions

String handling contains functions that are used for manipulating and performing special operations on strings. The file ***"string.h"*** is available in the C library and contains many string manipulation functions. We can use the functions in this file by including the header file "string.h" in our C program. Some of the common functions are:

- strlen() function
- strcpy() function
- strcmp() function
- strrev() function
- strcat() function

Let us look at these functions in detail in the following section:

1.5.1 strlen() Function

The function *strlen()* is used for finding the number of characters present in a given string. It gives back the length of the specified string variable. The syntax is as follows:

len = strlen(str1);

where 'len' is an integer variable which keeps the length of the string variable 'str1'. The following program determines the length of a string using *strlen()* function.

Example: Program to find the length of a string # include<stdio.h>

```
# include<conio.h> # include<string.h> void main()
{
// Program to find the length of a string char str1[10];
int len; clrscr();
printf("Enter string :\n"); scanf("%s",&str1);
len = strlen(str1);
printf("The length of string %s is %d",str1,len); getch();
}
```

1.5.2 strcpy() Function

The function *strcpy()* is used for coping the contents of one string to another string. It copies the contents of the source string to a destination string. The syntax is as follows:

strcpy(str1,str2);

where ‘str1’ is the destination string and ‘str2’ is the source string. The following program copies contents of one string to another using *strcpy()* function.

Example: Program to copy the contents of one string to another string*/

```
# include<stdio.h> # include<conio.h> # include<string.h> void main()
{
// Program to copy contents of one strin to another string


    char str1[20], str2[10]; clrscr();
printf(“\nEnter string str1:”); scanf(“%s”,&str1); printf(“\nEnter string
str2:”); scanf(“%s”,&str2);
printf(“\nValue of string str1 before copy is :%s”,str1); strcpy(str1,str2);
printf(“\nValue of string str1 after copy is :%s”,str1); getch();
}
```

1.5.3 strcmp() Function

The function *strcmp()* is used for comparing the contents of two strings. It compares the contents of the strings character by character and then returns an integer as the output. The syntax is as follows:

s = strcmp(str1,str2);

where ‘s’ is an integer and ‘str1’, ‘str2’ are the two strings whose contents are compared. The value of the s has the following meanings:

If s = 0, then str1 and str2 are equal If s = 1, then str2 > str1

If s = -1, then str1 > str2

The following program compares the values of two strings using strcmp() function.

Example: Program to compare the contents of two strings*/ # include<stdio.h>

```
# include<conio.h> # include<string.h> void main()
{
//Program to compare contents of one string to another  string

char str1[10], str2[10]; int result;
clrscr();
printf("\nEnter string str1:"); scanf("%s",&str1); printf("\nEnter string str2:"); scanf("%s",&str2);
result = strcmp(str1,str2); if(result == 0)
    printf("\nString str1 is equal to String str2 "); else
    printf("\nString str1 not equal String str2 "); getch();
}
```

1.5.4 strrev() Function

The function *strrev()* is used to reverse the contents of any given string. It reverses all the contents of the string except the null character which is used to indicate the end of a string. The syntax is as follows:

strrev(str1);

where 'str1' is the string whose contents are to be reversed. The following program reverses the contents of a string using *strrev()* function.

Example: Program to reverse the contents of a string # include<stdio.h>

```
# include<conio.h> # include<string.h> void main()
{
```

```
// Program to reverse the contents of a string char str1[10];
clrscr();
        printf(“\nEnter string :”); scanf(“%s”,&str1); printf(“\nOriginal
string: %s”,str1); strrev(str1);
printf(“\nAfter string reversal. String is %s”,str1); getch();
}
```

1.5.5 *strcat()* Function

The function *strcat()* is used to combine the contents of one string with another string. It concatenates the contents of the source string to a destination string. The syntax is as follows:

strcat(str1,str2);

where ‘str1’ is the destination string and ‘str2’ is the source string. The following program concatenates the contents of one string to another string using strcat() function.

Example: Program to concatenate two strings # include<stdio.h>

```
# include<conio.h> # include<string.h> void main()
{
// Program to concatenate two strings char str1[20], str2[10];
clrscr();
printf(“\nEnter string str1:”); scanf(“%s”,&str1); printf(“\nEnter string
str2:”); scanf(“%s”,&str2); strcat(str1,str2);
printf(“Conactenated string is %s\n”,str1); getch();
}
```

Check Your Progress

Q.3: Can we declare a string without mentioning the size of a string?

Q.4: What is string handling functions?

Q.5: Can we perform the operations done on strings by string handling functions without using these library functions?

1.6 Answer to Check Your Progress

Ans. to Q. No. 1: An array of characters is called strings. In other words, strings are arrays where the data type is character.

Ans. to Q. No. 2: We can declare a string in the following way:

char name_of_string [size_of_string];

The data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets.

Ans. to Q. No. 3: Yes, we can declare a string without mentioning its size provided we initialize it with the data elements.

Ans. to Q. No. 4: String handling contains functions that are used for manipulating and performing special operations on strings. The file **"string.h"** is available in the C library and contains many string manipulation functions.

Ans. to Q. No. 5: Yes, the operations performed by string handling functions can be performed without using them.

1.7 Model Questions

1. Define strings. How are strings represented in memory?
2. Write a program to find the length of a string without using library functions.
3. Write a program to copy one string to another without using library functions.
4. Write a program to combine two strings without using library functions.
5. Write a program to reverse a string without using library functions.
6. Write a programto compare between two strings without using library functions.
7. Write a program to take input a number and a string and then display the string that many numbers of times.

Unit-9
Dynamic Memory

1.1 Learning Objectives
1.2 Introduction
1.3 Dynamic Memory
    1.3.1 malloc()
    1.3.2 free()
    1.3.3 realloc()
    1.3.4 calloc()
1.4 Answer to Check Your Progress
1.5 Model Questions

1.1 Learning Objectives

After going through unit the learner will able to learn.

- About dynamic memory
- About malloc()
- About free()
- About realloc()
- About calloc()

1.2 Introduction

Memory allocation refers to the reservation of memory for storing data. Memory allocation is done in C language in two ways (i) Static allocation and (ii) Dynamic allocation. Static allocation is done by using array. The main disadvantage of static allocation is that the programmer must know the size of the array or data while writing the program. Generally, it is not possible to know the required memory in advance. To overcome this problem dynamic memory allocation is done. Dynamic

memory allocation refers to the allocation of memory during program execution. The basic difference between static and dynamic memory allocation is that in static allocation
memory is allocated during the compile time, whereas in dynamic allocation memory is allocated during the program execution time.

1.3 Dynamic Memory

Up until now, we've been talking about memory that pretty much is set up at the beginning of the program run. You have constant strings here and there, arrays of pre-declared length, and variables all declared ahead of time. But what if you have something like the following?

Assignment: Implement a program that will read an arbitrary number of integers from the
keyboard. The first line the user enters will be the number of `int`s to read. The `int`s themselves
will appear on subsequent lines, one `int` per line.

Yes, it's that time again: break it up into component parts that you can implement. You'll
need to read lines of text from the keyboard (there's a cool little function called **`fgets()`** that
can help here), and the first line you'll need to convert to an integer so you know how many
more lines to read. (You can use **`atoi()`**, read “ascii-to-integer” to do this conversion.) Then
you'll need to read that many more strings and store them...where?

Here's where dynamic memory can help out--we need to store a bunch of `int`s, but we
don't know how many until after the program has already started running. What we do is find

out how many `int`s we need, then we calculate how many bytes we need for each, multiply
those two numbers to get the total number of bytes we need to store everything, and then ask the
OS to allocate that many bytes for us on the heap for us to use in the manner we choose. In this
case, we're choosing to store `int`s in there.

There are three functions we're going to talk about here. Well, make that four functions,
but one is just a variant of another: **`malloc()`** (allocate some memory for us to use), **`free()`**
(release some memory that **`malloc()`** gave us earlier), **`realloc()`** (change the size of some
previously allocated memory), and **`calloc()`** (just like **`malloc()`**, except clears the memory to zero.)

Using these functions in unison results in a beautifully intricate dance of data, ebbing and
flowing with the strong tidal pull of the dedicated user's will.

Yeah. Let's cut the noise and get on with it here.

Check Your Progress

Fill in the blanks

Q.1: Memory allocation refers to the reservation of memory for……………

Q.2: Memory allocation is done in C language in …………..

Q.3: The main disadvantage of …………..is that the programmer must know the size of the array or data while writing the program

1.3.1 malloc()

malloc(): This function is used to allocate a block of memory. After allocating the memory it returns a pointer of type *void*. This means that one can assign it to any type of pointer. The syntax for using *malloc()* is as follows:

ptr =(cast-type *)malloc(no. of bytes);

Here, ptr is a pointer of type cast-type. For example,

```
int x;
x=(int *)malloc(100);
```

If it is unable to find the requested amount of memory, malloc() function returns NULL.

This is the big one: he's the guy that gives you memory when you ask for it. It returns to you a pointer to a chunk of memory of a specified number of bytes, or `NULL` if there is some kind of error (like you're out of memory). The return type is actually `void*`, so it can turn into a pointer to whatever you want.

Since **`malloc()`** operates in bytes of memory and you often operate with other data types e.g. “Allocate for me 12 `int`s.”), people often use the **`sizeof()`** operator to determine how many bytes to allocate, for example:

```
      int *p;
      p = malloc(sizeof(int) * 12); //
allocate for me 12 ints!
```

that was pretty much an example of how to use **malloc()**, too. You can reference the result using pointer arithmetic or array notation; either is fine since it's a pointer. But you should really check the result for errors:

```
      int *p;
      p = malloc(sizeof(float) * 3490); //
allocate 3490 floats!
if (p == NULL) {
```

```
printf("Horsefeathers!  We're  probably  out  of
memory!\n");
exit(1);
}
```

More commonly, people pack this onto one line:

```
if  ((p  =  malloc(100))  ==  NULL)  {  //
allocate 100 bytes
printf("Ooooo! Out of memory error!\n");
exit(1);
}
```

Now remember this: you're allocating memory on the heap and there are only two ways to ever get that memory back: 1) your program can exit, or 2) you can call **`free()`** to free a **`malloc()`**'d chunk. If your program runs for a long time and keeps **`malloc()`**ing and never

`free()`ing when it should, it's said to “leak” memory. This often manifests itself in a way such as, “Hey, Bob. I started your print job monitor program a week ago, and now it's using 13 terabytes of RAM. Why is that?”

Be sure to avoid memory leaks! **`free()`** that memory when you're done with it!

1.3.2 free()

free( ): A memory area that is dynamically allocated using either calloc() or malloc() doesn’t get freed automatically when the execution terminates. You must explicitly use free() library function to release the memory space.

Syntax: free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

Speaking of how to free memory that you've allocated, you do it with the implausibly-named **free()** function.

This function takes as its argument a pointer that you've picked up using **malloc()** (or

calloc()). And it releases the memory associated with that data. You really should never use
memory after it has been **free()**'d. It would be Bad.
So how about an example:

```
int *p;
p = malloc(sizeof(int) * 37); // 37 ints!
free(p); // on second thought, never mind!
```

Of course, between the **`malloc()`** and the **`free()`**, you can do anything with the memory your twisted little heart desires.

1.3.3 realloc()

realloc( ): The size of dynamically allocated memory can be changed by using realloc() library function.

Syntax: void *realloc(void *ptr, size_t size);

Example: int *ptr = (int *)malloc(sizeof(int)*2);//dynamic allocation
for two integer value
int *ptr_new;
ptr_new = (int *)realloc(ptr, sizeof(int)*3);// dynamic reallocation

realloc() is a fun little function that takes a chunk of memory you allocated with **malloc()** (or **calloc()**) and changes the size of the memory chunk. Maybe you thought you only needed 100 ints at first, but now you need 200. You can **realloc()** the block to give you the space you need.

This is all well and good, except that **realloc()** might have to *move your data* to another
place in memory if it can't, for whatever reason, increase the size of the current block. It's not
omnipotent, after all.

What does this mean for you, the mortal? Well in short, it means you should use **realloc()** sparingly since it could be an expensive operation. Usually the procedure is to keep track of how much room you have in the memory block, and then add another big chunk to it if you run out. So first you allocate what you'd guess is enough room to hold all the data you'd require, and then if you happened to run out, you'd reallocate the block with the next best guess of what you'd require in the future. What makes a good guess depends on the program. Here's an example that just allocates more "buckets" of space as needed:

```
#include <stdlib.h>
#define INITIAL_SIZE 10
#define BUCKET_SIZE 5
static int data_count; // how many ints we
have stored
static int data_size; // how many ints we
*can* store in this block
static int *data; // the block of data,
itself
int main(void)
{
void add_data(int new_data); // function
prototype
int i;
// first, initialize the data area:
data_count = 0;
data_size = INITIAL_SIZE;
data = malloc(data_size * sizeof(int)); //
allocate initial area
// now add a bunch of data
for(i = 0; i < 23; i++) {
```

```
add_data(i);
}
return 0;
}
void add_data(int new_data)
{
// if data_count == data_size, the area is
full and
// needs to be realloc()'d before we can
add another:
if (data_count == data_size) {
// we're full up, so add a bucket
data_size += BUCKET_SIZE;
data = realloc(data, data_size *
sizeof(int));
}
// now store the data
*(data+data_count) = new_data;
// ^^^ the above line could have used array
notation, like so:
// data[data_count] = new_data;
data_count++;
}
```

In the above code, you can see that a potentially expensive **realloc()** is only done after the first 10 ints have been stored, and then again only after each block of five after that. This beats doing a **realloc()** every time you add a number, hands down.

(Yes, yes, in that completely contrived example, since I know I'm adding 23 numbers right off the get-go, it would make much more sense to set *INITIAL_SIZE* to 25 or something, but that defeats the whole purpose of the example, now, doesn't it?)

1.3.4 calloc()

calloc(): It is a library function to allocate memory. The difference between *calloc()* and *malloc()* is that *calloc()* initialize the allocated memory to zero where as *malloc()* does not initialize allocated memory to zero.

Declaration Syntax: Following is the declaration for *calloc()* function.

void (cast-type*)calloc(no. of elements to be allocated,
size of each element)

For example:

ptr = (int*) calloc(10, sizeof(int));

This statement allocates contiguous space in memory for an array of 10 elements each of size of int, i.e., 2 bytes.

Since you've already read the section on **malloc()** (you have, right?), this part will be easy! Yay! Here's the scoop: **calloc()** is just like **malloc()**, except that it 1) clears the memory to zero for you, and 2) it takes two parameters instead of one.

The two parameters are the number of elements that are to be in the memory block, and the size of each element. Yes, this is exactly like we did in **malloc()**, except that **calloc()** is doing the multiply for you:

```
// this:
p = malloc(10 * sizeof(int));
// is just like this:
p = calloc(10, sizeof(int));
// (and the memory is cleared to zero when
using calloc())
```

The pointer returned by **calloc()** can be used with **realloc()** and **free()** just as if you had used **malloc()**.

The drawback to using **calloc()** is that it takes time to clear memory, and in most cases, you don't need it clear since you'll just be writing over it anyway. But if you ever find yourself **malloc()**ing a block and then setting the memory to zero right after, you can use **calloc()** to do that in one call.

I wish this section on **calloc()** were more exciting, with plot, passion, and violence, like any good Hollywood picture, but...this is C programming we're talking about. And that should be exciting in its own right.

Check Your Progress

Q.4: ……………allocation refers to the allocation of memory during program execution.

Q.5: ………………function is used to allocate a block of memory.

Q.6: ………..memory area that is dynamically allocated using either calloc() or malloc() doesn’t get freed automatically when the execution terminates.

Q.7: ………..is a memory area that is dynamically allocated using either calloc() or malloc() doesn’t get freed automatically when the execution terminates.

Q.8: ………………………is a library function to allocate memory.

1.4 Answer to Check Your Progress

Ans to Q.1: storing data

Ans to Q.2: two ways.

Ans to Q.3: static allocation

Ans to Q.4: Dynamic memory

Ans to Q.5: malloc()

Ans to Q.6: A free( )

Ans to Q.7: free( )

Ans to Q.8: calloc()

1.5 Model Questions

1. What is dynamic memory allocation?
2. What are the functions used to allocate memory dynamically?
3. What is the difference between malloc and calloc?
4. What is the purpose of realloc( )?
5. What is static memory allocation and dynamic memory allocation?

Block-III

Unit-10

Advance Topics

1.1 Learning Objectives

1.2 Introduction

1.3 Pointer Arithmetic

1.4 typedef

1.5 `enum`

1.6 More `struct` declarations

1.7 Command Line Arguments

1.8 Multidimensional Arrays

1.9 Casting and promotion

1.10 Incomplete types

1.11 `void` pointers

1.12 `NULL` pointers

1.13 More Static

1.14 Typical Multi-file Projects

1.15 The Almighty C Preprocessor

1.15.1 #include

1.15.2 #define

1.15.3 #if and #ifdef
1.16 Pointers to pointers
1.17 Pointers to Functions
1.18 Variable Argument Lists
1.18.1 vprintf()
1.19 Answer to Check Your Progress
1.20 Model Questions

1.1 Learning Objectives

After going through this unit the will able to learn:

- About the Pointer Arithmetic
- About struct declarations
- About Command Line Arguments
- About Void Pointer
- About Null Pointer
- About Variable Argument Lists

1.2 Introduction

Learning pointer arithmetic clarifies this equivalence between pointers an arrays. We can obtain the address of an array element by multiplying the element's index by the number of bytes that each element occupies and add that product to the array's starting address. Then, we can access the data at the resulting address simply by dereferencing that address.

