1. **Basics of C++**

**Getting Started**

A computer cannot understand our language that we use in our day to day conversations, and likewise, we cannot understand the binary language that the computer uses to do its tasks. It is therefore necessary for us to write instructions in some specially defined language like C++ which is like natural language and after converting with the help of compiler the computer can understand it.

**C++ Compiler**

A C++ compiler is itself a computer program which’s only job is to convert the C++ program from our form to a form the computer can read and execute. The original C++ program is called the **“source code”**, and the resulting compiled code produced by the compiler is usually called an **“object file”**.

Before compilation the **preprocessor** performs preliminary operations on C++ source files. Preprocessed form of the source code is sent to compiler.

After compilation stage object files are combined with predefined libraries by a **linker**, sometimes called a binder, to produce the final complete file that can be executed by the computer. A library is a collection of pre-compiled “object code” that provides operations that are done repeatedly by many computer programs.



**First program is C++**

*// This is my first program is C++*

*/\* this program will illustrate different components of*

*a simple program in C++ \*/*

#include <iostream>

using namespace std;

int main()

{

cout << "Hello World!";

return 0;

}

When the above program is compiled, linked and executed, the following output is displayed on the VDU screen.

Hello World!

Various components of this program are discussed below:

**Comments**

First three lines of the above program are comments and are ignored by the compiler. Comments are included in a program to make it more readable. If a comment is short and can be accommodated in a single line, then it is started with double slash sequence in the first line of the program. However, if there are multiple lines in a comment, it is enclosed between the two symbols /\* and \*/.

**#include <iostream>**

The line in the above program that start with # symbol are called directives and are instructions to the compiler. The word include with '#' tells the compiler to include the file iostream into the file of the above program. File iostream is a header file needed for input/ output requirements of the program. Therefore, this file has been included at the top of the program.

**using namespace std;**

All the elements of the standard C++ library are declared within std. This line is very frequent in C++ programs that use the standard library.

**int main ( )**

The word main is a function name. The brackets ( ) with main tells that main ( ) is a function. The word int before main ( ) indicates that integer value is being returned by the function main (). When program is loaded in the memory, the control is handed over to function main ( ) and it is the first function to be executed.

**Curly bracket and body of the function main ( )**

A C++ program starts with function called main(). The body of the function is enclosed between curly braces. The program statements are written within the brackets. Each statement must end by a semicolon, without which an error message in generated.

**cout<<"Hello World!";**

This statement prints our "Hello World!" message on the screen. cout understands that anything sent to it via the << operator should be printed on the screen.

**return 0;**

This is a new type of statement, called a return statement. When a program finishes running, it sends a value to the operating system. This particular return statement returns the value of 0 to the operating system, which means “everything went okay!”.

**Printing Multiple Lines of Text with a Single Statement**

*/\* This program illustrates how to print multiple lines of text*

*with a single statement \*/*

#include <iostream>

using namespace std;

int main()

{

cout << "Welcome\nto\nC++";

return 0;

}

**Output:**

Welcome to C++

The characters print exactly as they appear between the double quotes. However, if we type \n, the characters \n are not printed on the screen. The backslash (\) is called an **escape character**. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an **escape sequence**. The escape sequence \n means **newline**. It causes the cursor to move to the beginning of the next line on the screen.

The following table gives a listing of common escape sequences.

|  |  |
| --- | --- |
| **Escape Sequence** | **Description** |
| \n | Newline |
| \t | Horizontal tab |
| \a | Bell (beep) |
| \\ | Backslash |
| \' | Single quote |
| \'' | Double quote |

**Variable: Memory Concept**

Programs shown in the previous section print text on the screen. This section will introduce the concept of variable so that our program can perform calculation on data.

**Program: Adding two numbers**

We'll solve this problem in C++ with the following steps:

**STEP 1:** Allocate memory for storing three numbers.

**STEP 2:** Store first number in computer memory.

**STEP 3:** Store second number in computer memory.

**STEP 4:** Add these two numbers together and store the result of the addition in a third memory location.

**STEP 5:** Print the result.

**STEP 1**: Now first we'll allocate memory for storing numbers. Location of the computer memory which is used to store data and is given a symbolic name for reference is known as variable. We need three variables, two for storing input and third for storing result. Before a variable is used in a program, we must declare it. This activity enables the compiler to make available the appropriate type of location in the memory.

Following statements declare three variables of type integer to store whole numbers.

int x;

int y;

int z;

You can declare more than one variable of same type in a single statement like:

int x, y, z;

**STEP 2:** Following statement stores value in first variable.

x = 25;

**STEP 3:** Following statement stores value in second variable.

y = 10;

**STEP 4:** Now, add these two numbers together and store the result of the addition in third variable.

z = x + y;

**STEP 5:** Print the result.

cout << "The sum is ";

cout << sum;

You can combine above two statements in one statement

cout << "The sum is " << sum;

here, is the complete program:

#include <iostream>

using namespace std;

int main()

{

*//declare variables of integer type*

int x;

int y;

int z;

*//storing value in variables*

x = 25;

y = 10;

*//adding numbers and store the result in sum*

z = x + y;

*//print the result*

cout << "The sum is ";

cout << z; return 0;

}

**Output:**

The sum is 35

**Identifiers**

Symbolic names can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. In previous program x, y and z are identifiers of variables. The rule for the formation of an identifier are:

* An identifier can consist of alphabets, digits and/or underscores.
* It must not start with a digit.
* C++ is case sensitive that is upper case and lower case letters are considered different from each other.
* It should not be a reserved word.

**Keywords**

There are some reserved words in C++ which have predefined meaning to compiler called keywords. These words may not be used as identifiers. Some commonly used Keywords are given below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| asm | auto | | bool | break | | case |
| catch | char | | class | const | | const\_cast |
| continue | default | | delete | do | | double |
| dynamic\_cast | else | | enum | explicit | | export |
| extern | false | | float | for | | friend |
| goto | if | | inline | int | | long |
| mutable | namespace | | new | operator | | private |
| protected | public | | register | reinterpret\_cast | | return |
| short | signed | | sizeof | static | | static\_cast |
| struct | switch | | template | this | | throw |
| true | try | | typedef | typeid | | typename |
| union | unsigned | | using | virtual | | void |
| volatile | | wchar\_t | | | while | |

**C++ Basic Elements**

Programming language is a set of rules, symbols, and special words used to construct programs. There are certain elements that are common to all programming languages. Now, we will discuss these elements in brief:

**C++ Character Set**

Character set is a set of valid characters that a language can recognize.

|  |  |
| --- | --- |
| **Letters** | A-Z, a-z |
| **Digits** | 0-9 |
| **Special Characters** | Space + - \* / ^ \ () [] {} = != <> ‘ “ $ , ; : % ! & ? \_ # <= >= @ |
| **Formatting characters** | backspace, horizontal tab, vertical tab, form feed, and carriage return |

**Tokens**

A token is a group of characters that logically belong together. The programmer can write a program by using tokens. C++ uses the following types of tokens. Keywords, Identifiers, Literals, Punctuators, Operators.

1. **Keywords**

These are some reserved words in C++ which have predefined meaning to compiler called keywords. It is discussed in previous section.

1. **Identifiers**

Symbolic names can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. The rule for the formation of an identifier are:

* An identifier can consist of alphabets, digits and/or underscores.
* It must not start with a digit.
* C++ is case sensitive that is upper case and lower case letters are considered different from each other.
* It should not be a reserved word.

1. **Literals**

Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++:

* Integer-Constants
* Character-constants
* Floating-constants
* Strings-constants

**Integer Constants**

Integer constants are whole number without any fractional part. C++ allows three types of integer constants.

* **Decimal integer constants:** It consists of sequence of digits and should not begin with 0 (zero). For example, 124, - 179, +108.
* **Octal integer constants:** It consists of sequence of digits starting with 0 (zero). For example, 014, 012.
* **Hexadecimal integer constant:** It consists of sequence of digits preceded by ox or OX.

**Character constants**

A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example, 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character.

**Floating constants**

They are also called real constants. They are numbers having fractional parts. They may be written in fractional form or exponent form. A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits. For example, 3.0, -17.0, -0.627 etc.

**String Literals**

A sequence of character enclosed within double quotes is called a string literal. String literal is by default (automatically) added with a special character ‘\0' which denotes the end of the string. Therefore the size of the string is increased by one character. For example "COMPUTER" will re represented as "COMPUTER\0" in the memory and its size is 9 characters.

1. **Punctuators**

The following characters are used as punctuators in C++.

|  |  |
| --- | --- |
| Brackets [ ] | Opening and closing brackets indicate single and multidimensional array subscript. |
| Parentheses ( ) | Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc. |
| Braces { } | Opening and closing braces indicate the start and end of a compound statement. |
| Comma , | It is used as a separator in a function argument list. |
| Semicolon ; | It is used as a statement terminator. |
| Colon : | It indicates a labeled statement or conditional operator symbol. |
| Asterisk \* | It is used in pointer declaration or as multiplication operator. |
| Equal sign = | It is used as an assignment operator. |
| Pound sign # | It is used as pre-processor directive. |

**Operators**

Operators are special symbols used for specific purposes. C++ provides six types of operators. Arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator.

**Arithmetical operators**

Arithmetical operators +, -, \*, /, and % are used to performs an arithmetic (numeric) operation. You can use the operators +, -, \*, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type. Operators that have two operands are called binary operators.

**Relational operators**

The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

|  |  |
| --- | --- |
| **Relational Operators** | **Meaning** |
| < | Less than |
| <= | Less than or equal to |
| == | Equal to |
| > | Greater than |
| >= | Greater than or equal to |
| ! = | Not equal to |

**Logical operators**

The logical operators are used to combine one or more relational expression. The logical operators are:

|  |  |
| --- | --- |
| **Operators** | **Meaning** |
| || | OR |
| && | AND |
| ! | NOT |

**Unary operators**

C++ provides two unary operators for which only one variable is required. For Example:

a = - 50;

a = + 50;

Here plus sign (+) and minus sign (-) are unary because they are not used between two variables.

**Assignment operator**

The assignment operator '=' is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side. For example:

m = 5;

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

x = y = z = 32;

This code stores the value 32 in each of the three variables x, y, and z.

In addition to standard assignment operator shown above, C++ also support compound assignment operators.

**Compound Assignment Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **Equivalent to** |
| + = | A + = 2 | A = A + 2 |
| - = | A - = 2 | A = A - 2 |
| % = | A % = 2 | A = A % 2 |
| /= | A/ = 2 | A = A / 2 |
| \*= | A \* = 2 | A = A \* 2 |

**Increment and Decrement Operators**

C++ provides two special operators viz '++' and '--' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

**The syntax of the increment operator is:**

Pre-increment: ++variable

Post-increment: variable++

**The syntax of the decrement operator is:**

Pre-decrement: ––variable

Post-decrement: variable––

In Prefix form first variable is first incremented/decremented, then evaluated in Postfix form first variable is first evaluated, then incremented/decremented.

int x, y;

int i = 10, j = 10;

x = ++i; *//add one to i, store the result back in x*

y = j++; *//store the value of j to y then add one to j*

cout << x; *//11*

cout << y; *//10*

**Conditional operator**

The conditional operator ?: is called ternary operator as it requires three operands. The format of the conditional operator is: Conditional\_ expression ? expression1 : expression2; If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated.

int a = 5, b = 6;

big = (a > b) ? a : b;

The condition evaluates to false, therefore biggets the value from b and it becomes 6.

