http://www.assoc-amazon.com/e/ir?t=ucmbread-20&l=as2&o=1&a=1441919015

**Topic 1**

Basic definitions of C/C++ programming language.

C is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language. It keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers.

Local Environment Setup

If you want to set up your environment for C programming language, you need the following two software tools available on your computer, (a) Text Editor and (b) The C Compiler.

Text Editor

This will be used to type your program. Examples of few a editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The name and version of text editors can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on windows as well as on Linux or UNIX.

The files you create with your editor are called the source files and they contain the program source codes. The source files for C programs are typically named with the extension "**.c**".

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, compile it and finally execute it.

The C Compiler

The source code written in source file is the human readable source for your program. It needs to be "compiled", into machine language so that your CPU can actually execute the program as per the instructions given.

The compiler compiles the source codes into final executable programs. The most frequently used and free available compiler is the GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective operating systems.

The following section explains how to install GNU C/C++ compiler on various OS. We keep mentioning C/C++ together because GNU gcc compiler works for both C and C++ programming languages.

Installation on UNIX/Linux

If you are using **Linux or UNIX**, then check whether GCC is installed on your system by entering the following command from the command line −

$ gcc -v

If you have GNU compiler installed on your machine, then it should print a message as follows −

Using built-in specs.

Target: i386-redhat-linux

Configured with: ../configure --prefix=/usr .......

Thread model: posix

gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at <https://gcc.gnu.org/install/>

This tutorial has been written based on Linux and all the given examples have been compiled on the Cent OS flavor of the Linux system.

Installation on Mac OS

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's web site and follow the simple installation instructions. Once you have Xcode setup, you will be able to use GNU compiler for C/C++.

Xcode is currently available at [developer.apple.com/technologies/tools/](https://developer.apple.com/technologies/tools/).

Installation on Windows

To install GCC on Windows, you need to install MinGW. To install MinGW, go to the MinGW homepage, [www.mingw.org](http://www.mingw.org/), and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

While installing Min GW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable, so that you can specify these tools on the command line by their simple names.

After the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

Hello World Example

A C program basically consists of the following parts −

* Preprocessor Commands
* Functions
* Variables
* Statements & Expressions
* Comments

Let us look at a simple code that would print the words "Hello World" −

[Live Demo](http://tpcg.io/3Ty4QP)

#include <stdio.h>

int main() {

/\* my first program in C \*/

printf("Hello, World! \n");

return 0;

}

Let us take a look at the various parts of the above program −

* The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
* The next line *int main()* is the main function where the program execution begins.
* The next line /\*...\*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
* The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
* The next line **return 0;** terminates the main() function and returns the value 0.

Compile and Execute C Program

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps −

* Open a text editor and add the above-mentioned code.
* Save the file as *hello.c*
* Open a command prompt and go to the directory where you have saved the file.
* Type *gcc hello.c* and press enter to compile your code.
* If there are no errors in your code, the command prompt will take you to the next line and would generate *a.out*executable file.
* Now, type *a.out* to execute your program.
* You will see the output *"Hello World"* printed on the screen.

$ gcc hello.c

$ ./a.out

Hello, World!

Make sure the gcc compiler is in your path and that you are running it in the directory containing the source file hello.c.

You have seen the basic structure of a C program, so it will be easy to understand other basic building blocks of the C programming language.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens −

printf("Hello, World! \n");

The individual tokens are −

printf

(

"Hello, World! \n"

)

;

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements −

printf("Hello, World! \n");

return 0;

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /\* and terminate with the characters \*/ as shown below −

/\* my first program in C \*/

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '\_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a **case-sensitive** programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers −

mohd zara abc move\_name a\_123

myname50 \_temp j a23b9 retVal

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| auto | else | long | switch |
| break | enum | register | typedef |
| case | extern | return | union |
| char | float | short | unsigned |
| const | for | signed | void |
| continue | goto | sizeof | volatile |
| default | if | static | while |
| do | int | struct | \_Packed |
| double |  |  |  |

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement −

int age;

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement −

fruit = apples + oranges; // get the total fruit

no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish to increase readability.

**Topic №5 ,6.**

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

We will, in this chapter, look into the way each operator works.

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B**holds 20 then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands. | A + B = 30 |
| − | Subtracts second operand from the first. | A − B = -10 |
| \* | Multiplies both operands. | A \* B = 200 |
| / | Divides numerator by de-numerator. | B / A = 2 |
| % | Modulus Operator and remainder of after an integer division. | B % A = 0 |
| ++ | Increment operator increases the integer value by one. | A++ = 11 |
| -- | Decrement operator decreases the integer value by one. | A-- = 9 |

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not. If yes, then the condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true. | (A <= B) is true. |

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | !(A && B) is true. |

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12, i.e., 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) = 61, i.e., 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49, i.e., 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = -60, i.e,. 1100 0100 in 2's complement form. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240 i.e., 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 = 15 i.e., 0000 1111 |

Assignment Operators

The following table lists the assignment operators supported by the C language −

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator. Assigns values from right side operands to left side operand | C = A + B will assign the value of A + B to C |
| += | Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

Misc Operators ↦ sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_sizeof_operator.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of a variable. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of a variable. | &a; returns the actual address of the variable. |
| \* | Pointer to a variable. | \*a; |
| ? : | Conditional Expression. | If Condition is true ? then value X : otherwise value Y |

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has a higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

[Show Examples](https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm)

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

**C - Decision Making**

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages −



C programming language assumes any **non-zero** and **non-null**values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision making statements.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/cprogramming/if_statement_in_c.htm)  An if statement consists of a boolean expression followed by one or more statements. |
| 2 | [**i**f...else statement](https://www.tutorialspoint.com/cprogramming/if_else_statement_in_c.htm)  An **if statement** can be followed by an optional else statement, which executes when the Boolean expression is false. |
| 3 | [nested if statements](https://www.tutorialspoint.com/cprogramming/nested_if_statements_in_c.htm)  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| 4 | [switch statement](https://www.tutorialspoint.com/cprogramming/switch_statement_in_c.htm)  A **switch** statement allows a variable to be tested for equality against a list of values. |
| 5 | [nested switch statements](https://www.tutorialspoint.com/cprogramming/nested_switch_statements_in_c.htm)  You can use one **switch** statement inside another **switch** statement(s). |

## The ? : Operator

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this −

* Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
* If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

**IF statement**

An **if** statement consists of a Boolean expression followed by one or more statements.

## Syntax

The syntax of an 'if' statement in C programming language is −

if(boolean\_expression) {

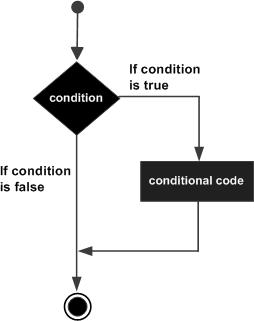
/\* statement(s) will execute if the boolean expression is true \*/

}

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

C programming language assumes any **non-zero** and **non-null**values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.

## Flow Diagram



## Example

[Live Demo](http://tpcg.io/iMjhoV)

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* check the boolean condition using if statement \*/

if( a < 20 ) {

/\* if condition is true then print the following \*/

printf("a is less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}

When the above code is compiled and executed, it produces the following result −

a is less than 20;

value of a is : 10

# C - if...else statement

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

## Syntax

The syntax of an **if...else** statement in C programming language is −

if(boolean\_expression) {

/\* statement(s) will execute if the boolean expression is true \*/

} else {

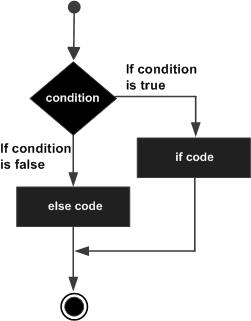
/\* statement(s) will execute if the boolean expression is false \*/

}

If the Boolean expression evaluates to **true**, then the **if block**will be executed, otherwise, the **else block** will be executed.

C programming language assumes any **non-zero** and **non-null**values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

## Flow Diagram



## Example

[Live Demo](http://tpcg.io/iWQk4P)

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 100;

/\* check the boolean condition \*/

if( a < 20 ) {

/\* if condition is true then print the following \*/

printf("a is less than 20\n" );

} else {

/\* if condition is false then print the following \*/

printf("a is not less than 20\n" );

}

printf("value of a is : %d\n", a);

return 0;

}

When the above code is compiled and executed, it produces the following result −

a is not less than 20;

value of a is : 100

## If...else if...else Statement

An **if** statement can be followed by an optional **else if...else**statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if..else statements, there are few points to keep in mind −

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

### Syntax

The syntax of an **if...else if...else** statement in C programming language is −

if(boolean\_expression 1) {

/\* Executes when the boolean expression 1 is true \*/

} else if( boolean\_expression 2) {

/\* Executes when the boolean expression 2 is true \*/

} else if( boolean\_expression 3) {

/\* Executes when the boolean expression 3 is true \*/

} else {

/\* executes when the none of the above condition is true \*/

}

### Example

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 100;

/\* check the boolean condition \*/

if( a == 10 ) {

/\* if condition is true then print the following \*/

printf("Value of a is 10\n" );

} else if( a == 20 ) {

/\* if else if condition is true \*/

printf("Value of a is 20\n" );

} else if( a == 30 ) {

/\* if else if condition is true \*/

printf("Value of a is 30\n" );

} else {

/\* if none of the conditions is true \*/

printf("None of the values is matching\n" );

}

printf("Exact value of a is: %d\n", a );

return 0;

}

When the above code is compiled and executed, it produces the following result −

None of the values is matching

Exact value of a is: 100

# C - switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

## Syntax

The syntax for a **switch** statement in C programming language is as follows −

switch(expression) {

case constant-expression :

statement(s);

break; /\* optional \*/

case constant-expression :

statement(s);

break; /\* optional \*/

/\* you can have any number of case statements \*/

default : /\* Optional \*/

statement(s);

}

The following rules apply to a **switch** statement −

* The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
* When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a **break**. If no **break**appears, the flow of control will *fall through* to subsequent cases until a break is reached.
* A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

## Flow Diagram



## Example

[Live Demo](http://tpcg.io/0KAp4U)

#include <stdio.h>

int main () {

/\* local variable definition \*/

char grade = 'B';

switch(grade) {

case 'A' :

printf("Excellent!\n" );

break;

case 'B' :

case 'C' :

printf("Well done\n" );

break;

case 'D' :

printf("You passed\n" );

break;

case 'F' :

printf("Better try again\n" );

break;

default :

printf("Invalid grade\n" );

}

printf("Your grade is %c\n", grade );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Well done

Your grade is B

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



C programming language provides the following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| Sr.No. | **Loop Type & Description** |
| 1 | [**while loop**](https://www.tutorialspoint.com/cprogramming/c_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| **2** | [**for loop**](https://www.tutorialspoint.com/cprogramming/c_for_loop.htm)  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| **3** | [do...while loop](https://www.tutorialspoint.com/cprogramming/c_do_while_loop.htm)  It is more like a while statement, except that it tests the condition at the end of the loop body. |
| 4 | [nested loops](https://www.tutorialspoint.com/cprogramming/c_nested_loops.htm)  You can use one or more loops inside any other while, for, or do..while loop. |

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/cprogramming/c_break_statement.htm)  Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [continue statement](https://www.tutorialspoint.com/cprogramming/c_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [goto statement](https://www.tutorialspoint.com/cprogramming/c_goto_statement.htm)  Transfers control to the labeled statement. |

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

#include <stdio.h>

int main () {

for( ; ; ) {

printf("This loop will run forever.\n");

}

return 0;

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

**NOTE** − You can terminate an infinite loop by pressing Ctrl + C keys.

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in C programming language is −

while(condition) {

statement(s);

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

[Live Demo](http://tpcg.io/p4bawJ)

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* while loop execution \*/

while( a < 20 ) {

printf("value of a: %d\n", a);

a++;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is −

for ( init; condition; increment ) {

statement(s);

}

Here is the flow of control in a 'for' loop −

* The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
* Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
* After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
* The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Flow Diagram



Example

[Live Demo](http://tpcg.io/Li6I3H)

#include <stdio.h>

int main () {

int a;

/\* for loop execution \*/

for( a = 10; a < 20; a = a + 1 ){

printf("value of a: %d\n", a);

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

The syntax of a **do...while** loop in C programming language is −

do {

statement(s);

} while( condition );

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example

[Live Demo](http://tpcg.io/pVAh6H)

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

do {

printf("value of a: %d\n", a);

a = a + 1;

}while( a < 20 );

return 0;

}

When the above code is compiled and executed, it produces the following result −

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**TOPIC 2**

**Pointers.**

Addresses, & operator, ***%p*** format. Introducton to pointers. Pointers and arrays. Pointer arithmetic.