For example, the syntax on the left is equivalent to that on the right

```
a[i]            *(a + i)
&a[i]            (a + i)
```

a + i evaluates to the address of the i+1-th element of a (&a[i]). The rules for pointer arithmetic stipulate that we multiply the element's index by the size of an element before adding the array's starting address.

1.3 Pointer Arithmetic

You can perform math on pointers. What does it mean to do that, though? Well, pay attention, because people use *pointer arithmetic* all the time to manipulate pointers and move around through memory.

You can add to and subtract from pointers. If you have a pointer to a `char`, incrementing that pointer moves to the next `char` in memory (one byte up). If you have a pointer to an `int`, incrementing that pointer moves to the next `int` in memory (which might be four bytes up,
or some other number depending on your CPU architecture.) It's important to know that the
number of bytes of memory it moves differs depending on the type of pointer, but that's actually
all taken care of for you.

```
/* This code prints: */
/* 50 */
/* 99 */
/* 3490 */
int main(void)
{
int a[4] = { 50, 99, 3490, 0 };
int *p;
p = a;
while(*p > 0) {
printf("%i\n", *p);
p++; /* go to the next int in memory */
}
return 0;
}
```

What have we done! How does this print out the values in the array? First of all, we point *p* at the first element of the array. Then we're going to loop until what *p* points at is less than or equal to zero. Then, inside the loop, we print what *p* is pointing at. Finally, *and here's the tricky part*, we *increment the pointer*. This causes the pointer to move to the next int in memory so
we can print it.
In this case, I've arbitrarily decided (yeah, it's shockingly true: I just make all this stuff up)
to mark the end of the array with a zero value so I know when to stop printing. This is known
as a *sentinel value*...that is, something that lets you know when some data ends. If this sounds
familiar, it's because you just saw it in the section on strings. Remember--strings end in a zero
character ('\0') and the string functions use this as a sentinel value to know where the string
ends.
Lots of times, you see a for loop used to go through pointer stuff. For instance, here's some
code that copies a string:

```
char *source = "Copy me!";
char dest[20]; /* we'll copy that string
into here */
char *sp; /* source pointer */
char *dp; /* destination pointer */
for(sp = source, dp = dest; *sp != '\0';
sp++, dp++) {
*dp = *sp;
}
printf("%s\n", dest); /* prints "Copy me!"
*/
```

Looks complicated! Something new here is the *comma operator* **(,)**. The comma operator
allows you to stick expressions together. The total value of the expression is the rightmost
expression after the comma, but all parts of the expression are evaluated, left to right.
So let's take apart this for loop and see what's up. In the initialization section, we point *sp*
and `dp` to the source string and the destination area we're going to copy it to. In the body of the loop, the actual copy takes place. We copy, using the assignment operator, the character that the source pointer points to, to the address that the destination pointer points to. So in this way, we're going to copy the string a letter at a time.
The middle part of the for loop is the continuation condition--we check here to see if the source pointer points at a NUL character which we know exists at the end of the source string.
Of course, at first, it's pointing at `'C'` (of the “Copy me!” string), so we're free to continue.
At the end of the for statement we'll increment both `sp` and `dp` to move to the next character to copy. Copy, copy, copy!

1.4 typedef

This one isn't too difficult to wrap your head around, but there are some strange nuances to it that you might see out in the wild. Basically typedef allows you to make up an alias for a certain type, so you can reference it by that name instead.

Why would you want to do that? The most common reason is that the other name is a little bit too unwieldy and you want something more concise...and this most commonly occurs when you have a `struct` that you want to use.

```
struct a_structure_with_a_large_name {
int a;
float b;
};
typedef                                  struct
a_structure_with_a_large_name NAMESTRUCT;
int main(void)
{
/* we can make a variable of the structure
like this: */
struct        a_structure_with_a_large_name
one_variable;
/* OR, we can do it like this: */
NAMESTRUCT another_variable;
return 0;
}
```

In the above code, we've defined a type, `NAMESTRUCT`, that can be used in place of the other type, `struct a_structure_with_a_large_name`. Note that this is now a full-blown type; you can use it in function calls, or whereever you'd use a “normal” type. (Don't tell typedef'd types they're not normal--it's impolite.)

You're probably also wondering why the new type name is in all caps. Historically, typedef'd types have been all caps in C by convention (it's certainly not necessary.) In C++, this is no

longer the case and people use mixed case more often. Since this is a C guide, we'll stick to
the old ways.

(One thing you might commonly see is a `struct` with an underscore before the `struct` tag name in the typedef. Though technically illegal, many programmers like to use the same name for the `struct` as they do for the new type, and putting the underscore there differentiates
the two visaully. But you shouldn't do it.)

You can also typedef “anonymous” `struct`s, like this

```
typedef struct
{
int a;
float b;
} someData;
```

So then you can define variables as type `someData`. Very exciting.

1.5 enum

Sometimes you have a list of numbers that you want to use to represent different things, but it's easier for the programmer to represent those things by name instead of number. You can use an enum to make symbolic names for integer numbers that programmers can use later in their
code in place of ints.

(I should note that C is more relaxed that C++ is here about interchanging ints and enums.
We'll be all happy and C-like here, though.)

Note that an enum is a type, too. You can typedef it, you can pass them into functions, and
so on, again, just like “normal” types.

Here are some enums and their usage. Remember--treat them just like ints, more or less.

```
enum fishtypes {
HALIBUT,
TUBESNOUT,
SEABASS,
ROCKFISH
};
int main(void)
{
enum fishtypes fish1 = SEABASS;
enum fishtypes fish2;
if (fish1 == SEABASS) {
fish2 = TUBESNOUT;
}
return 0;
}
```

Nothing to it--they're just symbolic names for unique numbers. Basically it's easier for other programmers to read and maintain.
Now, you can print them out using `%d` in **`printf()`**, if you want. For the most part, though, there's no reason to know what the actual number is; usually you just want the symbolic representation.
But, since I know you're dying of curiosity, I might as well tell you that the `enum`s start at zero by default, and increase from there. So in the above example, `HALIBUT` would be 0, `TUBESNOUT` would be 1, and `ROCKFISH` would be 3.
If you want, though, you can override any or all of these:

```
enum frogtypes {
THREELEGGED=3,
FOUREYED,
SIXHEADED=6
};
```

In the above case, two of the enums are explicitly defined. For `FOUREYED` (which isn't defined), it just increments one from the last defined value, so its value is 4.

1.6 More struct declarations

Remember how, many moons ago, I mentioned that there were a number of ways to declare structs and not all of them made a whole lot of sense. We've already seen how to declare a struct globally to use later, as well as one in a typedef situation, *comme ca*:

```
/* standalone: */
struct antelope {
int legcount;
float angryfactor;
};
/* or with typedef: */
typedef struct _goatcheese {
char victim_name[40];
float cheesecount;
} GOATCHEESE;
```

But you can also declare variables along with the `struct` declaration by putting them directly afterward:

```
struct breadtopping {
enum toppingtype type; /* BUTTER, MARGARINE
or MARMITE */
float amount;
} mytopping;
/* just like if you'd later declared: */
```

```
struct breadtopping mytopping;
```

So there we've kinda stuck the variable defintion on the tail end of the `struct` definition.

Pretty sneaky, but you see that happen from time to time in that so-called *Real Life* thing that I
hear so much about.

And, just when you thought you had it all, you can actually omit the struct name in many
cases. For example:

```
typedef struct { /* <--Hey! We left the
name off! */
char name[100];
int num_movies;
} ACTOR_PRESIDENT;
```

It's more *right* to name all your `struct`s, even if you don't use the proper name and only use the typedef'd name, but you still see those naked `struct`s here and there

1.7 Command Line Arguments

I've been lying to you this whole time, I must admit. I thought I could hide it from you and not get caught, but you realized that something was wrong...why doesn't the **`main()`** have a
return type or argument list?

Well, back in the depths of time, for some reason, !!!TODO research!!! it was perfectly acceptable to do that. And it persists to this day. Feel free to do that, in fact, But that's not telling
you the whole story, and it's time you knew the *whole truth*!

Welcome to the real world:

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

What is all that stuff? Before I tell you, though, you have to realize that programs, when executed from the command line, accept arguments from the command line program, and
return a result to the command line program. Using many Unix shells, you can get the return
value of the program in the shell variable `$?`. (This doesn't even work in the windows command
shell--use !!!TODO look up windows return variable!!! instead.) And you specify parameters
to the program on the command line after the program name. So if you have a program called
“makemoney”, you can run it with parameters, and then check the return value, like this:

```
$ makemoney fast alot
$ echo $?
2
```

In this case, we've passed two command line arguments, “fast” and “alot”, and gotten a return value back in the variable `$?`, which we've printed using the Unix **echo** command. How does the program read those arguments, and return that value?
Let's do the easy part first: the return value. You've noticed that the above prototype for
`main()` returns an `int`. Swell! So all you have to do is either return that value from **`main()`**
somewhere, or, alternatively, you can call the function **`exit()`** with an exit value as the
parameter:

```
int main(void)
{
int a = 12;
if (a == 2) {
```

```
exit(3); /* just like running (from main())
"return 3;" */
}
return 2; /* just like calling exit(2); */
}
```

For historical reasons, an exit status of 0 has meant success, while nonzero means failure.

Other programs can check your program's exit status and react accordingly. Ok that's the return status. What about those arguments? Well, that whole definition of *argv* looks too intimidating to start with. What about this *argc* instead? It's just an int, and an easy one at that. It contains the total count of arguments on the command line, *including the name of the program itself*. For example:

```
$ makemoney fast alot # <-- argc == 3
$ makemoney # <-- argc == 1
$ makemoney 1 2 3 4 5 # <-- argc == 6
```

(The dollar sign, above, is a common Unix command shell prompt. And that hash mark (#)

is the command shell comment character in Unix. I'm a Unix-dork, so you'll have to deal. If you

have a problem, talk to those friendly Stormtroopers over there.)

Good, good. Not much to that `argc business`, either. Now the biggie: `argv`. As you

might have guessed, this is where the arguments themselves are stored. But what about that

`char**` type? What do you do with that? Fortunately, you can often use array notation in the

place of a dereference, since you'll recall arrays and pointers are related beasties. In this case,

you can think of *argv* as an array of pointers to strings, where each string pointed to is one of
the command line arguments:

```
$ makemoney somewhere somehow
$ # argv[0] argv[1] argv[2] (and argc is 3)
```

Each of these array elements, *argv[0]*, *argv[1]*, and so on, is a string. (Remember a string is just a pointer to a `char` or an array of `char`s, the name of which is a pointer to the first element of the array.)

I haven't told you much of what you can do with strings yet, but check out the reference section for more information. What you do know is how to **`printf()`** a string using the “`%s`” format specifier, and you do know how to do a loop. So let's write a program that simply prints out its command line arguments, and then sets an exit status value of 4:

```
/* showargs.c */
#include <stdio.h>
int main(int argc, char **argv)
{
int i;
printf("There are %d things on the command
line\n", argc);
printf("The program name is \"%s\"\n",
argv[0];
printf("The arguments are:\n");
for(i = 1; i < argc; i++) {
printf(" %s\n", argv[i]);
}
return 4; /* exit status of 4 */
}
```

Note that we started printing arguments at index 1, since we already printed `argv[0]` before that. So sample runs and output (assuming we compiled this into a program called **showargs**):

```
$ showargs alpha bravo
There are 3 things on the command line
The program name is "showargs"
The arguments are:
alpha
bravo
$ showargs
There are 1 things on the command line
The program name is "showargs"
The arguments are:
$ showargs 12
There are 2 things on the command line
The program name is "showargs"
The arguments are:
12
```

(The actual thing in `argv[0]` might differ from system to system. Sometimes it'll contain
some path information or other stuff.)
So that's the secret for getting stuff into your program from the command line!

1.8 Multidimensional Arrays

Welcome to...the *Nth Dimension!* Bet you never thought you'd see that. Well, here we are.
Yup. The Nth Dimension.
Ok, then. Well, you've seen how you can arrange sequences of data in memory using an

array. It looks something like this:

!!!TODO image of 1d array

Now, imagine, if you will, a *grid* of elements instead of just a single row of them:

This is an example of a *two-dimensional* array, and can be indexed by giving a row number

and a column number as the index, like this: `a[2][10]`. You can have as many dimensions in

an array that you want, but I'm not going to draw them because 2D is already past the limit of

my artistic skills.

So check this code out--it makes up a two-dimensional array, initializes it in the definition

(see how we nest the squirrely braces there during the init), and then uses a *nested loop* (that is,

a loop inside another loop) to go through all the elements and pretty-print them to the screen.

```
#include <stdio.h>
int main(void)
{
int a[2][5] = { { 10, 20, 30, 40, 55 }, /*
[2][5] == [rows][cols] */
{ 10, 18, 21, 30, 44 } };
int i, j;
for(i = 0; i < 2; i++) { /* for all the
rows... */
for(j = 0; j < 5; j++) { /* print all the
columns! */
printf("%d ", a[i][j]);
}
/* at the end of the row, print a newline
for the next row */
printf("\n");
}
return 0;
}
```

As you might well imagine, since there really is no surprise ending for a program so simple
as this one, the output will look something like this:

```
10 20 30 40 55
10 18 21 30 44
```

Hold on for a second, now, since we're going to take this concept for a spin and learn a little bit more about how arrays are stored in memory, and some of tricks you can use to access them. First of all, you need to know that in the previous example, even though the array has two
rows and is multidimensional, the data is stored sequentially in memory in this order: 10, 20, 30,
40, 55, 10, 18, 21, 30, 44.
See how that works? The compiler just puts one row after the next and so on.
But hey! Isn't that just like a one-dimensional array, then? Yes, for the most part, it technically is! A lot of programmers don't even bother with multidimensional arrays at all, and just use single dimensional, doing the math by hand to find a particular row and column. You can't technically just switch dimensions whenever you feel like it, Buccaroo Bonzai, because the
types are different. And it'd be bad form, besides.
For instance...nevermind the “for instance”. Let's do the same example again using a single

dimensional array:

```
#include <stdio.h>
int main(void)
{
int a[10] = { 10, 20, 30, 40, 55, /* 10
elements (2x5) */
10, 18, 21, 30, 44 };
int i, j;
for(i = 0; i < 2; i++) { /* for all the
rows... */
for(j = 0; j < 5; j++) { /* print all the
columns! */
int index = i*5 + j; /* calc the index */
printf("%d ", a[index]);
}
/* at the end of the row, print a newline
for the next row */
printf("\n");
}
return 0;
}
```

So in the middle of the loop we've declared a local variable *index* (yes, you can do that--remember local variables are local to their block (that is, local to their surrounding squirrley braces)) and we calculate it using *i* and *j*. Look at that calculation for a bit to make sure it's correct. This is technically what the compiler does behind your back when you accessed the array using multidimensional notation.

1.9 Casting and promotion

Sometimes you have a type and you want it to be a different type. Here's a great example:

```
int main(void)
{
int a = 5;
int b = 10;
float f;
f = a / b; /* calculate 5 divided by 10 */
printf("%.2f\n", f);
return 0;
}
```

And this prints: 0

What? Five divided by 10 is zero? Since when? I'll tell you: since we entered the world of integer-only division. When you divide one int by another int, the result is an int, and
any fractional part is thrown away. What do we do if we want the result to become a float
somewhere along the way so that the result is correct?
Turns out, either integer (or both) in the divide can be made into a float, and then the result of the divide will be also be a float. So just change one and everything should work out. “Get on with it! How do you cast?” Oh yeah--I guess I should actually do it. You might recall the cast from other parts of this guide, but just in case, we'll show it again:

```
f = (float)a / b; /* calculate 5 divided by
10 */
```

Bam! There is is! Putting the new type in parens in front of the expression to be converted,
and it magically becomes that type!

You can cast almost anything to almost anything else, and if you mess it up somehow, it's entirely your fault since the compiler will blindly do whatever you ask. `:-)`

1.10 Incomplete types

This topic is a little bit more advanced, but bear with it for a bit. An incomplete type is simply the declaration of the name of a particular `struct`, put there so that you can use pointers to the `struct` without actually knowing the fields stored therein. It most often comes up when people don't want to #include another header file, which can happen for a variety of different reasons.

For example, here we use a pointer to a type without actually having it defined anywhere in **`main()`**. (It is defined elsewhere, though.)

```
struct foo; /* incomplete type! Notice
it's, well, incomplete. */
int main(void)
{
struct foo *w;
w = get_next_wombat(); /* grab a wombat */
process_wombat(w); /* use it somewhere */
return 0;
}
```

I'm telling you this in case you find yourself trying to include a header that includes another header that includes the same header, or if your builds are taking forever because you're

including too many headers, or...more likely you'll see an error along the lines of “cannot
reference incomplete type”. This error means you've tried to do too much with the incomplete
type (like you tried to dereference it or use a field in it), and you need to #include the right
header file with the full complete declaration of the `struct`.

