**GENERATIONS OF PROGRAMMING LANGUAGES:**

As we know that programming languages are the primary tool for creating software. The concept of generations of programming languages is closely connected to the advance in technology. The five generations of programming languages include machine language, assembly language, high-level language, very high-level language, and artificial intelligence.

**First Generation: Machine Languages**

1. This is the lowest level of programming languages and is the only language that a computer understands. All the command and data values are expressed using 0’s and 1’s. Code can be directly executed by the computer.

**Second Generation: Assembly Language**

This language is also called as 2nd Generation Language. All the instructions are written in Assembly level languages are in the form of mnemonics. Mnemonic means symbolic name. ADD for addition, MUL for multiplication, SUB for subtraction, DIV for division etc. So, there is a need of translator (Assembler). Assembler is a system, which translate assembly level instructions in to machine understandable form.

**Third Generation: High-level Language**

Third-Generation programming languages are a modification of 2GLs. The third generation was introduced to make the languages more programmers friendly. Programs were written in languages that were more English-like, making them more convenient to use. High-level programming languages make a complex programming simpler and easier to read, write and maintain.

A translator is needed to translate the instructions written in high-level language into the computer-executable machine language. Such translators are commonly known as interpreters and compilers. Programs written in such languages are portable between machines. For example, a program written in standard C can be compiled and executed on any computer that has a standard C compiler. BASIC, FORTAN, PASCAL, COBOL, C, C++ and JAVA are few examples of third generation programming languages.

**Fourth Generation: Very High-level Languages**

Fourth generation programming languages are non-procedural. While using non-procedural language, programmers define what they want the computer to do but they do not supply all the details of how it has to be done.

4GL characteristics:

1. The instruction of the code is written in English-like sentences.
2. They are non-procedural, so users concentrate on the ‘what’ instead of the ‘how’ aspect of the task.
3. The code written in 4GL is easy to maintain.
4. The written in 4GL enhances the productivity of the programmers.

A typical example of the 4GL is the query language, which allows a user to request information from a database. A very high-level language is usually limited to specific application and uses syntax that is never used in other programming languages.

**Fifth Generation Programming Language:**

Fifth generation programming languages are centred on solving problems using the constraints given to a program rather than using an algorithm written by a programmer. These languages are widely used in artificial intelligence research. It contains visual tools to help to develop a program. Examples of 5GLs include prolog, Mercury and Visual Basic.

Fifth generation programming allows people to interact with computers without needing any specialised knowledge. People can talk to computers and the voice recognition systems can convert spoken sounds into written words.

**Difference between compiler and interpreter**

|  |  |
| --- | --- |
| **Compiler** | **Interpreter** |
| 1. It translates the entire program in one go. 2. It generates error(s) after translating the entire program. 3. Execution of code is faster. 4. An object file is generated. 5. Code need not be recompiled every time it is executed. 6. It merely translates the code. 7. It requires some memory space. | 1. It interprets and executes one statement at a time. 2. It stops translation after getting the first error. 3. Execution of code is slower as every time reinterpretation of statements has to be done. 4. No object file is generated. 5. Code has to be reinterpreted every time it is executed. 6. It translates as well as executes the code. 7. It requires less memory space. |

**OBJECT-ORIENTED PROGRAMMING PARADIGM**

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allow us to decompose a problem into a number of entities called Objects and then builds data and functions around these objects.

Object A Object B

Data

Functions

Data

Functions

Data

Functions

**Features of object-oriented programming:**

1. Emphasis on data rather than procedure.
2. Programs are divided into what are known as Objects.
3. Data structures are designed such that they characterize the objects.
4. Methods that operate on the data of an object are tied together in the data structure.
5. Data is hidden and cannot be accessed by external functions.
6. Objects may communicate with each other through methods.
7. New data and methods can be easily added whenever necessary.
8. Follows bottom-up approach in program design.

**BASIC CONCEPTS OF OOPS**

The basic concepts of OOPs are

1. Objects
2. Classes
3. Data Abstraction
4. Data Encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic Binding
8. Message Passing

**Objects:** Objects are basic run time entities in an object oriented programming. An object may be defined as an identifiable entity with some state and behaviour. We are often surrounded by a variety of objects. The computer we may be working on, the chair we sit on, a telephone, a clock, etc. are all objects.

**Classes:** The most important feature of Object-Oriented Programming is the Classes. A class is the way to bind the data and its associated functions together. It allows the data to be hidden if necessary from the external use. **(OR)**

A Class can be defined as a template/blue print that describes the behaviours/states that object of its type support.

**Data Abstraction:** Abstraction refers to the act of representing essential features without including the background details or explanations.

**Data Encapsulation:** The wrapping of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.

**Inheritance:** Inheritance is one of the most powerful features of OOP’s. Inheritance is a process of creating a new class from the existing class. The new class inherits all the capabilities of the existing class.

**Polymorphism:** Polymorphism comes from the Greek words “Poly” and “morphism”, “Poly” means many and “Morphism” means form i.e., many forms. In object oriented programming polymorphism refers to identically named methods have different behaviour depending on the type of object.

**SHAPE**

**TRIANGLE**

**RECTANGLE**

**CIRCLE**

**Dynamic Binding:** Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.

**Message passing:** AMessage for an object is request in the receiving object that generates the desired result.

**BENEFITS OF OOP:**

1. The principle of data hiding helps the programmer to build secure programs.
2. It is easy to partition the work in a project based on objects.
3. Object-Oriented systems can be easily upgraded from small to large systems.
4. Through Inheritance, we can eliminate redundant code and extend the use of existing classes.
5. OOP provides an advantage in production and maintenance of software.
6. Object-Oriented program involves the identification and implementation of different classes of objects and their behaviour.
7. Software complexity can be easily managed.
8. It is possible to have multiple instances of an object to co-exist without any interference.
9. OOP leads to saving of development time and higher productivity.

**APPLICATIONS OF OOP:**

Applications of OOP are beginning to gain importance in many areas. The promising areas for application of OOP include:

1. Artificial intelligence and expert systems
2. Simulation and modelling studies.
3. Object-oriented database systems
4. Object-oriented operating systems
5. Real-time systems.
6. Office automation systems.
7. CAD/CAM systems
8. Multimedia applications
9. Graphical User Interface (GUI)
10. Computer – based training and education systems.

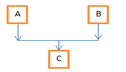
**DIFFERENCE BETWEEN OBJECT ORIENTED AND PROCEDURE ORIENTED PROGRAMMING.**

|  |  |
| --- | --- |
| **Procedure Oriented Programming** | **Object Oriented Programming** |
|  |  |
| Large programs are divided into smaller programs are called as functions. | Programs are divided into classes and objects. |
| Emphasis is on doing things. | Emphasis is on data. |
| It does not support real world applications. | It supports real world applications. |
| Follows top down approach in program design. | Follow bottom up approach in program design. |
| Functions share global data. | Data is hidden and can’t be accessed for external functions. |
| Functions transform data from one form to another. | Object transaction the data. |
| It does not support functions overloading. | It supports functions overloading. |
| Destructor doesn’t support. | Destructor supports. |
| Doesn’t support templates. | It supports |
| Operators overloading is not possible. | It support operator overloading. |
| Inheritance is not possible. | It supports inheritance. |

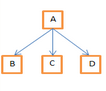
**TYPES OF INHERITANCE:-**

1. Single Inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance
6. Multipath inheritance

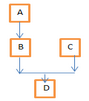
**Single Inheritance:** Derivation of a class from only one base class is called Single inheritance.