**The comma operator**

The comma operator gives left to right evaluation of expressions. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

int a = 1, b = 2, c = 3, i; *// comma acts as separator, not as an operator*

i = (a, b); *// stores b into i*

Would first assign the value of a to i, and then assign value of b to variable i. So, at the end, variable i would contain the value 2.

**The sizeof operator**

As we know that different types of Variables, constant, etc. require different amounts of memory to store them The sizeof operator can be used to find how many bytes are required for an object to store in memory. For example:

sizeof (char) returns 1

sizeof (float) returns 4

the sizeof operator determines the amount of memory required for an object at compile time rather than at run time.

**The order of Precedence**

The order in which the Arithmetic operators (+, -, \*, /, %) are used in a given expression is called the order of precedence. The following table shows the order of precedence.

|  |  |
| --- | --- |
| **Order** | **Operators** |
| First Second Third | () \*, /, % +, - |

The following table shows the precedence of operators.

|  |  |
| --- | --- |
| ++, --(post increment/decrement) | Highest  To  Lowest |
| ++ (Pre increment) -- (Pre decrement), sizeof ( ), !(not), -(unary), +(unary) |
| \*,/, % |
| +, - |
| <, <=, >, >= |
| ==,!= |
| && |
| ? : |
| = |
| Comma operator |

**Data Handling**

**Basic Data Types**

C++ supports a large number of data types. The built in or basic data types supported by C++ are integer, floating point and character. C++ also provides the data type bool for variables that can hold only the values True and false.

Some commonly used data types are summarized in table along with description.

|  |  |
| --- | --- |
| **Type** | **Description** |
| int | Small integer number |
| long int | Large integer number |
| float | Small real number |
| double | Double precision real number |
| long double | Long double precision real number |
| char | A Single Character |

The exact sizes and ranges of values for the fundamental types are implementation dependent. The header files <climits> (for the integral types) and <cfloat> (for the floating-point types) specify the ranges of values supported on your system.

**C++ string Class**

Because a char variable can store only one character in its memory location, another data type is needed for a variable able to hold an entire string. While C++ does not have a builtin data type able to do this, Standard C++ provides string class that allows the programmer to create a string type variable.

In order to declare and use objects (variables) of this type we need to include an additional header file in our source code: <string>

*// This program demonstrates the string class.*

#include <iostream>

#include <string> // Required for the string class.

using namespace std;

int main ()

{

string mystring = "This is a string";

cout << mystring;

return 0;

}

**Variable Initialization**

Variable is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program.

**Declaration of a variable**

Before a variable is used in a program, we must declare it. This activity enables the compiler to make available the appropriate type of location in the memory.

float total;

You can declare more than one variable of same type in a single statement.

int x, y;

**Initialization of variable**

When we declare a variable it's default value is undetermined. We can declare a variable with some initial value.

int a = 20;

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses () : For example:

int a (0);

Both ways of initializing variables are equivalent in C++.

**Constants**

A variable which does not change its value during execution of a program is known as a constant variable. Any attempt to change the value of a constant will result in an error message. A constant in C++ can be of any of the basic data types, const qualifier can be used to declare constant as shown below:

const float PI = 3.1415;

The above declaration means that PI is a constant of float types having a value 3.1415. Examples of valid constant declarations are:

const int RATE = 50;

const float PI = 3.1415;

const char CH = 'A';

**Type Conversion**

The process in which one pre-defined type of expression is converted into another type is called conversion. There are two types of conversion in C++:

* Implicit conversion
* Explicit conversion

**Implicit conversion**

Data type can be mixed in the expression. For example:

double a;

int b = 5;

float c = 8.5;

a = b \* c;

When two operands of different type are encountered in the same expression, the lower type variable is converted to the higher type variable. The following table shows the order of data types.

|  |  |
| --- | --- |
| **Order of data types** | |
| **Data type** | **Order**  Highest  To  Lowest |
| long double |
| double |
| float |
| long |
| int |
| char |

The int value of b is converted to type float and stored in a temporary variable before being multiplied by the float variable c. The result is then converted to double so that it can be assigned to the double variable a.

**Explicit conversion**

It is also called type casting. It temporarily changes a variable data type from its declared data type to a new one. It may be noted here that type casting can only be done on the right hand side the assignment statement.

totalPay = static\_cast<double>(salary) + bonus;

Initially variable salary is defined as float but for the above calculation it is first converted to double data type and then added to the variable bonus.

**Input/Output (I/O)**

The standard C++ library includes the header file iostream, which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. The following C++ stream objects can be used for the input/output purpose.

**cout** console output

**cin** console input

**cout object**

cout is used to print message on screen in conjunction with the insertion operator <<

cout << "Hello World"; *// prints Hello world on screen*

cout << 250; *// prints number 250 on screen*

cout << sum; *// prints the content of variable sum on screen*

To print constant strings of characters we must enclose them between double quotes (").

If we want to print out a combination of variables and constants, the insertion operator (<<) may be used more than once in a single statement

cout << "Area of rectangle is " << area << " square meter”;

If we assume the area variable to contain the value 24 the output of the previous statement would be:

Area of rectangle is 24 square meter

**cin object**

cin can be used to input a value entered by the user from the keyboard. However, the extraction operator >> is also required to get the typed value from cin and store it in the memory location. Let us consider the following program segment:

int marks;

cin >> marks;

In the above segment, the user has defined a variable marks of integer type in the first statement and in the second statement he is trying to read a value from the keyboard.

*// input output example*

#include <iostream>

using namespace std;

int main ()

{

int length;

int breadth;

int area;

cout << "Please enter length of rectangle: ";

cin >> length;

cout << "Please enter breadth of rectangle: ";

cin >> breadth;

area = length \* breadth;

cout << "Area of rectangle is " << area;

return 0;

}

Output :

Please enter length of rectangle: 6

Please enter breadth of rectangle: 4

Area of rectangle is 24

You can also use cin to request more than one input from the user:

cin >> length >> breadth;

is equivalent to:

cin >> length;

cin >> breadth;

**cin and strings**

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

cin >> mystring;

However, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction.

for example, if we want to get a sentence from the user, this extraction operation would not be useful. In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin:

*// cin and strings*

#include <iostream>

#include <string>

using namespace std;

int main ()

{

string name;

cout << "Enter your name";

getline (cin, name);

cout << "Hello " << name << "!\n";

return 0;

}

Output:

Enter your name: Aniket Rajput

Hello Aniket Rajput!

1. **Flow of Control**

**Flow of Control**

**Statements**

Statements are the instructions given to the computer to perform any kind of action. Action may be in the form of data movement, decision making etc. Statements form the smallest executable unit within a C++ program. Statements are always terminated by semicolon.

**Compound Statement**

A compound statement is a grouping of statements in which each individual statement ends with a semi-colon. The group of statements is called block. Compound statements are enclosed between the pair of braces ({}.). The opening brace ({) signifies the beginning and closing brace (}) signifies the end of the block.

**Null Statement**

Writing only a semicolon indicates a null statement. Thus ';' is a null or empty statement. This is quite useful when the syntax of the language needs to specify a statement but the logic of the program does not need any statement. This statement is generally used in for and while looping statements.

**Conditional Statements**

Sometimes the program needs to be executed depending upon a particular condition. C++ provides the following statements for implementing the selection control structure.

* if statement
* if else statement
* nested if statement
* switch statement

**if statement**

syntax of the if statement

if (condition)

{

statement(s);

}

From the flowchart it is clear that if the if condition is true, statement is executed, otherwise it is skipped. The statement may either be a single or compound statement.



**if else statement**

syntax of the if - else statement

if (condition)

statement1;

else

statement2;

From the above flowchart it is clear that the given condition is evaluated first. If the condition is true, statement1 is executed. If the condition is false, statement2 is executed. It should be kept in mind that statement and statement2 can be single or compound statement.

|  |  |
| --- | --- |
| **if example** | **if else example** |
| if (x == 100)  cout << "x is 100"; | if (x == 100)  cout << "x is 100";  else  cout << "x is not 100"; |

**Nested if statement**

The if block may be nested in another if or else block. This is called nesting of if or else block.

syntax of the nested if statement

if (condition 1)

{

if (condition 2)

{

statement(s);

}

}

|  |
| --- |
| if (condition 1)  statement 1;  else if (condition 2)  statement2;  else  statement3; |

|  |
| --- |
| **if-else-if example** |
| if(percentage>=60)  cout<<"Ist division";  else if(percentage>=50)  cout<<"IInd division";  else if(percentage>=40)  cout<<"IIIrd division";  else  cout<<"Fail" ; |

**switch statement**

The if and if-else statements permit two way branching whereas switch statement permits multiple branching. The syntax of switch statement is:

switch (var / expression)

{

case constant1:

statement 1;

break;

case constant2:

statement2;

break;

.

.

default:

statement3;

break;

}

The execution of switch statement begins with the evaluation of expression. If the value of expression matches with the constant, then the statements following this statement execute sequentially till it executes break. The break statement transfers control to the end of the switch statement. If the value of expression does not match with any constant, the statement with default is executed.

Some important points about switch statement:

* The expression of switch statement must be of type integer or character type.
* The default case need not to be used at last case. It can be placed at any place.
* The case values need not to be in specific order.

**Looping statement**

It is also called a Repetitive control structure. Sometimes we require a set of statements to be executed a number of times by changing the value of one or more variables each time to obtain a different result. This type of program execution is called looping. C++ provides the following construct:

* while loop
* do-while loop
* for loop

**While loop**

Syntax of while loop:

while(condition)

{

statement(s);

}

The flow diagram indicates that a condition is first evaluated. If the condition is true, the loop body is executed and the condition is re-evaluated. Hence, the loop body is executed repeatedly as long as the condition remains true. As soon as the condition becomes false, it comes out of the loop and goes to the statement next to the ‘while’ loop.



**do-while loop**

Syntax of do-while loop:

do

{

statements;

} while (condition);

**Note:** That the loop body is always executed at least once. One important difference between the while loop and the do-while loop the relative ordering of the conditional test and loop body execution. In the while loop, the loop repetition test is performed before each execution the loop body; the loop body is not executed at all if the initial test fail. In the do-while loop, the loop termination test is Performed after each execution of the loop body. hence, the loop body is always executed least once.



**for loop**

It is a count controlled loop in the sense that the program knows in advance how many times the loop is to be executed.

syntax of for loop:

for (initialization; decision; increment/decrement)

{

statement(s);

}

The flow diagram indicates that in for loop three operations take place:

* Initialization of loop control variable
* Testing of loop control variable
* Update the loop control variable either by incrementing or decrementing.

Operation (i) is used to initialize the value. On the other hand, operation (ii) is used to test whether the condition is true or false. If the condition is true, the program executes the body of the loop and then the value of loop control variable is updated. Again it checks the condition and so on. If the condition is false, it gets out of the loop.