C/C++ **memory management.**

**Pointers**

Every variable is a memory location and every memory location has its address defined which can be accessed using ***ampersand*** (&) operator which denotes ***an address*** in memory. Consider the following code that prints the address of the integer variables:

#include <stdio.h>

int i, j, k, l;

int main(void)

{

printf("%p %p %p %p\n",&i,&j,&k,&l);

return 0;

}

The following code prints the address of the char variables:

#include <stdio.h>

char i, j, k, l;

int main(void)

{

printf("%p %p %p %p\n",&i,&j,&k,&l);

return 0;

}

A ***pointer*** is a variable whose value is *memory address* (or the *address* of another variable). Pointers give you the ability to work directly and efficiently with computer memory. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

type \*var-name;

Here, ***type*** is the pointer's base type; it must be a valid C++ type and ***var-name*** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

int \*i, \*j;

double \*x, \*y;

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents ***a memory address***. The *size* of any memory address is usually 4 bytes because it has the form XXXX:XXXX. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

The *size* of any pointer is usually 4 bytes:

#include <stdio.h>

int \*i, \*j;

double \*x, \*y;

int main(void)

{

printf("%d %d\n",sizeof(i),sizeof(j));

printf("%d %d\n",sizeof(x),sizeof(y));

return 0;

}

#include <stdio.h>

int i, j, k, l;

int \*p;

int main(void)

{

i = 11; j = 22; k = 33; l = 44;

printf("%p %p %p %p\n",&i,&j,&k,&l);

p = &j;

printf("%p %p %d\n",&p,p,\*p);

return 0;

}

**Using Pointers**

There are few important operations, which we will do with the pointers:

* ***Define*** a pointer variables
* ***Assign*** the address of a variable to a pointer
* ***Access*** the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand.

Just as you dereference an iterator to access the object to which it refers, you dereference a pointer to access the object to which it points. You accomplish the dereferencing the same way – with \* , the ***dereference operator***.

#include <stdio.h>

int i;

int \*ptr;

int main(void)

{

i = 11; // Assign a variable

printf("The address of the variable i is: %p\n",&i);

printf("ptr is a NULL pointer: %p\n",ptr);

ptr = &i; // store address of i in pointer variable ptr

printf("The address stored in ptr variable is: %p\n",ptr);

printf("i = %d, value of \*ptr = %d\n",i,\*ptr);

printf("The address where the pointer is located: %p\n",&ptr);

return 0;

}

Tricky program “Catch an Address of a variable”.

#include <stdio.h>

int i = 1234;

int \*ptr;

int main(void)

{

printf("%p\n",&i);

ptr = (int \*)0x00405000; // put there the address of variable i

printf("%p %d\n",ptr,\*ptr); // so that \*ptr = i

return 0;

}

**Reassigning Pointers**

Pointers can point to different objects at different times during the life of a program. *Reassigning* a pointer works like reassigning any other variable.

#include <stdio.h>

int i = 1234, j = 5678;

int \*ptr;

int main(void)

{

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

printf("&i = %p, &j = %p\n",&i,&j);

ptr = &i; printf("ptr = %p, \*ptr = %d\n",ptr,\*ptr);

ptr = &j; printf("ptr = %p, \*ptr = %d\n",ptr,\*ptr);

return 0;

}

**Using Pointers to Objects**

The previous program has worked only with values of a built-in type *int*. But you can use pointers with objects just as easily. You can access an object through a pointer using the dereference operator.

You can call the *member functions* of an object through a pointer the same way you can call the member functions of an object through an iterator. One way to do this is by using the dereference operator and the member access operator.

Just as with iterators, you can use the -> operator with pointers for a more readable way to access object members.

#include <cstdio>

#include <vector>

using namespace std;

vector<int> v(10000);

vector<int> \*ptr;

int main(void)

{

ptr = &v;

printf("%d\n",(\*ptr).size());

printf("%d\n",ptr->size());

return 0;

}

The code (\*ptr).size() says “Take the result of dereferencing *ptr* and call that object’s size() member function”.

**NULL Pointer**

It is always a good practice to assign the pointer NULL to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a ***null pointer***.

The NULL pointer is a ***constant*** with a value of zero defined in several standard libraries. Consider the following program:

#include <stdio.h>

int \*ptr = NULL;

int main(void)

{

printf("ptr is a NULL pointer: %p\n",ptr);

printf("The address where the pointer is located: %p\n",&ptr);

return 0;

}

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to ***point to nothing***.

**Pointer Arithmetic**

Pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

To understand pointer arithmetic, let us consider that *ptr* is an integer pointer which points to the address 0041:717C. Assuming 32-bit integers, let us perform the following arithmatic operation on the pointer:

ptr++;

The *ptr* will point to the location 0041:7180 because each time *ptr* is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location.

#include <stdio.h>

int i, j, k, l;

int \*ptr = &k;

int main(void)

{

i = 11; j = 22; k = 33; l = 44;

printf("%d\n",\*ptr++); // 33

printf("%d\n",\*ptr++); // 22

printf("%d\n",\*ptr++); // 11

printf("%d\n",\*ptr++); // 44

return 0;

}

If *ptr* points to a character whose address is 0041:7178, then the operation *ptr*++ will point to the location 0041:7179 because next character will be available at 0041:7179.

#include <stdio.h>

char i, j, k, l;

char \*ptr = &k;

int main(void)

{

printf("%p %p %p %p\n",&i,&j,&k,&l);

i = 'A'; j = 'B'; k = 'C'; l = 'D';

printf("%c\n",\*ptr++); // C

printf("%c\n",\*ptr++); // B

printf("%c\n",\*ptr++); // A

printf("%c\n",\*ptr++); // D

return 0;

}

**Pointers and arrays**

The ***name of the array*** always points to the first element of an array. Address of first element of an array m is &m[0]. Hence, &m[0] is equivalent to m.



Also, &m[*i*] is equivalent to m + *i*:



#include <stdio.h>

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

int main(void)

{

printf("%p %p\n",m,&m[0]);

printf("%p %p\n",m+2,&m[2]);

return 0;

}

Value in address &m[0] is m[0] and value in address m is \*m. Hence, m[0] is equivalent to \*m.



Also, m[*i*] is equivalent to \*(m + *i*):



#include <stdio.h>

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

int main(void)

{

for(int i = 0; i < 5; i++)

printf("%d %d\n",m[i],\*(m+i));

return 0;

}

You can use pointer to access the data in array. Print all elements of array in different ways:

#include <stdio.h>

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

int \*ptr = m, i;

int main(void)

{

for(i = 0; i < 5; i++)

printf("%d %d %d\n",\*ptr++,\*(m+i),m[i]);

return 0;

}

**Dynamic Memory Allocation for Objects**

Allocate memory for integers.

#include <stdio.h>

int \*p, \*q;

int main(void)

{

p = new int(22);

q = new int; // memory allocated, but nothing assigned, garbage is here

printf("%d %d\n",\*p,\*q);

\*q = 55;

printf("%d %d\n",\*p,\*q);

delete p;

delete q;

return 0;

}

Allocate memory for array of integers.

#include <stdio.h>

int \*p;

int main(void)

{

p = new int[10];

for(int i = 0; i < 10; i++)

p[i] = i\*i;

for(int i = 0; i < 10; i++)

printf("%d ",p[i]);

printf("\n");

delete[] p;

return 0;

}

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

#include <stdio.h>

class Point

{

public:

Point()

{

printf("Point Constructor\n");

}

~Point()

{

printf("Point Destructor\n");

}

};

int main(void)

{

Point p;

return 0;

}

Define a pointer to the class:

#include <stdio.h>

class Point

{

public:

Point()

{

printf("Point Constructor\n");

}

~Point()

{

printf("Point Destructor\n");

}

};

int main(void)

{

Point \*p = new Point;

delete p;

return 0;

}

If you were to allocate an array of Point objects, the constructor would be called this number of times and similarly while deleting these objects, destructor will also be called same number of times:

#include <stdio.h>

class Point

{

public:

Point()

{

printf("Point Constructor\n");

}

~Point()

{

printf("Point Destructor\n");

}

};

int main(void)

{

Point \*p = new Point[5];

// delete p; // wrong, p points to array!

delete[] p;

return 0;

}

**TOPIC 3**

**Function.**

**Functions,** function **declaration.**

**Scope rules. local** variables. **global** variables.

**formal**  parameters.Ways to pass arguments in functions: pass by value and pass by reference.

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition**provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

## Defining a Function

The general form of a function definition in C programming language is as follows −

return\_type function\_name( parameter list ) {

body of the function

}

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

## Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

## Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

## Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

[Live Demo](http://tpcg.io/T4MSFr)

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

## Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

|  |  |
| --- | --- |
| **Sr.No.** | **Call Type & Description** |
| 1 | [Call by value](https://www.tutorialspoint.com/cprogramming/c_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [Call by reference](https://www.tutorialspoint.com/cprogramming/c_function_call_by_reference.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

FUNCTIONS

A ***function*** is a group of statements that together perform a task. Every C program has at least one function, which is **main**(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

**Types of functions**

* ***Predefined standard library functions*** – such as **puts**(), **gets**(), **printf**(), **scanf**() etc – these are the functions which already have a definition in header files (.h files like stdio.h), so we just call them whenever there is a need to use them.
* ***User Defined functions*** – the functions which we can create by ourselves, for example we can create function **abc**() and call it in **main**() in order to use it.

**Defining a Function**

The general form of a function definition in C programming language is as follows:

return\_type function\_name( parameter list )

{

body of the function

}

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function:

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return\_type** is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

**Example**

Given below is the source code for a function called **max**(). This function takes two parameters *a* and *b* and returns the maximum value between them.

int max(int a, int b)

{

if (a > b) return a;

return b;

}

Function **max()** can be written using ternary operator:

int max(int a, int b)

{

return (a > b) ? a : b;

}

Function **sqr()** returns square of a number.

int sqr(int n)

{

return n \* n;

}

**Calling a Function**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example:

#include <stdio.h>

int max(int a, int b)

{

if (a > b) return a;

else return b;

}

int main(void)

{

int a = 111, b = 222;

int res = max(a,b);

printf("a = %d\n",a);

printf("b = %d\n",b);

printf("max(a,b) = %d\n",res);

return 0;

}

**Example**

Function f(*n*) returns the sum of digits for two digital number *n*.