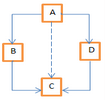
**Multiple Inheritance:** Derivation of a class from one or more base classes is called multiple inheritance.

........

**Hierarchical Inheritance:** Derivation of several classes from one base class i.e. one class may be inherited by more than one class is called hierarchical inheritance.

**Multilevel Inheritance:** Derivation of a class from another derived class is called Multilevel Inheritance.

**Hybrid Inheritance:** Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.

**Multipath Inheritance:** Derivation of a class from other derived classes which are derived from the same base class is called multi-path inheritance.

**OBJECT ORIENTED LANGUAGES**

Different types of object oriented programming languages are:

1. **C++:** C++ is a general purpose, object-oriented programming language developed by Bjarne Stroustrup at Bell labs in the year 1979. Initially it was named as “C with Classes”. Later this was named as C++. C++ supports object-oriented programming features like Encapsulation, Inheritance, and Polymorphism.
2. **Java:** Java is an object-oriented programming language introduced by Sun Microsystem in the year 1995. It was originally developed in the year 1991 by James Gosling and his team. Though the syntax and semantics of Java is similar to that of C++. Java is the first language to bring animation and interactivity to Internet-based applications.
3. **C#:** C# pronounced as “C-sharp” is a new object-oriented programming language developed by Microsoft in late 1990s. It combines the power of C++ with the programming ease of Visual Basic. C# contains features similar to those of Java.
4. **Smalltalk:** Smalltalk is purely an object-oriented programming language developed at Xerox’s Palo Alto Research Center (PARC). Smalltalk programs are written using smalltalk browser. Smalltalk use dynamic objects and memory is allocated from free store. It also provides automatic garbage collection and memory is released when objects is no longer in use.
5. **Charm++:** Charm++ is also object oriented programming language. This language supports inheritance, overloading and reusability.

**BRIEF HISTORY OF C++**

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980’s. Stroustrup wanted to combine the features of Simula 67 and C languages, to develop a more powerful language that could support object-oriented programming. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct features of Simula67. Stroustrup initially called the new language ‘C with classes’. In 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++. C++ is a superset of C.

**DIFFERENCE BETWEEN C AND C++**

|  |  |
| --- | --- |
| **C** | **C++** |
| 1. It is developed by Dennis Ritchie. | 1. It is developed by Bjarne Stroustrup. |
| 1. C follows procedural style programming. | 1. C++ follows both procedural and object-oriented. |
| 1. Data is less secured in C. | 1. Data is secured. |
| 1. C doest not support scope operator(::). | 1. C++ supports the scope operator(::). |
| 1. C follow top-down approach. | 1. C++ follows bottom-up approach. |
| 1. C language does not support the information hiding. | 1. C++ language supports the information hiding. |
| 1. C does not support exception handling. | 1. C++ supports the exception handling. |
| 1. C does not support access specifiers | 1. C++ supports access specifiers. |
| 1. C does not support inheritance. | 1. C++ supports inheritance. |
| 1. C does not support templates. | 1. C++ supports templates. |
| 1. C does not support polymorphism. | 1. C++ supports polymorphism. |
| 1. C does not support operator overloading. | 1. C++ supports operator overloading. |
| 1. C does not support constructor and destructor. | 1. C++ supports constructor and destructor. |
| 1. C does not support function overloading | 1. C++ supports function overloading. |

**GENERAL STRUCTURE OF C++**

**Include files**

**Class declaration**

**Member function definitions**

**Main function program**

1. **Include Files:** C++ programs depend upon some header files for function definitions. Each header file starts with **#include**.

**Ex: #include <iostream>**

1. **Class declaration or definition:** A class is declared in this section. A class contain data variables, function prototypes. The class definition is always terminated by a semicolon (;).

**Ex: class example**

**{**

**};**

1. **Member Function Definitions:** This contains definition of the functions. The function definition can be done outside or inside the class.
2. **The Main() function:** Execution of a program starts from the main function. Main is not a keyword it is a special word.

**Example:**

#include <iostream>

#include<conio.h>

main()

{

cout<< “ HELLO “;

}

**CHARACTER SET**

A character set denotes any alphabet, digit or special symbol used to represent information.

|  |  |
| --- | --- |
| Alphabets | A, B, ….., Y, Z  a,b ……, y, z |
| Digits | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| Special symbols | ~ ‘ ! @ # % ^ & \* ( ) \_ - + = | \ { }  [ ] : ; " ' < > , . ? / |

**COMMENTS**

Comments are like helping text in your C++ program and they are ignored by the compiler.

There are two types of comments.

1. Single Line Comments
2. Multi Line Comments

Single line comments are represented by double slash //.

**Example:** // printing information

Multi line comments are represented by slash asterisk /\*…\*/. It occupies of many lines of code.

**Example:**

/\*

Code

To be commented

\*/

**C++ TOKENS**

The smallest individual units in a program are known as tokens. C has the following tokens.

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators
6. **Keywords:** The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

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

1. **Identifiers**

Identifier refers to the name of variables, functions and arrays. These are the user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted. The underscore character is also permitted in identifiers.

**Rules for identifiers:**

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits, or underscore.
3. Cannot use a keyword.
4. Must not contain white space.

The difference between C and C++ is a limit on the length of a name. C recognizes only the first 32 characters in a name. C++ places no limit on its length.

**Example: int a\_;**

**int \_b;**

**int a\_b;**

1. **Constants**

Constants in C++ refer to fixed values that do not change during the execution of a program. C supports types of constants as explained below:

**Integer Constants:**

An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal integer, octal integer and hexadecimal integer

**Decimal integer:** It consists of a set of digits, 0-9. It may be either positive or negative.

Ex: 3445, 123, -123, -56, 0

**Octal integer:** It consists of any combination of digits from the set 0 to 7 with a leading 0. Octal values have sign

Ex: 034, 037, 0551

**Hexadecimal integer:** It consists of a set of digits 0 to 9 and alphabets A to F to represent the values of 10 to 15. Each Hexadecimal value begins with 0x.

Ex: 0x20xF5, 0x2, 0x9F

**Real Constants:**

A real constant refers to numbers containing fractional parts like 17.4. Such numbers are called real (or floating point) constants.

A real number may also be expressed in exponential notation.

**Ex:** 2.1565e2

e2 means multiply the number by 102.

then the above value becomes 215.65

The general form for is :

mantissa e exponent

**Single Character Constants:**

Any single character or symbol or digits enclosed within a pair of single quote marks is called single character constant.

Ex: ‘B’, ‘3’, ‘!’

**String Constants:**

A string constant is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Ex: “Ashwika”, “2016”, “ MAY1”

**Backslash Character Constants:**

C++ supports some special backslash characters constants that are used in output functions.

|  |  |
| --- | --- |
| **Constant** | **Meaning** |
| ‘\b’ | Back space |
| ‘\f’ | Form feed |
| ‘\n’ | New line |
| ‘\r’ | Carriage return |
| ‘\t’ | Horizontal tab |
| ‘\’’ | Single quote |
| ‘\”’ | Double quote |
| ‘\\’ | Black slash |
| ‘\a’ | Audible alert |
| ‘\v’ | Vertical tab |

1. **Strings**

**Single Character Constants:**

Any single character or symbol or digits enclosed within a pair of single quote marks is called single character constant.