**Jump Statements**

The jump statements unconditionally transfer program control within a function.

* goto statement
* break statement
* continue statement

**The goto statement**

goto allows to make jump to another point in the program.

goto pqr;

**pqr:** pqr is known as label. It is a user defined identifier.

After the execution of goto statement, the control transfers to the line after label pqr.

**The break statement**

The break statement, when executed in a switch structure, provides an immediate exit from the switch structure. Similarly, you can use the break statement in any of the loop. When the break statement executes in a loop, it immediately exits from the loop.

**The continue statement**

The continue statement is used in loops and causes a program to skip the rest of the body of the loop.

while (condition)

{

Statement 1;

If (condition)

continue;

statement;

}

The continue statement skips rest of the loop body and starts a new iteration.

**The exit ( ) function**

The execution of a program can be stopped at any point with exit ( ) and a status code can be informed to the calling program. The general format is

exit (code);

where code is an integer value. The code has a value 0 for correct execution. The value of the code varies depending upon the operating system.

1. **Functions**

**C++ Standard Library Function**

The C++ Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, error checking and many other useful operations. This makes the programmer's job easier, because these functions provide many of the capabilities programmers need. The C++ Standard Library functions are provided as part of the C++ programming environment.

*Header file names ending in .h are "old-style" header files that have been superseded by the C++ Standard Library header files.*

|  |  |
| --- | --- |
| **C++ Standard Library header file** | **Explanation** |
| <iostream> | Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file <iostream.h>. |
| <iomanip> | Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file <iomanip.h>. |
| <cmath> | Contains function prototypes for math library functions. This header file replaces header file <math.h>. |
| <cstdlib> | Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file <stdlib.h>. |
| <ctime> | Contains function prototypes and types for manipulating the time and date. This header file replaces header file <time.h>. |
| <cctype> | Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <ctype.h> |
| <cstring> | Contains function prototypes for C-style string-processing functions. This header file replaces header file <string.h>. |
| <cstdio> | Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <stdio.h>. |
| <fstream> | Contains function prototypes for functions that perform input from files on disk and output to files on disk. This header file replaces header file <fstream.h>. |
| <climits> | Contains the integral size limits of the system. This header file replaces header file <limits.h>. |
| <cassert> | Contains macros for adding diagnostics that aid program debugging. This replaces header file <assert.h> from pre-standard C++. |
| <cfloat> | Contains the floating-point size limits of the system. This header file replaces header file <float.h>. |
| <string> | Contains the definition of class string from the C++ Standard Library |
| <vector>,  <list>,  <deque>,  <queue>,  <stack>,  <map>,  <set>,  <bitset> | These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. |
| <typeinfo> | Contains classes for runtime type identification (determining data types at execution time). |
| <exception>, <stdexcept> | These header files contain classes that are used for exception handling. |
| <memory> | Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. |
| <sstream> | Contains function prototypes for functions that perform input from strings in memory and output to strings in memory. |
| <functional> | Contains classes and functions used by C++ Standard Library algorithms. |
| <iterator> | Contains classes for accessing data in the C++ Standard Library containers. |
| <algorithm> | Contains functions for manipulating data in C++ Standard Library containers. |
| <locale> | Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.). |
| <limits> | Contains classes for defining the numerical data type limits on each computer platform. |
| <utility> | Contains classes and functions that are used by many C++ Standard Library header files. |

**Mathematical Functions**

Some of the important mathematical functions in header file <**cmath>** are

|  |  |
| --- | --- |
| **Function** | **Meaning** |
| sin(x) | Sine of an angle x (measured in radians) |
| cos(x) | Cosine of an angle x (measured in radians) |
| tan(x) | Tangent of an angle x (measured in radians) |
| asin(x) | Sin-1 (x) where x (measured in radians) |
| acos(x) | Cos-1 (x) where x (measured in radians) |
| exp(x) | Exponential function of x (ex) |
| log(x) | logarithm of x |
| log 10(x) | Logarithm of number x to the base 10 |
| sqrt(x) | Square root of x |
| pow(x, y) | x raised to the power y |
| abs(x) | Absolute value of integer number x |
| fabs(x) | Absolute value of real number x |

**Character Functions**

All the character functions require **<cctype>** header file. The following table lists the function.

|  |  |
| --- | --- |
| **Function** | **Meaning** |
| isalpha(c) | It returns True if C is an uppercase letter and False if c is lowercase. |
| isdigit(c) | It returns True if c is a digit (0 through 9) otherwise False. |
| isalnum(c) | It returns True if c is a digit from 0 through 9 or an alphabetic character (either uppercase or lowercase) otherwise False. |
| islower(c) | It returns True if C is a lowercase letter otherwise False. |
| isupper(c) | It returns True if C is an uppercase letter otherwise False. |
| toupper(c) | It converts c to uppercase letter. |
| tolower(c) | It converts c to lowercase letter. |

**Function**

A function is a subprogram that acts on data and often returns a value. A program written with numerous functions is easier to maintain, update and debug than one very long program. By programming in a modular (functional) fashion, several programmers can work independently on separate functions which can be assembled at a later date to create the entire project. Each function has its own name. When that name is encountered in a program, the execution of the program branches to the body of that function. When the function is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.

**Creating User-Defined Functions**

**Declare the function:**

The declaration, called the **FUNCTION PROTOTYPE**, informs the compiler about the functions to be used in a program, the argument they take and the type of value they return.

**Define the function:**

The function definition tells the compiler what task the function will be performing. The function prototype and the function definition must be same on the return type, the name, and the parameters. The only difference between the function prototype and the function header is a semicolon. The function definition consists of the function header and its body. The header is EXACTLY like the function prototype, EXCEPT that it contains NO terminating semicolon.

**//Prototyping, defining and calling a function**

#include <iostream>

using namespace std;

void starline(); // prototype the function

int main()

{

starline( ); // function call

cout<< "\t\tBjarne Stroustrup\n"; starline( ); // function call

return 0;

}

**// function definition**

void starline()

{

int count; // declaring a LOCAL variable

for(count = 1; count <=65; count++)

cout<< "\*";

cout<<endl;

}

**Argument to a Function**

Sometimes the calling function supplies some values to the called function. These are known as parameters. The variables which supply the values to a calling function called **actual parameters**. The variable which receive the value from called statement are termed **formal parameters**.

**Consider the following example that evaluates the area of a circle.**

#include<iostream>

using namespace std;

void area(float);

int main()

{

float radius;

cin>>radius;

area(radius);

return 0;

}

void area (float r)

{

cout<< “the area of the circle is”<<3.14\*r\*r<<”\n”;

}

Here radius is called **actual parameter** and r is called **formal parameter**.

**Return Type of a Function**

**// Example program**

#include <iostream>

using namespace std;

int timesTwo(int num); // function prototype

int main()

{

int number, response;

cout<<"Please enter a number:";

cin>>number;

response = timesTwo(number); //function call

cout<< "The answer is "<<response; return 0;

}

//timesTwo function

int timesTwo (int num)

{

int answer; //local variable

answer = 2 \* num;

return (answer);

}

**Calling of a Function**

The function can be called using either of the following methods:

* Call by value
* Call by reference

**Call by Value**

In call by value method, the called function creates its own copies of original values sent to it. Any changes, that are made, occur on the function’s copy of values and are not reflected back to the calling function.

**Call by Reference**

In call be reference method, the called function accesses and works with the original values using their references. Any changes, that occur, take place on the original values are reflected back to the calling code. Consider the following program which will swap the value of two variables.

|  |  |
| --- | --- |
| **Using call by reference** | **Using call by value** |
| #include<iostream>  using namespace std;  void swap(int &, int &);  int main()  {  int a=10,b=20;  swap(a,b);  cout<<a<<" "<<b;  return 0;  }  void swap(int &c, int &d)  {  int t;  t=c;  c=d;  d=t;  } | #include<iostream>  using namespace std;  void swap(int , int );  int main()  {  int a=10,b=20;  swap(a,b);  cout<<a<<" "<< b;  return 0;  }  void swap(int c, int d)  {  int t;  t=c;  c=d;  d=t;  } |
| **output:** 20 10 | **output:** 10 20 |

**Function with Default Arguments**

C++ allows to call a function without specifying all its arguments. In such cases, the function assigns a default value to a parameter which does not have a matching arguments in the function call. Default values are specified when the function is declared. The complier knows from the prototype how many arguments a function uses for calling.

Example:

float result(int marks1, int marks2, int marks3=75);

a subsequent function call

average = result(60,70);

passes the value 60 to marks1, 70 to marks2 and lets the function use default value of 75 for marks3.

The function call

average = result(60,70,80);

passes the value 80 to marks3.

**Inline Function**

Functions save memory space because all the calls to the function cause the same code to be executed. The functions body need not be duplicated in memory. When the complier sees a function call, it normally jumps to the function. At the end of the function. it normally jumps back to the statement following the call. While the sequence of events may save memory space, it takes some extra time. To save execution time in short functions, inline function is used. Each time there is a function call, the actual code from the function is inserted instead of a jump to the function. The inline function is used only for shorter code.

inline int cube(int r)

{

return r\*r\*r;

}

**Some important points to be noted**

* Function is made inline by putting a word inline in the beginning.
* Inline function should be declared before main() function.
* It does not have function prototype.
* Only shorter code is used in inline function If longer code is made inline then compiler ignores the request and it will be executed as normal function.

**Global Variable and Local Variable**

**Local Variable**: A variable declared within the body of a function will be evaluated only within the function. The portion of the program in which a variable is retained in memory is known as the **scope of the variable**. The scope of the local variable is a function where it is defined. A variable may be local to function or compound statement.

**Global Variable**: A variable that is declared outside any function is known as a global variable. The scope of such a variable extends till the end of the program. these variables are available to all functions which follow their declaration. So it should be defined at the beginning, before any function is defined.

**Unary Scope Resolution Operator (::)**

It is possible to declare local and global variables of the same name. C++ provides the **unary scope resolution operator (::)** to access a global variable when a local variable of the same name is in scope. A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

**Variables and Storage Class**

The storage class of a variable determines which parts of a program can access it and how long it stays in existence. The storage class can be classified as automatic register static external.

**Automatic variable:**

All variables by default are auto i.e. the declarations int a and auto int a are equivalent. Auto variables retain their scope till the end of the function in which they are defined. An automatic variable is not created until the function in which it defined is called. When the function exits and control is returned to the calling program, the variables are destroyed and their values are lost. The name automatic is used because the variables are automatically created when a function is called and automatically destroyed when it returns.

**Register variable:**

A register declaration is an auto declaration. A register variable has all the characteristics of an auto variable. The difference is that register variable provides fast access as they are stored inside CPU registers rather than in memory.

**Static variable:**

A static variable has the visibility of a local variable but the lifetime of an external variable. Thus it is visible only inside the function in which it is defined, but it remains in existence for the life of the program.

**External variable:**

A large program may be written by a number of persons in different files. A variable declared global in one file will not be available to a function in another file. Such a variable, if required by functions in both the files, should be declared global in one file and at the same time declared external in the second file.

1. **Compound Data Types**

**Array**

An array is a collection of data elements of same data type. It is described by a single name and each element of an array is referenced by using array name and its subscript no.