1.11void pointers

Welcome to *THE VOID*! As Neo Anderson would say, “...Whoa.” What is this `void` thing?
Stop! Before you get confused, a `void` pointer isn't the same thing as a `void` return value from a function or a `void` argument list. I know that can be confusing, but there it is. Just wait
until we talk about all the ways you can use the `static` keyword.
A `void` pointer is a *pointer to any type*. It is automatically cast to whatever type you assign
into it, or copy from it. Why would you want to ever use such a thing? I mean, if you're going to
dereference a pointer so that you can get to the original value, doesn't the compiler need to know
what type the pointer is so that it can use it properly?
Yes. Yes, it does. Because of that, you cannot dereference a `void` pointer. It's against the law, and the C Police will be at your door faster than you can say Jack Robinson. Before you can use it, you have to cast it to another pointer type.

How on Valhalla is this going to be of any use then? Why would you even want a pointer you didn't know the type of?

The Specification: Write a function that can append pointers *of any type* to an array. Also write a function that can return a particular pointer for a certain index.

So in this case, we're going to write a couple useful little functions for storing off pointers, and returning them later. The function has be to *type-agnostic*, that is, it must be able to store pointers of any type. This is something of a fairly common feature to libraries of code that manipulate data--lots of them take `void` pointers so they can be used with any type of pointer the programmer might fancy.

Normally, we'd write a linked list or something to hold these, but that's outside the scope of this book. So we'll just use an array, instead, in this superficial example. Hmmm. Maybe I should write a beginning data structures book...

Anyway, the specification calls for two functions, so let's pound those puppies out right here:

```
#include <stdio.h>
void *pointer_array[10]; /* we can hold up
to 10 void-pointers */
int index=0;
void append_pointer(void *p)
{
pointer_array[index++] = p;
}
void *get_pointer(int i)
{
return pointer_array[i];
}
```

Since we need to store those pointers somewhere, I went ahead and made a global array of them that can be accessed from within both functions. Also, I made a global index variable to remember where to store the next appended pointer in the array.

So check the code out for **`append_pointer()`** there. How is all that crammed together into one line? Well, we need to do two things when we append: store the data at the current index, and move the index to the next available spot. We copy the data using the assignment operator, and then notice that we use the *post-increment* operator (**`++`**) to increment the index.

Remember what *post*-increment means? It means the increment is done *after* the rest of the
expression is evaluated, including that assignment.

The other function, **`get_pointer`**, simply returns the `void*` at the specified index, *`i`*. What you want to watch for here is the subtle difference between the return types of the two functions. One of them is declared `void`, which means it doesn't return anything, and the other one is declared with a return type of `void*`, which means it returns a `void` pointer. I know, I know, the duplicate usage is a little troublesome, but you get used to it in a big hurry. Or else! Finally, we have that code all written--now how can we actually use it? Let's write a **`main()`** function that will use these functions:

```
int main(void)
{
char *s = "some data!"; /* s points to a
constant string (char*) */
int a = 10;
int *b;
char *s2; /* when we call get_pointer(),
we'll store them back here */
int *b2;
b = &a; /* b is a pointer to a */
/* now let's store them both, even though
they're different types */
append_pointer(s);
append_pointer(b);
/* they're stored! let's get them back! */
s2 = get_pointer(0); /* this was at index 0
*/
        b2 = get_pointer(1); /* this was at index 1 */
        return 0;
        }
```

See how the pointer types are interchangable through the `void*`? C will let you convert the `void*` into any other pointer with impunity, and it really is up to you to make sure you're getting them back to the type they were originally. Yes, you can make mistakes here that crash

the program, you'd better believe it. Like they say, “C gives you enough rope to hang yourself.”

1.12 NULL pointers

I think I have just enough time before the plane lands to talk about `NULL` when it comes to
pointers.
`NULL` simply means a pointer to nothing. Sometimes it's useful to know if the pointer is
valid, or if it needs to be initialized, or whatever. `NULL` can be used as a sentinel value for a
variety of different things. Rememeber: it means “this pointer points to nothing”! Example:

```
int main(void)
```

```
{
int *p = NULL;
if (p == NULL) {
printf("p is uninitialized!\n");
} else {
printf("p points to %d\n", *p);
}
return 0;
}
```

Note that pointers aren't pre initialized to `NULL` when you declare them--you have to explicitly do it. (No non-static local variables are pre initialized, pointers included.)

Check Your Progress

Choose the correct one

Q.1: The maximum combined length of the command-line arguments including the spaces between adjacent arguments is

A. 128 characters

B. 256 characters

C. 67 characters

D. It may vary from one operating system to another

Q.2: What is a multidimensional array in C Language ?

A. It is like a matrix or table with rows and columns

B. It is an array of arrays

C. To access 3rd tow 2nd element use ary[2][1] as the index starts from 0 row or column

D. All the above.

Q.3 Void pointers in C are used to implement ……………in C

A. generic functions

B. Data type

C. Variable

D. None of the above

Q.4 What is (void*)0?

A. Representation of NULL pointer

B. Representation of void pointer

C. Error

D. None of above

1.13 More Static

Modern technology has landed me safely here at LAX, and I'm free to continue writing while I wait for my plane to Ireland. Tell me you're not jealous at least on some level, and I won't believe you.

But enough about me; let's talk about programming. (How's that for a geek pick-up line? If
you use it, do me a favor and don't credit me.)

You've already seen how you can use the `static` keyword to make a local variable persist
between calls to its function. But there are other exciting completely unrelated uses for `static`
that probably deserve to have their own keyword, but don't get one. You just have to get used to

the fact that `static` (and a number of other things in C) have different meanings depending on
context.
So what about if you declare something as `static` in the global scope, instead of local to a function? After all, global variables already persist for the life of the program, so `static` can't mean the same thing here. Instead, at the global scope, `static` means that the variable or function declared `static` *is only visible in this particular source file*, and cannot be referenced
from other source files. Again, this definition of `static` only pertains to the global scope.
`static` still means the same old thing in the local scope of the function.
You'll find that your bigger projects little bite-sized pieces themselves fit into larger bite-sized pieces (just like that picture of the little fish getting eaten by the larger fish being eaten by the fish that's larger, still.) When you have enough related smaller bite-sized pieces, it often makes sense to put them in their own source file.
I'm going to rip the example from the section on `void` pointers wherein we have a couple
functions that can be used to store any types of pointers.
One of the many issues with that example program (there are all kinds of shortcomings
and bugs in it) is that we've declared a global variable called *`index`*. Now, "index" is a pretty
common word, and it's entirely likely that somewhere else in the project, someone will make up
their own variable and name it the same thing as yours. This could cause all kinds of problems,

not the least of which is they can modify your value of *index*, something that is very important
to you.
One solution is to put all your stuff in one source file, and then declare *index* to be
`static` global. That way, no one from outside your source file is allowed to use it. You are
King! `static` is a way of keeping the implementation details of your portion of the code out
of the hands of others. Believe me, if you let other people meddle in your code, they will do so
with playful abandon! Big Hammer Smash!
So here is a quick rewrite of the code to be stuck it its own file:

```
/** file parray.c **/
static void *pointer_array[10]; /* now no one can see it except this file! */
static int index=0; /* same for this one! */
/* but these functions are NOT static, */
/* so they can be used from other files: */
void append_pointer(void *p)
{
pointer_array[index++] = p;
}
void *get_pointer(int i)
{
return pointer_array[i];
}
/** end of file parray.c **
```

What would be proper at this point would be to make a file called `parray.h` that has the function prototypes for the two functions in it. Then the file that has **`main()`** in it can #include `parray.h` and use the functions when it is all linked together.

1.14 Typical Multi file Projects

Like I'm so fond of saying, projects generally grow too big for a single file, very similarly to how The Blob grew to be enormous and had to be defeated by Steve McQueen.
Unfortunately, McQueen has already made his great escape to Heaven, and isn't here to help you
with your code. Sorry.
So when you split projects up, you should try to do it in bite-sized modules that make sense
to have in individual files. For instance, all the code responsible for calculating Fast Fourier
Transforms (a mathematical construct, for those not in the know), would be a good candidate
for its own source file. Or maybe all the code that controls a particular AI bot for a game could
be in its own file. It's more of a guideline than a rule, but if something's not a least in some way
related to what you have in a particular source file already, maybe it should go elsewhere. A
perfect illustrative question for this scenario might be, "What is the 3D rendering code doing in
the middle of the sound driver code?"
When you do move code to its own source file, there is almost always a header file that you should write to go along with it. The code in the new source file (which will be a bunch of

functions) will need to have prototypes and other things made visible to the other files so they
can be used. The way the other source files use other files is to $#include their header files.
So for example, let's make a small set of functions and stick them in a file called *simplemath.c*:

```
/** file simplemath.c **/
int plusone(int a)
{
return a+1;
}
int minusone(int a)
{
return a-1;
}
/** end of file simplemath.c **/
```

A couple simple functions, there. Nothing too complex, yes? But by themselves, they're not much use, and for other people to use them we need to distribute their prototypes in a header
file. Get ready to absorbe this...you should recognize the prototypes in there, but I've added
some new stuff:

```
/** file simplemath.h **/
#ifndef _SIMPLEMATH_H_
#define _SIMPLEMATH_H_
/* here are the prototypes: */
int plusone(int a);
int minusone(int a);

#endif
/** end of file simplemath.h **/
```

Icky. What is all that #ifndef stuff? And the #define and the #endif? They are *boilerplate code* (that is, code that is more often than not stuck in a file) that prevents the header file from being included multiple times.

The short version of the logic is this: if the symbol *`_SIMPLEMATH_H_`* isn't defined then define it, and then do all the normal header stuff. Else if the symbol *`_SIMPLEMATH_H_`* *is* already defined, do nothing. In this way, the bulk of the header file is included only once for the build, no matter how many other files try to include it. This is a good idea, since at best it's a redundant waste of time to re-include it, and at worst, it can cause compile-time errors.

Well, we have the header and the source files, and now it's time to write a file with **`main()`**
in it so that we can actually use these things:

```
/** file main.c **/
#include "simplemath.h"
int main(void)
{
int a = 10, b;
b = plusone(a); /* make that processor
work! */
return 0;
}
/** end of file main.c **/
```

Check it out! We used double-quotes in the #include instead of angle brackets! What this tells the preprocessor to do is, “include this file from the current directory instead of the standard system directory.” I'm assuming that you're putting all these files in the same place for the purposes of this exercise.

Recall that including a file is exactly like bringing it into your source at that point, so, as such, we bring the prototypes into the source right there, and then the functions are free to be
used in **`main()`**. Huzzah!

One last question! How do we actually build this whole thing? From the command line:

```
$ cc -o main main.c simplemath.c
```

You can lump all the sources on the command line, and it'll build them together, nice and easy.

1.15 The Almighty C Preprocessor

Remember back about a million years ago when you first started reading this guide and I mentioned something about spinach? That's right--you remember how spinach relates to the
whole computing process?

Of course you don't remember. I just made it up just now; I've never mentioned spinach in this guide. I mean, c'mon. What does spinach have to do with anything? Sheesh!

Let me steer you to a less leafy topic: the C preprocessor. As the name suggests, this little program will process source code before the C compiler sees it. This gives you a little bit more control over what gets compiled and how it gets compiled.

You've already seen one of the most common preprocessor directives: #include. Other sections of the guide have touched upon various other ones, but we'll lay them all out here for fun.

1.15.1 #include

The well-known #include directive pulls in source from another file. This other file should
be a header file in virtually every single case.

On each system, there are a number of standard include files that you can use for various

tasks. Most popularly, you've seen *stdio.h* used. How did the system know where to find
it? Well, each compiler has a set of directories it looks in for header files when you specify
the file name in angle brackets. (On Unix systems, it commonly searches the */usr/include*
directory.)
If you want to include a file from the same directory as the source, use double quotes
around the name of the file. Some examples:

```
/* include from the system include directory: */
#include <stdio.h>
#include <sys/types.h>
/* include from the local directory: */
#include "mutants.h"
#include "fishies/halibut.h"
```

As you can see from the example, you can also specify a *relative path* into subdirectories out of the main directory. (Under Unix, again, there is a file called *types.h* in the directory */usr/include/sys*.)

1.15.2 #define

The #define is one of the more powerful C preprocessor directives. With it you can declare constants to be substituted into the source code before the compiler even sees them. Let's say
you have a lot of constants scattered all over your program and each number is *hard-coded* (that
is, the number is written explicitly in the code, like “2”).

Now, you thought it was a constant when you wrote it because the people you got the specification swore to you up and down on pain of torture that the number would be “2”, and it
would never change in 100 million years so strike them blind right now.
Hey--sounds good. You even have someone to blame if it did change, and it probably won't
anyway since they seem so sure.

Don't be a fool.
The spec will change, and it will do so right after you have put the number “2” in
approximately three hundred thousand places throughout your source, they're going to say, “You
know what? Two just isn't going to cut it--we need three. Is that a hard change to make?”
Blame or not, you're going to be the one that has to change it. A good programmer will realize that hard-coding numbers like this isn't a good idea, and one way to get around it is to use a #define directive. Check out this example:

```
#define PI 3.14159265358979 /* more pi than
you can handle */
int main(void)
{
float r =10.0;
printf("pi: %f\n", PI);
printf("pi/2: %f\n", PI/2);
printf("area: %f\n", PI*r*r);
printf("circumference: %f\n", 2*PI*r);
return 0;
}
```

(Typically #defines are all capitals, by convention.) So, hey, we just printed that thing out
as if it was a float. Well, it *is* a float. Remember--the C preprocessor substitutes the value of `PI`
before the compiler even sees it. It is just as if you had typed it there yourself.
Now let's say you've used `PI` all over the place and now you're just about ready to ship, and
the designers come to you and say, “So, this whole pi thing, um, we're thinking it needs to be
four instead of three-point-one-whatever. Is that a hard change?”
No problem in this case, no matter how deranged the change request. All you have to do
is change the one #define at the top, and it's therefore automatically changed all over the code
when the C preprocessor runs through it:

```
#define PI 4.0 /* whatever you say, boss */
```

Pretty cool, huh. Well, it's perhaps not as good as to be “cool”, but you have not yet witnessed the destructive power of this fully operational preprocessor directive! You can actually use #define to write little *macros* that are like miniature functions that the preprocessor evaluates, again, before the C compiler sees the code. To make a macro like this, you give an argument list (without types, because the preprocessor knows nothing about types, and then you list how that is to be used. For instance, if we want a macro that evaluates to a number you pass it times 3490, we could do the following:

```
#define   TIMES3490(x)   ((x)*3490)   /*   no
semicolon, notice! */
void evaluate_fruit(void)
{
printf("40 * 3490 = %d\n", TIMES3490(40));
}
```

In that example, the preprocessor will take the macro and *expand* it so that it looks like this to the compiler:

```
void evaluate_fruit(void)
{
printf("40 * 3490 = %d\n", ((40)*3490));
}
```

(Actually the preprocessor can do basic math, so it'll probably reduce it directly to “139600”. But this is my example and I'll do as I please!)

Now here's a question for you: are you taking it on blind faith that you need all those parenthesis in the macro, like you're some kind of LISP superhero, or are you wondering, “Why am I wasting precious moments of my life to enter all these parens?”

Well, you *should* be wondering! It turns out there are cases where you can really generate
some *evil* code with a macro without realizing it. Take a look at this example which builds
without a problem:

```
#define TIMES3490(x) x*3490
void walrus_discovery_unit(void)
{
int tuskcount = 7;
printf("(tuskcount+2)   *   3490   =   %d\n",
TIMES3490(tuskcount + 2));
}
```

What's wrong here? We're calculating `tuskcount+2`, and then passing that through the **`TIMES3490()`** macro. But look at what it expands to:

```
printf("(tuskcount+2)   *   3490   =   %d\n",
tuskcount+2*3490);
```

Instead of calculating (tuskcount+2)*3490, we're calculating tuskcount+(2*3490), because multiplication comes before addition

in the order of operations! See, adding all those extra parens to the macro prevents this sort of thing from happening. So programmers with good
practices will automatically put a set of parens around each usage of the parameter variable in
the macro, as well as a set of parens around the outside of the macro itself.

1.15.3 #if and #ifdef

There are some *conditionals* that the C preprocessor can use to discard blocks of code so that the compiler never sees them. The #if directive operates like the C if, and you can pass it an expression to be evaluated. It is most common used to block off huge chunks of code like a
comment, when you don't want it to get built:

```
void set_hamster_speed(int warpfactor)
{
#if 0
uh this code isn't written yet. someone should really write it.
#endif
}
```

You can't nest comments in C, but you can nest #if directives all you want, so it can be very helpful for that.
The other if-statement, #ifdef is true if the subsequent macro is already defined. There's
a negative version of this directive called #ifndef (“if not defined”). #ifndef is very commonly
used with header files to keep them from being included multiple times:

```
/** file aardvark.h **/
```

```
#ifndef _AARDVARK_H_
#define _AARDVARK_H_
int get_nose_length(void);
void set_nose_length(int len);
#endif
/** end of file aardvark.h **/
```

The first time this file is included, `_AARDVARK_H_` is not yet defined, so it goes to the next line, and defines it, and then does some function prototypes, and you'll see there at the end, the whole #if-type directive is culminated with an #endif statement. Now if the file is included again (which can happen when you have a lot of header files are including other header files *ad infinininininini*--cough!), the macro `_AARDVARK_H_` will already be defined, and so the #ifndef
will fail, and the file up to the #endif will be discarded by the preprocessor.
Another extremely useful thing to do here is to have certain code compile for a certain platform, and have other code compile for a different platform. Lots of people like to build software with a macro defined for the type of platform they're on, such as *LINUX* or *WIN32*. And you can use this to great effect so that your code will compile and work on different types of systems:

```
void run_command_shell(void)
{
#ifdef WIN32
system("COMMAND.EXE");
#elifdef LINUX
system("/bin/bash");
#else
#error We don't have no steenkin shells!
#endif
}
```

A couple new things there, most notable #elifdef. This is the contraction of “else ifdef”, which must be used in that case. If you're using #if, then you'd use the corresponding #elif.