#include <stdio.h>

int f(int n)

{

int a = n / 10; // number of tens

int b = n % 10; // number of units

return a + b;

}

int main(void)

{

printf("%d\n",f(34));

return 0;

}

**Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the ***formal parameters*** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function **Call by value** and **Call by reference**.

**Call by value**

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. ***Different memory*** is allocated for both actual and formal parameters.

* ***Actual parameter*** is the argument which is used in function call.
* ***Formal parameter*** is the argument which is used in function definition



Function f has two formal parameters *a* and *b*. Inside the body of the function f we work with formal parameters, changing them.

#include <stdio.h>

void f(int a, int b)

{

a++; b = b + 2;

printf("%d %d\n",a,b); // 11 22

}

int main(void)

{

int a = 10, b = 20;

f(a,b);

printf("%d %d\n",a,b); // 10 20

return 0;

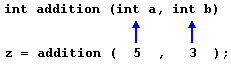
}

**Call by reference**

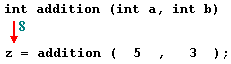
This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Functions allow to structure programs in segments of code to perform individual tasks.  
  
In C, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:  
  
type name ( parameter1, parameter2, ...) { statements }  
  
Where:  
- type is the type of the value returned by the function.  
- name is the identifier by which the function can be called.  
- parameters (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example: int x), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.  
- statements is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.  
  
Let's have a look at an example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // function example  #include <iostream>  using namespace std;  int addition (int a, int b)  {  int r;  r=a+b;  return r;  }  int main ()  {  int z;  z = addition (5,3);  cout << "The result is " << z;  } | The result is 8 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions/) |

This program is divided in two functions: addition and main. Remember that no matter the order in which they are defined, a C++ program always starts by calling main. In fact, main is the only function called automatically, and the code in any other function is only executed if its function is called from main (directly or indirectly).  
  
In the example above, main begins by declaring the variable z of type int, and right after that, it performs the first function call: it calls addition. The call to a function follows a structure very similar to its declaration. In the example above, the call to addition can be compared to its definition just a few lines earlier:  
  
   
The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these correspond to the parameters a and b, declared for function addition.  
  
At the point at which the function is called from within main, the control is passed to function addition: here, execution of main is stopped, and will only resume once the addition function ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables int a and int b within the function.  
  
Then, inside addition, another local variable is declared (int r), and by means of the expression r=a+b, the result of aplus b is assigned to r; which, for this case, where a is 5 and b is 3, means that 8 is assigned to r.  
  
The final statement within the function:

|  |  |  |
| --- | --- | --- |
|  | return r; |  |

Ends function addition, and returns the control back to the point where the function was called; in this case: to function main. At this precise moment, the program resumes its course on main returning exactly at the same point at which it was interrupted by the call to addition. But additionally, because addition has a return type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended addition: in this particular case, the value of the local variable r, which at the moment of the return statement had a value of 8.  
  
   
Therefore, the call to addition is an expression with the value returned by the function, and in this case, that value, 8, is assigned to z. It is as if the entire function call (addition(5,3)) was replaced by the value it returns (i.e., 8).  
  
Then main simply prints this value by calling:

|  |  |  |
| --- | --- | --- |
|  | cout << "The result is " << z; |  |

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // function example  #include <iostream>  using namespace std;  int subtraction (int a, int b)  {  int r;  r=a-b;  return r;  }  int main ()  {  int x=5, y=3, z;  z = subtraction (7,2);  cout << "The first result is " << z << '\n';  cout << "The second result is " << subtraction (7,2) << '\n';  cout << "The third result is " << subtraction (x,y) << '\n';  z= 4 + subtraction (x,y);  cout << "The fourth result is " << z << '\n';  } | The first result is 5  The second result is 5  The third result is 2  The fourth result is 6 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions/) |

Similar to the addition function in the previous example, this example defines a subtract function, that simply returns the difference between its two parameters. This time, main calls this function several times, demonstrating more possible ways in which a function can be called.  
  
Let's examine each of these calls, bearing in mind that each function call is itself an expression that is evaluated as the value it returns. Again, you can think of it as if the function call was itself replaced by the returned value:

|  |  |  |
| --- | --- | --- |
| 1 2 | z = subtraction (7,2);  cout << "The first result is " << z; |  |

If we replace the function call by the value it returns (i.e., 5), we would have:

|  |  |  |
| --- | --- | --- |
| 1 2 | z = 5;  cout << "The first result is " << z; |  |

With the same procedure, we could interpret:

|  |  |  |
| --- | --- | --- |
|  | cout << "The second result is " << subtraction (7,2); |  |

as:

|  |  |  |
| --- | --- | --- |
|  | cout << "The second result is " << 5; |  |

since 5 is the value returned by subtraction (7,2).  
  
In the case of:

|  |  |  |
| --- | --- | --- |
|  | cout << "The third result is " << subtraction (x,y); |  |

The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. The function is called with the values x and y have at the moment of the call: 5 and 3 respectively, returning 2 as result.   
  
The fourth call is again similar:

|  |  |  |
| --- | --- | --- |
|  | z = 4 + subtraction (x,y); |  |

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

|  |  |  |
| --- | --- | --- |
|  | z = subtraction (x,y) + 4; |  |

With exactly the same result. Note also that the semicolon does not necessarily go after the function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

|  |  |  |
| --- | --- | --- |
| 1 2 | z = 4 + 2; // same as z = 4 + subtraction (x,y);  z = 2 + 4; // same as z = subtraction (x,y) + 4; |  |

**Functions with no type. The use of void**

The syntax shown above for functions:  
  
type name ( argument1, argument2 ...) { statements }  
  
Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is void, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | // void function example  #include <iostream>  using namespace std;  void printmessage ()  {  cout << "I'm a function!";  }  int main ()  {  printmessage ();  } | I'm a function! | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions/) |

void can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, printmessage could have been declared as:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | void printmessage (void)  {  cout << "I'm a function!";  } |  |

In C++, an empty parameter list can be used instead of void with same meaning, but the use of void in the argument list was popularized by the C language, where this is a requirement.  
  
Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an empty pair of parentheses shall always be appended to the function name. See how printmessage was called in an earlier example:

|  |  |  |
| --- | --- | --- |
|  | printmessage (); |  |

The parentheses are what differentiate functions from other kinds of declarations or statements. The following would not call the function:

|  |  |  |
| --- | --- | --- |
|  | printmessage; |  |

**The return value of main**

You may have noticed that the return type of main is int, but most examples in this and earlier chapters did not actually return any value from main.  
  
Well, there is a catch: If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement:

|  |  |  |
| --- | --- | --- |
|  | return 0; |  |

Note that this only applies to function main for historical reasons. All other functions with a return type shall end with a proper return statement that includes a return value, even if this is never used.  
  
When main returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by main, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for main that are guaranteed to be interpreted in the same way on all platforms are:

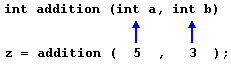
|  |  |
| --- | --- |
| **value** | **description** |
| 0 | The program was successful |
| [EXIT\_SUCCESS](http://www.cplusplus.com/EXIT_SUCCESS) | The program was successful (same as above). This value is defined in header [<cstdlib>](http://www.cplusplus.com/%3Ccstdlib%3E). |
| [EXIT\_FAILURE](http://www.cplusplus.com/EXIT_FAILURE) | The program failed. This value is defined in header [<cstdlib>](http://www.cplusplus.com/%3Ccstdlib%3E). |

Because the implicit return 0; statement for main is a tricky exception, some authors consider it good practice to explicitly write the statement.

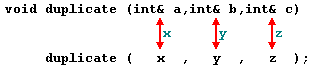
**Arguments passed by value and by reference**

In the functions seen earlier, arguments have always been passed *by value*. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example, take:

|  |  |  |
| --- | --- | --- |
| 1 2 | int x=5, y=3, z;  z = addition ( x, y ); |  |

In this case, function addition is passed 5 and 3, which are copies of the values of x and y, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the function's definition, but any modification of these variables within the function has no effect on the values of the variables x and y outside it, because x and y were themselves not passed to the function on the call, but only copies of their values at that moment.  
  
   
In certain cases, though, it may be useful to access an external variable from within a function. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the function duplicate in this code duplicates the value of its three arguments, causing the variables used as arguments to actually be modified by the call:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // passing parameters by reference  #include <iostream>  using namespace std;  void duplicate (int& a, int& b, int& c)  {  a\*=2;  b\*=2;  c\*=2;  }  int main ()  {  int x=1, y=3, z=7;  duplicate (x, y, z);  cout << "x=" << x << ", y=" << y << ", z=" << z;  return 0;  } | x=2, y=6, z=14 | [Edit & Run](http://www.cplusplus.com/doc/tutorial/functions/) |

To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (&) following the parameter type, as in the parameters taken by duplicate in the example above.  
  
When a variable is passed *by reference*, what is passed is no longer a copy, but the variable itself, the variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call.  
  
   
  
In fact, a, b, and c become aliases of the arguments passed on the function call (x, y, and z) and any change on awithin the function is actually modifying variable x outside the function. Any change on b modifies y, and any change on c modifies z. That is why when, in the example, function duplicate modifies the values of variables a, b, and c, the values of x, y, and z are affected.  
  
If instead of defining duplicate as:

|  |  |  |
| --- | --- | --- |
|  | void duplicate (int& a, int& b, int& c) |  |

Was it to be defined without the ampersand signs as:

|  |  |  |
| --- | --- | --- |
|  | void duplicate (int a, int b, int c) |  |

The variables would not be passed *by reference*, but *by value*, creating instead copies of their values. In this case, the output of the program would have been the values of x, y, and z without being modified (i.e., 1, 3, and 7).

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

/\* function definition to swap the values \*/

void swap(int \*x, int \*y) {

int temp;

temp = \*x; /\* save the value at address x \*/

\*x = \*y; /\* put y into x \*/

\*y = temp; /\* put temp into y \*/

return;

}

Let us now call the function **swap()** by passing values by reference as in the following example −

#include <stdio.h>

/\* function declaration \*/

void swap(int \*x, int \*y);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

printf("Before swap, value of a : %d\n", a );

printf("Before swap, value of b : %d\n", b );

/\* calling a function to swap the values.

\* &a indicates pointer to a ie. address of variable a and

\* &b indicates pointer to b ie. address of variable b.

\*/

swap(&a, &b);

printf("After swap, value of a : %d\n", a );

printf("After swap, value of b : %d\n", b );

return 0;

}

Let us put the above code in a single C file, compile and execute it, to produce the following result −

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

**Topic № 4.**

## Arrays. Declaring Arrays. Accessing Array Elements. Multidimensional Array. Two-dimensional array. Relationship Between Pointers and Arrays.