Ex: ‘B’, ‘3’, ‘!’

**String Constants:**

A string constant is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Ex: “Ashwika”, “2016”, “ MAY1”

1. **Operators**

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

C++ operators can be classified into following types,

1. Arithmetic operators
2. Relation operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

An expression is a sequence of operands and operators that reduces to a single value.

1. **Arithmetic operators:** Arithmetic operators are used to perform arithmetic calculations.

|  |  |
| --- | --- |
| Operator | Meaning |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division (which returns quotient value) |
| % | Modulo division (which returns reminder value) |

#include <iostream>

using namespace std;

main()

{

int a,b,c;

a=20,b=3,c=2;

cout<<" \n The sum of a, b, and c is "<<(a+b+c);

cout<<" \n The subtraction of a and b is" <<(a-b);

cout<<" \n Multiplication of a and b and c is "<<(a\*b\*c);

cout<<" \n Division of a and b is "<<a/b;

cout<<" \n Modulo of a and b is "<<a%b;

}

1. **Relational operators:** The relational operators are used to compare two values and gives either true (1) or false (0) result. The following are the relational operators.

|  |  |
| --- | --- |
| Operator | Meaning |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| == | Equal to |
| != | Not equal to |

#include <iostream>

using namespace std;

main()

{

int a,b;

a=20,b=3;

cout<<" \n a>b "<< (a>b);

cout<<" \n a<b "<< (a<b);

cout<<" \n a==b "<< (a==b);

cout<<" \n a<=b "<< (a<=b);

cout<<" \n a>=b "<< (a>=b);

cout<<"\n a!=b "<< (a!=b);

}

1. **Logical operators**: These are used to combine two or more relational expressions and give the result either true or false.

|  |  |
| --- | --- |
| Operator | Meaning |
| && | Logical and |
| || | Logical or |
| ! | Logical not |

#include<iostream>

using namespace std;

main()

{

int a=280,b=34,c=22;

cout<<((a<b)&&(a<c));

}

1. **Assignment operators:** Assignment operators are used to assign the result of an expression to a variable. v op = exp;

|  |  |  |
| --- | --- | --- |
| Operator | Examples | Meaning |
| += | a+=1 | a=a+1 |
| -= | a-=b | a=a-b |
| \*= | a\*=b | a=a\*b |
| /= | a/=b | a=a+b |
| %= | a%=b | a=a%b |

1. **Increment / Decrement operator**: ( ++ /--)

The increment operator is a unary operator. It is used to increase the value of an operand by 1.

The decrement operator is a unary operator. It is used to decrease the value of an operand by 1.

The increment operator '++' and decrement operator ‘- -‘has different meaning depending on the position it is used. It means this operator is used again two ways. They are pre-increment, post-increment and pre-decrement, post-decrement.

Pre-increment: ++m; Post-increment: m++;

Pre-decrement: --m; Post-decrement: m--;

#include<iostream>

using namespace std;

main()

{

int a=1,b=3;

cout<< " \n Post increment of a is "<< a++;

cout<<" \n post decrement of b is "<< b--;

cout<<" \n pre increment of a is "<< ++a;

cout<<" \n pre decrement of b is "<< --b;

}

1. **Conditional operators:**

It is also known as Ternary Operator. The general form of conditional operator is as follows:

**Syntax**: (Exp1)? Exp2: Exp3;

**Working**: Exp1 is evaluated first, if it is true Exp2 will be executed. If Exp1 is false Exp3 will be executed.

#include<iostream>

using namespace std;

main()

{

int a,b,c,max;

cout<< " Enter values for a,b and c \n";

cin>>a>>b>>c;

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

max=(max>c)?max:c;

cout<< " The large number is "<<max;

}

1. **Bitwise Operators:** C++ supports special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing bits or shifting them to right or left. Bitwise operators may not be applied to float or double data type.

|  |  |
| --- | --- |
| Operator | Meaning |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise XOR (Exclusive OR) |
| ~ | 1’s complement |
| >> | Right shift |
| << | Left shift |

**8. Special operators:**

The following are the special operators used in ‘C++’ language.

* Comma operator (,)
* Sizeof operator : This operator returns the size of an operand

**The sizeof Operator:**

The sizeof is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

**Examples:** m=sizeof(sum);

n=sizeof(int);

**VARIABLE**

A **variable** is a name of memory location. It is used to store data. A variable takes different values at different times during execution. The syntax of variable:

Type variable\_list;

C++ supports two types of variables. They are

1. **Numeric variables:** These variables can be used to store either integer values or floating point values.
2. **Character variable:** Character variable can include any letter from the alphabet or from the ASCII chart and numbers 0-9 that are given within single quotations.

Example of declaring a variable is given below:

int a;

float b;

char c;

From above a, b, c are variables and int, float, char are data types.

**Initializing variable:** The process of giving initial values to variables is called initialization.

Example: **int a=10;**

**float b=2.2;**

**Rules for defining variables:**

* A variable can have alphabets, digits and underscore.
* A variable name can start with alphabet and underscore only. It can't start with digit.
* No white space is allowed within variable name.
* A variable name must not be any reserved word or keyword e.g. int, float etc.

**DATA TYPES**

Data types specify what type of data we enter into our programs. The data types can be broadly classified into the following categories.

**Primary Data Types:** These are fundamental or built-in data types in C++ namely integer (int), floating (float), character (char), double.

**Integer type:**

Integers are used to store whole numbers.

**Size and range of Integer type on 16-bit machine**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| int or signed int | 2 | -32,768 to 32767 |
| unsigned int | 2 | 0 to 65535 |
| short int or signed short int | 1 | -128 to 127 |
| long int or signed long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |

**Floating type:**

Floating types are used to store real numbers.

**Size and range of Integer type on 16-bit machine**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| Float | 4 | 3.4E-38 to 3.4E+38 |
| double | 8 | 1.7E-308 to 1.7E+308 |
| long double | 10 | 3.4E-4932 to 1.1E+4932 |

**Character type:**

Character types are used to store characters value.

**Size and range of Integer type on 16-bit machine**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| char or signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |

**User-defined Data Types:** User-defined data types like structure, union, classes, typedef, enum.

**Derived Data Types:** Derived data types like arrays, pointers, functions and references.

**STREAM BASED I/O**

Streams in C++ are classified into

1. **Output Streams (or) Output Statement (cout):** The output stream is allowed to perform write operations on output devices such as screen, disk, etc. The **cout** is a predefined object of **ostream** class. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console.

**Example:**

#include <iostream>

**using** **namespace** std;

**int** main( ) {

**char** ary[] = "Welcome to C++ tutorial";

   cout << "Value of ary is: " << ary << endl;

}

1. **Input Streams (or) Input Statement(cin):** The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

**Example:**

#include <iostream>

**using** **namespace** std;

**int** main( ) {

**int** age;

   cout << "Enter your age: ";

   cin >> age;

   cout << "Your age is: " << age << endl;

}

**SCOPE RESOLUTION OPERATOR**

The scope resolution operator(::) allows a programmer to access a global variable even it is hidden by a local variable.