**Declaration of Array**

Type arrayName[numberOfElements];

**For example:**

int Age[5] ;

float cost[30];

****

**Initialization of One Dimensional Array**

An array can be initialized along with declaration. For array initialization it is required to place the elements separated by commas enclosed within braces.

int A[5] = {11,2,23,4,15};

It is possible to leave the array size open. The compiler will count the array size.

int B[] = {6,7,8,9,15,12};

**Referring to Array Elements**

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value.

The format is as simple as:

name[index]

**Examples:**

cout << age[4]; //print an array element

age[4] = 55; // assign value to an array element

cin >> age[4]; //input element 4

**Using Loop to input an Array from user**

int age [10], i ;

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

{

cin >> age[i];

}

**Arrays as Parameters**

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. For example, the following function:

void print(int A[])

accepts a parameter of type "array of int" called A.

In order to pass to this function an array declared as:

int arr[20];

we need to write a call like this:

print(arr);

**Here is a complete example:**

#include <iostream>

using namespace std;

void print(int A[], int length)

{

for (int n = 0; n < length; n++)

cout << A[n] << " ";

cout << "\n";

}

int main ()

{

int arr[] = {5, 10, 15};

print(arr,3);

return 0;

}

**Basic Operation On One Dimensional Array**

**Function to traverse the array A**

void display(int A[], int n)

{

cout << "The elements of the array are:\n";

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

cout << A[i];

}

**Function to Read elements of the array A**

void Input(int A[], int n)

{

cout << "Enter the elements:";

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

cin >> A[i];

}

**Function to Search for an element from A by Linear Search**

void lsearch(int A[], int n, int data)

{

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

{

if(A[i] == data)

{

cout << "Data Found at : " << i;

return;

}

}

cout << "Data Not Found in the array" << endl;

}

**Function to Search for an element from Array A by Binary Search**

int BsearchAsc(int A[], int n, int data)

{

int Mid, Lbound = 0, Ubound = n-1, Found=0;

while((Lbound <= Ubound) && !(Found))

{

Mid =(Lbound+Ubound)/2; //Searching The Item

if(data > A[Mid])

Lbound = Mid+1;

else if(data < A[Mid])

Ubound = Mid-1;

else

Found++;

}

if(Found)

return(Mid+1); //returning 1ocation, if present

else

return(-1); //returning -1,if not present

}

**Function to Sort the array A by Bubble Sort**

void BSort(int A[], int n)

{

int I, J, Temp;

for(I = 0; I < n-1; I++) //sorting

{

for(J = 0; J < (n-1-I); J++)

if(A[J] > A[J+1])

{

Temp = A[J]; //swapping

A[J] = A[J+1];

A[J+1] = Temp;

}

}

}

**Function to Sort the array ARR by Insertion Sort**

void ISort(int A[], int n)

{

int I, J, Temp;

for(I = 1; I < n; I++) //sorting

{

Temp = A[I];

J = I-1;

while((Temp < A[J]) && (J >= 0))

{

A[J+1] = A[J];

J--;

}

A[J+1]=Temp;

}

}

**Function to Sort the array by Selection Sort**

void SSort(int A[], int n)

{

int I, J, Temp, Small;

for(I = 0; I < n-1; I++)

{

Small = I;

for(J = I+1; J < n; J++) //finding the smallest element

if(A[J] < A[Small])

Small = J;

if(Small != I)

{

Temp = A[I]; //Swapping

A[I] = A[Small];

A[Small] = Temp;

}

}

}

**Function to merge A and B arrays of lenghts N and M**

void Merge(int A[], int B[], int C[], int N, int M, int &K)

{

int I = 0, J = 0; K = 0;

while (I < N && J < M)

{

if (A[I] < B[J])

C[K++] = A[I++];

else if (A[I] > B[J])

C[K++] = B[J++];

else

{

C[K++] = A[I++];

J++;

}

}

int T;

for (T = I; T < N; T++)

C[K++] = A[T];

for (T = J; T < M; T++)

C[K++] = B[T];

}

**Two Dimensional Array**

It is a collection of data elements of same data type arranged in rows and columns (that is, in two dimensions).

**Declaration of Two-Dimensional Array**

Type arrayName[numberOfRows][numberOfColumn];

**For example:**

int Sales[3][5];



**Initialization of Two-Dimensional Array**

A two-dimensional array can be initialized along with declaration. For two-dimensional array initialization, elements of each row are enclosed within curly braces and separated by commas. All rows are enclosed within curly braces.

int A[4][3] = {{22, 23, 10},

{15, 25, 13},

{20, 74, 67},

{11, 18, 14}};

**Referring to Array Elements**

To access the elements of a two-dimensional array, we need a pair of indices: one for the row position and one for the column position. The format is as simple as:

name[rowIndex][columnIndex].

**Examples:**

cout << A[1][2]; //print an array element

A[1][2] = 13; // assign value to an array element

cin >> A[1][2]; //input element

**Using Loop to input a Two-Dimensional Array from user**

int mat[3][5], row, col ;

for (row = 0; row < 3; row++)

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

cin >> mat[row][col];

**Arrays as Parameters**

Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference. When declaring a two-dimensional array as a formal parameter, we can omit the size of the first dimension, but not the second; that is, we must specify the number of columns. For example:

void print(int A[][3], int N, int M)

In order to pass to this function an array declared as:

int arr[4][3];

we need to write a call like this:

print(arr);

**Here is a complete example:**

#include <iostream>

using namespace std;

void print(int A[][3], int N, int M)

{

for (R = 0; R < N; R++)

for (C = 0; C < M; C++)

cout << A[R][C];

}

int main ()

{

int arr[4][3] ={{12, 29, 11},

{25, 25, 13},

{24, 64, 67},

{11, 18, 14}};

print(arr,4,3);

return 0;

}

**Function to read the array A**

void Read(int A[][20], int N, int M)

{

for(int R = 0; R < N; R++)

for(int C = 0; C < M; C++)

{

cout << "(R<<','<<")?"; cin >> A[R][C];

}

}

**Function to display content of a two dimensional array A**

void Display(int A[][20], int N, int M)

{

for(int R = 0;R < N; R++)

{

for(int C = 0; C < M; C++)

cout << setw(10) << A[R][C];

cout << endl;

}

}

**Function to find the sum of two dimensional arrays A and B**

void Addition(int A[][20], int B[][20], int N, int M)

{

for(int R = 0; R < N; R++)

for(int C = 0;C < M; C++)

C[R][C] = A[R][C] + B[R][C];

}

**Function to multiply two dimensional arrays A and B of order NxL and LxM**

void Multiply(int A[][20], int B[][20], int C[][20],int N, int L, int M)

{

for(int R = 0; R < N; R++)

for(int C = 0; C < M; C++)

{

C[R][C] = 0;

for(int T = 0; T < L; T++)

C[R][C] += A[R][T] \* B[T][C];

}

}

**Function to find & display sum of rows & sum of cols. of array A**

void SumRowCol(int A[][20], int N, int M)

{

for(int R = 0; R < N; R++)

{

int SumR = 0;

for(int C = 0; C < M; C++)

SumR += A[R][C];

cout << "Row("<<R<<")=" << SumR << endl;

}

for(int R = 0;R < N; R++)

{

int SumR = 0;

for(int C = 0; C < M; C++)

SumR += A[R][C];

cout << "Row("<<R<<")=" << SumR << endl;

}

}

**Function to find sum of diagonal elements of a square matrix A**

void Diagonal(int A[][20], int N, int &Rdiag, int &LDiag)

{

for(int I = 0, Rdiag = 0; I < N; I++)

Rdiag += A[I][I];

for(int I = 0, Ldiag = 0; I < N; I++)

Ldiag += A[N-I-1][I];

}

**Function to find out transpose of a two dimensional array A**

void Transpose(int A[][20], int B[][20], int N, int M)

{

for(int R = 0; R < N; R++)

for(int C = 0;C < M; C++)

B[R][C] = A[C][R];

}

**C-Strings (Character Arrays)**

**STRING:** It is an array of type char.

**Syntax for declaration**

char <array/string name> [max. number of characters to be stored +1];

The number of elements that can be stored in a string is always n-1, if the size of the array specified is n. This is because 1 byte is reserved for the NULL character '\0' i.e. backslash zero. A string is always terminated with the NULL character.

**Example:**

char str[80];

In the above example, str can be used to store a string with 79 characters.

**Initializing a string**

A string can be initialized to a constant value when it is declared.

char str[ ] = "Good";

Or

char str[]={'G','o','o','d','\0'};

Here. 'G' will be stored in str[0], 'o' in str[1] and so on.

**Note:** When the value is assigned to the complete string at once, the computer automatically inserts the NULL character at the end of the string. But, if it is done character by character, then we have to insert it at the end of the string.

**Reading strings with/without embedded blanks**

To read a string without blanks cin can be used

cin>>str;

To read a string with blanks cin.getline() or gets() can be used.

cin.getline(str,80);

-Or-

gets(str);

**Printing strings**

cout and puts () can be used to print a string.

cout<<str;

Or

puts(str);

**Note:** For gets() and puts(), the header file <cstdio> (formally stdio.h) has to be included. puts() can be used to display only strings. It takes a line feed after printing the string.

|  |  |
| --- | --- |
| **Cin** | **gets()** |
| It can be used to take input of a value of any data type. | It can be used to take input of a string. |
| It takes the white space i.e. a blank, a tab, or a new line character as a string terminator. | It does not take the white space i.e. a blank, a tab, or a new line character, as a string terminator. |
| It requires header file <iostream> | It requires the header file <cstdio> |
| **Example:**  char S[80];  cout << "Enter a string:”;  cin>>S; | **Example:**  char S[80];  cout << "Enter a string:";  gets(S); |

|  |  |
| --- | --- |
| **cout** | **puts()** |
| It can be used to display the value of any data type. | It can be used to display the value of a string. |
| It does not take a line feed after displaying the string. | It takes a line feed after displaying the string. |
| It requires the header file iostream | It requires the header file cstdio |
| **Example:**  char S[80] = "Computers";  cout<<S<<S;  **Output:**  ComputersComputers | **Example:**  char S[80] = "Computers";  puts(S);  puts(S);  **Output:**  Computers Computers |

**Counting the number of characters in a string and printing it backwards**

#include<iostream>

using namespace std;

int main( )

{

char str[80];

cout<<"Enter a string:";

cin.getline(str,80);

for(int l=0; str[l]!='\0';l++) //Loop to find length

cout<<"The length of the string is : "<<l<<endl ;

for(int i=l-1;i>=0;i--) //Loop to display the string backwards

cout<<str[i];

return 0;

}

**Function to count the number of words in a string**

void count(char S[])

{

int words=0;

for(int i=0;S[i]!='\0';i++)

{

if (S[i]==' ')

words++; //Checking for spaces

}

cout<<"The number of words="<<words+1<<endl;

}

**Function to find the length of a string**

int length(char S[ ])

{

for(int i=0;S[i]!='\0';i++)

return i;

}

**Function to copy the contents of string S2 to S1**

void copy(char S1[ ], char S2[ ])

{

for(int i=0;S2[i]!='\0';i++)