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



## Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

double balance[10];

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

## Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



## Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

#include <stdio.h>

int main () {

int n[ 10 ]; /\* n is an array of 10 integers \*/

int i,j;

/\* initialize elements of array n to 0 \*/

for ( i = 0; i < 10; i++ ) {

n[ i ] = i + 100; /\* set element at location i to i + 100 \*/

}

/\* output each array element's value \*/

for (j = 0; j < 10; j++ ) {

printf("Element[%d] = %d\n", j, n[j] );

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

## Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

|  |  |
| --- | --- |
| **Sr.No.** | **Concept & Description** |
| 1 | [Multi-dimensional arrays](https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm)  C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| 2 | [Passing arrays to functions](https://www.tutorialspoint.com/cprogramming/c_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 3 | [Return array from a function](https://www.tutorialspoint.com/cprogramming/c_return_arrays_from_function.htm)  C allows a function to return an array. |
| 4 | [Pointer to an array](https://www.tutorialspoint.com/cprogramming/c_pointer_to_an_array.htm)  You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |

**Topic № 5.**

**Passing Arrays to Functions**

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Way-1

Formal parameters as a pointer −

void myFunction(int \*param) {

.

.

.

}

Way-2

Formal parameters as a sized array −

void myFunction(int param[10]) {

.

.

.

}

Way-3

Formal parameters as an unsized array −

void myFunction(int param[]) {

.

.

.

}

Example

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows −

double getAverage(int arr[], int size) {

int i;

double avg;

double sum = 0;

for (i = 0; i < size; ++i) {

sum += arr[i];

}

avg = sum / size;

return avg;

}

Now, let us call the above function as follows −

#include <stdio.h>

/\* function declaration \*/

double getAverage(int arr[], int size);

int main () {

/\* an int array with 5 elements \*/

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

/\* pass pointer to the array as an argument \*/

avg = getAverage( balance, 5 ) ;

/\* output the returned value \*/

printf( "Average value is: %f ", avg );

return 0;

}

When the above code is compiled together and executed, it produces the following result −

Average value is: 214.400000

As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters.

**Topic № 6.**

Recursion

The recursive function is

* + a kind of function that calls itself, or
  + a function that is part of a cycle in the sequence of function calls.

Let’s we want to find the *factorial* of a number: f(*n*) = *n*! We know that

*n*! = 1 \* 2 \* 3 \* … \* (*n* – 1) \* *n*

For example, f(5) = 1 \* 2 \* 3 \* 4 \* 5. We also know that f(4) = 1 \* 2 \* 3 \* 4. So

f(5) = (1 \* 2 \* 3 \* 4) \* 5 = f(4) \* 5

The problem of calculating f(5) is *reduced* to the problem of calculating f(4): in order to find f(5) we first must find f(4) and then multiply the result by 5. This process can be continues like

f(5) = f(4) \* 5 = f(3) \* 4 \* 5 = f(2) \* 3 \* 4 \* 5 = …

How long shall we continue this process? We know that 0! = 1, but there is no sence for calculating factorial for negative numbers. The equality 0! = 1 or f(0) = 1 is called *simple case* or *terminating* *case* or *base case*. When we need to find f(0), we do not continue the reduction like f(0) = f(-1) \* 0 because it has no sence, but simply substitute the value of f(0) by 1. So

f(2) = f(1) \* 2 = f(0) \* 1 \* 2 = 1 \* 1 \* 2 = 2

A recursive function consists of two types of cases:

* *a base case(s)*
* *a recursive case*

The base case is a small problem

* the solution to this problem should not be recursive, so that the function is guaranteed to terminate
* there can be more than one base case

The recursive case defines the problem in terms of a smaller problem of the same type

* the recursive case includes a recursive function call
* there can be more than one recursive case

From the definition of factorial we can conclue that

*n*! = (1 \* 2 \* 3 \* … \* (*n* – 1)) \* *n* = (*n* – 1)! \* *n*

Or if we denoted f(*n*) = *n*! then f(*n*) = f(*n* – 1) \* *n*. This is called *recursive case*. We continue the recursive process till *n* = 0, when 0! = 1. So f(0) = 1. This is called the *base case*.



int fact(int n)

{

if (n == 0) return 1;

return fact(n-1) \* n;

}

In order to calculate *n*! we simply call a function: fact(*n*).

Below we’ll give some recursive functions:

* sum of digits: *sum*(*n*) = 
* number of digits in an integer: *digits*(*n*) = 
* Fibonacci numbers: *fib*(*n*) = : 0, 1, 1, 2, 3, 5,…

int fib(int n)

{

if (n == 0) return 0;

if (n == 1) return 1;

return fib(n-1) + fib(n - 2);

}

* Binomial coefficient:  = 

int Cnk(int n, int k)

{

if (n == k) return 1;

if (k == 0) return 1;

return Cnk(n - 1, k - 1) + Cnk(n - 1, k);

}

* Power:  = 

int f(int x, int n)

{

if (n == 0) return 1;

if (n % 2 == 0) return f(x \* x, n / 2);

return x \* f(x, n - 1);

}

* Greater Common Divisor: GCD(*a*, *b*) = ,

or simplified version GCD(*a*, *b*) = 

int gcd(int a, int b)

{

if (a == 0) return b;

if (b == 0) return a;

if (a >= b) return gcd(a % b, b);

return gcd(a, b % a);

}

or

int gcd(int a, int b)

{

return (b) ? gcd(b,a % b) : a;

}

What do the next functions do (calculate):

Quiz 1

int f(int n)

{

if (n == 0) return 0;

return f(n-1) + n;

}

Quiz 2

int f(int n)

{

if (n == 0) return 0;

return f(n-1) + 1;

}

Quiz 3

int f(int n)

{

if (n == 0) return 1;

return f(n-1) \* 2;

}

Quiz 4

int f(int n)

{

if (n == 0) return 0;

return f(n-1) + 5;

}

What will be printed with the next code

Quiz 5

#include <stdio.h>

void f(int n)

{

if (n == 0) return;

printf("%d ",n);

f(n-1);

}

int main(void)

{

int n;

scanf("%d",&n);

f(n);

return 0;

}

Quiz 6

#include <stdio.h>

void f(int n)

{

if (n == 0) return;

f(n-1);

printf("%d ",n);

}

int main(void)

{

int n;

scanf("%d",&n);

f(n);

return 0;

}

**Topic № 7.**

**Char arrays. Two dimentional arrays.**

**String** is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

**For example:** The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

**Declaring and Initializing a string variables**

There are different ways to initialize a character array variable.

char name[13] = "StudyTonight"; // valid character array initialization

char name[10] = {'L','e','s','s','o','n','s','\0'}; // valid initialization

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

char ch[3] = "hell"; // Illegal

char str[4];

str = "hell"; // Illegal

**String Input and Output**

Input function scanf() can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using scanf() function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

#include<stdio.h>

#include<string.h>

void main()

{

char str[20];

printf("Enter a string");

scanf("%[^\n]", &str); //scanning the whole string, including the white spaces

printf("%s", str);

}

Another method to read character string with white spaces from terminal is by using the gets()function.

char text[20];

gets(text);

printf("%s", text);

**String Handling Functions**

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| strcat() | It is used to concatenate(combine) two strings |
| strlen() | It is used to show length of a string |
| strrev() | It is used to show reverse of a string |
| strcpy() | Copies one string into another |
| strcmp() | It is used to compare two string |

**strcat() function**

strcat("hello", "world");

strcat() function will add the string **"world"** to **"hello"** i.e it will ouput helloworld.

**strlen() function**

strlen() function will return the length of the string passed to it.

int j;

j = strlen("studytonight");

printf("%d",j);

12

**strcmp() function**

strcmp() function will return the ASCII difference between first unmatching character of two strings.

int j;

j = strcmp("study", "tonight");

printf("%d",j);

-1

**strcpy() function**

It copies the second string argument to the first string argument.

#include<stdio.h>

#include<string.h>

int main()

{

char s1[50];

char s2[50];

strcpy(s1, "StudyTonight"); //copies "studytonight" to string s1

strcpy(s2, s1); //copies string s1 to string s2

printf("%s\n", s2);

return(0);

}

StudyTonight

**strrev() function**

It is used to reverse the given string expression.

#include<stdio.h>

int main()

{

char s1[50];

printf("Enter your string: ");

gets(s1);

printf("\nYour reverse string is: %s",strrev(s1));

return(0);

}

**Topic № 8.**

## Searching Arrays.

**Topic № 9**

Often in real life, we are supposed to arrange data in a particular order. For instance, during our school days, we are told to stand in the queue based on our heights. Another example is of the attendance register at school/college which contains our names arranged in the alphabetical order.

These data arrangements give easier access to data for future use for ex. finding “Joe” in an attendance register of 100 students. The arrangement of data in a particular order is called as sorting of the data by that order. 2 of the most commonly used orders are:

* **Ascending order:** while sorting the data in ascending order, we try to arrange the data in a way such that each element is in some way “smaller than” its successor. This “smaller than” relation is an ordered relation over the set from which the data is taken. As a simple example, the numbers 1, 2, 3, 4, 5 are sorted in ascending order. Here, the “smaller than” relation is actually the “<” operator. As can be seen, 1 < 2 < 3 < 4 < 5.
* **Descending order:** descending order is the exact opposite of ascending order. Given a data that is sorted in ascending order, reverse it and you will get the data in descending order.

Due to the similar nature of the 2 orders, we often drop the actual order and we say – we want to sort the data. This generally means that we want the data to be sorted in *ascending* order.

Before we get into the details of the sorting algorithm, let us understand the problem statement.

**Problem statement**

We are given an array (or a list) of data. We are also given a way to “order” the elements present in the data. Now, we are asked to arranged the data as per the given order.

As an example, we are given an array of integers: [5, 1, 4, 2, 3]. We are given the “order” as “smaller than”. So, we are asked to arrange the elements of this array in such a way that each element is smaller than its successor. Basically, we need to find a way to sort this array so that the final array obtained is [1, 2, 3, 4, 5].

There are several techniques/algorithms to achieve this ordered output array. One such well-known technique that we will discuss in this blog is called as Bubble Sort.

**Bubble Sort Algorithm in C – Introduction**

Bubble Sort in C is a sorting algorithm where we repeatedly iterate through the array and swap adjacent elements that are unordered. We repeat this until the array is sorted.

As an example, for the array mentioned above – [5, 1, 4, 2, 3] we can see that 5 should not be on the left of 1 and so, we swap them to get: [1, 5, 4, 2, 3]. Next, we see that 5 should again not be on the left of 4. We swap 5 and 4 to get [1, 4, 5, 2, 3]. We repeat this for 5 and 2 and subsequently for 5 and 3 to get [1, 4, 2, 3, 5].

As can be seen – after one “pass” over the array, the largest element (5 in this case) has reached its correct position – extreme right. Let us try to repeat this process.

(1, 4) is correct. However, (4, 2) is an incorrect order. Therefore, we swap 4 and 2 to get [1, 2, 4, 3, 5]. Now again, (4, 3) is incorrect so we do another swap and get [1, 2, 3, 4, 5].

*As can be seen, the array is sorted!*

This exactly is how bubble sort in C works.

**Bubble Sort – Explanation**

In the first “pass” through the array, the largest element will always get swapped until it is placed to the extreme right. This is because this largest element will always break the desired order. So, at the end of the first pass, the largest element will always reach its correct position.