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

int a=10;

main()

{

int a=20;

cout<<" Global Variable a ="<<::a<<endl;

cout<<"Local Variable a= "<<a<<endl;

::a=100;

cout<<"Global variable a="<<::a;

getch();

}

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

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

1. if statement
2. if-else statement
3. nested if statement
4. if-else-if ladder
5. switch statement

**SIMPLE IF:** The general form of the simple if statement is

The ‘statement - block’ may be a single statement or a group of statements. If the test expression is true, the ‘statement – block’ will be executed, otherwise the ‘statement – block’ will be skipped and the execution will jump to ‘statement –x’.

**Flowchart of simple if:**

#include <iostream>

**using** **namespace** std;

**int** main () {

**int** num = 10;

**if** (num % 2 == 0)

            {

                cout<<"It is even number";

            }

**return** 0;

}

**IF-ELSE STATEMENT:** The general form is

If the test expression is true, then the true-block statement(s) immediately following if statement, are executed. Otherwise, the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both.

**Flow Chart:**

 #include <iostream>

**using** **namespace** std;

**int** main () {

**int** num = 11;

**if** (num % 2 == 0)

            {

                cout<<"It is even number";

            }

**else**

            {

                cout<<"It is odd number";

            }

**return** 0;

}

**NESTING OF IF....ELSE STATEMENT:** The general form of the nested if-else statement is



If the test condition1 is false statement -3 will be executed, otherwise it continues to perform the second test. If the condition – 2 is true, the statement -1 will be evaluated, otherwise the statement -2 will be evaluated and the control is transferred to the statement – x.

**Flow chart:**



#include<iostream>

using namespace std;

main()

{

int a=325,b=13332,c=4978;

if(a>b)

{

if(a>c)

cout<<a;

else

cout<<c;

}

else

{

if(b>c)

cout<<b;

else

cout<<c;

}

}

**ELSE IF LADDER:** The general form is



This construct is known as the else if ladder. The conditions are evaluated from top downwards. As soon as the true condition is found, the statement associated with it is executed and the control is transferred to the ‘statement-x’ (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the default-statement will be executed.

#include<iostream>

#include<conio.h>

using namespace std;

main()

{

int mark;

cout<< " Enter the mark : ";

cin>>mark;

if(mark >= 75)

{

cout<<" Distinction ";

}

else if(mark >=60 && mark <75)

{

cout<<" First Class ";

}

else if (mark >=50 && mark <60)

{

cout<<" Second Class ";

}

else if(mark >=35 && mark <50)

{

cout<< " Third class";

}

else

{

cout<<" Fail ";

}

getch();

}

**Flow chart**:



**SWITCH STATEMENT:**

The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.

The expression is an integer expression or characters. Value-1, value-2... are constants or constant expressions and are known as case labels.

Block-1, block-2 ... are statement lists and may contain zero or more statements. It is important to note that case labels end with a colon (:).

The break statement at the end of each block signal the end of a particular case and causes an exit from the switch statement. The default is an optional case.

The general form is :



#include<iostream>

#include<conio.h>

using namespace std;

main()

{

int dayno;

cout<<"Enter the day number: ";

cin>>dayno;

switch(dayno)

{

case 1: cout<<" Sunday \n";

break;

case 2: cout<<" Monday \n";

break;

case 3: cout<<" Tuesday \n";

break;

case 4: cout<<" Wednesday \n";

break;

case 5: cout<<" Thrusday \n";

break;

case 6 : cout<<" Friday \n";

break;

case 7: cout<<" Saturday \n";

break;

default:

cout<<" out of range";

}

getch();

}

**Flowchart:**

****

**DECISION MAKING AND LOOPING:**

The process of repeatedly executing a block of statements is known as Looping.

The programme uses a loop construct to instruct the computer to perform repetitive tasks for a finite number of times based on conditions. Each loop construct contains two parts namely condition part and body part. The body part of the loop will be executed repeatedly as long as the given condition evaluates to true.

**Looping operations:**

1. **while Loop**
2. **do ..while Loop**
3. **for Loop**



**The while statement:** The general form is

#include<iostream>

using namespace std;

main()

{

int x=1,sum=0;

while(x<=10)

{

sum=sum+x;

x++;

}

cout<<" The sum is “<<sum;

}



The while is an entry-controlled loop statement. The simplest of all the looping structure in C is the while statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process is repeated until the test condition becomes false and the control is transferred out of the loop.

**The do..while Statement:** The general form is



do..while loop is an exit-controlled loop statement as it allows the body of the loop to be executed for the first time without any condition.

At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

**Example:**

#include<iostream>

using namespace std;

main()

{

int x=1, sum=0;

do

{

sum=sum+x;

x++;

}

while(x<=10);

cout<<" The sum is “<<sum;

}

**The for statement:** The general form is



It is a entry-controlled loop.

Execution of the **for** statement is as follows

1. Initialization of the control variable is done first.
2. The value of the control variable is tested using the test condition. The test condition is a relational expression. When the loop is true, the body of the loop is executed. Otherwise the loop is terminated.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now the control variable is incremented or decremented. The new value of the control variable is tested again to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

**Example:**

#include<iostream>

using namespace std;

main()

{

int x,sum=0;

for(x=1;x<=10;x++)

{

sum=sum+x;

}

cout<<" The sum is “<<sum;

}

**Nested Loops:**

Nesting of loops, that is, one **for** statement within another **for** statement is allowed in C++.

**Example:**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

int row=1,col=1;

for(row=1;row<=10;row++)

{

for(col=1;col<=10;col++)

{

cout<<setw(4)<<(col\*row);

}

cout<<endl;

}

getch();

}

**Difference b/w while and do-while:**

|  |  |
| --- | --- |
| **While** | **Do –while** |
| It is an entry-control loop | it is an exit-controlled loop |
| Test condition is tested first | Test condition is tested after executing the body of the loop |
| It is pre-testing loop | It is post-testing loop |
| It will not give guarantee to execute the body of the loop minimum once | It will give the guarantee to execute the body of the loop minimum once |
| Syntax: while (condition)  Simple or compound statement; | Syntax: do  Simple (or) compound statement;  while(condition); |

### GOTO STATEMENT:

The goto statement is an unconditional statement used to transfer the control to any place in a program.

**Syntax**: goto label;

Here the label is a variable used to name the target location to which the control is transferred. The labels name must be unique with in a program. The label can be placed any where in the program either before or after. The target statement must be named with label name followed by a colon as follows:

Label: statement;

|  |  |
| --- | --- |
| Ex: Forward Jump  main()  {  int num;  cout<<"Enter a value";  cin>>num;  if(num<0)  goto end;  cout<<" value= "<<num;  end:cout<<"end of program";  } | Ex: Backward Jump  main()  {  int num;  Input:cout<<“Enter a value”;  cin>>num;  if(num<0)  goto input;  cout<<“value= ”<<num;  } |

**BREAK STATEMENT:**

The break statement is used to terminate the loop. When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

**Syntax:** Break ;

**Example :**

**#**include<iostream>

#include<conio.h>

using namespace std;

main ( )

{ int i,num,n,sum=0;

cout<<” how many values?”;

cin>>n;

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

{

cin>>num;

if (num<0)

break;

sum=sum+num;

}

cout<<” Result = “<<sum;

getch();

}

**CONTINUE STATEMENT:**

Like break statement, C supports another similar statement called the continue statement. However, unlike the break which caused the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler. “SKIP THE FOLLOWING STATEMENTS AND CONTIUNE WITH THE NEXT ITERATION”. The format of the continue statement is simply

**Syntax:** **Continue ;**

**Example:**

**#**include<iostream>

#include<conio.h>

using namespace std;

main ( )

{

int i, n,sum=0,num;

cout<<“how many values?”;

cin>>n;

for(i=1;i<=n,i++)

{

cin>>num;

if(num<0)

continue;

else

sum=sum+num;

}

cout<<“Result= ”<<sum;

getch();

}

**FUNCTIONS:**

A function is a block of code that performs a particular task. There are times when we need to write a particular block of code for more than once in our program. This may lead to bugs and irritation for the programmer. C++ language provides an approach in which you need to declare and define a group of statements once and that can be called and used whenever required. This saves both time and space.