S1[i]=S2[i];

S1[i]='\0';

}

**Function to concatenate the contents of string S2 to S1**

void concat(char S1[ ], char S2[ ])

{

for(int l=0;S1[l]!='\0';l++)

for(int i=0;S2[i]!='\0';i++)

S1[l++]=S2[i];

S1[l]='\0';

}

**Function to compare strings STR1 to STR2.**

The function returns a value>0 if //STR1>STR2, a value<0 if STR1<STR2, and value 0 if STR1=STR2.

int compare(char STR1[ ],char STR2[])

{

for(int I=0;STR1[I]==STR2[I] && STR1[I]!='\0'&&STR2[I]!='\0';I++)

return STR1[I]-STR2[I];

}

**To reverse the contents of string S and store it in string Rev**

void Reverse(char S[], char Rev[])

{

for(int C1=0; S[C1]!='\0'; C1++)

C1--;

for(int C2=0;C1>=0;C2++,C1--)

Rev[C2]=S[C1];

Rev[C2]='\0';

}

**Function to check whether a string S is a palindrome or not**

int Palin(char S[])

{

for(int L=0;S[L]!='\0';L++) //To find length

for(int C=0;(C<L/2) && (S[C]==S[L-C-1]);C++)

return (C==L/2)?1:0; //Returns 1 if Palindrome else 0

}

**Function to change the case of string S to uppercase**

void Upper(char S[])

{

for(int i=0;S[i]!='\0';i++)

S[i] = (S[i]>='a' && S[i]<='z')?(S[i]-32):S[i];

}

**Function to change the case of string S to lower case**

void Lower(char S[])

{

for(int i=0;S[i]!='\0';i++)

S[i] = (S[i]>='A' && S[i]<='Z')?(S[i]+32):S[i];

}

**Function to extract n characters from left side of the string and store it in a different string.**

Example: 4 characters from ENVIRONMENT=ENVI

int SLeft(char S[ ], int n, char result[ ])

{

for(int l=0;S[l]!='\0';l++)

if(n<=I) //characters extracted should be <=length

{

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

result[i]=S[i];

result[i]='\0';

return 1;

}

Else

return 0;

}

**Function to extract n characters from right side of the string and store it in a different string.**

Example: 4 characters from ENVIRONMENT=MENT

int SRight(char S[ ], int n, char result[ ])

{

for(int l=0;S[l]!='\0';l++);

if(n<=I) //characters extracted should be <=length

{

for(int j=0;i=l-n;S[i]!=’/0’;i++,j++)

result[j]=S[i];

result[j]='\0';

return 1;

}

else

return 0;

}

**Structure**

A structure is a collection of variable which can be same or different types. You can refer to a structure as a single variable and to its parts as **members** of that variable by using the dot (.) operator. The power of structures lies in the fact that once defined, the structure name becomes a **user-defined data type** and may be used the same way as other built-in data types, such as int, double, char.

struct Student

{

int rollno, age;

char name[80];

float marks;

};

int main()

{

// declare two variables of the new type

Student s1, s3;

//accessing of data members

cin >> s1.rollno >> s1.age >> s1.name >> s1.marks;

cout << s1.rollno << s1.age << s1.name << s1.marks;

//initialization of structure variable

Student s2 = {100, 17, "Aniket", 92};

cout << s2.rollno << s2.age << s2.name << s2.marks;

//structure variable in assignment statement

s3 = s2;

cout << s3.rollno << s3.age << s3.name << s3.marks;

return 0;

}

**Defining a structure**

When dealing with the students in a school, many variables of different types are needed. It may be necessary to keep track of name, age, Rollno, and marks point for example.

struct Student

{

int rollno, age;

char name[80];

float marks;

};

**Student** is called the **structure tag,** and is your brand new data type, like int, double or char.

**rollno, name, age,** and **marks** are **structure members.**

**Declaring Variables of Type struct**

The most efficient method of dealing with structure variables is to define the structure **globally**. This tells "the whole world", namely main and any functions in the program, that a new data type exists. To declare a structure globally, place it **BEFORE** void main(). The structure variables can then be defined locally in main, for example…

struct Student

{

int rollno, age;

char name[80];

float marks;

};

int main()

{

// declare two variables of the new type

Student s1, s3;

………

………

return 0;

}

**Alternate method of declaring variables of type struct:**

struct Student

{

int rollno, age;

char name[80];

float marks;

} s1, s3;

**Accessing of data members**

The accessing of data members is done by using the following format: structure variable.member name for example

cin >> s1.rollno >> s1.age >> s1.name >> s1.marks;

**Initialization of structure variable**

Initialization is done at the time of declaration of a variable. For example

Student s2 = {100, 17, "Aniket", 92};

**Structure variable in assignment statement**

s3 = s2;

The statement assigns the value of each member of s2 to the corresponding member of s3. Note that one structure variable can be assigned to another only when they are of the same structure type, otherwise complier will give an error.

**Nested structure (Structure within structure)**

It is possible to use a structure to define another structure. This is called nesting of structure. Consider the following program

struct Day

{

int month, date, year;

};

struct Student

{

int rollno, age;

char name[80];

Day date\_of\_birth;

float marks;

};

**Accessing Member variables of Student**

To access members of date\_of\_birth we can write the statements as below:

Student s; // Structure variable of Student

s.date\_of\_birth.month = 11;

s.date\_of\_birth.date = 5;

s.date\_of\_birth.year = 1999;

**typedef**

It is used to define new data type for an existing data type. It provides and alternative name for standard data type. It is used for self-documenting the code by allowing descriptive name for the standard data type.

The general format is:

typedef existing datatype new datatype

for example:

typedef float real;

Now, in a program one can use datatype real instead of float.

Therefore, the following statement is valid:

real amount;

**Enumerated data type**

The enum specifier defines the set of names which are stored internally as integer constant. The first name was given the integer value 0, the second value 1 and so on.

for example:

enum months{jan, feb, mar, apr, may} ;

It has the following features:

* It is user defined.
* It works if you know in advance a finite list of values that a data type can take.
* The list cannot be input by the user or output on the screen.

**#define preprocessor directive**

The #define preprocessor allows to define symbolic names and constants e.g.

#define pi 3.14159.

This statement will translate every occurrence of PI in the program to 3.14159.

**Macros**

Macros are built on the #define preprocessor. Normally a macro would look like:

#define square(x) x\*x.

Its arguments substituted for replacement text, when the macro is expanded.

**Pointer**

**Accessing address of a variable**

Computer‟s memory is organized as a linear collection of bytes. Every byte in the computer‟s memory has an address. Each variable in program is stored at a unique address. We can use address operator & to get address of a variable:

int num = 23;

cout << &num; // prints address in hexadecimal

**POINTER**

A pointer is a variable that holds a memory address, usually the location of another variable in memory.

**Defining a Pointer Variable**

int \*iptr;

iptr can hold the address of an int.

**Pointer Variables Assignment:**

int num = 25;

int \*iptr;

iptr = &num;

**Memory layout**



**To access num using iptr and indirection operator \***

cout << iptr; // prints 0x4a00

cout << \*itptr; // prints 25

Similary, following declaration shows:

char \*cptr;

float \*fptr;

cptr is a pointer to character and fptr is a pointer to float value.

**Pointer Arithmetic**

Some arithmetic operators can be used with pointers:

* Increment and decrement operators ++, --.
* Integers can be added to or subtracted from pointers using the operators +, -, +=, and -=.

Each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.

If “p” is a character pointer, then “p++” will increment “p” by 1 byte.

If “p” were an integer pointer its value on “p++” would be incremented by 4 bytes.

**Pointers and Arrays**

Array name is base address of array

int vals[] = {4, 7, 11};

cout << vals; // displays 0x4a00

cout << vals[0]; // displays 4

Lets takes an example:

int arr[]={4,7,11};

int \*ptr = arr;

What is ptr + 1?

It means (address in ptr) + (1 \* size of an int)

cout << \*(ptr+1); // displays 7

cout << \*(ptr+2); // displays 11

**Array Access**

Array notation arr[i] is equivalent to the pointer notation \*(arr + i)

Assume the variable definitions

int arr[]={4,7,11};

int \*ptr = arr;

Examples of use of ++ and –

ptr++; // points at 7

ptr--; // now points at 4

**Character Pointers and Strings**

Initialize to a character string.

char\* a = “Hello”;

a is pointer to the memory location where “H” is stored. Here “a” can be viewed as a character array of size 6, the only difference being that a can be reassigned another memory location.

char\* a = “Hello”;

a gives address of “H”

\*a gives “H”

a[0] gives “H”

a++ gives address of “e”

\*a++ gives “e”

**Pointers as Function Parameters**

A pointer can be a parameter. It works like a reference parameter to allow change to argument from within function.

#include<iostream>

using namespace std;

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

int main()

{

int a=10,b=20;

swap(&a, &b);

cout<<a<<" "<<b;

return 0;

}

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

{

int t;

t=\*x;

\*x=\*y;

\*y=t;

}

**Output:** 20 10

**Pointers to Constants and Constant Pointers**

**Pointer to a constant**: cannot change the value that is pointed at.

**Constant pointer**: address in pointer cannot change once pointer is initialized.

**Pointers to Structures**

We can create pointers to structure variables.

struct Student

{

int rollno;

float fees;

}; Student stu1;

Student \*stuPtr = &stu1;

(\*stuPtr).rollno= 104;

**-or-**

Use the form ptr->member:

stuPtr->rollno = 104;

**Static allocation of memory**

In the static memory allocation, the amount of memory to be allocated is predicted and preknown. This memory is allocated during the compilation itself. All the declared variables declared normally, are allocated memory statically.

**Dynamic allocation of memory**

In the dynamic memory allocation, the amount of memory to be allocated is not known. This memory is allocated during run-time as and when required. The memory is dynamically allocated using new operator.

**Free store**

Free store is a pool of unallocated heap memory given to a program that is used by the program for dynamic allocation during execution.

**Dynamic Memory Allocation**

We can allocate storage for a variable while program is running by using new operator.

**To allocate memory of type integer**

int \*iptr=new int;

**To allocate array**

double \*dptr = new double[25];

**To allocate dynamic structure variables or objects**

Student sptr = new Student; //Student is tag name of structure

**Releasing Dynamic Memory**

**Use delete to free dynamic memory**

delete iptr;

**To free dynamic array memory**

delete [] dptr;

**To free dynamic structure**

delete Student;

**Memory Leak**

If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program. Such memory blocks are known as orphaned memory blocks. These orphaned memory blocks when increase in number, bring adverse effect on the system. This situation is called memory leak

**Self-Referential Structure**

The self-referential structures are structures that include an element that is a pointer to another structure of the same type.

struct node

{

int data;

node\* next;

}

1. **Object oriented Programming**

**OOP Concepts**

There are two common programming methods: procedural programming and object-oriented programming (OOP). So far you have been creating procedural programs.

**Procedural Programming**

In a procedural program data is typically stored in a collection of variables and there is a set of functions that perform operations on the data. The data and the functions are separate entities. Usually the variables are passed to the functions that perform the desired operations. As you might imagine, the focus of procedural programming is on creating the functions, or procedures, that operate on the program’s data. Procedural programming works well. However, as programs become larger and more complex, the separation of a program’s data and the code that operates on the data can lead to problems.