Also I threw in an #error directive, there. This will cause the preprocessor to bomb out right at that point with the given message.

1.16 Pointers to pointers

You've already seen how you can have a pointer to a variable...and you've already seen how a pointer *is* a variable, so is it possible to have a *pointer to a pointer*?

No, it's not.

I'm kidding--of course it's possible. Would I have this section of the guide if it wasn't? There are a few good reasons why we'd want to have a pointer to a pointer, and we'll give you the simple one first: you want to pass a pointer as a parameter to a function, have the function modify it, and have the results reflected back to the caller.

Note that this is exactly the reason why we use pointers in function calls in the first place:

we want the function to be able to modify the thing the pointer points to. In this case, though, the thing we want it to modify is another pointer. For example:

```
void get_string(int a, char **s)
{
switch(a) {
case 0:
*s = "everybody";
break;
case 1:
*s = "was";
break;
case 2:
*s = "kung-foo fighting";
break;
default:
*s = "errrrrrnt!";
}
}
int main(void)
{
char *s;
get_string(2, &s);
printf("s is \"%s\"\n", s); /* 's is "kung-
foo fighting"' */
return 0;
}
```

What we have, above, is some code that will deliver a string (pointer to a `char`) back to the caller via pointer to a pointer. Notice that we pass the *address of* the pointer *s* in **`main()`**.

This gives the function **`get_string()`** a pointer to *s*, and so it can dereference that pointer to
change what it is pointing at, namely *s* itself.

There's really nothing mysterious here. You have a pointer to a thing, so you can dereference the pointer to change the thing. It's just like before, except for that fact that we're operating on a pointer now instead of just a plain base type.

What else can we do with pointers to pointers? You can dynamically make a construction similar to a two-dimensional array with them. The following example relies on your knowledge that the function call **`malloc()`** returns a chunk of sequential bytes of memory that you can use as you will. In this case, we'll use them to create a number of `char*`s. And we'll have a pointer to that, as well, which is therefore of type `char**`.

```
int main(void)
{
char **p;
p = malloc(sizeof(char*) * 10); // allocate
10 char*s
return 0;
}
```

Swell. Now what can we do with those? Well, they don't point to anything yet, but we can call **`malloc()`** for each of them in turn and then we'll have a big block of memory we can store strings in.

```
int main(void)
{
char **p;
int i;
p = malloc(sizeof(char*) * 10); // allocate
10 char*s-worth of bytes
for(i = 0; i < 10; i++) {
*(p+i) = malloc(30); // 30 bytes for each
pointer
// alternatively we could have written,
above:
// p[i] = malloc(30);
// but we didn't.
sprintf(*(p+i), "this is string #%d", i);
}
for(i = 0; i < 10; i++) {
```

```
printf("%d: %s\n", i, p[i]); // p[i] same
as *(p+i)
}
return 0;
}
```

Ok, as you're probably thinking, this is where things get completely wacko-jacko. Let's
look at that second **`malloc()`** line and dissect it one piece at a time.
You know that `p` is a pointer to a pointer to a `char`, or, put another way, it's a pointer to a
`char*`. Keep that in mind.
And we know this `char*` is the first of a solid block of 10, because we just **`malloc()`**'d that many before the for loop. With that knowledge, we know that we can use some pointer arithmetic to hop from one to the next. We do this by adding the value of `i` onto the `char**` so
that when we dereference it, we are pointing at the next `char*` in the block. In the first iteration
of the loop `i` is zero, so we're just referring to the first `char*`.
And what do we do with that `char*` once we have it? We point it at the return value of
`malloc()` which will point it at a fresh ready-to-use 30 bytes of memory.
And what do we use that memory for (sheesh, this could go on forever!)--well, we use a variant of **`printf()`** called **`sprintf()`** that writes the result into a string instead of to the console.
And there you have it. Finally, for fun, we print out the results using array notation to access the strings instead of pointer arithmetic.

1.17 Pointers to Functions

You've completely mastered all that pointer stuff, right? I mean, you are the *Pointer*
Master! No, really, I insist!
So, with that in mind, we're going to take the whole pointer and address idea to the next phase and learn a little bit about the machine code that the compiler produces. I know this seems like it has nothing to do with this section, Pointers to Functions, but it's background that will only make you stronger. (Provided, that is, it doesn't kill you first. Admittedly, the chances of death from trying to understand this section are slim, but you might want to read it in a padded room just as a precautionary measure.)
Long ago I mentioned that the compiler takes your C source code and produces *machine code* that the processor can execute. These machine code instructions are small (taking between one and four bytes or memory, typically, with optionally up to, say, 32 bytes of arguments per instruction--these numbers vary depending on the processor in question). This isn't so important
as the fact that these instructions have to be stored somewhere. Guess where.
You thought that was a rhetorical command, but no, I really do want you to guess where,
generally, the instructions are stored.
You have your guess? Good. Is it animal, vegetable, or mineral? Can you fly in it? Is it a rocketship? Yay!
But, cerebral digression aside, yes, you are correct, the instructions are stored in memory, just like variables are stored in memory. Instructions themselves have addresses, and a special variable in the CPU (generally known as a "register" in CPU-lingo) points to the address of the
currently executing instruction.

What what? I said “points to” and “address-of”! Suddenly we have a connection back to pointers and all that...familiar ground again. But what did I just say? I said: instructions are held in addresses, and, therefore, you have have a pointer to a block of instructions. A block of instructions in C is held in a function, and, therefore, you can have a pointer to a function. *Voila*!

Ok, so if you have a function, how do you get the address of the function? Yes, you can use
the **&**, but most people don't. It's similar to the situation with arrays, where the name of the array
without square brackets is a pointer to the first element in the array; the name of the function
without parens is a pointer to the first instruction in the function. That's the easy part.

The hard part is declaring a variable to be of type “pointer to function”. It's hard because the syntax is funky:

```
// declare p as a pointer to a function that takes two int
// parameters, and returns a float:
float (*p)(int, int);
```

Again, note that this is a *declaration of a pointer* to a function. It doesn't yet point to anything in particular. Also notice that you don't have to put dummy parameter names in the declaration of the pointer variable. All right, let's make a function, point to it, and call it:

```
int deliver_fruit(char *address, float speed)
{
printf("Delivering fruit to %s at speed %.2f\n", address, speed);
return 3490;
}
int main(void)
{
int   (*p)(char*,float);   //   declare   a
function pointer variable
p = deliver_fruit; // p now points to the
deliver_fruit() function
deliver_fruit("My  house",  5280.0);  //  a
normal call
p("My  house",  5280.0);  //  the  same  call,
but using the pointer
return 0;
}
```

What the heck good is this? The usual reasons are these:

- You want to change what function is called at runtime.
- You have a big array of data including pointers to functions.
- You don't know the function at compile-time; maybe it's in a shared library that you load a runtime and query to find a function, and that query returns a pointer to the function. I II know this is a bit beyond the scope of the section, but bear with me. know this is a bit beyond the scope of the section, but bear with me.know this is a bit beyond the scope of the section, but bear with me.

For example, long ago a friend of mine and I wrote a program that would simulate a bunch of creatures running around a grid. The creatures each had a `struct` associated with them that held their position, health, and other information. The `struct` also held a pointer to a function that was their behavior, like this:

```
struct creature {
int xpos;
int ypos;
float health;
int (*behavior)(struct useful_data*);
};
```

So for each round of the simulation, we'd walk through the list of creatures and call their behavior function (passing a pointer to a bunch of useful data so the function could see other
creatures, know about itself, etc.) In this way, it was easy to code bugs up as having different
behaviors.

Indeed, I wrote a creature called a “brainwasher” that would, when it got close to another creature, change that creature's behavior pointer to point to the brainwasher's behavior code! Of course, it didn't take long before they were all brainwashers, and then starved and cannibalized
themselves to death. Let that be a lesson to you.

1.18 Variable Argument Lists

Ever wonder, in your spare time, while you lay awake at night thinking about the C Programming Language, how functions like **`printf()`** and **`scanf()`** seem to take an arbitrary
number of arguments and other functions take a specific number? How do you even write a
function prototype for a function that takes a variable number of arguments?

(Don't get confused over terminology here--we're not talking about variables. In this case, “variable” retains its usual boring old meaning of “an arbitrary number of”.)

Well, there are some little tricks that have been set up for you in this case. Remember how

all the arguments are pushed onto the stack when passed to a function? Well, some macros have
been set up to help you walk along the stack and pull arguments off one at a time. In this way,
you don't need to know at compile-time what the argument list will look like--you just need to
know how to parse it.
For instance, let's write a function that averages an arbitrary number of positive numbers.
We can pull numbers off the stack one at a time and average them all, but we need to know
when to stop pulling stuff off the stack. One way to do this is to have a sentinel value that you
watch for--when you hit it, you stop. Another way is to put some kind of information in the
mandatory first argument. Let's do option B and put the count of the number of arguments to be
averaged in as the first argument to the function.
Here's the prototype for the function--this is how we declare a variable argument list. The
first argument (at least) must be specified, but that's all:

```
      float average(int count, ...);
```

It's the magical “...” that does it, see? This lets the compiler know that there can be more arguments after the first one, but doesn'r require you to say what they are. So *this* is how we are
able to pass many or few (but at least one, the first argument) arguments to the function.
But if we don't have names for the variables specified in the function header, how do we
use them in the function? Well, aren't we the demanding ones, actually wanting to *use* our

function! Ok, I'll tell you!

There is a special type declared in the header *stdarg.h* called `va_list`. It holds data
about the stack and about what arguments have been parsed off so far. But first you have to tell
it where the stack for this function starts, and fortunately we have a variable right there at the
beginning of our **`average()`** function: *a*.

We operate on our `va_list` using a number of preprocessor macros (which are like
mini-functions if you haven't yet read the section on macros.) First of all, we use **`va_start()`**
to tell our `va_list` where the stack starts. Then we use **`va_arg()`** repeatedly to pull arguments
off the stack. And finally we use **`va_end()`** to tell our `va_list` that we're done with it.

(The language specification says we *must* call **`va_end()`**, and we must call it in the same
function from which we called **`va_start()`**. This allows the compiler to do any cleanup that is
necessary, and keeps the Vararg Police from knocking on your door.

So an example! Let's write that **`average()`** function. Remember: **`va_start()`**,
`va_arg()`, **`va_arg()`**, **`va_arg()`**, etc., and then **`va_end()`**!

```
float average(int count, ...)
{
float ave = 0;
int i;
va_list args; // here's our va_list!
va_start(args, count); // tell it the stack
starts with "count"
// inside the while(), pull int args off
the stack:
for(i = 0; i < count; i++) {
int val = va_arg(args, int); // get next
int argument
ave += (float)val; // cast the value to a
float and add to total
}
va_end(args); // clean this up
return ave / count; // calc and return the
average
}
```

So there you have it. As you see, the **`va_arg()`** macro pulls the next argument off the stack of the given type. So you have to know in advance what type the thing is. We know for our **`average()`** function, all the types are `int`s, so that's ok. But what if they're different types
mixed all together? How do you tell what type is coming next?
Well, if you'll notice, this is exactly what our old friend **`printf()`** does! It knows what
type to call **`va_arg()`** with, since it says so right in the format string.

1.18.1 `vprintf()`

There are a number of functions that helpfully accept a `va_list` as an argument that you
can pass. This enables you to wrap these functions up easily in your own functions that take a
variable number of arguments themselves. For instance:

Assignment: Implement a version of `printf()` called `timestamp_printf()` that works
exactly like `printf()` except it prints the time followed by a newline followed by the data
output specified by the `printf()`-style format string.
Holy cow! At first glance, it looks like you're going to have to implement a clone of
`printf()` just to get a timestamp out in front of it! And `printf()` is, as we say in the industry,
“nontrivial”! See you next year!

Wait, though--wait, wait...there *must* be a way to do it easily, or this author is complete
insane to give you this assignment, and that couldn't be. Fruit! Where is my cheese!?
Blalalauugh!!
Ahem. I'm all right, really, Your Honor. I'm looking into my crystal ball and I'm seeing...a
type `va_list` in your future. In fact, if we took our variable argument list, processed it with
`va_start()` and got our `va_list` back, we could, if such a thing existed, just pass it to an
already-written version of `printf()` that accepted just that thing.
Welcome to the world of `vprintf()`! It does exactly that, by Jove! Here's a lovely
prototype:

```
    int    vprintf(const    char    *format,
va_list args);
```

All righty, so what building blocks do we need for this assignment? The spec says we need
to do something just like **`printf()`**, so our function, like **`printf()`** is going to accept a format
string first, followed by a variable number of arguments, something like this:

```
     int    timestamp_printf(char    *format,
...);
```

But before it prints its stuff, it needs to output a timestamp followed by a newline. The
exact format of the timestamp wasn't specified in the assignment, so I'm going to assume
something in the form of “weekday month day hh:mm:ss year”. By amazing coincidence, a
function exists called **`ctime()`** that returns a string in exactly that format, given the current
system time.
So the plan is to print a timestamp, then take our variable argument list, run it through
`va_start` to get a `va_list` out of it, and pass that `va_list` into **`vprintf()`** and let it work
its already-written **`printf()`** magic. And...GO!

```
#include <stdio.h>
#include <stdarg.h>
#include   <time.h>   //   for   time()   and
ctime();
int timestamp_printf(char *format, ...)
{
va_list args;
time_t system_time;
char *timestr;
int return_value;
system_time = time(NULL); // system time in
seconds since epoch
timestr = ctime(&system_time); // ready-to-
print timestamp
// print the timestamp:
printf("%s",  timestr);  //  timestr  has  a
newline on the end already
// get our va_list:
va_start(args, format);
// call vprintf() with our arg list:
return_value = vprintf(format, args);
//  done  with  list,  so  we  have  to  call
va_end():
va_end(args);
//  since  we  want  to  be  *exactly*  like
printf(), we have saved its
// return value, and we'll pass it on right
here:
return return_value;
}

int main(void)
{
// example call:
timestamp_printf("Brought  to  you  by  the
number %d\n", 3490);
return 0;
}
```

And there you have it! Your own little **`printf()`**-like functionality!

Now, not every function has a “v” in front of the name for processing variable argument
lists, but most notably all the variants of **`printf()`** and **`scanf()`** do, so feel free to use them as
you see fit!
TODO order of operations, arrays of pointers to functions.

Check Your Progress

Q.5: Fill in the blanks

i. The well-known ……………..directive pulls in source from another file.
ii. The………….. is one of the more powerful C preprocessor directives.
iii. There are some *conditionals* that the………. can use to discard blocks of code so that the compiler never sees them.
iv. A function exists called……… that returns a string in exactly that format, given the current system time.

1.19 Answer to Check Your Progress

Ans to Q.1: D

Ans to Q.2: D

Ans to Q.3: A

Ans to Q.4: A

Ans to Q.5: i. #include ii. #define iii. C preprocessor iv. `ctime()`

1.20Model Questions

1. What is Command Line Arguments?
2. What is Multidimensional Arrays? Explain with the help of example.
3. What is Null Pointer?
4. Explain Pointer Arithmetic with an example.
5. What is the most common preprocessor directives?

Unit 11
Standard I/O Library

1.1 Learning Objectives
1.2 Introduction
1.3 fopen()
1.4 freopen()
1.5 fclose()
1.6 printf(), fprintf()
1.7 scanf(), fscanf()
1.8 gets(), fgets()

1.9 getc(), fgetc(), getchar()
1.10 puts(), fputs()
1.11 putc(), fputc(), putchar()
1.12 fseek(), rewind()
1.13 ftell()
1.14 fgetpos(), fsetpos()
1.15 ungetc()
1.16 fread()
1.17 fwrite()
1.18 feof(), ferror(), clearerr()
1.19 perror()
1.20 remove()
1.21 rename()
1.22 tmpfile()
1.23 tmpnam()
1.24 setbuf(), setvbuf()
1.25 fflush()
1.26 Check Your Progress
1.27 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will be able to:

- learn about header file and its use
- learn formatted input function like scanf()
- learn formatted output function like printf()
- describe control string used in printf() and scanf()
- describe unformatted input functions like getch(), getche(), getchar(), gets()
- formatted input functions like putch(), putchar(), puts()

1.2 Introduction

A computer program basically consists of three sections i.e, input, processing, and output. The input section receives data from the environment through some input device like keyboard, mouse, secondary storage etc. The processing section is responsible for calculating the required data or output. Different logic like selective logic and iterative logic are implemented in this section. The last section is the output section.

This section is responsible for providing output in the appropriate manner. C language provides a set of library functions for doing these activities.

These functions are predefined and stored in some file known as header file.

1.3 fopen()

Opens a file for reading or writing.

Prototypes

```
               #include <stdio.h>
     FILE *fopen(const char *path, const
char *mode);
```

Description

The **fopen()** opens a file for reading or writing.

Parameter *path* can be a relative or fully-qualified path and file name to the file in
question.