# Advantages of User-defined Functions :(or salient features )

* Modular programming
* Testing and debugging (correction of errors) is easy.
* Understanding the program is easy.
* It reduces size of the program due to code reusability
* Functions can be accessed without redevelopment, which in turn prompts reuse code.
* Code duplication can be avoided
* Can be invoked any number of times and from any locations of the application.

Elements/Component of user defined function are

1) Function Declaration/prototype

2) Function definition (or) called function

3) Function call (or) calling function

**Function Declaration/ Function prototype:**

General syntax of function declaration is:

*return-type function-name (parameter-list);*

Like variables, all functions in a C++ program must be declared, before they are invoked. A function declaration (also known as function prototype) consists of four parts.

1. Function type (return type).
2. Function name.
3. Parameter list.
4. Terminating semicolon.

**Example:**

int mul (int m, int n);

**Points to remember:**

1. The parameters list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.
5. If the function has no formal parameters, the list is written as (void).
6. The return type must be **void** if no value is returned.
7. When the declared type does not match with the types in the function definition, compiler will produce an error.

**Function Definition (or) called function:**

The general syntax of function definition is

return-type function-name(parameter-list)

{

Local variable declarations;

Executable statement1;

Executable statement2; function body

…………..

Return statement;

}

The function definition includes the following elements:

**return type:** return type specifies the type of value(int,float,char,double) that function is expected to return.

**function-name:** function name specifies the name of the function.

**parameter-list:** The parameter list declares the variables that will receive the data sent by calling program. They often referred to as formal parameters. These parameters are also used to send values to calling program. A function can also defined without arguments.

**Function body:** The function body contains the declarations and statements necessary for performing the required task. The body is enclosed with curly braces { } and consists of three parts.

* Local variable declaration.
* Executable statements.
* Return statement.

**Example:**

float mul( float x, float y)

{

float result;

result= x+y;

return (result);

}

**Function Call or calling function:**

A function can be called by simply using the function name followed by a list of actual parameters, if any, enclosed in parentheses.

Whenever a function is called, the execution control is transferred to that function. After completion of function execution, the control returns to the calling point.

The general form of calling a function is as follows:

**Syntax:**

**f**unction-name(data-type arg1, data-type arg2, ……., data-type argn);

Where, ‘function name’ is name of user-defined functio. Arg1, arg2, ….., arg-n are the values that are send to the function. These are referred as actual arguments.

While accessing a function, the following points must be remembered.

1. The number and order of actual parameters and formal parameters must be same.
2. When a function returns a value to its calling point then the returned value must be assigned to a variable or it must be used in the output function.

**Example:**

sum(a,b);

square(10);

**Return Statement:(Return values and their Types)**

The return statement is used to return a value from a user-defined function to its calling point. The statement can be used in the following ways.

**Syntax-1**: return;

**Syntax-2**: return (value);

**Examples:**

1.return;

2. return (25);

3. return (x+y);

4. return x+y;

There are two types of parameters. They are

1. Actual Parameters and
2. Formal parameters.

**Actual Parameters:** The parameters that are included in function calling point are called “actual parameters”. These are used to send vales to the called function.

**Formal Parameters:** The parameters that are included in function definition are called “formal parameters”. These are used to receive values from the calling point.

While using actual and formal parameters the following rules must be followed:

1. The number of actual parameters must be equal to the number of formal parameters.
2. The data types of actual parameters must be same to their corresponding formal parameters.
3. The actual and formal parameters may or may not be having the same names.

#include<iostream>

#include<conio.h>

using namespace std;

int sum(int a,int b); // function prototype

main()

{

int x,y,z;

cout<<” Enter x,y values: ";

cin>>x>>y;

z=sum(x,y); //actual arguments

Function call or calling function

cout<<” sum: "<<z;

getch();

}

int sum(int a,int b) // formal arguments

Function definition or called function

{

return(a+b);

}

**FUNCTION OVERLOADING OR FUNCTION POLYMORPHISM**

Function overloading means, that we can use the same function name to create functions that performs a variety of different tasks.

**Example:**

#include<iostream>

using namespace std;

test(int a)

{

cout<<a<<" ";

}

test(int a, int b)

{

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

}

main()

{

int x;

cin>>x;

test( x);

int x1=23, y1=233;

test( x1, y1);

}

**INLINE FUNCTIONS**

The C++ provides a mechanism called inline function. When a function is declared as inline, the compiler copies the code of the function body and inserted in place of function call during compilation. Passing of control between function call and function definition is avoided.

**Example:**

#include<iostream>

using namespace std;

inline int square(int x)

{

return (x\*x);

}

inline int area(int l ,int b ,int h)

{

return(l\*b\*h);

}

main()

{

int n;

cout<<" Enter the number ";

cin>>n;

cout<<" Square ="<<square(n);

cout<<" area ="<<area(1,3,5);

}

**ARRAYS**

An array is a collection of elements of same type that share a common name. The elements of the array are stored in consecutive memory locations and are referred by an index (also known as subscript).

**Array types:**

1. One Dimensional Arrays
2. Two dimensional Arrays
3. Multi Dimensional Arrays

**ONE DIMENSIONAL ARRAYS**

When array is declared with only one dimension (subscript) then it is called one dimensional array or single dimensional array.

**Declaration of one-dimensional array:**

**Syntax:**

**Type array\_name[size];**

1. The type specifies the type of element that will be contained in the array, such as int, float, or char.
2. Size indicates the maximum number of elements that can be stored inside the array.
3. Array\_name is an identifier that specifies the name of the array.

**Ex**:

**int arr[5];**

0 1 2 3 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

a[0] a[1] a[2] a[3] a[4]

**Initialization of arrays:**

After an array is declared, its elements must be initialized. Otherwise, they will contain garbage value.

Arrays can be initialized as follows:

**Syntax:**

Datatype array\_name[size]={elements separated by commas};

**Ex:**

int arr[5]={5,7,3,9,1};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 3 | 9 | 1 |

arr[0] arr[1] arr[2] arr[3] arr[5]

**Accessing array elements:**

Once an array is defined, its elements can be accessed by using an index or subscript.

**Syntax:** arr\_name[index];

**Example:** arr[4];

Points to the element five of the given list.