**Object Oriented programming**

The *object oriented programming design* models the real world well and overcomes the shortcomings of procedural paradigm. It views a problem in terms of objects and thus emphasizes on both procedures as well as data.

An **object** is an entity that combines both data and procedures in a single unit. An object’s data items, also referred to as its attributes, are stored in member variables. The procedures that an object performs are called its member functions. This wrapping of an object’s data and procedures together is called **encapsulation**.

Not only objects encapsulate associated data and procedures, they also permit data hiding. **Data hiding** refers to an object’s ability to hide its data from code outside the object. Only the object’s member functions can directly access and make changes to the object’s data.

**Advantages of Object oriented programming:**

* Software complexity can be easily managed.
* Object-oriented systems can be easily upgraded.
* It is quite easy to partition the work in a project based on object.

**Class & Objects**

The mechanism that allows you to combine data and the function in a single unit is called a class. Once a class is defined, you can declare variables of that type. A class variable is called object or instance. In other words, a class would be the data type, and an object would be the variable. Classes are generally declared using the keyword class, with the following format:

class class\_name

{

private:

members1;

protected:

members2;

public:

members3;

};

Where class\_name is a valid identifier for the class. The body of the declaration can contain members, that can be either data or function declarations, The members of a class are classified into three categories: private, public, and protected. private, protected, and public are reserved words and are called member access specifiers. These specifiers modify the access rights that the members following them acquire.

**Private members** of a class are accessible only from within other members of the same class. You cannot access it outside of the class.

**Protected members** are accessible from members of their same class and also from members of their derived classes.

Finally, **public members** are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access.

Here is a complete example:

class Circle

{

private:

double radius;

public:

void setRadius(double r)

{

radius = r;

}

double getArea()

{

return 3.14 \* radius \* radius;

}

};

**Object Declaration**

Once a class is defined, you can declare objects of that type. The syntax for declaring an object is the same as that for declaring any other variable. The following statements declare two objects of type circle:

Circle c1, c2;

**Accessing Class Members**

Once an object of a class is declared, it can access the public members of the class.

c1.setRadius(2.5);

**Defining Member function of class**

You can define Functions inside the class as shown in above example. Member functions defined inside a class this way are created as inline functions by default. It is also possible to declare a function within a class but define it elsewhere. Functions defined outside the class are not normally inline. When we define a function outside the class we cannot reference them (directly) outside of the class. In order to reference these, we use the scope resolution operator, :: (double colon). In this example, we are defining function setRadius outside the class:

void Circle :: setRadius(double r)

{

radius = r;

}

The following program demonstrates the general feature of classes. Member functions setRadius() and getArea() defined outside the class.

#include <iostream>

using namespace std;

class Circle *//specify a class*

{

private :

double radius; *//class data members*

public:

void setRadius(double r);

double getArea(); *//member function to return area*

};

void Circle :: setRadius(double r)

{

radius = r;

}

double Circle :: getArea()

{

return 3.14 \* radius \* radius;

}

int main()

{

Circle c1; *//define object of class circle*

c1.setRadius(2.5); *//call member function to initialize radius*

cout << c1.getArea(); *//display area of circle object*

return 0;

}

**Constructor and Destructor**

**Constructor**

It is a member function having same name as it’s class and which is used to initialize the objects of that class type with a legal initial value. Constructor is automatically called when object is created.

**Types of Constructor**

**Default Constructor:**

A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

Circle :: Circle()

{

radius = 0;

}

**Parameterized Constructor:**

A constructor that receives arguments/parameters, is called parameterized constructor.

Circle :: Circle(double r)

{

radius = r;

}

**Copy Constructor**:

A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

Circle :: Circle(Circle &t)

{

radius = t.radius;

}

There can be multiple constructors of the same class, provided they have different signatures.

**Destructor**

A destructor is a member function having sane name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object’s scope is over.

~Circle() {}

**Example:** In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading.

#include<iostream>

using namespace std;

class Circle *//specify a class*

{

private:

double radius; *//class data members*

public:

Circle() *//default constructor*

{

radius = 0;

}

Circle(double r) *//parameterized constructor*

{

radius = r;

}

Circle(Circle &t) *//copy constructor*

{

radius = t.radius;

}

void setRadius(double r) *//function to set data*

{

radius = r;

}

double getArea()

{

return 3.14 \* radius \* radius;

}

~Circle() *//destructor*

{}

};

int main()

{

Circle c1; *//default constructor invoked*

Circle c2(2.5); *//parmeterized constructor invoked*

Circle c3(c2); *//copy constructor invoked*

cout << c1.getArea()<<endl;

cout << c2.getArea()<<endl;

cout << c3.getArea()<<endl;

return 0;

}

**Another way of Member initialization in constructors**

The constructor for this class could be defined, as usual, as:

Circle :: Circle(double r)

{

radius = r;

}

It could also be defined using member initialization as:

Circle :: Circle(double r) : radius(r)

{}

**Time Class Case Study**

In the preceding section, we introduced many basic terms and concepts of C++ object oriented programming. In this section, we take a deeper look at classes. In this Time class case study, we will demonstrate several class construction features. We begin with a Time class that reviews several of the features presented in the preceding section.

#include<iostream>

#include<iomanip>

using namespace std;

class Time

{

private :

int hour;

int minute;

int second;

public :

*//constructor with default value 0*

Time(int h = 0, int m = 0, int s = 0);

*//setter function*

void setTime(int h, int m, int s);

*//print description of object in hh:mm:ss*

void print();

*//compare two time object*

bool equals(Time);

};

Time :: Time(int h, int m, int s)

{

hour = h;

minute = m;

second = s;

}

void Time :: setTime(int h, int m, int s)

{

hour = h;

minute = m;

second = s;

}

void Time :: print()

{

cout << setw(2) << setfill('0') << hour << ":"

<< setw(2) << setfill('0') << minute << ":"

<< setw(2) << setfill('0') << second << "\n";

}

bool Time :: equals(Time otherTime)

{

if(hour == otherTime.hour && minute == otherTime.minute &&

second == otherTime.second)

return **true**;

else

return **false**;

}

int main()

{

Time t1(10, 50, 59);

t1.print(); *// 10:50:59*

Time t2; *//object created with default value*

t2.print(); *// 00:00:00*

t2.setTime(6, 39, 9); *//set the new time in object*

t2.print(); *// 06:39:09*

if(t1.equals(t2))

cout << "Two objects are equals\n";

else

cout << "Two objects are not equals\n";

return 0;

}

**Output:**

10:50:59

00:00:00

06:39:09

Two objects are not equals

Let's us discuss our Time class and add some new concepts of programming.

**Constructors with Default Arguments**

In Circle class we have created explicit default constructor and parameterized constructor. Compiler overloads constructor based on match.

You can combine both statements in one as in Time class example

*//constructor with default value*

Time(int h = 0, int m = 0, int s = 0);

It works same way just matter of styling code.

**Constant Function**

In some cases, you will need that the member function should not change any private member variables of the calling object. You can do this by adding const to the end of the function declaration (prototype) and in the function definition.

Let’s make member functions in our Time class const if appropriate:

class Time

{

...

...

*//print description of object in hh:mm:ss*

void print() const;

*...*

*...*

};

....

....

void Time :: print() const

{

cout << setw(2) << setfill('0') << hour << ":"

<< setw(2) << setfill('0') << minute << ":"

<< setw(2) << setfill('0') << second << "\n";

}

....

**Constant Parameters**

You can make a parameter in a function as being a *const* parameter by preceding its type with *const*. This tells the compiler to disallow that parameter changing its value inside that function. This mechanism protects you from making inadvertent mistakes.

Let’s make parameter constant in our Time class, if appropriate:

class Time

{

.....

*//constructor with default value 0*

Time(const int h = 0, const int m = 0, const int s = 0);

*//setter function*

void setTime(const int h, const int m, const int s);

*.....*

};

Time :: Time(const int h, const int m, const int s)

{

hour = h;

minute = m;

second = s;

}

void Time :: setTime(const int h, const int m, const int s)

{

hour = h;

minute = m;

second = s;

}

......

**Passing Objects to Functions**

Object can be passed by value, by reference or by pointer. In our Time class we passed object by value.

bool Time :: equals(Time otherTime)

{

if(hour == otherTime.hour && minute == otherTime.minute &&

second == otherTime.second)

return **true**;

else

return **false**;

}

This means that equals() receives a copy of object t2 with name otherTime. If a function needs to store or change data in an object’s member variables, the object must be passed to it by reference.

**Constant Reference Parameters**

Passing by const reference is the preferred way to pass objects as an alternative to pass-by-value. When you pass by const reference, you take the argument in by reference, but cannot make any changes to the original object.

*//object passing by reference with const parameter*

bool equals(const Time&);

**Separate Header and Implementation Files**

In this section, we demonstrate how to make class reusable by separating it into other files.

**Header File**

Class declarations are stored in a separate file. A file that contains a class declaration is called header file. The name of the class is usually the same as the name of the class, with a .h extension. For example, the Time class would be declared in the file Time .h.

#ifndef TIME\_H

#define TIME\_H

class Time

{

private :

int hour;

int minute;

int second;

public :

*//with default value*

Time(const int h = 0, const int m = 0, const int s = 0);

*// setter function*

void setTime(const int h, const int m, const int s);

*// Print a description of object in " hh:mm:ss"*

void print() const;

*//compare two time object*

bool equals(const Time&);

};

#endif

**Implementation File**

The member function definitions for a class are stored in a separate .cpp file, which is called the class implementation file. The file usually has the same name as the class, with the .cpp extension. For example the Time class member functions would be defined in the file Time.cpp.

#include <iostream>

#include <iomanip>

#include "Time.h"

using namespace std;

Time :: Time(const int h, const int m, const int s)

: hour(h), minute (m), second(s)

{}

void Time :: setTime(const int h, const int m, const int s)

{

hour = h;

minute = m;

second = s;

}

void Time :: print() const

{

cout << setw(2) << setfill('0') << hour << ":"

<< setw(2) << setfill('0') << minute << ":"

<< setw(2) << setfill('0') << second << "\n";

}

bool Time :: equals(const Time &otherTime)

{

if(hour == otherTime.hour && minute == otherTime.minute

&& second == otherTime.second)

return **true**;

else

return **false**;

}

**Client Code**

Client code, is the one that includes the main function. This file should be stored by the name main.cpp

#include <iostream>

using namespace std;

#include "Time.h"

int main()

{

Time t1(10, 50, 59);

t1.print(); *// 10:50:59*

Time t2;

t2.print(); *// 06:39:09*

t2.setTime(6, 39, 9);

t2.print(); *// 06:39:09*

if(t1.equals(t2))

cout << "Two objects are equal\n";

else

cout << "Two objects are not equal\n";

return 0;

}

The advantages of storing class definition in separate file are:

* The class is reusable.
* The clients of the class know what member functions the class provides, how to call them and what return types to expect.
* The clients do not know how the class's member functions are implemented.

**Static Members of a Class**

In the previous sections we have shown examples of classes where each object of a class had its own set of data. Member function could access the object's own version of the data.