Paramter *mode* tells **fopen()** how to open the file (reading, writing, or both), and whether
or not it's a binary file. Possible modes are:

r

Open the file for reading (read-only).

w

Open the file for writing (write-only). The file is created if it doesn't exist.

r+

Open the file for reading and writing. The file has to already exist.

w+

Open the file for writing and reading. The file is created if it doesn't already exist.

a

Open the file for append. This is just like opening a file for writing, but it positions the file pointer at the end of the file, so the next write appends to the end. The file is created if it doesn't exist.

a+

Open the file for reading and appending. The file is created if it doesn't exist.

Any of the modes can have the letter “b” appended to the end, as is “wb” (“write binary”), to signify that the file in question is a *binary* file. (“Binary” in this case generally means that the

file contains non-alphanumeric characters that look like garbage to human eyes.) Many systems (like Unix) don't differentiate between binary and non-binary files, so the “b” is extraneous. But if your data is binary, it doesn't hurt to throw the “b” in there, and it might help someone who is trying to port your code to another system.

Return Value

fopen() returns a FILE* that can be used in subsequent file-related calls.

If something goes wrong (e.g. you tried to open a file for read that didn't exist), **fopen()**

will return NULL.

Example

```
int main(void)
{
FILE *fp;
if  ((fp  =  fopen("datafile.dat",  "r"))  ==
NULL) {
printf("Couldn't   open   datafile.dat   for
reading\n");
exit(1);
}
//  fp  is  now  initialized  and  can  be  read
from
return 0;
}
```

See Also
fclose()
freopen()

1.4 freopen()

Reopen an existing `FILE*`, associating it with a new path

Prototypes

```
#include <stdio.h>
FILE  *freopen(const  char  *filename,  const
char *mode, FILE *stream);
```

Description

Let's say you have an existing FILE* stream that's already open, but you want it to suddenly use a different file than the one it's using. You can use **freopen()** to “re-open” the stream with a new file.

Why on Earth would you ever want to do that? Well, the most common reason would be if you had a program that normally would read from *stdin*, but instead you wanted it to read from a file. Instead of changing all your **scanf()**s to **fscanf()**s, you could simply reopen *stdin* on the file you wanted to read from.

Another usage that is allowed on some systems is that you can pass NULL for *filename*, and specify a new *mode* for *stream*. So you could change a file from “r+” (read and write) to just “r” (read), for instance. It's implementation dependent which modes can be changed.

When you call **freopen()**, the old *stream* is closed. Otherwise, the function behaves just like the standard **fopen()**.

Return Value

freopen() returns *stream* if all goes well.

If something goes wrong (e.g. you tried to open a file for read that didn't exist), **fopen()**
will return NULL.

Example

```
#include <stdio.h>
int main(void)
{
int i, i2;
scanf("%d", &i); // read i from stdin
// now change stdin to refer to a file
instead of the keyboard
freopen("someints.txt", "r", stdin);
scanf("%d", &i2); // now this reads from
the file "someints.txt"
printf("Hello, world!\n"); // print to the
screen
// change stdout to go to a file instead of
the terminal:
freopen("output.txt", "w", stdout);
printf("This goes to the file
\"output.txt\"\n");
// this is allowed on some systems--you can
change the mode of a file:
freopen(NULL, "wb", stdout); // change to
"wb" instead of "w"
return 0;
}
```

See Also

fclose()

fopen()

1.5 fclose()

The opposite of **`fopen()`**--closes a file when you're done with it so that it frees system resources.

Prototypes

#include <stdio.h>

int fclose(FILE *stream);

Description

When you open a file, the system sets aside some resources to maintain information about that open file. Usually it can only open so many files at once. In any case, the Right Thing to do is to close your files when you're done using them so that the system resources are freed.

Also, you might not find that all the information that you've written to the file has actually
been written to disk until the file is closed. (You can force this with a call to **fflush()**.)
When your program exits normally, it closes all open files for you. Lots of times, though, you'll have a long-running program, and it'd be better to close the files before then. In any case, not closing a file you've opened makes you look bad. So, remember to **fclose()** your file when you're done with it!

Return Value

On success, 0 is returned. Typically no one checks for this. On error EOF is returned.
Typically no one checks for this, either.

Example

```
FILE *fp;
fp = fopen("spoonDB.dat", r"); // (you
should error-check this)
sort_spoon_database(fp);
fclose(fp); // pretty simple, huh.
```

See Also

`fopen()`

1.6 printf(), fprintf()

Print a formatted string to the console or to a file.

Prototypes

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

Description

These functions print formatted strings to a file (that is, a `FILE*` you likely got from **`fopen()`**), or to the console (which is usually itself just a special file, right?)

The **`printf()`** function is legendary as being one of the most flexible outputting systems ever devisied. It can also get a bit freaky here or there, most notably in the *`format`* string. We'll take it a step at a time here.

The easiest way to look at the format string is that it will print everything in the string as-is, *unless* a character has a percent sign (`%`) in front of it. That's when the magic happens: the next argument in the **`printf()`** argument list is printed in the way described by the percent code.

Here are the most common percent codes:

%d

Print the next argument as a signed decimal number, like 3490. The argument printed
this way should be an int.

%f

Print the next argument as a signed floating point number, like 3.14159. The
argument printed this way should be a float.

%c

Print the next argument as a character, like 'B'. The argument printed this way should
be a char.

%s

Print the next argument as a string, like "Did you remember your mittens?".
The argument printed this way should be a char* or char[].

%%

No arguments are converted, and a plain old run-of-the-mill percent sign is printed.
This is how you print a '%' using **printf)()**.

So those are the basics. I'll give you some more of the percent codes in a bit, but let's get some more breadth before then. There's actually a lot more that you can specify in there after the percent sign.
For one thing, you can put a field width in there--this is a number that tells **`printf()`** how
many spaces to put on one side or the other of the value you're printing. That helps you line
things up in nice columns. If the number is negative, the result becomes left-justified instead of
right-justified. Example:

```
printf("%10d", x); /* prints X on the right
side of the 10-space field */
printf("%-10d", x); /* prints X on the left
side of the 10-space field */
```

If you don't know the field width in advance, you can use a little kung-foo to get it from the argument list just before the argument itself. Do this by placing your seat and tray tables in the fully upright position. The seatbelt is fastened by placing the--**cough**. I seem to have been doing way too much flying lately. Ignoring that useless fact completely, you can specify a dynamic field width by putting a `*` in for the width. If you are not willing or able to perform this
task, please notify a flight attendant and we will reseat you.

```
int width = 12;
int value = 3490;
printf("%*d\n", width, value);
```

You can also put a “0” in front of the number if you want it to be padded with zeros:

```
int x = 17;
printf("%05d", x); /* "00017" */
```

When it comes to floating point, you can also specify how many decimal places to print by
making a field width of the form “`x.y`” where `x` is the field width (you can leave this off if you
want it to be just wide enough) and `y` is the number of digits past the decimal point to print:

```
float f = 3.1415926535;
printf("%.2f", f); /* "3.14" */
printf("%7.3f",  f);  /*  "  3.141"  <--  7
spaces across */
```

Ok, those above are definitely the most common uses of **`printf()`**, but there are still more
modifiers you can put in after the percent and before the field width:

0

This was already mentioned above. It pads the spaces before a number with zeros, e.g.
"%05d".

-

This was also already mentioned above. It causes the value to be left-justified in the
field, e.g. "%-5d".

' ' (space)

This prints a blank space before a positive number, so that it will line up in a column
along with negative numbers (which have a negative sign in front of them). "% d".

`+`

Always puts a + sign in front of a number that you print so that it will line up in a
column along with negative numbers (which have a negative sign in front of them).
"%+d".

#

This causes the output to be printed in a different form than normal. The results vary
based on the specifier used, but generally, hexidecimal output ("%x") gets a "0x"
prepended to the output, and octal output ("%o") gets a "0" prepended to it. These
are, if you'll notice, how such numbers are represented in C source. Additionally,
floating point numbers, when printed with this # modified, will print a trailing decimal
point even if the number has no fractional part. Example: "%#x".

Now, I know earlier I promised the rest of the format specifiers...so ok, here they are:

`%i`

Just like `%d`, above.

%o

Prints the integer number out in octal format. Octal is a base-eight number
representation scheme invented on the planet Krylon where all the inhabitants have
only eight fingers.

%u

Just like %d, but works on unsigned ints, instead of ints.

%x or %X

Prints the unsigned int argument in hexidecimal (base-16) format. This is for
people with 16 fingers, or people who are simply addicted hex, like you should
be. Just try it! "%x" prints the hex digits in lowercase, while "%X" prints them in
uppercase.

%F

Just like “%f”, except any string-based results (which can happen for numbers like
infinity) are printed in uppercase.

%e or %E

Prints the float argument in exponential (scientific) notation. This is your classic
form similar to “three times 10 to the 8th power”, except printed in text form: “3e8”.
(You see, the “e” is read “times 10 to the”.) If you use the "%E" specifier, the the
exponent “e” is written in uppercase, a la “3E8”.

%g or %G

Another way of printing doubles. In this case the precision you specific tells it how
many significant figures to print.

%p

Prints a pointer type out in hex representation. In other words, the address that the
pointer is pointing to is printed. (Not the value in the address, but the address number
itself.)

%n

This specifier is cool and different, and rarely needed. It doesn't actually print anything, but stores the number of characters printed so far in the next pointer argument in the list.

```
int numChars;
float a = 3.14159;
int b = 3490;
printf("%f %d%n\n", a, b, &numChars);
printf("The    above    line    contains    %d
characters.\n", numChars);
```

The above example will print out the values of *a* and *b*, and then store the number of characters printed so far into the variable *numChars*. The next call to **printf()** prints out that result.

So let's recap what we have here. We have a format string in the form:

`"%[modifier][fieldwidth][.precision][length modifier][formatspecifier]"`

Modifier is like the `"-"` for left justification, the field width is how wide a space to print the result in, the precision is, for `float`s, how many decimal places to print, and the format specifier is like `%d`.

That wraps it up, except what's this "lengthmodifier" I put up there?! Yes, just when you thought things were under control, I had to add something else on there. Basically, it's to tell **`printf()`** in more detail what size the arguments are. For instance, `char`, `short`, `int`, and `long` are all integer types, but they all use a different number of bytes of memory, so you can't use plain old "`%d`" for *all* of them, right? How can **`printf()`** tell the difference?

The answer is that you tell it explicitly using another optional letter (the length modifier, this) before the type specifier. If you omit it, then the basic types are assumed (like `%d` is for `int`, and `%f` is for `float`).

Here are the format specifiers:

`h`

Integer referred to is a `short` integer, e.g. "`%hd`" is a `short` and "`%hu`" is an
`unsigned short`.

`l` ("ell")

Integer referred to is a `long` integer, e.g. "`%ld`" is a `long` and "`%lu`" is an `unsigned`
`long`.

`hh`

Integer referred to is a `char` integer, e.g. "`%hhd`" is a `char` and "`%hhu`" is an
`unsigned char`.

`ll` ("ell ell")

Integer referred to is a `long long` integer, e.g. "`%lld`" is a `long long` and "`%llu`"
is an `unsigned long long`.

I know it's hard to believe, but there might be *still more* format and length specifiers on
your system. Check your manual for more information.

Return Value

Example

```
int a = 100;
float b = 2.717;
char *c = "beej!";
char d = 'X';
int e = 5;
printf("%d", a); /* "100" */
printf("%f", b); /* "2.717000" */
printf("%s", c); /* "beej!" */
printf("%c", d); /* "X" */
printf("110%%"); /* "110%" */
printf("%10d\n", a); /* " 100" */
printf("%-10d\n", a); /* "100 " */
printf("%*d\n", e, a); /* " 100" */
printf("%.2f\n", b); /* "2.71" */
printf("%hhd\n", c); /* "88" <-- ASCII code
for 'X' */
printf("%5d %5.2f %c\n", a, b, d); /* " 100
2.71 X" */
```

See Also

`sprintf()`, **`vprintf()`**, **`vfprintf()`**, **`vsprintf()`**

1.7 scanf(), fscanf()

Read formatted string, character, or numeric data from the console or from a file.

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

Description

The **scanf()** family of functions reads data from the console or from a FILE stream, parses it, and stores the results away in variables you provide in the argument list.

The format string is very similar to that in **printf()** in that you can tell it to read a "%d", for instance for an int. But it also has additional capabilities, most notably that it can eat up other characters in the input that you specify in the format string.

But let's start simple, and look at the most basic usage first before plunging into the depths of the function. We'll start by reading an int from the keyboard:

```
int a;
scanf("%d", &a);
```

`scanf()` obviously needs a pointer to the variable if it is going to change the variable
itself, so we use the address-of operator to get the pointer.

In this case, **`scanf()`** walks down the format string, finds a "`%d`", and then knows it needs to read an integer and store it in the next variable in the argument list, *`a`*.

Here are some of the other percent-codes you can put in the format string:

`%d`

Reads an integer to be stored in an `int`. This integer can be signed.

`%f` (`%e`, `%E`, and `%g` are equivalent)

Reads a floating point number, to be stored in a `float`.

`%s`

Reads a string. This will stop on the first whitespace character reached, or at the specified field width (e.g. "%10s"), whichever comes first.

And here are some more codes, except these don't tend to be used as often. You, of course, may use them as often as you wish!

%u

Reads an unsigned integer to be stored in an unsigned int.

%x (%X is equivalent) Reads an unsigned hexidecimal integer to be stored in an unsigned int.

%o

Reads an unsigned octal integer to be stored in an unsigned int.

%i

Like %d, except you can preface the input with “0x” if it's a hex number, or “0” if it's an octal number.

`%c`

Reads in a character to be stored in a `char`. If you specify a field width (e.g. “`%12c`”,
it will read that many characters, so make sure you have an array that large to hold them.

`%p`

Reads in a pointer to be stored in a `void*`. The format of this pointer should be the same as that which is outputted with **`printf()`** and the “`%p`” format specifier.

`%n`

Reads nothing, but will store the number of characters processed so far into the next `int` parameter in the argument list.

`%%`

Matches a literal percent sign. No conversion of parameters is done. This is simply
how you get a standalone percent sign in your string without **`scanf()`** trying to do something with it.

`%[`

This is about the weirdest format specifier there is. It allows you to specify a set of characters to be stored away (likely in an array of `char`s). Conversion stops when a character that is not in the set is matched.
For example, `%[0-9]` means “match all numbers zero through nine.” And `%[AD-G34]`means “match A, D through G, 3, or 4”.
Now, to convolute matters, you can tell **`scanf()`** to match characters that are *not* in the set by putting a caret (^) directly

after the `%[` and following it with the set, like this: `%[^A-C]`, which means “match all characters that are *not* A through C.”

To match a close square bracket, make it the first character in the set, like this:

`%[]A-C]` or `%[^]A-C]`. (I added the “`A-C`” just so it was clear that the “`]`” was first in the set.)

To match a hyphen, make it the last character in the set: `%[A-C-]`.

So if we wanted to match all letters *except* “%”, “^”, “]”, “B”, “C”, “D”, “E”, and “-”, we could use this format string: `%[^]%^B-E-]`.

So those are the basics! Phew! There's a lot of stuff to know, but, like I said, a few of these format specifiers are common, and the others are pretty rare.

Got it? Now we can go onto the next--no wait! There's more! Yes, still more to know about **scanf()**. Does it never end? Try to imagine how I feel writing about it!

So you know that “%d” stores into an int. But how do you store into a long, short, or double?

Well, like in **printf()**, you can add a modifier before the type specifier to tell **scanf()** that you have a longer or shorter type. The following is a table of the possible modifiers:

`h`

The value to be parsed is a `short int` or `short unsigned`. Example: `%hd` or `%hu`.

`l`

The value to be parsed is a `long int` or `long unsigned`, or `double` (for `%f` conversions.) Example: `%ld`, `%lu`, or `%lf`.

`L`

The value to be parsed is a `long long` for integer types or `long double` for `float` types. Example: `%Ld`, `%Lu`, or `%Lf`.

`*`

Tells **`scanf()`** do to the conversion specified, but not store it anywhere. It simply
discards the data as it reads it. This is what you use if you want **`scanf()`** to eat some data but you don't want to store it anywhere; you don't give **`scanf()`** an argument for
this conversion. Example: `%*d`.

Return Value

`scanf()` returns the number of items assigned into variables. Since assignment into variables stops when given invalid input for a certain format specifier, this can tell you if you've input all your data correctly.

Also, **`scanf()`** returns `EOF` on end-of-file.

Example

```
int a;
long int b;
unsigned int c;
float d;
double e;
long double f;
char s[100];
scanf("%d", &a); // store an int
scanf(" %d", &a); // eat any whitespace,
then store an int
scanf("%s", s); // store a string
scanf("%Lf", &f); // store a long double
// store an unsigned, read all whitespace,
then store a long int:
```

```
scanf("%u %ld", &c, &b);
// store an int, read whitespace, read
"blendo", read whitespace,
// and store a float:
scanf("%d blendo %f", &a, &d);
// read all whitespace, then store all
characters up to a newline
scanf(" %[^\n]", s);
// store a float, read (and ignore) an int,
then store a double:
scanf("%f %*d %lf", &d, &e);
// store 10 characters:
scanf("%10c", s);
```

See Also

`sscanf()`, **`vscanf()`**, **`vsscanf()`**, **`vfscanf()`**

1.8 gets(), fgets()

Read a string from console or file

Prototypes

```
#include <stdio.h>
char *fgets(char *s, int size, FILE
*stream);
char *gets(char *s);
```

Description

These are functions that will retrieve a newline-terminated string from the console or a file.