**Example:**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

int a[8]={1,3,6,36,4,7,8,9};

int i;

for(i=1;i<8;i++)

{

cout<<a[i]<<setw(3);

}

getch();

}

**Sorting program:**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

int arr[6]={6,3,7,2,1,9};

int i,j;

printf(" \n Elements after sorting are : \n ");

for(i=0;i<6;i++)

{

for(j=i+1;j<6;j++)

{

if(arr[i]>arr[j])

{

int temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

}

for(i=0;i<6;i++)

{

cout<<arr[i]<<setw(4);

}

getch();

}

**TWO-DIMENSIONAL ARRAY**

When an array uses only two subscripts then it is called “Two dimensional array”. It can be viewed as table of elements, which contains rows and columns. A Two dimensional array is useful for matrix operations.

**Declaration of two-dimensional arrays:**

**Syntax:** data\_type array\_name[row\_size][col\_size]

Example: int arr[3][3];

**Initialization of 2-D arrays:**

int marks[2][3]={90,85,74,88,66,86};

int marks[2][3]={{86,89,66},{56,75,78}};

0 1 2

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

0

1

Two rows and three columns marks[2][3].

If we omit values to an array default it is going to take a value 0.

**Example:**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};

int i,j;

cout<<" The elements of Matrx A are :"<<endl;

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

{

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

{

cout<<setw(3)<<a[i][j];

}

cout<<endl;

}

getch();

}

**Addition and subtraction of two matrices program:**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

int a[3][3]={{4,5,6},{7,8,9},{10,11,12}};

int b[3][3]={{1,2,3},{4,5,6},{7,8,9}};

int c[3][3],d[3][3],i,j;

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

{

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

{

c[i][j]=a[i][j]+b[i][j];

}

}

cout<< "Elements after the sum of two matrices is: \n"<<endl;

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

{

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

{

cout<<setw(3)<<c[i][j];

}

cout<<endl;

}

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

{

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

{

d[i][j]=a[i][j]-b[i][j];

}

}

cout<< "Elements after the subtraction of two matrices is: \n"<<endl;

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

{

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

{

cout<<setw(3)<<d[i][j];

}

cout<<endl;

}

getch();

}

**Transpose matrix:**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

int a[3][3]={1,2,3,4,5,6,7,8,9};

int i,j;

cout<<" Elements of Matrix A are: \n";

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

{

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

{

cout<<setw(3)<<a[i][j];

}

cout<<endl;

}

cout<<" After transpose the elements in matrix A are \n";

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

{

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

{

cout<<setw(3)<<a[j][i];

}

cout<<endl;

}

getch();

}

**MATRIX MULTIPLICATION**

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

main()

{

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

int b[2][2]={{5,6},{7,8}};

int c[2][2],i,j,k;

cout<< " Martix A \n ";

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

cout<<setw(3)<<a[i][j];

}

cout<<"\n";

}

cout<< " Martix B \n" ;

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

cout<<setw(3)<< b[i][j];

}

cout<<"\n";

}

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

c[i][j]=0;

for(k=0;k<2;k++)

{

c[i][j]=c[i][j]+a[i][k]\*b[k][j];

}

}

}

cout<< "\n The product of two matrices result is below \n" ;

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

cout<<setw(3)<<c[i][j];

}

cout<<"\n";

}

getch();

}

**STRINGS**

A string is a sequence of characters enclosed within the double quotes. A string may include letters, digits and various special characters such as +,-,\*,/ and %.

**Example:**

1. “cnu”

2. “ashwika”

3. “1+2=3”

A string is nothing but a null-terminated character array. This means that after the last character, a null character (‘\0’) is stored to signify the end of the character array.

**Example:**

char str[]=”hello”;

**Declaring and initializing string variables:**

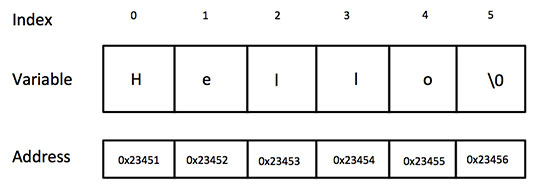
A string variable is declared as an array. The general form is:

**Syntax:** char string\_name[size];

**Example:** char name[10];

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the *null* character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array.

**Difference between integer and character array**

|  |  |
| --- | --- |
| **Integer Array** | **Character Array** |
| It is represented as array of integers | It is represented as array of characters |
| Each element in the integer array occupies 2 bytes | Each element in the character array occupies 1 byte |
| No null value inserted | Null value is inserted |
| Allows integer values | Allows characters, digits and special symbols |
| Syn: int arr[2]; | Syn: char str[24]; |
| Initialized as int[ ]={2,3,4}; | Initialized as char str[ ]={‘a’,’e’,’I’,’\o’};  char str[10]=”kadapa”; |

**SORTING PROGRAM:**

#include<iostream>

#include<conio.h>

#include<string.h>

using namespace std;

int main()

{

char str[10][20], temp[20];

int i,j,n;

cout<< " Enter how many strings to sort : ";

cin>>n;

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

cin>>str[i];

cout<<"\n before sorting \n" ;

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

cout<<str[i]<<endl;

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

{

for(j=i+1;j<n;j++)

{

if(strcmp(str[i],str[j])>0)

{

strcpy(temp,str[i]);

strcpy(str[i],str[j]);

strcpy(str[j],temp);

}

}

}

cout<<" After sorting \n";

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

cout<<str[i]<<endl;

getch();

}

**STRING HANDLING FUNCTIONS**

The string handling or manipulation functions are contained in the header file string.h and therefore the statement #include<string.h> must be included in the program.

|  |  |
| --- | --- |
| **FUNCTION** | **USE** |
| strcat | Appends one string at the end of another. |
| strcmp | Compares two strings |
| strcpy | Copies one string over another. |
| strlen | Finds the length of a string |
| strlwr | Converts a string to lowercase |
| strupr | Converts a string to uppercase |
| strdup | Duplicates a string |
| strrev | Reverse a string |
| strncat | Appends first n characters of a string at the end of another |
| strcmpi | Compares two strings without case sensitivity. |
| strchr | Find first occurrence of a given character in a string |
| strrchr | Find last occurrence of a given character in a string |
| strncmp | Compares first n characters of two string |

#include<iostream>

#include<conio.h>

#include<string.h>

using namespace std;

main()

{

char str1[20]="i love";

char str2[20]=" my country";

char str3[20];

cout<< "\n The length of string1 is= "<<strlen(str1);

cout<<"\n String concatenation= " <<strcat(str1,str2);

cout<<"\n String copy= "<<strcpy(str3,str1);

cout<<"\n String lower= "<<strlwr(str2);

cout<<"\n String uppper= "<<strupr(str2);

cout<<"\n String reverse= "<<strrev(str2);

cout<<"\n String comparison= "<<strcmp(str2,str3);

getch();

}

**PALINDROME:**

#include<iostream>

#include<conio.h>

#include<string.h>

using namespace std;

main()

{

char str[30];

int i,j,n,flag=1;

cout<<" \n Enter any string :";

gets(str);

n=strlen(str);

cout<<n;

for(i=0,j=n-1;i<n/2;i++,j--)

{

if(str[i]!=str[j])

{

flag=0;

break;

}

}

if(flag==1)

cout<<"\n The given string is palindrome ";

else

cout<<" \n The given string is not palindrome ";

getch();

}

**SORTING:**

#include<iostream>

#include<conio.h>

#include<string.h>

using namespace std;

main()

{

char str[10][20], temp[20];

int i,j,n;

cout<<"\n Enter how many strings to sort : ";

cin>>n;

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

cin >>str[i];

cout<<"\n before sorting \n";

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

cout<<str[i]<<"\t";

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

{

for(j=i+1;j<n;j++)

{

if(strcmp(str[i],str[j])>0)

{

strcpy(temp,str[i]);

strcpy(str[i],str[j]);

strcpy(str[j],temp);

}

}

}

cout<<"\n After sorting \n";

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

cout<<str[i]<<"\t";

getch();

}

**POINTERS**

A pointer is a variable which stores the memory address of another variable. Pointers are used to store the address of variables.