In some situations, it may be desirable that one or more common data fields should exist, which are accessible to all objects of the class. In C++, a static member is shared by all objects of the class.

**Static Data Members**

A data member of a class can be declared static; be it in the public or private part of the class definition. Such a data member is created and initialized only once. Static data members which are declared public can be accessed by using class name and the scope resolution operator.

We only include the declaration of static data in the class declaration. Initialization of a static data-member is done outside the class. This is illustrated in the following code fragment:

#include <iostream>

using namespace std;

class Circle

{

private:

double radius; *// Radius of a circle*

public:

static int count;

*// Constructor definition*

Circle(double r = 1.0)

{

radius = r;

*// Increase every time object is created*

count++;

}

double getArea()

{

return 3.14 \* radius \* radius;

}

};

*// Initialize static member of class Circle*

int Circle::count = 0;

int main()

{

Circle c1(3.3); *// Declare object c1*

Circle c2(4.5); *// Declare object c2*

*// Print total number of objects.*

cout << "Total objects: " << Circle::count << endl;

return 0;

}

**Output:**

Total objects: 2

**Static Member Functions**

The static functions can access only the static data of a class. Similarly, static functions cannot call non-static functions of the class.

Functions which are static and which are declared in the public section of a class can be called without specifying an object of the class. This is illustrated in the following code fragment:

#include <iostream>

using namespace std;

class Circle

{

private:

static int count;

double radius; *// Radius of a circle*

public:

*// Constructor definition*

Circle(double r = 1.0)

{

radius = r;

*// Increase every time object is created*

count++;

}

double getArea()

{

return 3.14 \* radius \* radius;

}

static int getCount()

{

return count;

}

};

*// Initialize static member of class Circle*

int Circle::count = 0;

int main()

{

Circle c1(3.3); *// Declare object c1*

Circle c2(4.5); *// Declare object c2*

*// Print total number of objects.*

cout << "Total objects: " << Circle::getCount() << endl;

return 0;

}

**Output:**

Total objects: 2

**Friend Functions**

As we have seen in the previous sections, private and protected data or function members are normally only accessible by the code which is part of same class. However, situations may arise in which it is desirable to allow the explicit access to private members of class to other functions.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend. This is illustrated in the following code fragment:

#include <iostream>

using namespace std;

class Rectangle

{

private:

int length;

int width;

public:

void setData(int len, int wid)

{

length = len;

width = wid;

}

int getArea()

{

return length \* width ;

}

friend double getCost(Rectangle); *//friend of class Rectangle*

};

*//friend function getCost can access private member of class*

double getCost (Rectangle rect)

{

double cost;

cost = rect.length \* rect.width \* 5;

return cost;

}

int main ()

{

Rectangle floor;

floor.setData(20,3);

cout << "Expense " << getCost(floor) << endl;

return 0;

}

**Output:**

Expense 300

The getCost function is a friend of Rectangle. From within that function we have been able to access the member’s length and width, which are private members.

**Friend Classes**

One class member function can access the private and protected members of other class. We do it by declaring a class as friend of other class. This is illustrated in the following code fragment:

#include <iostream>

using namespace std;

class CostCalculator;

class Rectangle

{

private:

int length;

int width;

public:

void setData(int len, int wid)

{

length = len;

width = wid;

}

int getArea()

{

return length \* width ;

}

friend class CostCalculator; *//friend of class Rectangle*

};

*//friend class costCalculator can access private member of class Rectangle*

class CostCalculator

{

public:

double getCost (Rectangle rect)

{

double cost;

cost = rect.length \* rect.width \* 5;

return cost;

}

};

int main ()

{

Rectangle floor;

floor.setData(20,3);

CostCalculator calc;

cout << "Expense " << calc.getCost(floor) << endl;

return 0;

}

**Output:**

Expense 300

**Note:** An empty declaration of class costCalculator at top is necessary.

**Operator Overloading in C++**

Operator overloading is giving new functionality to an existing operator. It means the behavior of operators when applied to objects of a class can be redefined. It is similar to overloading functions except the function name is replaced by the keyword operator followed by the operator’s symbol. There are 5 operators that are forbidden to overload. They are **:: . .\* sizeof ?:**

In the following code fragment, we will overload binary + operator for Complex number class object.

#include <iostream>

using namespace std;

class Complex

{

private :

double real;

double imag;

public:

Complex () {};

Complex (double, double);

Complex operator + (Complex);

void print();

};

Complex::Complex (double r, double i)

{

real = r;

imag = i;

}

Complex Complex::operator+ (Complex param)

{

Complex temp;

temp.real = real + param.real;

temp.imag = imag + param.imag;

return (temp);

}

void Complex::print()

{

cout << real << " + i" << imag << endl;

}

int main ()

{

Complex c1 (3.1, 1.5);

Complex c2 (1.2, 2.2);

Complex c3;

c3 = c1 + c2; *//use overloaded + operator*

c1.print();

c2.print();

c3.print();

return 0;

}

**Output:**

3.1 + i1.5

1.2 + i2.2

4.3 + i3.7

In C++ we can cause an operator to invoke a member function by giving that member function a special name (of the form: operator<symbol>). Hence for the sum operation, the special name is: operator+. So, by naming the member function operator+ we can call the function by statement

c3 = c1 + c2

That is similiar to

c3 = c1.operator+(c2);

**Inheritance**

The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance.

Inheritance lets you create new classes from existing class. Any new class that you create from an existing class is called **derived class**, existing class is called **base class**.

The inheritance relationship enables a derived class to inherit features from its base class. Furthermore, the derived class can add new features of its own. Therefore, rather than create completely new classes from scratch, you can take advantage of inheritance and reduce software complexity.

**Forms of Inheritance**

**Single Inheritance:**

It is the inheritance hierarchy wherein one derived class inherits from one base class.

**Multiple Inheritance:**

It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es).

**Hierarchical Inheritance:**

It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

**Multilevel Inheritance:**

It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

**Hybrid Inheritance:**

The inheritance hierarchy that reflects any legal combination of other four types of inheritance.



In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

class derived class: memberAccessSpecifier base class

{

...

};

Where derived\_class is the name of the derived class and base\_class is the name of the class on which it is based. The member Access Specifier may be public, protected or private. This access specifier describes the access level for the members that are inherited from the base class.

|  |  |
| --- | --- |
| **Member Access Specifier** | **How Members of the Base Class Appear in the Derived Class** |
| Private | Private members of the base class are inaccessible to the derived class. |
| Protected members of the base class become private members of the derived class. |
| Public members of the base class become private members of the derived class. |
| Protected | Private members of the base class are inaccessible to the derived class. |
| Protected members of the base class become protected members of the derived class. |
| Public members of the base class become protected members of the derived class. |
| Public | Private members of the base class are inaccessible to the derived class. |
| Protected members of the base class become protected members of the derived class. |
| Public members of the base class become public members of the derived class. |

*In principle, a derived class inherits every member of a base class except constructor and destructor. It means private members are also become members of derived class. But they are inaccessible by the members of derived class.*

Following example further explains concept of inheritance:



class Shape

{

protected:

float width, height;

public:

void set\_data (float a, float b)

{

width = a;

height = b;

}

};

class Rectangle: public Shape

{

public: float area ()

{

return (width \* height);

}

};

class Triangle: public Shape

{

public: float area ()

{

return (width \* height / 2);

}

};

int main ()

{

Rectangle rect;

Triangle tri;

rect.set\_data (5,3);

tri.set\_data (2,5);

cout << rect.area() << endl;

cout << tri.area() << endl;

return 0;

}

**Output:**

15 5

The object of the class Rectangle contains:

width, height inherited from Shape becomes the protected member of Rectangle.

set\_data() inherited from Shape becomes the public member of Rectangle area is Rectangle’s own public member.

The object of the class Triangle contains:

width, height inherited from Shape becomes the protected member of Triangle.

set\_data() inherited from Shape becomes the public member of Triangle area is Triangle’s own public member.

set\_data () and area() are public members of derived class and can be accessed from outside class i.e. from main().

**Constructor and Inheritance**

The compiler automatically calls a base class constructor before executing the derived class constructor. The compiler’s default action is to call the default constructor in the base class. You can specify which of several base class constructors should be called during the creation of a derived class object.

This is done by specifying the arguments to the selected base class constructor in the definition of the derived class constructor.

class Rectangle

{

private :

float length;

float width;

public:

Rectangle ()

{

length = 0;

width = 0;

}

Rectangle (float len, float wid)

{

length = len;

width = wid;

}

float area()

{

return length \* width ;

}

};

class Box : public Rectangle

{

private :

float height;

public:

Box ()

{

height = 0;

}

Box (float len, float wid, float ht) : Rectangle(len, wid)

{

height = ht;

}

float volume()

{

return area() \* height;

}

};

int main ()

{

Box bx;

Box cx(4,8,5);

cout << bx.volume() << endl;

cout << cx.volume() << endl;

return 0;

}

**Output:**

0 160

**Overriding Base Class Functions**

A derived class can override a member function of its base class by defining a derived class member function with the same name and parameter list. It is often useful for a derived class to define its own version of a member function inherited from its base class. This may be done to specialize the member function to the needs of the derived class. When this happens, the base class member function is said to be overridden by the derived class.

class mother

{

public:

void display ()

{

cout << "mother: display function\n";

}

};

class daughter : public mother

{

public:

void display ()

{

cout << "daughter: display function\n\n";

}

};

int main ()

{

daughter rita;

rita.display();

return 0;

}

**Output:**

daughter: display function

**Gaining Access to an Overridden Function**

It is occasionally useful to be able to call the overridden version. This is done by using the scope resolution operator to specify the class of the overridden member function being accessed.

class daughter : public mother

{

public:

void display ()

{

cout << "daughter: display function\n\n";

mother::display();

}

};

**Output:**

daughter: display function

mother: display function

**Virtual Base Class**

Multipath inheritance may lead to duplication of inherited members from a grandparent base class. This may be avoided by making the common base class a virtual base class. When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited.



class A

{

.....

.....

};

class B1 : virtual public A

{

.....

.....

};

class B2 : virtual public A

{

.....

.....

};

class C : public B1, public B2

{

..... // only one copy of A

..... // will be inherited

};

**Polymorphism, Virtual Functions and Abstract Class**

In C++, a pointer variable of a base class type can point to an object of its derived class. There are situations when this feature of C++ can be used to develop generic code for a variety of applications.

**Pointer of base class**

Consider the following program to understand pointer compatibility property

#include <iostream>

using namespace std;

class Shape

{

protected:

double width, height;

public:

void set\_data (double a, double b)

{

width = a;

height = b;

}

};

class Rectangle: public Shape

{

public:

double area ()

{

return (width \* height);

}

};

int main ()

{

Shape \*sPtr; *//declare pointer variables of type Shape*

Rectangle Rect; *//create the object rect of type Rectangle*

sPtr = &Rect; *//make sPtr point to the object rect.*

sPtr->set\_data (5,3); *//set length and width of object rect*

cout << sPtr -> area() << endl; *//Compile Error !!*

return 0;

}

Notice that even though rectPtr is pointing to rect (object of type Rectangle), when the program executes, the statement sets length and width of rectangle. If you tried to access area function of class Rectangle with sPtr it will give you compiler error.

sPtr -> area()

is a compiler error !