In other normal words, it reads a line of text. The behavior is slightly different, and, as such, so

is the usage. For instance, here is the usage of **`gets()`**:

Don't use **`gets()`**.

Admittedly, rationale would be useful, yes? For one thing, **`gets()`** doesn't allow you to
specify the length of the buffer to store the string in. This would allow people to keep entering
data past the end of your buffer, and believe me, this would be Bad News.
I was going to add another reason, but that's basically the primary and only reason not to
use **`gets()`**. As you might suspect, **`fgets()`** allows you to specify a maximum string length.
One difference here between the two functions: **`gets()`** will devour and throw away
the newline at the end of the line, while **`fgets()`** will store it at the end of your string (space
permitting).
Here's an example of using **`fgets()`** from the console, making it behave more like
`gets()`:

```
char s[100];
gets(s); // read a line (from stdin)
fgets(s, sizeof(s), stdin); // read a line
from stdin
```

In this case, the **sizeof()** operator gives us the total size of the array in bytes, and since a
char is a byte, it conveniently gives us the total size of the array.
Of course, like I keep saying, the string returned from **fgets()** probably has a newline at
the end that you might not want. You can write a short function to chop the newline off, like so:

```
char *remove_newline(char *s)
```

```
{
int len = strlen(s);
if (len > 0 && s[len-1] == '\n') // if
there's a newline
s[len-1] = '\0'; // truncate the string
return s;
}
```

So, in summary, use **fgets()** to read a line of text from the keyboard or a file, and don't
use **gets()**.

Return Value

Both **fgets()** and **fgets()** return a pointer to the string passed.

On error or end-of-file, the functions return NULL.

Example

```
char s[100];

gets(s); // read from standard input (don't use
this--use fgets()!)
fgets(s, sizeof(s), stdin); // read 100 bytes
from standard input
fp = fopen("datafile.dat", "r"); // (you should
error-check this)
fgets(s, 100, fp); // read 100 bytes from the
file datafile.dat
fclose(fp);
fgets(s, 20, stdin); // read a maximum of 20
bytes from stdin
```

See Also

`getc()`, **`fgetc()`**, **`getchar()`**, **`puts()`**, **`fputs()`**, **`ungetc()`**

1.9 getc(), fgetc(), getchar()

Get a single character from the console or from a file.
Prototypes
#include <stdio.h>

int getc(FILE *stream);

int fgetc(FILE *stream);

int getchar(void);

Description

All of these functions in one way or another, read a single character from the console or
from a FILE. The differences are fairly minor, and here are the descriptions:

getc() returns a character from the specified FILE. From a usage standpoint, it's equivalent to the same **fgetc()** call, and **fgetc()** is a little more common to see. Only the implementation of the two functions differs.

fgetc() returns a character from the specified FILE. From a usage standpoint, it's equivalent to the same **getc()** call, except that **fgetc()** is a little more common to see. Only the implementation of the two functions differs.

Yes, I cheated and used cut-n-paste to do that last paragraph.

getchar() returns a character from *stdin*. In fact, it's the same as calling getc(stdin).

Return Value

All three functions return the unsigned char that they read, except it's cast to an int.

If end-of-file or an error is encountered, all three functions return EOF.

Example

```
// read all characters from a file,
outputting only the letter 'b's
```

```
// it finds in the file
#include <stdio.h>
int main(void)
{
FILE *fp;
int c;
fp = fopen("datafile.txt", "r"); // error
check this!
// this while-statement assigns into c, and
then checks against EOF:
while((c = fgetc(fp)) != EOF) {
if (c == 'b') {
putchar(c);
}
}
fclose(fp);
return 0;
}
```

1.10 puts(), fputs()

Write a string to the console or to a file.

Prototypes

```
#include <stdio.h>
int puts(const char *s);
int fputs(const char *s, FILE *stream);
```

Description

Both these functions output a NUL-terminated string. **puts()** outputs to the console, while
fputs() allows you to specify the file for output.

Return Value

Both functions return non-negative on success, or EOF on error.

Example

```
// read strings from the console and save
them in a file
#include <stdio.h>
int main(void)
{
FILE *fp;
char s[100];
fp = fopen("datafile.txt", "w"); // error
check this!
while(fgets(s, sizeof(s), stdin) != NULL) {
// read a string
fputs(s, fp); // write it to the file we
opened
}
fclose(fp);
return 0;
}
```

1.11 putc(), fputc(), putchar()

Write a single character to the console or to a file.

Prototypes

```
#include <stdio.h>
int putc(int c, FILE *stream);
int fputc(int c, FILE *stream);
```

int putchar(int c);

Description

All three functions output a single character, either to the console or to a FILE.

putc() takes a character argument, and outputs it to the specified FILE. **fputc()** does
exactly the same thing, and differs from **putc()** in implementation only. Most people use
fputc().

putchar() writes the character to the console, and is the same as calling putc(c,
stdout).

Return Value

All three functions return the character written on success, or EOF on error.

Example

```
// print the alphabet
#include <stdio.h>
int main(void)
{
char i;
for(i = 'A'; i <= 'Z'; i++)
putchar(i);
putchar('\n'); // put a newline at the end
to make it pretty
return 0;
}
```

1.12 fseek(), rewind()

Position the file pointer in anticipition of the next read or write.

Prototypes

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int
whence);
```

```
void rewind(FILE *stream);
```

Description

When doing reads and writes to a file, the OS keeps track of where you are in the file using
a counter generically known as the file pointer. You can reposition the file pointer to a different
point in the file using the **fseek()** call. Think of it as a way to randomly access you file.

The first argument is the file in question, obviously. *offset* argument is the position that you want to seek to, and *whence* is what that offset is relative to.

Of course, you probably like to think of the offset as being from the beginning of the file. I
mean, "Seek to position 3490, that should be 3490 bytes from the beginning of the file." Well, it
can be, but it doesn't have to be. Imagine the power you're wielding here. Try to command your
enthusiasm.

You can set the value of `whence` to one of three things:

`SEEK_SET`

`offset` is relative to the beginning of the file. This is probably what you had in mind
anyway, and is the most commonly used value for `whence`.

`SEEK_CUR`

`offset` is relative to the current file pointer position. So, in effect, you can say,
"Move to my current position plus 30 bytes," or, "move to my current position minus
20 bytes."

`SEEK_END`

`offset` is relative to the end of the file. Just like `SEEK_SET` except from the other end

of the file. Be sure to use negative values for *`offset`* if you want to back up from the

end of the file, instead of going past the end into oblivion.

Speaking of seeking off the end of the file, can you do it? Sure thing. In fact, you can seek
way off the end and then write a character; the file will be expanded to a size big enough to hold
a bunch of zeros way out to that character.
Now that the complicated function is out of the way, what's this **rewind()** that I briefly
mentioned? It repositions the file pointer at the beginning of the file:

```
fseek(fp, 0, SEEK_SET); // same as rewind()
rewind(fp); // same as fseek(fp, 0, SEEK_SET)
```

Return Value

For **fseek()**, on success zero is returned; -1 is returned on failure.

The call to **rewind()** never fails.

Example

```
fseek(fp, 100, SEEK_SET); // seek to the 100th byte of the file
fseek(fp, -30, SEEK_CUR); // seek backward 30 bytes from the current pos
fseek(fp, -10, SEEK_END); // seek to the 10th byte before the end of file

    fseek(fp, 0, SEEK_SET); // seek to the beginning of the file
    rewind(fp); // seek to the beginning of the file
```

See Also

`ftell()`, **`fgetpos()`**, **`fsetpos()`**

Check Your Progress

Fill in the blanks

Q.1: You can reposition the file pointer to a different point in the file using the…………

Q.2: …………………..writes the character to the console

Q.3: ……………returns a character from the specified FILE

Q.4: ……………returns a character from *stdin*.

Q.5: the ………operator gives us the total size of the array in bytes, and since a char is a byte, it conveniently gives us the total size of the array.

Q.6: ……………allows you to specify a maximum string length.

Q.7: The……….. family of functions reads data from the console or from a FILE stream.

Q.8: You can use ……………to "re-open" the stream with a new file.

Q.9: The ……………..opens a file for reading or writing.

1.13 ftell()

Tells you where a particular file is about to read from or write to.

Prototypes

```
#include <stdio.h>

long ftell(FILE *stream);
```

Description

This function is the opposite of **fseek()**. It tells you where in the file the next file

operation will occur relative to the beginning of the file.

It's useful if you want to remember where you are in the file, **fseek()** somewhere else, and then come back later. You can take the return value from **ftell()** and feed it back into **fseek()** (with *whence* parameter set to SEEK_SET) when you want to return to your previous position.

Return Value

Returns the current offset in the file, or -1 on error.

Example

```
long pos;
// store the current position in variable
"pos":
pos = ftell(fp);
// seek ahead 10 bytes:
fseek(fp, 10, SEEK_CUR);
// do some mysterious writes to the file
do_mysterious_writes_to_file(fp);
// and return to the starting position,
stored in "pos":
fseek(fp, pos, SEEK_SET);
```

See Also

`fseek()`, **`rewind()`**, **`fgetpos()`**, **`fsetpos()`**

1.14 fgetpos(), fsetpos()

Get the current position in a file, or set the current position in a file. Just like **ftell()** and **fseek()** for most systems.

Prototypes

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

Description

These functions are just like **ftell()** and **fseek()**, except instead of counting in bytes, they use an *opaque* data structure to hold positional information about the file. (Opaque, in this case, means you're not supposed to know what the data type is made up of.)

On virtually every system (and certainly every system that I know of), people don't use these functions, using **ftell()** and

fseek() instead. These functions exist just in case your system can't remember file positions as a simple byte offset.

Since the *pos* variable is opaque, you have to assign to it using the **fgetpos()** call itself.

Then you save the value for later and use it to reset the position using **fsetpos()**.

Return Value

Both functions return zero on success, and -1 on error.

Example

```
char s[100];
fpos_t pos;
fgets(s,  sizeof(s),  fp);  //  read  a  line
from the file
fgetpos(fp, &pos); // save the position
fgets(s,  sizeof(s),  fp);  //  read  another
line from the file
fsetpos(fp,  &pos);  //  now  restore  the
position to where we saved
```

See Also

`fseek()`, **`ftell()`**, **`rewind()`**

1.15 ungetc()

Pushes a character back into the input stream.

Prototypes

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

You know how **getc()** reads the next character from a file stream? Well, this is the

opposite of that--it pushes a character back into the file stream so that it will show up again on the very next read from the stream, as if you'd never gotten it from **getc()** in the first place.

Why, in the name of all that is holy would you want to do that? Perhaps you have a stream

of data that you're reading a character at a time, and you won't know to stop reading until you

get a certain character, but you want to be able to read that character again later. You can read

the character, see that it's what you're supposed to stop on, and then **ungetc()** it so it'll show up on the next read.

Yeah, that doesn't happen very often, but there we are.

Here's the catch: the standard only guarantees that you'll be able to push back *one character*. Some implementations might allow you to push back more, but there's really no way to tell and still be portable.

Return Value

On success, **ungetc()** returns the character you passed to it. On failure, it returns EOF.

Example

```
// read a piece of punctuation, then everything after it up to the next
// piece of punctuation. return the punctuation, and store the rest
// in a string
//
// sample input: !foo#bar*baz
// output: return value: '!', s is "foo"
// return value: '#', s is "bar"
// return value: '*', s is "baz"
//
char read_punctstring(FILE *fp, char *s)
```

```
{
char origpunct, c;
origpunct = fgetc(fp);
if (origpunct == EOF) // return EOF on end-
of-file
return EOF;
while(c = fgetc(fp), !ispunct(c) && c !=
EOF) {
*s++ = c; // save it in the string
}
*s = '\0'; // nul-terminate the string!
// if we read punctuation last, ungetc it
so we can fgetc it next
// time:
if (ispunct(c))
ungetc(c, fp);
}
     return origpunct;
}
```

See Also

`fgetc()`

1.16 fread()

Read binary data from a file.

Prototypes

```
#include <stdio.h>
size_t fread(void *p, size_t size, size_t
nmemb, FILE *stream);
```

Description

You might remember that you can call **`fopen()`** with the "`b`" flag in the open mode string

to open the file in “binary” mode. Files open in not-binary (ASCII or text mode) can be read
using standard character-oriented calls like **`fgetc()`** or **`fgets()`**. Files open in binary mode are
typically read using the **`fread()`** function.
All this function does is says, “Hey, read this many things where each thing is a certain
number of bytes, and store the whole mess of them in memory starting at this pointer.”
This can be very useful, believe me, when you want to do something like store 20 `int`s in a
file.

But wait--can't you use **`fprintf()`** with the “`%d`” format specifier to save the `int`s to a
text file and store them that way? Yes, sure. That has the advantage that a human can open the
file and read the numbers. It has the disadvantage that it's slower to convert the numbers from
`int`s to text and that the numbers are likely to take more space in the file. (Remember, an `int` is
likely 4 bytes, but the string “12345678” is 8 bytes.)
So storing the binary data can certainly be more compact and faster to read.
(As for the prototype, what is this `size_t` you see floating around? It's short for “size
type” which is a data type defined to hold the size of something. Great--would I stop beating
around the bush already and give you the straight story?! Ok, `size_t` is probably an `int`.)

Return Value

This function returns the number of items successfully read. If all requested items are read,
the return value will be equal to that of the parameter `nmemb`. If EOF occurs, the return value
will be zero.
To make you confused, it will also return zero if there's an error. You can use the functions
`feof()` or **`ferror()`** to tell which one really happened.

Example

```
// read 10 numbers from a file and store
them in an array
int main(void)
{
int i;
int n[10]
FILE *fp;
fp = fopen("binarynumbers.dat", "rb");
fread(n, sizeof(int), 10, fp); // read 10
ints
fclose(fp);
// print them out:
for(i = 0; i < 10; i++)
printf("n[%d] == %d\n", i, n[i]);
return 0;
}
```

See Also

`fopen()`, **`fwrite()`**, **`feof()`**, **`ferror()`**

1.17 fwrite()

Write binary data to a file.

Prototypes

```
#include <stdio.h>
size_t  fwrite(const  void  *p,  size_t  size,
size_t nmemb, FILE *stream);
```

Description

This is the counterpart to the **fread()** function. It writes blocks of binary data to disk. For
a description of what this means, see the entry for **fread()**.

Return Value

fwrite() returns the number of items successfully written, which should hopefully be
nmemb that you passed in. It'll return zero on error.

Example

```
// save 10 random numbers to a file
int main(void)
{
int i;
int r[10];
FILE *fp;
// populate the array with random numbers:
for(i = 0; i < 10; i++) {
r[i] = rand();
}
// save the random numbers (10 ints) to the
file
fp = fopen("binaryfile.dat", "wb");
fwrite(r, sizeof(int), 10, fp); // write 10
ints
fclose(fp);
return 0;
}
```

See Also

`fopen()`, `fread()`

1.18 feof(), ferror(), clearerr()

Determine if a file has reached end-of-file or if an error has occurred.

Prototypes

```
#include <stdio.h>

int feof(FILE *stream);

int ferror(FILE *stream);

void clearerr(FILE *stream);
```

Description

Each FILE* that you use to read and write data from and to a file contains flags that the system sets when certain events occur. If you get an error, it sets the error flag; if you reach the end of the file during a read, it sets the EOF flag. Pretty simple really.

The functions **feof()** and **ferror()** give you a simple way to test these flags: they'll return non-zero (true) if they're set.

Once the flags are set for a particular stream, they stay that way until you call **`clearerr()`**
to clear them.

Return Value

feof() and **ferror()** return non-zero (true) if the file has reached EOF or there has been an error, respectively.

Example

```
// read binary data, checking for eof or
error
int main(void)
{
int a;
FILE *fp;
fp = fopen("binaryints.dat", "rb");
// read single ints at a time, stopping on
EOF or error:
while(fread(&a,   sizeof(int),   1,   fp),
!feof(fp) && !ferror(fp)) {
printf("I read %d\n", a);
}
if (feof(fp))
printf("End of file was reached.\n");
if (ferror(fp))
printf("An error occurred.\n");
fclose(fp);
return 0;
}
```

See Also

`fopen()`, **`fread()`**

1.19 perror()

Print the last error message to *`stderr`*

Prototypes

```
#include <stdio.h>
#include <errno.h> // only if you want to
directly use the "errno" var
void perror(const char *s);
```

Description

Many functions, when they encounter an error condition for whatever reason, will set a global variable called *`errno`* for you. *`errno`* is just an interger representing a unique error.

But to you, the user, some number isn't generally very useful. For this reason, you can call **`perror()`** after an error occurs to print what error has actually happened in a nice human-readable string.

And to help you along, you can pass a parameter, `s`, that will be prepended to the error
string for you.

One more clever trick you can do is check the value of the `errno` (you have to include `errno.h` to see it) for specific errors and have your code do different things. Perhaps you want to ignore certain errors but not others, for instance.

The catch is that different systems define different values for `errno`, so it's not very portable. The standard only defines a few math-related values, and not others. You'll have to check your local man-pages for what works on your system.

Return Value

Returns nothing at all! Sorry!

Example

fseek() returns -1 on error, and sets *errno*, so let's use it. Seeking on *stdin* makes no sense, so it should generate an error:

```
#include <stdio.h>
#include <errno.h> // must include this to
see "errno" in this example
int main(void)
{
if (fseek(stdin, 10L, SEEK_SET) < 0)
perror("fseek");
fclose(stdin); // stop using this stream
if (fseek(stdin, 20L, SEEK_CUR) < 0) {
// specifically check errno to see what
kind of
// error happened...this works on Linux,
but your
// mileage may vary on other systems!
if (errno == EBADF) {
```

```
perror("fseek again, EBADF");
} else {
perror("fseek again");
}
}
     return 0;
     }
```

And the output is:
fseek: Illegal seek

fseek again, EBADF: Bad file descriptor

See Also

feof(), ferror(), clearerr()

1.20 remove()

Delete a file

Prototypes

```
#include <stdio.h>
int remove(const char *filename);
```

Description

Removes the specified file from the filesystem. It just deletes it. Nothing magical. Simply
call this function and sacrifice a small chicken and the requested file will be deleted.

Return Value

Returns zero on success, and -1 on error, setting *errno*.

Example

```
char *filename = "/home/beej/evidence.txt";
remove(filename);
remove("/disks/d/Windows/system.ini");
```

See Also

rename()

1.21 rename()

Renames a file and optionally moves it to a new location

Prototypes

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

Renames the file `old` to name `new`. Use this function if you're tired of the old name of the file, and you are ready for a change. Sometimes simply renaming your files makes them feel new again, and could save you money over just getting all new files!

One other cool thing you can do with this function is actually move a file from one directory to another by specifying a different path for the new name.

Return Value

Returns zero on success, and `-1` on error, setting `errno`.

Example

```
rename("foo", "bar"); // changes the name of the file "foo" to "bar"
// the following moves the file "evidence.txt" from "/tmp" to
// "/home/beej", and also renames it to "nothing.txt":
rename("/tmp/evidence.txt", "/home/beej/nothing.txt");
```

See Also

remove()

1.22 tmpfile()

Create a temporary file

Prototypes

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

This is a nifty little function that will create and open a temporary file for you, and will
return a `FILE*` to it that you can use. The file is opened with mode "`r+b`", so it's suitable for
reading, writing, and binary data.

By using a little magic, the temp file is automatically deleted when it is **`close()`**'d or when
your program exits. (Specifically, **`tmpfile()`** unlinks the file right after it opens it. If you don't
know what that means, it won't affect your **`tmpfile()`** skill, but hey, be curious! It's for your
own good!)

Return Value

This function returns an open `FILE*` on success, or `NULL` on failure.

Example

```
#include <stdio.h>
int main(void)
{
FILE *temp;
char s[128];
temp = tmpfile();
fprintf(temp,   "What   is   the   frequency,
Alexander?\n");
rewind(temp); // back to the beginning
fscanf(temp, "%s", s); // read it back out
fclose(temp);   //   close   (and   magically
delete)
return 0;
}
```

See Also

`fopen()`

`fclose()`

`tmpnam()`

1.23 tmpnam()

Generate a unique name for a temporary file

Prototypes

```
#include <stdio.h>
char *tmpnam(char *s);
```

Description

This function takes a good hard look at the existing files on your system, and comes up
with a unique name for a new file that is suitable for temporary file usage.
Let's say you have a program that needs to store off some data for a short time so you
create a temporary file for the data, to be deleted when the program is done running. Now
imagine that you called this file *`foo.txt`*. This is all well and good, except what if a user
already has a file called *`foo.txt`* in the directory that you ran your program from? You'd
overwrite their file, and they'd be unhappy and stalk you forever. And you wouldn't want that,
now would you?
Ok, so you get wise, and you decide to put the file in *`/tmp`* so that it won't overwrite any

important content. But wait! What if some other user is running your program at the same time
and they both want to use that filename? Or what if some other program has already created that
file?

See, all of these scary problems can be completely avoided if you just use **`tmpnam()`** to get
a safe-ready-to-use filename.
So how do you use it? There are two amazing ways. One, you can declare an array (or
`malloc()` it--whatever) that is big enough to hold the temporary file name. How big is that?
Fortunately there has been a macro defined for you, `L_tmpnam`, which is how big the array must
be.
And the second way: just pass `NULL` for the filename. **`tmpnam()`** will store the temporary
name in a static array and return a pointer to that. Subsequent calls with a `NULL` argument will
overwrite the static array, so be sure you're done using it before you call **`tmpnam()`** again.
Again, this function just makes a file name for you. It's up to you to later **`fopen()`** the file
and use it.
One more note: some compilers warn against using **`tmpnam()`** since some systems have
better functions (like the Unix function **`mkstemp()`**.) You might want to check your local
documentation to see if there's a better option. Linux documentation goes so far as to say,

“Never use this function. Use **`mkstemp()`** instead.”

I, however, am going to be a jerk and not talk about **`mkstemp()`** because it's not in the
standard I'm writing about. Nyaah.

Return Value

Returns a pointer to the temporary file name. This is either a pointer to the string you
passed in, or a pointer to internal static storage if you passed in NULL. On error (like it can't find
any temporary name that is unique), **tmpnam()** returns NULL.

Example

```
char filename[L_tmpnam];
char *another_filename;
if (tmpnam(filename) != NULL)
printf("We got a temp file named: \"%s\"\n", filename);

else
printf("Something went wrong, and we got nothing!\n");
another_filename = tmpnam(NULL);
printf("We got another temp file named: \"%s\"\n", another_filename);
printf("And we didn't error check it because we're too lazy!\n");
```

On my Linux system, this generates the following output:

```
We got a temp file named: "/tmp/filew9PMuZ"
We got another temp file named: "/tmp/fileOwrgPO"
And we didn't error check it because we're too lazy!
```

See Also

`fopen()`
`tmpfile()`

1.24 setbuf(), setvbuf()

Configure buffering for standard I/O operations

Prototypes

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description

Now brace yourself because this might come as a bit of a surprise to you: when you
`printf()` or **`fprintf()`** or use any I/O functions like that, *it does not normally work*
immediately. For the sake of efficiency, and to irritate you, the I/O on a `FILE*` stream is
buffered away safely until certain conditions are met, and only then is the actual I/O performed.
The functions **`setbuf()`** and **`setvbuf()`** allow you to change those conditions and the
buffering behavior.

So what are the different buffering behaviors? The biggest is called "full buffering",
wherein all I/O is stored in a big buffer until it is full, and then it is dumped out to disk (or
whatever the file is). The next biggest is called "line buffering"; with line buffering, I/O is
stored up a line at a time (until a newline (`'\n'`) character is encountered) and then that line
is processed. Finally, we have "unbuffered", which means I/O is processed immediately with
every standard I/O call.

You might have seen and wondered why you could call **putchar()** time and time again
and not see any output until you called `putchar('\n')`; that's right--*`stdout`* is line-buffered!
Since **`setbuf()`** is just a simplified version of **`setvbuf()`**, we'll talk about **`setvbuf()`**
first.
The *`stream`* is the `FILE*` you wish to modify. The standard says you *must* make your call
to **`setvbuf()`** *before* any I/O operation is performed on the stream, or else by then it might be
too late.
The next argument, *`buf`* allows you to make your own buffer space (using **`malloc()`** or
just a `char` array) to use for buffering. If you don't care to do this, just set *`buf`* to `NULL`.
Now we get to the real meat of the function: *`mode`* allows you to choose what kind of
buffering you want to use on this *`stream`*. Set it to one of the following:

_IOFBF

stream will be fully buffered.

_IOLBF

stream will be line buffered.

_IONBF

stream will be unbuffered.

Finally, the *size* argument is the size of the array you passed in for *buf*...unless you passed NULL for *buf*, in which case it will resize the existing buffer to the size you specify.

Now what about this lesser function **setbuf()**? It's just like calling **setvbuf()** with some specific parameters, except **setbuf()** doesn't return a value. The following example shows the equivalency:

```
// these are the same:
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ); //
fully buffered
     // and these are the same:
     setbuf(stream, NULL);
setvbuf(stream, NULL, _IONBF, BUFSIZ); //
unbuffered
```

Return Value

setvbuf() returns zero on success, and nonzero on failure. **setbuf()** has no return
value.

Example

```
FILE *fp;
char lineBuf[1024];
fp = fopen("somefile.txt", "r");
setvbuf(fp, lineBuf, _IOLBF, 1024); // set
to line buffering
// ...
fclose(fp);
fp = fopen("another.dat", "rb");
setbuf(fp, NULL); // set to unbuffered
// ...
fclose(fp);
```

See Also

`fflush()`

1.25 fflush()

Process all buffered I/O for a stream right now

Prototypes

```
#include <stdio.h>

int fflush(FILE *stream);
```

Description

When you do standard I/O, as mentioned in the section on the **setvbuf()** function, it is
usually stored in a buffer until a line has been entered or the buffer is full or the file is closed.
Sometimes, though, you really want the output to happen *right this second*, and not wait around
in the buffer. You can force this to happen by calling **fflush()**.
The advantage to buffering is that the OS doesn't need to hit the disk every time you call
fprintf(). The disadvantage is that if you look at the file on the disk after the **fprintf()**
call, it might not have actually been written to yet. ("I called **fputs()**, but the file is still zero
bytes long! Why?!") In virtually all circumstances, the advantages of buffering outweigh the
disadvantages; for those other circumstances, however, use **fflush()**.

Note that **fflush()** is only designed to work on output streams according to the spec.
What will happen if you try it on an input stream? Use your spooky voice: *who knooooows!*

Return Value

On success, **fflush()** returns zero. If there's an error, it returns EOF and sets the error
condition for the stream (see **ferror()**.)

Example

In this example, we're going to use the carriage return, which is '\r'. This is like newline ('\n'), except that it doesn't move to the next line. It just returns to the front of the current line.

What we're going to do is a little text-based status bar like so many command line programs implement. It'll do a countdown from 10 to 0 printing over itself on the same line.

What is the catch and what does this have to do with **fflush()**? The catch is that the terminal is most likely “line buffered” (see the section on **setvbuf()** for more info), meaning that it won't actually display anything until it prints a newline. But we're not printing newlines; we're just printing carriage returns, so we need a way to force the output to occur even though we're on the same line. Yes, it's **fflush()!**

```
#include <stdio.h>
#include <unistd.h> // for prototype for
sleep()
int main(void)
{
int count;
for(count = 10; count >= 0; count--) {
printf("\rSeconds until launch: "); // lead
with a CR
if (count > 0)
printf("%2d", count);
else
printf("blastoff!\n");
// force output now!!
fflush(stdout);
// the sleep() function is non-standard,
but virtually every
// system implements it--it simply delays
for the specificed
// number of seconds:
sleep(1);
}
return 0;
          }
```

See Also

`setbuf()`, **`setvbuf()`**

Check Your Progress

Q.10: Write true and false against the following.

i. **fflush()** returns NULL
ii. **setvbuf()** returns zero on success, and nonzero on failure.
iii. **tmpnam()** returns Not NULL
iv. **rename()** renames a file and optionally moves it to a new location
v. **fwrite()** write binary data to a file.
vi. **ftell()** Tells you where a particular file is about to read from or write to.

1.26 Answer to Check Your Progress

Ans to Q.1: fseek() call.

Ans to Q.2: putchar()

Ans to Q.3: getc()

Ans to Q.4: getchar()

Ans to Q.5: sizeof()

Ans to Q.6: `fgets()`

Ans to Q.7: scanf()

Ans to Q.8: freopen()

Ans to Q.9: fopen()

Ans to Q.10: **i.** False **ii.** True **iii.** False **iv.** True **v.** True **vi.** True

1.27Model Questions

2. What is the use of fopen()? Give an example .
3. What is the use of freopen()?Give an example .
4. What is the use gets() and fgets()? Explain with the help of example.
5. What is the difference between getc(), fgetc() and getchar()?
6. Explain the fseek() and rewind().
7. Explain the following functions with the help of example
    i. fgetpos() and fsetpos()
    ii. ungetc()
    iii. fread()
    iv. fwrite()
    v. feof(), ferror() and clearerr()
    vi. remove()
    vii. rename()
    viii. setbuf() and setvbuf()
    ix. fflush()

Unit 12
String Manipulation

1.1 Learning Objectives

1.2 Introduction

1.3 strlen()

1.4 strcmp(), strncmp()

1.5 strcat(), strncat()

1.6 strchr(), strrchr()

1.7 strcpy(), strncpy()

1.8 strspn(), strcspn()

1.9 strstr()

1.10 strtok()

1.11 Answer to Check Your Progress

1.12 Model Questions

1.1 Learning Objectives

After going through this unit, the learner will able to learn:

- strlen() function
- strcmp(), strncmp() function
- strcat(), strncat() function
- strchr(), strrchr() function
- strcpy(), strncpy() function
- strspn(), strcspn() function
- strstr() function
- strtok() function

1.2 Introduction

As has been mentioned earlier in the guide, a string in C is a sequence of bytes in memory, terminated by a NULL character ('\0'). The NULL at the end is important, since it lets all these string functions (and **`printf()`** and **`puts()`** and everything else that deals with a string) know where the end of the string actually is.

Fortunately, when you operate on a string using one of these many functions available to you, they add the NULL terminator on for you, so you actually rarely have to keep track of it yourself. (Sometimes you do, especially if you're building a string from scratch a character at a time or something.)

In this section you'll find functions for pulling substrings out of strings, concatenating strings together, getting the length of a string, and so forth and so on.

1.3 strlen()

Returns the length of a string.

Prototypes

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

Description

This function returns the length of the passed null-terminated string (not counting the NUL
character at the end). It does this by walking down the string and counting the bytes until the
NUL character, so it's a little time consuming. If you have to get the length of the same string
repeatedly, save it off in a variable somewhere.

Return Value

Returns the number of characters in the string.

Example

```
char *s = "Hello, world!"; // 13 characters
// prints "The string is 13 characters
long.":
printf("The string is %d characters
long.\n", strlen(s));
```

1.4 strcmp(), strncmp()

Compare two strings and return a difference.

Prototypes

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

Description

Both these functions compare two strings. **strcmp()** compares the entire string down to the end, while **strncmp()** only compares the first *n* characters of the strings.

It's a little funky what they return. Basically it's a difference of the strings, so if the strings are the same, it'll return zero (since the difference is zero). It'll return non-zero if the strings differ; basically it will find the first mismatched character and return less-than zero if that character in *s1* is less than the corresponding character in *s2*. It'll return greater-than zero if that character in *s1* is greater than that in *s2*.

For the most part, people just check to see if the return value is zero or not, because, more
often than not, people are only curious if strings are the same.

These functions can be used as comparison functions for **qsort()** if you have an array of
char*s you want to sort.

Return Value

Returns zero if the strings are the same, less-than zero if the first different character in *s1* is
less than that in *s2*, or greater-than zero if the first difference character in *s1* is greater than than
in *s2*.

Example

```
char *s1 = "Muffin";
char *s2 = "Muffin Sandwich";
char *s3 = "Muffin";
strcmp("Biscuits", "Kittens"); // returns <
0 since 'B' < 'K'
strcmp("Kittens", "Biscuits"); // returns >
0 since 'K' > 'B'
if (strcmp(s1, s2) == 0)
printf("This won't get printed because the
strings differ");
if (strcmp(s1, s3) == 0)
printf("This will print because s1 and s3
are the same");
// this is a little weird...but if the
strings are the same, it'll
// return zero, which can also be thought
of as "false". Not-false
// is "true", so (!strcmp()) will be true
if the strings are the
// same. yes, it's odd, but you see this
all the time in the wild
// so you might as well get used to it:
if (!strcmp(s1, s3))
printf("The strings are the same!")
```

```
if (!strncmp(s1, s2, 6))
printf("The first 6 characters of s1 and s2
are the same");
```

See Also

`memcmp()`, **`qsort()`**

1.5 strcat(), strncat()

Concatenate two strings into a single string.

Prototypes

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

Description

"Concatenate", for those not in the know, means to "stick together". These functions take two strings, and stick them together, storing the result in the first string.

These functions don't take the size of the first string into account when it does the concatenation. What this means in practical terms is that you can try to stick a 2 megabyte string into a 10 byte space. This will lead to unintended consequences, unless you intended to lead to unintended consequences, in which case it will lead to intended unintended consequences.

Technical banter aside, your boss and/or professor will be irate.

If you want to make sure you don't overrun the first string, be sure to check the lengths of the strings first and use some highly technical subtraction to make sure things fit. You can actually only concatenate the first *n* characters of the second string by using **strncat()** and specifying the maximum number of characters to copy.

Return Value

Both functions return a pointer to the destination string, like most of the string-oriented functions.

Example

```
char dest[20] = "Hello";
char *src = ", World!";
char numbers[] = "12345678";
printf("dest  before  strcat:  \"%s\"\n",
dest); // "Hello"
strcat(dest, src);
printf("dest  after  strcat:  \"%s\"\n",
dest); // "Hello, world!"
strncat(dest, numbers, 3); // strcat first
3 chars of numbers
printf("dest  after  strncat:  \"%s\"\n",
dest); // "Hello, world!123"
```

Notice I mixed and matched pointer and array notation there with *`src`* and *`numbers`*; this is just fine with string functions.

See Also

`strlen()`

Check Your Progress

Fill in the blanks

Q.1: …………….is used to Concatenate two strings into a single string.

Q.2: ……………….only compares the first *n* characters of the strings.

Q.3: …………………returns the length of a string.

Q.4: A string in C is a sequence of bytes in memory, terminated by a…………………

1.6 strchr(), strrchr()

Find a character in a string.

Prototypes