Now that the largest element has reached its correct position (for instance, 5 reached the last position), we can simply ignore it and concentrate on the rest of the array ([1, 4, 2, 3] in the above case). Here, the largest element in the rest of the array (which is 4) will be nothing but the second largest element in the array. By the above recursive argument, this second largest array will then reach the last position in the remaining array ([1, 2, 3, 4]). This is nothing but a recursive argument on the remaining array.

This continues until for n iterations where n = number of elements in the array. Finally, the array gets sorted.

**Bubble Sort Program in C**

|  |  |
| --- | --- |
|  | #include <stdio.h> |
|  | void bubble\_sort(int a[], int n) { |
|  | int i = 0, j = 0, tmp; |
|  | for (i = 0; i < n; i++) { // loop n times - 1 per element |
|  | for (j = 0; j < n - i - 1; j++) { // last i elements are sorted already |
|  | if (a[j] > a[j + 1]) { // swop if order is broken |
|  | tmp = a[j]; |
|  | a[j] = a[j + 1]; |
|  | a[j + 1] = tmp; |
|  | } |
|  | } |
|  | } |
|  | } |
|  | int main() { |
|  | int a[100], n, i, d, swap; |
|  | printf("Enter number of elements in the array:\n"); |
|  | scanf("%d", &n); |
|  | printf("Enter %d integers\n", n); |
|  | for (i = 0; i < n; i++) |
|  | scanf("%d", &a[i]); |
|  | bubble\_sort(a, n); |
|  | printf("Printing the sorted array:\n"); |
|  | for (i = 0; i < n; i++) |
|  | printf("%d\n", a[i]); |
|  | return 0; |
|  | } |

[**view raw**](https://gist.github.com/hackrio1/20bd70ac3f4b3659baa1175b4cbd29f6/raw/99ab51fd8ff4853ec1bc3e51d478c8e4fbfb4708/bubble-sort.c)[**bubble-sort.c**](https://gist.github.com/hackrio1/20bd70ac3f4b3659baa1175b4cbd29f6#file-bubble-sort-c) hosted with  by **[GitHub](https://github.com/)**

We loop n times – once for each element of the array. When i = 0, with the j loop, the largest element of the array reaches its correct position. When i = 1, with the j loop, the second largest element of the array reaches its correct position. So on and so forth.

**Conclusion**

Bubble sort is a fairly simple algorithm. It forms an interesting example of how simple computations can be used to perform more complex tasks. However, there is one issue with the algorithm – it is relatively slower compared to other sorting algorithms. To understand that, let us take a look at the loops involved – there are 2 loops:

* First, the outer loop of variable i that goes from i = 0 to i = n – 1.
* For each iteration of the outer i loop, the inner loop of variable j goes from j = 0 to j = n – i – 2.

We can consolidate the number of iterations to see that:

* When i = 0, the inner j loop goes from j = 0 to j = n – 2
* When i = 1, the inner j loop goes from j = 0 to j = n – 3
* When i = 2, the inner j loop goes from j = 0 to j = n – 4

..

..

* When i = n – 2, the inner j loop goes from j = 0 to j = 0

We can sum this up to see that the total iterations are (n – 2) + (n – 3) + (n – 4) … + 1 + 0 = (n – 2) \* (n – 3) / 2 = (n2  – 5n + 6) / 2 = n2/2 – 2.5n + 3

As can be seen, this term is proportional to n2 (the largest power of n is n2). Mathematically, this is stated as – bubble sort algorithm is of O(n2) complexity. This isn’t the best because when n is large (say n = 106), n2 is huge (n2 = 1012). Therefore, it will take a lot of iterations for the algorithm to complete. This is undesirable. There are some better algorithms like [merge sort in C](https://hackr.io/blog/merge-sort-in-c), etc that take O(nlog2n) iterations. logn is much smaller than n. As an example, when n = 230 (which is approximately 109), log2n is just 30).

Nevertheless, bubble sort is an interesting algorithm and is a great way for beginners to understand how sorting works.

**Topic**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

|  |
| --- |
| // C program for implementation of selection sort  #include <stdio.h>    void swap(int \*xp, int \*yp)  {      int temp = \*xp;      \*xp = \*yp;      \*yp = temp;  }    void selectionSort(int arr[], int n)  {      int i, j, min\_idx;        // One by one move boundary of unsorted subarray      for (i = 0; i < n-1; i++)      {          // Find the minimum element in unsorted array          min\_idx = i;          for (j = i+1; j < n; j++)            if (arr[j] < arr[min\_idx])              min\_idx = j;            // Swap the found minimum element with the first element          swap(&arr[min\_idx], &arr[i]);      }  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", arr[i]);      printf("\n");  }    // Driver program to test above functions  int main()  {      int arr[] = {64, 25, 12, 22, 11};      int n = sizeof(arr)/sizeof(arr[0]);      selectionSort(arr, n);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Output:

Sorted array:

11 12 22 25 64

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)  
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

**Topic № 10.**

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} book;

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program −

[Live Demo](http://tpcg.io/7DL5Jk)

#include <stdio.h>

#include <string.h>

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main( ) {

struct Books Book1; /\* Declare Book1 of type Book \*/

struct Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info \*/

printf( "Book 1 title : %s\n", Book1.title);

printf( "Book 1 author : %s\n", Book1.author);

printf( "Book 1 subject : %s\n", Book1.subject);

printf( "Book 1 book\_id : %d\n", Book1.book\_id);

/\* print Book2 info \*/

printf( "Book 2 title : %s\n", Book2.title);

printf( "Book 2 author : %s\n", Book2.author);

printf( "Book 2 subject : %s\n", Book2.subject);

printf( "Book 2 book\_id : %d\n", Book2.book\_id);

return 0;

}

When the above code is compiled and executed, it produces the following result −

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book\_id : 6495700

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

[Live Demo](http://tpcg.io/1Kk8Zs)

#include <stdio.h>

#include <string.h>

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

/\* function declaration \*/

void printBook( struct Books book );

int main( ) {

struct Books Book1; /\* Declare Book1 of type Book \*/

struct Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info \*/

printBook( Book1 );

/\* Print Book2 info \*/

printBook( Book2 );

return 0;

}

void printBook( struct Books book ) {

printf( "Book title : %s\n", book.title);

printf( "Book author : %s\n", book.author);

printf( "Book subject : %s\n", book.subject);

printf( "Book book\_id : %d\n", book.book\_id);

}

When the above code is compiled and executed, it produces the following result −

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable −

struct Books \*struct\_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

struct\_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

struct\_pointer->title;

Let us re-write the above example using structure pointer.

[Live Demo](http://tpcg.io/WOoGiV)

#include <stdio.h>

#include <string.h>

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

/\* function declaration \*/

void printBook( struct Books \*book );

int main( ) {

struct Books Book1; /\* Declare Book1 of type Book \*/

struct Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info by passing address of Book1 \*/

printBook( &Book1 );

/\* print Book2 info by passing address of Book2 \*/

printBook( &Book2 );

return 0;

}

void printBook( struct Books \*book ) {

printf( "Book title : %s\n", book->title);

printf( "Book author : %s\n", book->author);

printf( "Book subject : %s\n", book->subject);

printf( "Book book\_id : %d\n", book->book\_id);

}

When the above code is compiled and executed, it produces the following result −

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include −

* Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
* Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

C allows us to do this in a structure definition by putting :bit length after the variable. For example −

struct packed\_struct {

unsigned int f1:1;

unsigned int f2:1;

unsigned int f3:1;

unsigned int f4:1;

unsigned int type:4;

unsigned int my\_int:9;

} pack;

Here, the packed\_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit my\_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

**Topic № 11.**

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

|  |  |  |
| --- | --- | --- |
| **Standard File** | **File Pointer** | **Device** |
| Standard input | stdin | Keyboard |
| Standard output | stdout | Screen |
| Standard error | stderr | Your screen |

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example −

#include <stdio.h>

int main( ) {

int c;

printf( "Enter a value :");

c = getchar( );

printf( "\nYou entered: ");

putchar( c );

return 0;

}

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

$./a.out

**Enter a value :** this is test

**You entered:** t

The gets() and puts() Functions

The **char \*gets(char \*s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char \*s)** function writes the string 's' and 'a' trailing newline to **stdout**.

**NOTE:** Though it has been deprecated to use gets() function, Instead of using gets, you want to use [fgets()](https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm" \o "fgets();).

#include <stdio.h>

int main( ) {

char str[100];

printf( "Enter a value :");

gets( str );

printf( "\nYou entered: ");

puts( str );

return 0;

}

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows −

$./a.out

**Enter a value :** this is test

**You entered:** this is test

The scanf() and printf() Functions

The **int scanf(const char \*format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char \*format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better −

#include <stdio.h>

int main( ) {

char str[100];

int i;

printf( "Enter a value :");

scanf("%s %d", str, &i);

printf( "\nYou entered: %s %d ", str, i);

return 0;

}

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows −

$./a.out

**Enter a value :** seven 7

**You entered:** seven 7

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

**Topic № 23.**

In C programming, file is a place on your physical disk where information is stored.

**Why files are needed?**

* When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
* If you have to enter a large number of data, it will take a lot of time to enter them all.  
  However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
* You can easily move your data from one computer to another without any changes.

**Types of Files**

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

**1. Text files**

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

**2. Binary files**

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

**File Operations**

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

**Working with files**

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE \*fptr;

**Opening a file - for creation and edit**

Opening a file is performed using the [library function](https://www.programiz.com/c-programming/library-function) in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode")

For Example:

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

* Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txtand opens it for writing as per the mode 'w'.  
  The writing mode allows you to create and edit (overwrite) the contents of the file.
* Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'.  
  The reading mode only allows you to read the file, you cannot write into the file.

| Opening Modes in Standard I/O | | |
| --- | --- | --- |
| File Mode | Meaning of Mode | During Inexistence of file |
| r | Open for reading. | If the file does not exist, fopen() returns NULL. |
| rb | Open for reading in binary mode. | If the file does not exist, fopen() returns NULL. |
| w | Open for writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb | Open for writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a | Open for append. i.e, Data is added to end of file. | If the file does not exists, it will be created. |
| ab | Open for append in binary mode. i.e, Data is added to end of file. | If the file does not exists, it will be created. |
| r+ | Open for both reading and writing. | If the file does not exist, fopen() returns NULL. |
| rb+ | Open for both reading and writing in binary mode. | If the file does not exist, fopen() returns NULL. |
| w+ | Open for both reading and writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb+ | Open for both reading and writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a+ | Open for both reading and appending. | If the file does not exists, it will be created. |
| ab+ | Open for both reading and appending in binary mode. | If the file does not exists, it will be created. |

**Closing a File**

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function fclose().

fclose(fptr); //fptr is the file pointer associated with file to be closed.

**Reading and writing to a text file**

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

**Writing to a text file**

**Example 1: Write to a text file using fprintf()**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int num;

FILE \*fptr;

fptr = fopen("C:\\program.txt","w");

if(fptr == NULL)

{

printf("Error!");

exit(1);

}

printf("Enter num: ");

scanf("%d",&num);

fprintf(fptr,"%d",num);

fclose(fptr);

return 0;

}

This program takes a number from user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

**Reading from a text file**

**Example 2: Read from a text file using fscanf()**

#include <stdio.h>

#include <stdlib.h>

int main()

{

int num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.txt","r")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

fscanf(fptr,"%d", &num);

printf("Value of n=%d", num);

fclose(fptr);

return 0;

}

This program reads the integer present in the program.txt file and prints it onto the screen.