It means **base class pointer can not access the additional member function of its derived class**. If we want to do this we need to type cast the base class pointer.

**Using Type Casts with Base Class Pointers**

We can use a type cast to get the compiler to accept the statement:

static\_cast <Rectangle \*> (sPtr)->area()

so we should write the statment

cout << static\_cast <Rectangle \*> (sPtr) -> area() << endl;

The type cast informs the compiler that sPtr is actually pointing to a Rectangle object derived from the Shape base class. In general, **a pointer to a base class that actually points to a derived class object must first be appropriately cast before the additional features of the derived class can be used.**

**Virtual Function and Polymorphism**

Virtual functions are used in C++ to support polymorphic behavior. We are modifing the above program and will introduce you the concept of virtual function by following example:

#include <iostream>

using namespace std;

class Shape

{

protected:

double width, height;

public:

void set\_data (double a, double b)

{

width = a;

height = b;

}

virtual double area()

{

return 0;

}

};

class Rectangle: public Shape

{

public:

double area ()

{

return (width \* height);

}

};

int main ()

{

Shape \*sPtr;

Rectangle Rect;

sPtr = &Rect;

sPtr -> set\_data (5,3);

cout << sPtr -> area() << endl;

return 0;

}

**Output:**

15

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual. The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. **The advantage of having virtual function is that we are able to access** area **function of derived class by pointer variable of base class.**

**Pure Virtual Function and Abstract Class**

In above example, base class Shape member function area do not need any implementation because it is overriding in derived class. If this is the case, the C++ language permits the programmer to declare the function a pure virtual function. The C++ way of declaring a pure virtual function is to put the expression = 0 in the class declaration. For example, if a member function double area() is being declared pure virtual, then its declaration in its class looks like

virtual double area() = 0;

A pure virtual function is sometimes called an abstract function, and a class with at least one pure virtual function is called an abstract class. The C++ compiler will not allow you to instantiate an abstract class. Abstract classes can only be subclassed: that is, you can only use them as base classes from which to derive other classes.

A class derived from an abstract class inherits all functions in the base class, and will itself be an abstract class unless it overrides all the abstract functions it inherits. The usefulness of abstract classes lies in the fact that they define an interface that will then have to be supported by objects of all classes derived from it.

#include <iostream>

using namespace std;

class Shape

{

protected:

double width, height;

public:

void set\_data (double a, double b)

{

width = a;

height = b;

}

virtual double area() = 0;

};

class Rectangle: public Shape

{

public:

double area ()

{

return (width \* height);

}

};

class Triangle: public Shape

{

public:

double area ()

{

return (width \* height)/2;

}

};

int main ()

{

Shape \*sPtr;

Rectangle Rect;

sPtr = &Rect;

sPtr -> set\_data (5,3);

cout << "Area of Rectangle is " << sPtr -> area() << endl;

Triangle Tri;

sPtr = &Tri;

sPtr -> set\_data (4,6);

cout << "Area of Triangle is " << sPtr -> area() << endl;

return 0;

}

**Output:**

Area of Rectangle is 15

Area of Triangle is 12

1. **File Handling**

**Data File Handling in C++**

**File**

The information / data stored under a specific name on a storage device, is called a file.

**Stream:**

It refers to a sequence of bytes.

**Text file:**

It is a file that stores information in ASCII characters. In text files, each line of text is terminated with a special character known as EOL (End of Line) character or delimiter character. When this EOL character is read or written, certain internal translations take place.

**Binary file:**

It is a file that contains information in the same format as it is held in memory. In binary files, no delimiters are used for a line and no translations occur here.

**Classes for file stream operation**

**ofstream**: Stream class to write on files.

**ifstream**: Stream class to read from files.

**fstream:** Stream class to both read and write from/to files.

**Opening a file**

**OPENING FILE USING CONSTRUCTOR**

ofstream outFile("sample.txt"); //output only

ifstream inFile(“sample.txt”); //input only.

**OPENING FILE USING open()**

Stream-object.open(“filename”, mode)

ofstream outFile;

outFile.open("sample.txt");

ifstream inFile;

inFile.open("sample.txt");

|  |  |
| --- | --- |
| **File mode parameter** | **Meaning** |
| ios::app | Append to end of file |
| ios::ate | go to end of file on opening |
| ios::binary | file open in binary mode |
| ios::in | open file for reading only |
| ios::out | open file for writing only |
| ios::nocreate | open fails if the file does not exist |
| ios::noreplace | open fails if the file already exist |
| ios::trunc | delete the contents of the file if it exist |

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open():

fstream file;

file.open ("example.bin", ios::out | ios::app | ios::binary);

**Closing File**

outFile.close();

inFile.close();

**INPUT AND OUTPUT OPERATION**

**put() and get() function**

the function put() writes a single character to the associated stream. Similarly, the function get() reads a single character form the associated stream.

**Example:**

file.get(ch);

file.put(ch);

**write() and read() function**

write() and read() functions write and read blocks of binary data.

**Example:**

file.read((char \*)&obj, sizeof(obj));

file.write((char \*)&obj, sizeof(obj));

**ERROR HANDLING FUNCTION**

|  |  |
| --- | --- |
| **FUNCTION** | **RETURN VALUE AND MEANING** |
| eof() | returns true (non zero) if end of file is encountered while reading; otherwise return false(zero) |
| fail() | return true when an input or output operation has failed |
| bad() | returns true if an invalid operation is attempted or any unrecoverable error has occurred. |
| good() | returns true if no error has occurred. |

**File Pointers and Their Manipulation**

All i/o streams objects have, at least, one internal stream pointer:

ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.

ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

|  |  |
| --- | --- |
| seekg() | moves get pointer(input) to a specified location |
| seekp() | moves put pointer (output) to a specified location |
| tellg() | gives the current position of the get pointer |
| tellp() | gives the current position of the put pointer |

The other prototype for these functions is:

seekg(offset, refposition );

seekp(offset, refposition );

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition takes one of the following three constants defined in the ios class.

**ios::beg** start of the file.

**ios::cur** current position of the pointer.

**ios::end** end of the file.

**Example:**

file.seekg(-10, ios::cur);

**Basic Operation On Text File in C++**

File I/O is a five-step process:

* Include the header file fstream in the program.
* Declare file stream object.
* Open the file with the file stream object.
* Use the file stream object with >>, <<, or other input/output functions.
* Close the files.

Following program shows how the steps might appear in program.

**Program to write in a text file**

#include <fstream>

using namespace std;

int main()

{

ofstream fout;

fout.open("out.txt");

char str[300] = "Time is a great teacher but unfortunately

it kills all its pupils.Berlioz"; //Write string to the file.

fout << str;

fout.close();

return 0;

}

**Program to read from text file and display it**

#include<fstream>

#include<iostream>

using namespace std;

int main()

{

ifstream fin;

fin.open("out.txt");

char ch;

while(!fin.eof())

{

fin.get(ch);

cout << ch;

}

fin.close();

return 0;

}

**Program to count number of characters**

#include<fstream>

#include<iostream>

using namespace std;

int main()

{

ifstream fin;

fin.open("out.txt");

int count = 0;

char ch;

while(!fin.eof())

{

fin.get(ch);

count++;

}

cout << "Number of characters in file are " << count;

fin.close();

return 0;

}

**Program to count number of words**

#include<fstream>

#include<iostream>

using namespace std;

int main()

{

ifstream fin;

fin.open("out.txt");

int count = 0;

char word[30];

while(!fin.eof())

{

fin >> word;

count++;

}

cout << "Number of words in file are " << count;

fin.close();

return 0;

}

**Program to count number of lines**

#include<fstream>

#include<iostream>

using namespace std;

int main()

{

ifstream fin;

fin.open("out.txt");

int count = 0;

char str[80];

while(!fin.eof())

{

fin.getline(str,80);

count++;

}

cout << "Number of lines in file are " << count;

fin.close();

return 0;

}

**Program to copy contents of file to another file**

#include<fstream>

using namespace std;

int main()

{

ifstream fin;

fin.open("out.txt");

ofstream fout;

fout.open("sample.txt");

char ch;

while(!fin.eof())

{

fin.get(ch);

fout << ch;

}

fin.close();

return 0;

}

**Basic Operation on Binary File in C++**

When data is stored in a file in the binary format, reading and writing data is faster because no time is lost in converting the data from one format to another format. Such files are called binary files. This following program explains how to create binary files and also how to read, write, search, delete and modify data from binary files.

#include<iostream>

#include<fstream>

#include<cstdio>

using namespace std;

class Student

{

int admno;

char name[50];

public:

void setData()

{

cout << "\nEnter admission no. ";

cin >> admno;

cout << "Enter name of student ";

cin.getline(name,50);

}

void showData()

{

cout << "\nAdmission no. : " << admno;

cout << "\nStudent Name : " << name;

}

int retAdmno()

{

return admno;

}

};

/\* \* function to write in a binary file. \*/

void write\_record()

{

ofstream outFile;

outFile.open("student.dat", ios::binary | ios::app);

Student obj;

obj.setData();

outFile.write((char\*)&obj, sizeof(obj));

outFile.close();

}

/\* \* function to display records of file \*/

void display()

{

ifstream inFile;

inFile.open("student.dat", ios::binary);

Student obj;

while(inFile.read((char\*)&obj, sizeof(obj)))

{

obj.showData();

}

inFile.close();

}

/\* \* function to search and display from binary file \*/

void search(int n)

{

ifstream inFile;

inFile.open("student.dat", ios::binary);

Student obj;

while(inFile.read((char\*)&obj, sizeof(obj)))

{

if(obj.retAdmno() == n)

{

obj.showData();

}

}

inFile.close();

}

/\* \* function to delete a record \*/

void delete\_record(int n)

{

Student obj;

ifstream inFile;

inFile.open("student.dat", ios::binary);

ofstream outFile;

outFile.open("temp.dat", ios::out | ios::binary);

while(inFile.read((char\*)&obj, sizeof(obj)))

{

if(obj.retAdmno() != n)

{

outFile.write((char\*)&obj, sizeof(obj));

}

}

inFile.close();

outFile.close();

remove("student.dat");

rename("temp.dat", "student.dat");

}

/\* \* function to modify a record \*/

void modify\_record(int n)

{

fstream file;

file.open("student.dat",ios::in | ios::out);

Student obj;

while(file.read((char\*)&obj,sizeof(obj)))

{

if(obj.retAdmno() == n)

{

cout << "\nEnter the new details of student";

obj.setData();

int pos = -1 \* sizeof(obj);

file.seekp(pos, ios::cur);

file.write((char\*)&obj, sizeof(obj));

}

}

file.close();

}

int main()

{

//Store 4 records in file

for(int i = 1; i <= 4; i++)

write\_record(); //Display all records

cout << "\nList of records";

display(); //Search record

cout << "\nSearch result";

search(100); //Delete record

delete\_record(100);

cout << "\nRecord Deleted"; //Modify record

cout << "\nModify Record 101 ";

modify\_record(101);

return 0;

}