```
#include <string.h>
char *strchr(char *str, int c);
char *strrchr(char *str, int c);
```

Description

The functions **strchr()** and **strrchr** find the first or last occurrence of a letter in a string, respectively. (The extra “r” in **strrchr()** stands for “reverse”--it looks starting at the end of the string and working backward.) Each function returns a pointer to the char in question, or NULL if the letter isn't found in the string.

Quite straight forward.

One thing you can do if you want to find the next occurrence of the letter after finding the first, is call the function again with the previous return value plus one. (Remember pointer arithmetic?) Or minus one if you're looking in reverse. Don't accidentally go off the end of the string!

Return Value

Returns a pointer to the occurance of the letter in the string, or NULL if the letter is not
found.

Example

```
// "Hello, world!"
// ^ ^
// A B
char *str = "Hello, world!";
char *p;
p = strchr(str, ','); // p now points at
position A
```

```
p = strrchr(str, 'o'); // p now points at
position B
// repeatedly find all occurances of the
letter 'B'
char *str = "A BIG BROWN BAT BIT BEEJ";
char *p;
for(p = strchr(str, 'B'); p != NULL; p =
strchr(p + 1, 'B')) {
printf("Found a 'B' here: %s\n", p);
}
// output is:
//
// Found a 'B' here: BIG BROWN BAT BIT BEEJ
// Found a 'B' here: BROWN BAT BIT BEEJ
// Found a 'B' here: BAT BIT BEEJ
// Found a 'B' here: BIT BEEJ
// Found a 'B' here: BEEJ
```

1.7 strcpy(), strncpy()

Copy a string

Prototypes

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

Description

These functions copy a string from one address to another, stopping at the NUL terminator on the *src*string.

strncpy() is just like **strcpy()**, except only the first *n* characters are actually copied.

Beware that if you hit the limit, *n* before you get a NUL terminator on the *src* string, your *dest*
string won't be NUL-terminated. Beware! BEWARE!
(If the *src* string has fewer than *n* characters, it works just like **strcpy()**.)
You can terminate the string yourself by sticking the '\0' in there yourself:

```
char s[10];
char foo = "My hovercraft is full of
eels."; // more than 10 chars
strncpy(s, foo, 9); // only copy 9 chars
into positions 0-8
s[9] = '\0'; // position 9 gets the
terminator
```

Return Value
Both functions return *dest* for your convenience, at no extra charge.
Example

```
char *src = "hockey hockey hockey hockey
hockey hockey hockey hockey";
char dest[20];
int len;
strcpy(dest, "I like "); // dest is now "I
like "
len = strlen(dest);
// tricky, but let's use some pointer
arithmetic and math to append
// as much of src as possible onto the end
of dest, -1 on the length to
// leave room for the terminator:
```

```
strncpy(dest+len, src, sizeof(dest)-len-1);
// remember that sizeof() returns the size
of the array in bytes
// and a char is a byte:
dest[sizeof(dest)-1] = '\0'; // terminate
// dest is now: v null terminator
// I like hockey hocke
// 01234567890123456789012345
```

See Also

`memcpy()`, **`strcat()`**, **`strncat()`**

1.8 strspn(), strcspn()

Return the length of a string consisting entirely of a set of characters, or of not a set of
characters.

Prototypes

```
#include <string.h>
size_t   strspn(char   *str,   const   char
*accept);
size_t   strcspn(char   *str,   const   char
*reject);
```

Description

`strspn()` will tell you the length of a string consisting entirely of the set of characters in *`accept`*. That is, it starts walking down *`str`* until it finds a character that is *not* in the set (that is, a character that is not to be accepted), and returns the length of the string so far.

`strcspn()` works much the same way, except that it walks down *`str`* until it finds a character in the *`reject`* set (that is, a character that is to be rejected.) It then returns the length of the string so far.

Return Value

The lenght of the string consisting of all characters in *`accept`* (for **`strspn()`**), or the length of the string consisting of all characters except *`reject`* (for **`strcspn()`**

Example

```
char str1[] = "a banana";
char  str2[]  =  "the  bolivian  navy  on
manuvers in the south pacific";
// how many letters in str1 until we reach
something that's not a vowel?
n = strspn(str1, "aeiou"); // n == 1, just
"a"
// how many letters in str1 until we reach
something that's not a, b,
// or space?
n = strspn(str1, "ab "); // n == 4, "a ba"
// how many letters in str2 before we get a
"y"?
n  =  strcspn(str2,  "y");  //  n  =  16,  "the
bolivian nav"
```

See Also

`strchr()`, **`strrchr()`**

1.9 strstr()

Find a string in another string.

Prototypes

#include <string.h>

char *strstr(const char *str, const char *substr);

Description

Let's say you have a big long string, and you want to find a word, or whatever substring

strikes your fancy, inside the first string. Then **strstr()** is for you! It'll return a pointer to the

substr within the *str*!

Return Value

You get back a pointer to the occurance of the *substr* inside the *str*, or *NULL* if the
substring can't be found.

Example

```
char *str = "The quick brown fox jumped
over the lazy dogs.";
char *p;
p = strstr(str, "lazy");
printf("%s\n", p); // "lazy dogs."
// p is NULL after this, since the string
"wombat" isn't in str:
p = strstr(str, "wombat");
```

See Also

`strchr()`, **`strrchr()`**, **`strspn()`**, **`strcspn()`**

1.10 strtok()

Tokenize a string.

Prototypes

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

Description

If you have a string that has a bunch of separators in it, and you want to break that string up
into individual pieces, this function can do it for you.

The usage is a little bit weird, but at least whenever you see the function in the wild, it's
consistently weird.

Basically, the first time you call it, you pass the string, *str* that you want to break up in as the first argument. For each subsequent call to get more tokens out of the string, you pass

NULL. This is a little weird, but **strtok()** remembers the string you originally passed in, and continues to strip tokens off for you.

Note that it does this by actually putting a NUL terminator after the token, and then returning a pointer to the start of the token. So the original string you pass in is destroyed, as it were. If you need to preserve the string, be sure to pass a copy of it to **strtok()** so the original isn't destroyed.

Return Value

A pointer to the next token. If you're out of tokens, *NULL* is returned.

Example

```
// break up the string into a series of
space or
// punctuation-separated words
char *str = "Where is my bacon, dude?";
char *token;
// Note that the following if-do-while
construct is very very
// very very very common to see when using
strtok().
// grab the first token (making sure there
is a first token!)
if ((token = strtok(str, ".,?! ")) != NULL)
{
do {
printf("Word: \"%s\"\n", token);
// now, the while continuation condition
grabs the
// next token (by passing NULL as the first
param)
// and continues if the token's not NULL:
} while ((token = strtok(NULL, ".,?! ")) !=
NULL);
}
// output is:
//
// Word: "Where"
// Word: "is"
// Word: "my"
// Word: "bacon"
// Word: "dude"
//
```

See Also

strchr(), **strrchr()**, **strspn()**, **strcspn()**

Check Your Progress

Q.5: Write true and false against the following.

i. **strcpy()** is used to Tokenize a string.

ii. **strstr()** is used to find a string in another string.

iii. **strchr() and strncpy() is used to** copy a string.

iv. The functions **strchr()** and **strrchr** find the first or last occurrence of a letter in a string, respectively

1.11 Answer to Check Your Progress

Ans to Q.1: strcat() and strncat()

Ans to Q.2: strncmp()

Ans to Q.3: strlen()

Ans to Q.4: NULL character ('\0')

Ans to Q.5: i. False ii. True iii. False iv. True

1.12 Model Questions

1. What is the use of NULL character ('\0')?
2. Define strlen() function with the help of example.
3. What is the difference btween strcmp() and strncmp()?
4. Explain strchr() and strrchr() with the help of example.
5. Define strstr() function with example.

Unit 13

Mathematical Functions

1.1 Learning Objectives

1.2 Introduction

1.3 sin(), sinf(), sinl()

1.4 cos(), cosf(), cosl()

1.5 tan(), tanf(), tanl()

1.6 asin(), asinf(), asinl()

1.7 acos(), acosf(), acosl()

1.8 atan(), atanf(), atanl(), atan2(), atan2f(), atan2l()

1.9 sqrt()

1.10 Answer to Check Your Progress

1.1 Learning Objectives

After going through this unit, the learner will able to learn;

- About sin(), sinf(), sinl() function
- About cos(), cosf(), cosl()function

- About tan(), tanf(), tanl() function
- About asin(), asinf(), asinl()function
- About acos(), acosf(), acosl()function
- About atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() function
- About sqrt() function

1.2 Introduction

It's your favorite subject: Mathematics! Hello, I'm Doctor Math, and I'll be making math FUN and EASY!
[vomiting sounds]
Ok, I know math isn't the grandest thing for some of you out there, but these are merely functions that quickly and easily do math you either know, want, or just don't care about. That
pretty much covers it.
For you trig fans out there, we've got all manner of things, including sine, cosine, tangent, and, conversely, arc sine, arc cosine, and arc tangent. That's very exciting.
And for normal people, there is a slurry of your run-of-the-mill functions that will serve your general purpose mathematical needs, including absolute value, hypotenuse length, square
root, cube root, and power.
In short, you're a fricking MATHEMATICAL GOD!
Oh wait, before then, I should tell you that the trig functions have three variants with different suffixes. The "f" suffix (e.g. **sinf()**) returns a float, while the "l" suffix (e.g. **sinl()**) returns a massive and nicely accurate long double. Normal **sin()** just returns a double. These are extensions to ANSI C, but they should be supported by modern compilers.

Also, there are several values that are defined in the *math.h* header file.

`M_E`

e

`M_LOG2E`

log_2 e

`M_LOG10E`

log_10 e

`M_LN2`

log_e 2

`M_LN10`

log_e 10

`M_PI`

pi

`M_PI_2`

pi/2

`M_PI_4`

pi/4

`M_1_PI`

1/pi

`M_2_PI`

2/pi

`M_2_SQRTPI`

2/sqrt(pi)

`M_SQRT2`

sqrt(2)

`M_SQRT1_2`

1/sqrt(2)

1.3 sin(), sinf(), sinl()

Calculate the sine of a number.

Prototypes

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

```
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Description

Calculates the sine of the value *x*, where *x* is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just

like degrees. To convert from degrees to radians or the other way around, use the following

code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the sine of *x*. The variants return different types.

Example

```
double sinx;
long double ldsinx;
sinx = sin(3490.0); // round and round we
go!
ldsinx = sinl((long double)3.490);
```

See Also

cos(), **tan()**, **asin()**

1.4 cos(), cosf(), cosl()

Calculate the cosine of a number.

Prototypes

```
#include <math.h>
double cos(double x)
float cosf(float x)
long double cosl(long double x)
```

Description

Calculates the cosine of the value *x*, where *x* is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just

like degrees. To convert from degrees to radians or the other way around, use the following

code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the cosine of *x*. The variants return different types.

Example

```
double sinx;

long double ldsinx;

sinx = sin(3490.0); // round and round we
go!

ldsinx = sinl((long double)3.490);
```

See Also

`sin()`, **`tan()`**, **`acos()`**

Check Your Progress

Choose the Correct one

1. The cos function computes the cosine of x.
   - A. measured in radians
   - B. measured in degrees
   - C. measured in gradian
   - D. measured in milliradian
2. Which of the following header declares mathematical functions and macros?
   - A. math.h
   - B. assert.h
   - C. stdmat. H
   - D. stdio. H
3. What type of inputs are accepted by mathematical functions?
   - A. Short
   - B. Int
   - C. Float
   - D. double
4. Which of the following is not a valid mathematical function?
   - A. frexp(x);

B. atan2(x,y);

C. srand(x);

D. fmod(x);

1.5 tan(), tanf(), tanl()

Calculate the tangent of a number.

Prototypes

```
#include <math.h>
double tan(double x)
float tanf(float x)
long double tanl(long double x)
```

Description

Calculates the tangent of the value *x*, where *x* is in radians.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the tangent of *x*. The variants return different types.

Example

```
double tanx;
long double ldtanx;
tanx = tan(3490.0); // round and round we
go!
ldtanx = tanl((long double)3.490);
```

See Also

sin(), cos(), atan(), atan2()

1.6 asin(), asinf(), asinl()

Calculate the arc sine of a number.

Prototypes

```
#include <math.h>
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Description

Calculates the arc sine of a number in radians. (That is, the value whose sine is *x*.) The number must be in the range -1.0 to 1.0.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the arc sine of *x*, unless *x* is out of range. In that case, *errno* will be set to EDOM
and the return value will be NaN. The variants return different types.

Example

```
double asinx;
long double ldasinx;
asinx = asin(0.2);
ldasinx = asinl((long double)0.3);
```

See Also

acos(), **atan()**, **atan2()**, **sin()**

1.7 acos(), acosf(), acosl()

Calculate the arc cosine of a number.

Prototypes

```
#include <math.h>
double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Description

Calculates the arc cosine of a number in radians. (That is, the value whose cosine is *x*.) The

number must be in the range -1.0 to 1.0.

For those of you who don't remember, radians are another way of measuring an angle, just

like degrees. To convert from degrees to radians or the other way around, use the following

code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

Returns the arc cosine of *x*, unless *x* is out of range. In that case, *errno* will be set to

EDOM and the return value will be NaN. The variants return different types.

Example

```
double acosx;

long double ldacosx;

acosx = acos(0.2);

ldacosx = acosl((long double)0.3);
```

See Also

asin(), **atan()**, **atan2()**, **cos()**

1.8 atan(), atanf(), atanl(), atan2(), atan2f(), atan2l()

Calculate the arc tangent of a number.

Prototypes

```
#include <math.h>
double atan(double x);
float atanf(float x);
long double atanl(long double x);
double atan2(double y, double x);
float atan2f(float y, float x);
     long double atan2l(long double y, long
double x);
```

Description

Calculates the arc tangent of a number in radians. (That is, the value whose tangent is *x*.)

The **atan2()** variants are pretty much the same as using **atan()** with *y*/*x* as the argument...except that **atan2()** will use those values to determine the correct quadrant of the
result.

For those of you who don't remember, radians are another way of measuring an angle, just like degrees. To convert from degrees to radians or the other way around, use the following code:

```
degrees = radians * 180.0f / M_PI;
radians = degrees * M_PI / 180;
```

Return Value

The **atan()** functions return the arc tangent of *x*, which will be between PI/2 and -PI/2.

The **atan2()** functions return an angle between PI and -PI.

Example

```
double atanx;
```

```
long double ldatanx;
atanx = atan(0.2);
ldatanx = atanl((long double)0.3);
atanx = atan2(0.2);
ldatanx = atan2l((long double)0.3);
```

See Also

`tan()`, **`asin()`**, **`atan()`**

1.9 sqrt()

Calculate the square root of a number

Prototypes

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Description

Computes the square root of a number. To those of you who don't know what a square root is, I'm not going to explain. Suffice it to say, the square root of a number delivers a value that when squared (multiplied by itself) results in the original number.

Ok, fine--I did explain it after all, but only because I wanted to show off. It's not like I'm giving you examples or anything, such as the square root of nine is three, because when you
multiply three by three you get nine, or anything like that. No examples. I hate examples!

And I suppose you wanted some actual practical information here as well. You can see the usual trio of functions here--they all compute square root, but they take different types as arguments. Pretty straightforward, really.

Return Value

Returns (and I know this must be something of a surprise to you) the square root of *x*. If
you try to be smart and pass a negative number in for *x*, the global variable *errno* will be set to
EDOM (which stands for DOMain Error, not some kind of cheese.)

Example

```
// example usage of sqrt()
float something = 10;
double x1 = 8.2, y1 = -5.4;
double x2 = 3.8, y2 = 34.9;
double dx, dy;
printf("square  root  of  10  is  %.2f\n",
sqrtf(something));
dx = x2 - x1;
dy = y2 - y1;
printf("distance  between  points  (x1,  y1)
and (x2, y2): %.2f\n",
sqrt(dx*dx + dy*dy));
```

And the output is:

```
square root of 10 is 3.16
distance between points (x1, y1) and (x2, y2): 40.54
```

See Also

`hypot()`

Check Your Progress

Q.5: Fill in the blanks

i. …………….is used to calculate the square root of a number.

ii. ……………….is used to calculate the arc tangent of a number.

iii. ……………………is used to calculate the arc cosine of a number.

iv. ……………………………is used to calculate the arc sine of a number.

v. …………………………………..is used to calculate the tangent of a number.

1.10 Answer to Check Your Progress

Ans to Q.1: A. measured in radians

Ans to Q.2: A. math.h

Ans to Q.3: D. double

Ans to Q.4: D. fmod(x);

Ans to Q.5: i. sqrt**()** **ii.** atan(), atanf(), atanl(), atan2(), atan2f() and atan2l() **iii.** acos(), acosf(), acosl() **iv.** asin(), asinf(), asinl() **v.** tan(), tanf(), tanl()

1.11 Model Questions

1. Define sin(), sinf() and sinl() function with an example
2. Explain cos(), cosf() and cosl() function with an example.
3. What is the use of tan(), tanf() and tanl() funcation?
4. Define sqrt() funcation witn an example.

5. What is the use of the following functions give an example of each function?
    i. atan(), atanf(), atanl(), atan2(), atan2f() and atan2l()
    ii. acos(), acosf() and acosl()