If you succesfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in similar way.

**Reading and writing to a binary file**

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

**Writing to a binary file**

To write into a binary file, you need to use the function fwrite(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

fwrite(address\_data,size\_data,numbers\_data,pointer\_to\_file);

**Example 3: Writing to a binary file using fwrite()**

#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","wb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

for(n = 1; n < 5; ++n)

{

num.n1 = n;

num.n2 = 5\*n;

num.n3 = 5\*n + 1;

fwrite(&num, sizeof(struct threeNum), 1, fptr);

}

fclose(fptr);

return 0;

}

In this program, you create a new file program.bin in the C drive.

We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using fwrite.

The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.

Since, we're only inserting one instance of num, the third parameter is 1. And, the last parameter \*fptr points to the file we're storing the data.

Finally, we close the file.

**Reading from a binary file**

Function fread() also take 4 arguments similar to fwrite() function as above.

fread(address\_data,size\_data,numbers\_data,pointer\_to\_file);

**Example 4: Reading from a binary file using fread()**

#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

for(n = 1; n < 5; ++n)

{

fread(&num, sizeof(struct threeNum), 1, fptr);

printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);

}

fclose(fptr);

return 0;

}

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by \*fptr into the structure num.

You'll get the same records you inserted in Example 3.

**Getting data using fseek()**

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().

As the name suggests, fseek() seeks the cursor to the given record in the file.

**Syntax of fseek()**

fseek(FILE \* stream, long int offset, int whence)

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

| Different Whence in fseek | |
| --- | --- |
| Whence | Meaning |
| SEEK\_SET | Starts the offset from the beginning of the file. |
| SEEK\_END | Starts the offset from the end of the file. |
| SEEK\_CUR | Starts the offset from the current location of the cursor in the file. |

**Example of fseek()**

#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

// Moves the cursor to the end of the file

fseek(fptr, -sizeof(struct threeNum), SEEK\_END);

for(n = 1; n < 5; ++n)

{

fread(&num, sizeof(struct threeNum), 1, fptr);

printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);

fseek(fptr, -2\*sizeof(struct threeNum), SEEK\_CUR);

}

fclose(fptr);

return 0;

}

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

Check out these examples to learn more:

**Topic № 12.**

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

**Writing to a binary file**

To write into a binary file, you need to use the function fwrite(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

fwrite(address\_data,size\_data,numbers\_data,pointer\_to\_file);

**Example 3: Writing to a binary file using fwrite()**

#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","wb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

for(n = 1; n < 5; ++n)

{

num.n1 = n;

num.n2 = 5\*n;

num.n3 = 5\*n + 1;

fwrite(&num, sizeof(struct threeNum), 1, fptr);

}

fclose(fptr);

return 0;

}

In this program, you create a new file program.bin in the C drive.

We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using fwrite.

The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.

Since, we're only inserting one instance of num, the third parameter is 1. And, the last parameter \*fptr points to the file we're storing the data.

Finally, we close the file.

**Reading from a binary file**

Function fread() also take 4 arguments similar to fwrite() function as above.

fread(address\_data,size\_data,numbers\_data,pointer\_to\_file);

**Example 4: Reading from a binary file using fread()**

#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

for(n = 1; n < 5; ++n)

{

fread(&num, sizeof(struct threeNum), 1, fptr);

printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);

}

fclose(fptr);

return 0;

}

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by \*fptr into the structure num.

You'll get the same records you inserted in Example 3.

**Getting data using fseek()**

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().

As the name suggests, fseek() seeks the cursor to the given record in the file.

**Syntax of fseek()**

fseek(FILE \* stream, long int offset, int whence)

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

| Different Whence in fseek | |
| --- | --- |
| Whence | Meaning |
| SEEK\_SET | Starts the offset from the beginning of the file. |
| SEEK\_END | Starts the offset from the end of the file. |
| SEEK\_CUR | Starts the offset from the current location of the cursor in the file. |

**Example of fseek()**

#include <stdio.h>

#include <stdlib.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

// Moves the cursor to the end of the file

fseek(fptr, -sizeof(struct threeNum), SEEK\_END);

for(n = 1; n < 5; ++n)

{

fread(&num, sizeof(struct threeNum), 1, fptr);

printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);

fseek(fptr, -2\*sizeof(struct threeNum), SEEK\_CUR);

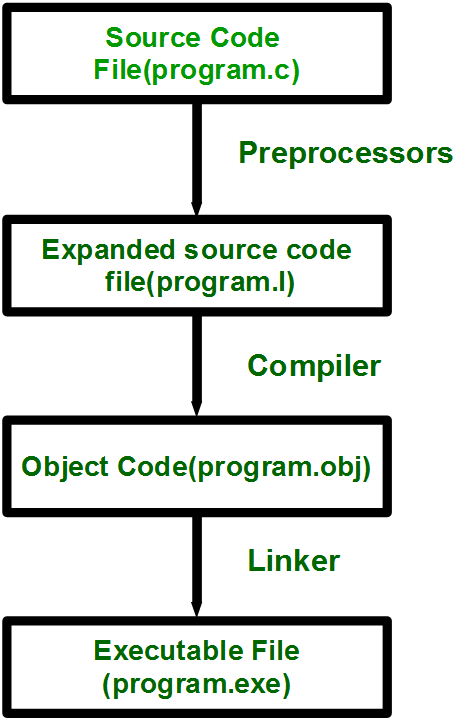
}

fclose(fptr);

return 0;

}

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

these steps before we actually start learning about Preprocessors.  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/preprocessors.png)

You can see the intermediate steps in the above diagram. The source code written by programmers is stored in the file program.c. This file is then processed by preprocessors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program.obj . Finally the linker links this object code file to the object code of the library functions to generate the executable file program.exe .

Preprocessor programs provides preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directive begins with a ‘#’ (hash) symbol. This (‘#’) symbol at the beginning of a statement in a C/C++ program indicates that it is a pre-processor directive. We can place these pre processor directives anywhere in our program. Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef* etc.

**There are 4 main types of preprocessor directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

Let us now learn about each of these directives in details.

* **Macros**: Macros are piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The ‘#define’ directive is used to define a macro. Let us now understand macro definition with the help of a program:

|  |
| --- |
| #include <iostream>    // macro definition  #define LIMIT 5  int main()  {      for (int i = 0; i < LIMIT; i++) {          std::cout << i << "\n";      }        return 0;  } |

* Copy CodeRun on IDE
* Output:
* 0
* 1
* 2
* 3
* 4
* In the above program, when the compiler executes the word LIMIT it replaces it with 5. The word ‘LIMIT’ in macro definition is called macro template and ‘5’ is macro expansion.  
  **Note**: There is no semi-colon(‘;’) at the end of macro definition. Macro definitions do not need a semi-colon to end.
* **Macros with arguments**: We can also pass arguments to macros. Macros defined with arguments works similarly as functions. Let us understand this with a program:

|  |
| --- |
| #include <iostream>    // macro with parameter  #define AREA(l, b) (l \* b)  int main()  {      int l1 = 10, l2 = 5, area;        area = AREA(l1, l2);        std::cout << "Area of rectangle is: " << area;        return 0;  } |

* Copy CodeRun on IDE
* Output:
* Area of rectangle is: 50
* We can see from the above program that whenever the compiler finds AREA(l, b) in the program it replaces it with the statement (l\*b) . Not only this, the values passed to the macro template AREA(l, b) will also be replaced in the statement (l\*b). Therefore AREA(10, 5) will be equal to 10\*5.
* **File Inclusion**: This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files which can be included by the user in the program:
  1. **Header File or Standard files**: These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O funuctions are in ‘iostream’ file whereas functions which perform string operations are in ‘string’ file.  
     **Syntax**:
  2. #include< *file\_name* >

where *file\_name* is the name of file to be included. The ‘<‘ and ‘>’ brackets tells the compiler to look for the file in standard directory.

* 1. **user defined files**: When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:
  2. #include"*filename*"
* **Conditional Compilation**: Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions. This can be done with the help of two preprocessing commands ‘**ifdef**‘ and ‘**endif**‘.  
  **Syntax**:
* ifdef macro\_name
* statement1;
* statement2;
* statement3;
* .
* .
* .
* statementN;
* endif

If the macro with name as ‘*macroname*‘ is defined then the block of statements will execute normally but if it is not defined, the compiler will simply skip this block of statements.

* **Other directives**: Apart from the above directives there are two more directives which are not commonly used. These are:
  1. **#undef Directive**: The #undef directive is used to undefine an existing macro. This directive works as:
  2. #undef LIMIT

Using this statement will undefine the existing macro LIMIT. After this statement every “#ifdef LIMIT” statement will evaluate to false.

* 1. **#pragma Directive**: This directive is a special purpose directive and is used to turn on or off some features. This type of directives are compiler-specific i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:
     + **#pragma startup** and **#pragma exit**: These directives helps us to specify the functions that are needed to run before program startup( before the control passes to main()) and just before program exit (just before the control returns from main()).  
       **Note:** Below program will not work with GCC compilers.  
       Look at the below program:

|  |
| --- |
| #include <stdio.h>    void func1();  void func2();    #pragma startup func1  #pragma exit func2    void func1()  {      printf("Inside func1()\n");  }    void func2()  {      printf("Inside func2()\n");  }    int main()  {      void func1();      void func2();      printf("Inside main()\n");        return 0;  } |

* + - Copy CodeRun on IDE
    - Output:
    - Inside func1()
    - Inside main()
    - Inside func2()
    - The above code will produce the output as given below when run on GCC compilers:
    - Inside main()
    - This happens because GCC does not supports #pragma startup or exit. However you can use the below code for a similar output on GCC compilers.

|  |
| --- |
| #include <stdio.h>    void func1();  void func2();    void \_\_attribute\_\_((constructor)) func1();  void \_\_attribute\_\_((destructor)) func2();    void func1()  {      printf("Inside func1()\n");  }    void func2()  {      printf("Inside func2()\n");  }    int main()  {      printf("Inside main()\n");        return 0;  } |

* + - Copy CodeRun on IDE
    - **#pragma warn Directive:** This directive is used to hide the warning message which are displayed during compilation.  
      We can hide the warnings as shown below:
      * **#pragma warn -rvl**: This directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.
      * **#pragma warn -par**: This directive hides those warning which are raised when a function does not uses the parameters passed to it.
      * **#pragma warn -rch**: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the *return* statement in a function is unreachable.

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the preprocessing directive **#include**. It has the following two forms −

#include <file>

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

#include "file"

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as follows −

char \*test (void);

and a main program called *program.c* that uses the header file, like this −

int x;

#include "header.h"

int main (void) {

puts (test ());

}

the compiler will see the same token stream as it would if program.c read.

int x;

char \*test (void);

int main (void) {

puts (test ());

}

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this −

#ifndef HEADER\_FILE

#define HEADER\_FILE

the entire header file file

#endif

This construct is commonly known as a wrapper **#ifndef**. When the header is included again, the conditional will be false, because HEADER\_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes

Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows −

#if SYSTEM\_1

# include "system\_1.h"

#elif SYSTEM\_2

# include "system\_2.h"

#elif SYSTEM\_3

...

#endif

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a **computed include**. Instead of writing a header name as the direct argument of **#include**, you simply put a macro name there −

#define SYSTEM\_H "system\_1.h"

...