**Declaring a pointer:**

A pointer is a variable that holds the memory address of another variable as its value. The general form to declare a pointer variable is:

**Syntax:**

datatype \*ptr\_var;

**Ex:** int \*ptr;

char \*c;

Here ‘ptr’ is a pointer variable that contains address of a variable which is of integer type.

All the data types of pointer variables can hold 2 bytes of memory.

**Initializing a pointer:**

A pointer variable can be declared first and later assigned a value as shown below:

**Syntax:** pointer\_variable= &variable;

Ex: int a,\*p;

a=10;

p=&a;

**Example:**

#include <iostream>

#include <conio.h>

using namespace std;

main()

{

int n,\*p;

p=&n;

cout<<"Enter the number: ";

cin>>n;

cout<< "\n The number that was entered is " <<\*p;

cout<<"\n The address of number in memory is "<< &n;

getch();

}

**ADVANTAGES AND DISADVANTAGES**

**Advantages:**

1. Pointers reduce the length and complexity of the program.
2. Array elements can be easily accessed.
3. Execution of a program is faster.
4. Pointers are frequently used to pass the information in functions.
5. Arguments passed to function can be modified.
6. Memory can be efficiently used through dynamic memory allocation.

**Disadvantages:**

1. If sufficient memory is not available during runtime for the storage of pointers, the program may crash.
2. If the programmer is not careful and consistent with the use of pointers, the program may crash (very possible).
3. Uninitialized pointers might cause segmentation fault.

**PASSING ARGUMENTS TO FUNCTION USING POINTERS**

#include<iostream>

#include<conio.h>

using namespace std;

void swap(int \*a,int \*b)

{

int t;

t=\*a;

\*a=\*b;

\*b=t;

}

main()

{

int x,y;

x=10;

y=20;

cout<< "\n values before swapping";

cout<< " x ="<<x << "y= "<<y;

swap(&x,&y);

cout<< "\n values after swapping";

cout<< " x ="<<x << "y= "<<y;

getch();

}

**STRUCTURES**

A structure is a collection of data items of different data types referenced under a same name.

**Syntax: structure declaration:**

struct structure-name

{

datatype member1;

datatype member2;

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

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

};

A structure can be defined by using the keyword ‘struct’. Each individual item of a structure is called a member (member variable).

**Ex:**

Struct student

{

int sno;

char sname[10];

int s1,s2;

};

**Defining structure variable:**

A structure variable can be defined in the main() function, before or after.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 10 | | | | | | | | | | 2 | 2 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| s.sno | s.sname | | | | | | | | | | s.s1 | s.s2 |

**Syntax:** struct structure-name variable-list;

**Ex:** struct student s; (or)

struct student s1,s2,s3;

The structure variables are created with separate address location in the memory.

**Initializing of structure:**

Like simple and array variables, structure type variables can also be initialized while they are declaring. Initializing a structure means assigning some constants to the members of the structure.

**Syntax:**

struct structure-name var1 = { values }, var2 = {values},……., var-n = {values};

**Ex:**

1) struct student s={a,”raju”,77,88};

2) struct stud s1 = {1,”ABC”, 45,55 }, s2 = {2,”XYZ”,90,89};

**Accessing Structure Elements**:

Individual members of a structure are accessed through the use of dot (.) operator. This operator is placed between a structure variable and structure members.

**Syntax:** structure\_variable.member

**Ex:** printf(“\n sno=%d”,s.sno);

printf(“\n sname=%s”,s.sname);

**Program:**

/\* write a program to find sum and average of student marks using structures \*/

#include<iostream>

#include<conio.h>

using namespace std;

struct student

{

int sno,s1,s2,tot;

char sname[10];

float avg;

};

main()

{

struct student s;

cout<<"\n Enter sno,sname,s1,s2:\n";

cin>>s.sno>>s.sname>>s.s1>>s.s2;

s.tot=s.s1+s.s2;

s.avg=s.tot/2.0;

cout<<"\n total="<<s.tot;

cout<<"\n average="<<s.avg;

getch();

}

**NESTED STRUCTURES**

When a structure is an element of another structure, it is called a nested structure.

#include<iostream>

#include<conio.h>

using namespace std;

struct marks

{

int inter,exter;

};

struct student

{

int sno;

char sname[10];

struct marks m;

};

main()

{

struct student s;

int tot;

cout<<"\n Enter sno & sname:\n";

cin>>s.sno>>s.sname;

cout<<"\n Enter internal marks:";

cin>>s.m.inter;

cout<<"\n Enter external marks:";

cin>>s.m.exter;

tot=s.m.inter+s.m.exter;

cout<<"\n total="<<tot;

getch();

}

**CLASS -OBJECTS**

A class is a way to bind the data and its associated functions together. It allows the data to be hidden, if necessary from external use.

The general form of a class declaration is:

|  |
| --- |
| class class\_name  {  private:  variable declarations;  function declarations;  public:  variable declarations;  function declarations;  }; |

The class is terminated with a semicolon. The class body contains variables and functions. These function and variables are collectively called class members. These class members are grouped under two sections, namely **private** and **public.** The class members that are declared as private can be accessed only from within the class. The public members can be accessed from outside the class also. Default members of a class are private.

**Creating objects:**

An object is an instance of a class. When is class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

**Syntax:** classname objectname;

**Accessing class members:**

The data members and member functions of class can be accessed using the dot(‘.’) operator with the object.

**Syntax obj.datamember;**

**obj.functionname(actual arguments);**

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

class test

{

int a;

char str[20];

public:

void setdata()

{

cout<<" Enter the number :";

cin>>a;

cout << "\n Enter the name :";

cin>>str;

}

void showdata()

{

cout<<" Number is :"<<a<<"\n";

cout<<" Name is :"<<str<<"\n";

}

};

main()

{

test s;

s.setdata();

s.showdata();

getch();

}

**ACCESS SPECIFIERS**

C++ supports three types of access specifiers. They are

1. private
2. public
3. protected

**private:** The data members and member functions that are declared as **private** can be accessed only within the class and not from the outside the class.

**Public:** The data members and member function that are declared as **public** can be accessed within the class as well as from outside the class.

**Protected:** The data members and member functions declared as **protected** cannot be accessed outside the class, but can be accessed from a derived class. The protected keyword is frequently used in inheritance of class.

**DEFINING MEMBER FUNCTIONS**

The data member of a class must be declared within the body of the class, whereas the member functions of the class can be defined two ways as below:

1. Inside the class definition
2. Outside the class definition.

**Inside the class definition**

**Syntax:**

returntype memberfunction(arguments)

{

Body of the class.