#include SYSTEM\_H

SYSTEM\_H will be expanded, and the preprocessor will look for system\_1.h as if the **#include** had been written that way originally. SYSTEM\_H could be defined by your Makefile with a -D option.

**Topic № 27.**

This lecture explains dynamic memory management in C. The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

|  |  |
| --- | --- |
| **Sr.No.** | **Function & Description** |
| 1 | **void \*calloc(int num, int size);**  This function allocates an array of **num** elements each of which size in bytes will be **size**. |
| 2 | **void free(void \*address);**  This function releases a block of memory block specified by address. |
| 3 | **void \*malloc(int num);**  This function allocates an array of **num** bytes and leave them uninitialized. |
| 4 | **void \*realloc(void \*address, int newsize);**  This function re-allocates memory extending it upto **newsize**. |

Allocating Memory Dynamically

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows −

char name[100];

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below example −

[Live Demo](http://tpcg.io/osfk0O)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* allocate memory dynamically \*/

description = malloc( 200 \* sizeof(char) );

if( description == NULL ) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcpy( description, "Zara ali a DPS student in class 10th");

}

printf("Name = %s\n", name );

printf("Description: %s\n", description );

}

When the above code is compiled and executed, it produces the following result.

Name = Zara Ali

Description: Zara ali a DPS student in class 10th

Same program can be written using **calloc();** only thing is you need to replace malloc with calloc as follows −

calloc(200, sizeof(char));

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of realloc() and free() functions −

[Live Demo](http://tpcg.io/mQPTlp)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char name[100];

char \*description;

strcpy(name, "Zara Ali");

/\* allocate memory dynamically \*/

description = malloc( 30 \* sizeof(char) );

if( description == NULL ) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcpy( description, "Zara ali a DPS student.");

}

/\* suppose you want to store bigger description \*/

description = realloc( description, 100 \* sizeof(char) );

if( description == NULL ) {

fprintf(stderr, "Error - unable to allocate required memory\n");

} else {

strcat( description, "She is in class 10th");

}

printf("Name = %s\n", name );

printf("Description: %s\n", description );

/\* release memory using free() function \*/

free(description);

}

When the above code is compiled and executed, it produces the following result.

Name = Zara Ali

Description: Zara ali a DPS student.She is in class 10th

You can try the above example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.

**Topic № 28,29**

Fibonacci numbers

The Fibonacci sequence is named after Italian mathematician Leonardo of Pisa, known as Fibonacci:

The Fibonacci numbers *fn* = f(*n*) are the numbers characterized by the fact that every number after the first two is the sum of the two preceding ones. They are defined with the next recurrent relation:



So *f*0 = 0, *f*1 = 1, *fn* = *fn*-1 + *fn*-2.

The Fibonacci sequence has the form

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Example. Fill integer array *fib* with Fibonacci numbers (fib[*i*] = *fi*):

#include <stdio.h>

int i, n, fib[47];

int main(void)

{

scanf("%d",&n);

fib[0] = 0; fib[1] = 1;

for(i = 2; i <= n; i++)

fib[i] = fib[i-1] + fib[i-2];

printf("%d\n",fib[n]);

return 0;

}

The biggest Fibonacci number that fits into int type is

*f*46 = 1836311903

The biggest Fibonacci number that fits into long long type is

*f*92 = 7540113804746346429

If you want to find Fibonacci number *fn* for *n* > 92, use BigInteger type.

Example. Find f(*n*) – the *n*-th Fibonacci number with recursion:

#include <stdio.h>

int n;

int fib(int n)

{

if (n == 0) return 0;

if (n == 1) return 1;

return fib(n-1) + fib(n - 2);

}

int main(void)

{

scanf("%d",&n);

printf("%d\n",fib(n));

return 0;

}

Euclidean algorithms (Basic and Extended)

GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common factors.

* If we subtract smaller number from larger (we reduce larger number), GCD doesn’t change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
* Now instead of subtraction, if we divide smaller number, the algorithm stops when we find remainder 0.

Below is a recursive function to evaluate gcd using Euclid’s algorithm.

|  |
| --- |
| // C++ program to demonstrate  // Basic Euclidean Algorithm  #include <bits/stdc++.h>  using namespace std;    // Function to return  // gcd of a and b  int gcd(int a, int b)  {      if (a == 0)          return b;      return gcd(b % a, a);  }    // Driver Code  int main()  {      int a = 10, b = 15;      cout << "GCD(" << a << ", "           << b << ") = " << gcd(a, b)                          << endl;      a = 35, b = 10;      cout << "GCD(" << a << ", "           << b << ") = " << gcd(a, b)                          << endl;      a = 31, b = 2;      cout << "GCD(" << a << ", "           << b << ") = " << gcd(a, b)                          << endl;      return 0;  }    // This code is contributed  // by Nimit Garg |

Copy CodeRun on IDE

**Output :**

GCD(10, 15) = 5

GCD(35, 10) = 5

GCD(31, 2) = 1

**Time Complexity:** O(Log min(a, b))

The extended Euclidean algorithm updates results of gcd(a, b) using the results calculated by recursive call gcd(b%a, a). Let values of x and y calculated by the recursive call be x1 and y1. x and y are updated using below expressions.

x = y1 - ⌊b/a⌋ \* x1

y = x1

|  |
| --- |
| // C program to demonstrate working of extended  // Euclidean Algorithm  #include <stdio.h>    // C function for extended Euclidean Algorithm  int gcdExtended(int a, int b, int \*x, int \*y)  {      // Base Case      if (a == 0)      {          \*x = 0;          \*y = 1;          return b;      }        int x1, y1; // To store results of recursive call      int gcd = gcdExtended(b%a, a, &x1, &y1);        // Update x and y using results of recursive      // call      \*x = y1 - (b/a) \* x1;      \*y = x1;        return gcd;  }    // Driver Program  int main()  {      int x, y;      int a = 35, b = 15;      int g = gcdExtended(a, b, &x, &y);      printf("gcd(%d, %d) = %d", a, b, g);      return 0;  } |

The time() function is defined in time.h (ctime in C++) header file. This function returns the time since 00:00:00 UTC, January 1, 1970 (Unix timestamp) in seconds. If second is not a null pointer, the returned value is also stored in the object pointed to by second.

**Syntax:**

time\_t time( time\_t \*second )

**Parameter:** This function accepts single parameter second. This parameter is used to set the time\_t object which store the time.

**Return Value:** This function returns current calender time as a object of type time\_t.

The **time.h** header file contains definitions of functions to get and manipulate date and time information.

* It describes three time related **datatypes**.
  1. **clock\_t**: clock\_t represents the date as integer which is a part of the calendar time.
  2. **time\_t**: time\_t represents the clock time as integer which is a part of the calendar time.

**Topic № 13.**

Introduction to Object Technology.

Object-oriented programming (OOP)

Basic Object Technology Concepts.

Object-oriented programming (OOP) is a software programming model constructed around objects. This model compartmentalizes data into objects (data fields) and describes object contents and behavior through the declaration of classes (methods).

OOP features include the following:

* Encapsulation: This makes the program structure easier to manage because each object’s implementation and state are hidden behind well-defined boundaries.
* Polymorphism: This means abstract entities are implemented in multiple ways.
* Inheritance: This refers to the hierarchical arrangement of implementation fragments.

Object-oriented programming allows for simplified programming. Its benefits include reusability, refactoring, extensibility, maintenance and efficiency.

OOP has been the programming model of choice for the last decade or more. OOP's modular design enables programmers to build software in manageable chunks rather than in large amounts of sequential code.

One of the great benefits of OOP is that of scalability, with objects and definitions having no finite limitation. Also, the separation of data from method prevents a common problem found in older linear software languages. If a bug appears in a linear code, it can be translated through a system and create masses of hard-to-trace errors. Conversely, an OOP program, with its separation of method and data, is not susceptible to such proliferated errors.

Popular OOP languages include Java, the C-family of languages,VB.NETand Python.

So-called "pure" OOP languages include Scala, Ruby, Eiffel, JADE, Smalltalk and Emerald.

Encapsulation, in the context of C#, refers to an object's ability to hide data and behavior that are not necessary to its user. Encapsulation enables a group of properties, methods and other members to be considered a single unit or object.  
  
The following are the benefits of encapsulation:

* Protection of data from accidental corruption
* Specification of the accessibility of each of the members of a class to the code outside the class
* Flexibility and extensibility of the code and reduction in complexity
* Lower coupling between objects and hence improvement in code maintainability

Encapsulation is used to restrict access to the members of a class so as to prevent the user of a given class from manipulating objects in ways that are not intended by the designer. While encapsulation hides the internal implementation of the functionalities of class without affecting the overall functioning of the system, it allows the class to service a request for functionality and add or modify its internal structure (data or methods) to suit changing requirements.  
  
Encapsulation is also known as information hiding.

Encapsulation in C# is implemented with different levels of access to object data that can be specified using the following access modifiers:

* Public: Access to all code in the program
* Private: Access to only members of the same class
* Protected: Access to members of same class and its derived classes
* Internal: Access to current assembly
* Protected Internal: Access to current assembly and types derived from containing class

Encapsulation can be illustrated with an example of an Employee object that stores details of that object. By using encapsulation, the Employee object can expose the data (like Name, EmployeeID, etc.) and methods (like GetSalary) necessary for using the object, while hiding its irrelevant fields and methods from other objects. It is easy to see a situation in which all users could access basic information about an employee while restricting salary information.   
  
C# allows encapsulation of data through the use of accessors (to get data) and mutators (to modify data), which help in manipulating private data indirectly without making it public. Properties are an alternate mechanism for private data to be encapsulated in a C# object and accessed in either read-only mode or in read-write mode. Unlike the accessor and mutator, a property provides a single point of access to an object's "set" and "get" values.

*This definition was written in the context of C#*

Polymorphism, in C#, is the ability of objects of different types to provide a unique interface for different implementations of methods. It is usually used in the context of late binding, where the behavior of an object to respond to a call to its method members is determined based on object type at run time. Polymorphism enables redefining methods in derived classes.   
  
Polymorphism forms one of the fundamental concepts of object-oriented programming, along with encapsulation and inheritance.

Method overloading, constructor overloading and operator overloading are considered compile-time (also called static or ad-hoc) polymorphism, or early binding. Method overriding, which involves inheritance and virtual functions, is called runtime (also called dynamic, inclusion, or subtyping) polymorphism, or late binding. In the case of compile-time polymorphism, identification of the overloaded method to be executed is carried out at compile time. However, in runtime polymorphism, the type of the object from which the overridden method will be called is identified at run time.  
  
In C#, polymorphism is implemented through inheritance and the use of the keyword "virtual". Derived classes inherit the base class members, except constructors, based on their accessibility levels. Hence, the compiler generates the code to check and identify the correct object type (that is pointed to by the reference type) at runtime and the appropriate method to be called.   
  
An example of polymorphism is an employee base class, which includes all the basic details about employees. Classes such as clerk and manager could inherit from the employee base

class with specific implementations (by overriding virtual methods) wherever necessary in the derived classes.

Inheritance is a mechanism wherein a new class is derived from an existing class. In Java, classes may inherit or acquire the properties and methods of other classes.

A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass. A subclass can have only one superclass, whereas a superclass may have one or more subclasses.

Inheritance is the process wherein characteristics are inherited from ancestors. Similarly, in Java, a subclass inherits the characteristics (properties and methods) of its superclass (ancestor). For example, a vehicle is a superclass and a car is a subclass. The car (subclass) inherits all of the vehicle’s properties. The inheritance mechanism is very useful in code reuse. The following are some limitations of Java class inheritance: A subclass cannot inherit private members of its superclass. Constructor and initializer blocks cannot be inherited by a subclass. A subclass can have only one superclass.

The keyword “extends” is used to derive a subclass from the superclass, as illustrated by the following syntax: class Name\_of\_subclass extends Name\_of superclass { //new fields and methods that would define the subclass go here } If you want to derive a subclass Rectangle from a superclass Shapes, you can do it as follows: class Rectangle extends Shapes { …. }

Object-oriented programming language (OOPL) is a high-level programming language based on the object-oriented programming (OOP) model.

OOPL incorporates logical classes, objects, methods, relationships and other processes with the design of software and applications. The first OOPL was Simula, a simulation creation tool developed in 1960.

Unlike conventional procedural languages, the programming syntax of object-oriented programming language is based on one or more objects, whereas procedural language incorporates logical procedures. In OOPL, objects interact with each other; have their own methods, procedures and functions; are part of a class and may be reused in one or more program. An OOPL must exhibit native object-oriented functions, including data abstraction, inheritance, encapsulation, class creation and associated objects.

Most modern programming languages are object-oriented or support the OOP model to an extent. Popular OOPLs include Java, C++, Python and SmallTalk.

**Topic № 14.**

**Classes. Classes and Objects.**

Class Scope and Accessing Class Members.

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

## C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows −

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object.

## Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

## Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear −

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

# C++ Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base**class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total area: 35

Encapsulation, in the context of C#, refers to an object's ability to hide data and behavior that are not necessary to its user. Encapsulation enables a group of properties, methods and other members to be considered a single unit or object.  
  
The following are the benefits of encapsulation:

* Protection of data from accidental corruption
* Specification of the accessibility of each of the members of a class to the code outside the class
* Flexibility and extensibility of the code and reduction in complexity
* Lower coupling between objects and hence improvement in code maintainability

Encapsulation is used to restrict access to the members of a class so as to prevent the user of a given class from manipulating objects in ways that are not intended by the designer. While encapsulation hides the internal implementation of the functionalities of class without affecting the overall functioning of the system, it allows the class to service a request for functionality and add or modify its internal structure (data or methods) to suit changing requirements.  
Encapsulation is also known as information hiding.

Encapsulation in C# is implemented with different levels of access to object data that can be specified using the following access modifiers:

* Public: Access to all code in the program
* Private: Access to only members of the same class
* Protected: Access to members of same class and its derived classes
* Internal: Access to current assembly
* Protected Internal: Access to current assembly and types derived from containing class

Encapsulation can be illustrated with an example of an Employee object that stores details of that object. By using encapsulation, the Employee object can expose the data (like Name, EmployeeID, etc.) and methods (like GetSalary) necessary for using the object, while hiding its irrelevant fields and methods from other objects. It is easy to see a situation in which all users could access basic information about an employee while restricting salary information.   
C# allows encapsulation of data through the use of accessors (to get data) and mutators (to modify data), which help in manipulating private data indirectly without making it public. Properties are an alternate mechanism for private data to be encapsulated in a C# object and accessed in either read-only mode or in read-write mode. Unlike the accessor and mutator, a property provides a single point of access to an object's "set" and "get" values.

with successive layers of control information before transmission across a network. The reverse of data encapsulation is decapsulation, which refers to the successive layers of data being removed (essentially unwrapped) at the receiving end of a network.

When a network device sends a message, the message will take the form of a packet. Each OSI (open system interconnection) model layer adds a header to the packet. The packet is then covered with some information directing it onward to a destination; this is analogous to the address on a letter in which the actual message is carried inside the envelope. Similarly, the message in the packet is encapsulated with some information such as the address of next node, protocol information, the type of data and the source and destination addresses.

Decapsulation is the process of opening up encapsulated data that are usually sent in the form of packets over a communication network. It can be literally defined as the process of opening a capsule, which, in this case, refers to encapsulated or wrapped-up data.

Access modifiers are keywords used to specify the accessibility of a class (or type) and its members. These modifiers can be used from code inside or outside the current application.   
  
Access modifiers in .NET are used to control the accessibility of each of the members of a type from different possible areas of code. This can be handled from within the current assembly or outside it. An assembly represents a logical unit of functionality and consists of types and resources located in one or more files.   
  
The purpose of using access modifiers is to implement encapsulation, which separates the interface of a type from its implementation. With this, the following benefits can be derived:

* Prevention of access to the internal data set by users to invalid state.
* Provision for changes to internal implementation of the types without affecting the components using it.
* Reduction in complexity of the system by reducing the interdependencies between software components.

The .NET framework provides an option of having five types of access modifiers:

1. Private – code within the type can only access the members of that type, and hence accessibility is limited to current type
2. Public – code from anywhere within the current assembly, or another assembly that references it, can access the members of the type, and hence allows accessibility from anywhere
3. Protected – code within the type, or its derived classes, can access the members of the type and hence accessibility is limited to current type and derived classes
4. Internal – code in the current assembly, but not from another assembly, can access the members of the type, hence accessibility is limited to current assembly
5. Protected Internal – code in the current assembly can access the members of the type and also from the assembly that references it. Hence, accessibility is from derived classes in the current assembly, and must take place through an instance of derived class type in the assembly referencing it.

There are several rules that apply to the access modifiers:

* When there is no access modifier specified to type members, the default access level is private and internal.
* There are no access modifiers allowed for namespaces, since they are public.
* The nested classes and struct members declared within a type are, to the containing class, private by default.
* Struct members cannot be declared protected since it does not support inheritance.
* Destructors cannot have access modifiers.
* Derived type cannot have greater accessibility than its base type.
* The member of a containing type should have accessibility lesser than that of its containing type. This can be illustrated with an example: A public method in a containing type cannot have “A” as a parameter, if type A is not in public visibility.
* Interfaces are declared public and internal, and cannot have other access modifiers, since interfaces are mainly used for access by classes to derive from it.
* Access modifiers are used not only to class members, but also to other code constructs with the same intention.

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

* **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private**members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
* **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
* **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private**members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax −

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

#include <iostream>

using namespace std;

// Base class Shape

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Base class PaintCost

class PaintCost {

public:

int getCost(int area) {

return area \* 70;

}

};

// Derived class

class Rectangle: public Shape, public PaintCost {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

// Print the total cost of painting

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total area: 35

Total paint cost: $2450

# C++ Overloading (Operator and Function)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

## Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types −

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

When the above code is compiled and executed, it produces the following result −

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

Polymorphism, in C++, is the ability of objects of different types to provide a unique interface for different implementations of methods. It is usually used in the context of late binding, where the behavior of an object to respond to a call to its method members is determined based on object type at run time. Polymorphism enables redefining methods in derived classes.   
  
Polymorphism forms one of the fundamental concepts of object-oriented programming, along with encapsulation and inheritance.

Method overloading, constructor overloading and operator overloading are considered compile-time (also called static or ad-hoc) polymorphism, or early binding. Method overriding, which involves inheritance and virtual functions, is called runtime (also called dynamic, inclusion, or subtyping) polymorphism, or late binding. In the case of compile-time polymorphism, identification of the overloaded method to be executed is carried out at compile time. However, in runtime polymorphism, the type of the object from which the overridden method will be called is identified at run time.  
  
In C#, polymorphism is implemented through inheritance and the use of the keyword "virtual". Derived classes inherit the base class members, except constructors, based on their accessibility levels. Hence, the compiler generates the code to check and identify the correct object type (that is pointed to by the reference type) at runtime and the appropriate method to be called.   
  
An example of polymorphism is an employee base class, which includes all the basic details about employees. Classes such as clerk and manager could inherit from the employee base class with specific implementations (by overriding virtual methods) wherever necessary in the derived classes.

## Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below −

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

## Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | || |
| += | -= | /= | %= | ^= | &= |
| |= | \*= | <<= | >>= | [] | () |
| -> | ->\* | new | new [] | Delete | delete [] |

# Polymorphism in C++

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0){

width = a;

height = b;

}

int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape {

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// Main function for the program

int main() {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

When the above code is compiled and executed, it produces the following result −

Parent class area :

Parent class area :

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this −

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

virtual int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

Rectangle class area

Triangle class area

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0) {

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

# Data Abstraction in C++

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

#include <iostream>

using namespace std;

int main() {

cout << "Hello C++" <<endl;

return 0;

}

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels −

* Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
* Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages −

* Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
* The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example −

#include <iostream>

using namespace std;

class Adder {

public:

// constructor

Adder(int i = 0) {

total = i;

}

// interface to outside world

void addNum(int number) {

total += number;

}

// interface to outside world

int getTotal() {

return total;

};

private:

// hidden data from outside world

int total;

};

int main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total 60

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Designing Strategy

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

# Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements −

* **Program statements (code)** − This is the part of a program that performs actions and they are called functions.
* **Program data** − The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected**and **public** members. By default, all items defined in a class are private. For example −

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

#include <iostream>

using namespace std;

class Adder {

public:

// constructor

Adder(int i = 0) {

total = i;

}

// interface to outside world

void addNum(int number) {

total += number;

}

// interface to outside world

int getTotal() {

return total;

};

private:

// hidden data from outside world

int total;

};

int main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Designing Strategy

Most of us have learnt to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This is applied most frequently to data members, but it applies equally to all members, including virtual functions.

**Topic № 15.**

**Constructors and Destructors.**

## Class Constructor

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor −

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(); // This is the constructor

private:

double length;

};

// Member functions definitions including constructor

Line::Line(void) {

cout << "Object is being created" << endl;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main() {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Object is being created

Length of line : 6

## Parameterized Constructor

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example −

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(double len); // This is the constructor

private:

double length;

};

// Member functions definitions including constructor

Line::Line( double len) {

cout << "Object is being created, length = " << len << endl;

length = len;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main() {

Line line(10.0);

// get initially set length.

cout << "Length of line : " << line.getLength() <<endl;

// set line length again

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Object is being created, length = 10

Length of line : 10

Length of line : 6

## Using Initialization Lists to Initialize Fields

In case of parameterized constructor, you can use following syntax to initialize the fields −

Line::Line( double len): length(len) {

cout << "Object is being created, length = " << len << endl;

}

Above syntax is equal to the following syntax −

Line::Line( double len) {

cout << "Object is being created, length = " << len << endl;

length = len;

}

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows −

C::C( double a, double b, double c): X(a), Y(b), Z(c) {

....

}

## The Class Destructor

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor −

#include <iostream>

using namespace std;

class Line {

public:

void setLength( double len );

double getLength( void );

Line(); // This is the constructor declaration

~Line(); // This is the destructor: declaration

private:

double length;

};

// Member functions definitions including constructor

Line::Line(void) {

cout << "Object is being created" << endl;

}

Line::~Line(void) {

cout << "Object is being deleted" << endl;

}

void Line::setLength( double len ) {

length = len;

}

double Line::getLength( void ) {

return length;

}

// Main function for the program

int main() {

Line line;

// set line length

line.setLength(6.0);

cout << "Length of line : " << line.getLength() <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Object is being created

Length of line : 6

Object is being deleted