}

**Outside the class definition**

**Syntax:**

returntype classname :: memberfunction(arguments)

{

Body of the class

}

|  |
| --- |
| **Example: inside the class definition**  **Example: outside the class definition**  #include<iostream>  #include<conio.h>  using namespace std;  class test  {  int a;  char str[20];  public:  void setdata();  void showdata();  };    void test :: setdata()  {  cout<<" Enter the number :";  cin>>a;  cout << "\n Enter the name :";  cin>>str;  }  void test :: showdata()  {  cout<<" Number is :"<<a<<"\n";  cout<<" Name is :"<<str<<"\n";  }  main()  {  test s;  s.setdata();  s.showdata();  getch();  }  #include<iostream>  #include<conio.h>  using namespace std;  class test  {  int a;  char str[20];  public:  void setdata()  {  cout<<" Enter the number :";  cin>>a;  cout << "\n Enter the name :";  cin>>str;  }  void showdata()  {  cout<<" Number is :"<<a<<"\n";  cout<<" Name is :"<<str<<"\n";  }  };    main()  {  test s;  s.setdata();  s.showdata();  getch();  } |

**MEMBER FUNCTION WITH OBJECT AS ARGUMENTS:**

Like any other data type, an object may be used as a function argument. This can be done in two ways.

1. Pass by value
2. Pass by reference

**Pass by value:** In pass by value, a copy of the object is passed to the function, any changes made to the object inside the function do not affect the actual object.

**Pass by reference:** In pass by reference mechanism, an address of the object is passed to the function and any modification made to the object inside function is reflected in the actual object.

|  |
| --- |
| **PASS BY VALUE**  #include <iostream>  **PASS BY REFERENCE**  #include<iostream>  using namespace std;  class Demo  {  private:  int a;  public:  void set(int x)  {  a = x;  }  void sum(Demo &ob1, Demo &ob2)  {  a = ob1.a + ob2.a;  }  void print()  {  cout<<"Value of A : "<<a<<endl;  }  };  int main()  {  Demo d1;  Demo d2;  Demo d3;  d1.set(10);  d2.set(20);  d3.sum(d1,d2);  d1.print();  d2.print();  d3.print();  }  **using** **namespace** std;  **class** Demo  {  **private**:  **int** a;  **public**:  **void** set(**int** x)  {  a = x;  }  **void** sum(Demo ob1, Demo ob2)  {  a = ob1.a + ob2.a;  }  **void** print()  {  cout<<"Value of A : "<<a<<endl;  }  };  **int** main()  {  Demo d1;  Demo d2;  Demo d3;  d1.set(10);  d2.set(20);  d3.sum(d1,d2);  d1.print();  d2.print();  d3.print();  } |

**RETURNING OBJECTS FROM FUNCTIONS**

A function cannot only receive objects as arguments but also can return them.

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

class test

{

int a,b;

public:

void get();

void disp();

test sum(test);

};

void test::get()

{

cout<<" Enter a and b values ";

cin>>a>>b;

}

void test::disp()

{

cout<<"\n a ="<<a;

cout<<"\n b= "<<b;

}

test test::sum(test t2)

{

test t3;

t3.a=a+t2.a;

t3.b=b+t2.b;

return t3;

}

main()

{

test t1,t2,t3;

t1.get();

t2.get() ;

t3=t1.sum(t2);

t1.disp();

t2.disp();

t3.disp();

}

**ARRAY OBJECTS**

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object. As we know, an array is a collection of similar type; therefore an array can be a collection of class type.

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

class student

{

int sno;

char sname[20];

float fees;

public:

void setdata()

{

cout<<"\n Enter the student number ";

cin>>sno;

cout<<"\n Enter the student name ";

cin>>sname;

cout<<"\n Enter the student fee ";

cin>>fees;

}

void showdata()

{

cout<<"\n Student number :"<<sno;

cout<<"\n Student Name :"<<sname;

cout<<"\n Student fees :"<<fees;

}

};

main()

{

int i;

student s[2];

for(i=0;i<2;i++)

{

cout<<"Enter student "<<i+1<<" details ";

s[i].setdata();

}

for(i=0;i<2;i++)

{

cout<<" \n student "<<i+1<<" details ";

s[i].showdata();

}

getch();

}

**FRIEND FUNCTION**

Private members are accessed only within the class they are declared. Friend function is used to access the private and protected members of different classes. It works as bridge between classes.

* Friend function must be declared with **friend** keyword.
* Friend function must be declared in all the classes from which we need to access private or protected members.
* Friend function will be defined outside the class without specifying the class name.
* Friend function will be invoked like normal function, without any object.
* Usually, it has the objects as arguments.

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

class sample

{

int a, b;

public:

void set()

{

a=12;

b=23;

}

friend int sum(sample s);

};

int sum(sample s)

{

return (s.a+s.b);

}

main()

{

sample x;

x.set();

cout<<" sum is "<<sum(x);

}

**FRIEND CLASS**

Like friend function, a class can also be a friend of another class. A friend class can access all the private and protected members of other class.

In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

class example

{

private:

int a, b;

public:

void set(int x, int y)

{

a=x;

b=y;

}

friend class test;

};

class test

{

private:

float m;

public:

void meanval(example e)

{

m=(e.a+e.b)/2.0;

}

void showmean()

{

cout<<"mean value ="<<m;

}

};

main()

{

example e;

test t;

e.set(2,3);

t.meanval(e);

t.showmean();

getch();

}

**CONSTRUCTORS AND DESTRUCTORS**

A constructor is a special member function, whose name is same as class name. The constructor is invoked whenever an object of the associated class is created.

**Syntax:**

class <classname>

{

private members:

public:

classname();

};

classname :: classname()

{

constructor body definition

}

**Characteristics:**

1. It has the same name as that of the class to which it belongs.
2. It is executed automatically whenever the class is instantiated.
3. It does not have any return type, not even void also.
4. We can default, parameterized, copy constructors.
5. It is normally used to initialize the data members of a class.
6. It can be declared in public section only.

**Destructor:**

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor.

It is also a member function, whose name is same as the class name but is preceded by tilde (~) symbol.

**Syntax:**

class <classname>

{

private members:

public:

~classname();

};

classname :: ~classname()

{

destructor body definition

}

**Characteristics:**

1. It has the same name as that of the class to which it belongs and preceded by tilde(~).
2. It is executed automatically upon exit from the program to clean up storage that is no longer accessible.
3. It does not have any return type and not even void also.
4. Only one destructor can be defined in the class.
5. The destructor does not have any argument.
6. It is a good practice to declare destructors in a program since it releases memory space for future use.

Different types of constructors are

1. Default constructor
2. Parameterized constructor
3. Copy constructor
4. Dynamic constructor

**Default Constructor:** A constructor that accepts no parameters is called the default constructor.

**Example:**

**Paramerized constructor example**

#include<iostream>

#include<conio.h>

using namespace std;

class integer

{

private:

int n,m;

public:

integer(int x, int y);

void showdata()

{

cout<<"m= "<<m<<"\n";

cout<<"n= "<<n<<"\n";

}

};

integer::integer(int a, int b)

{

m=a;

n=b;

}

main()

{

integer i(4,5);

i.showdata();

}

|  |
| --- |
| #include<iostream>  #include<conio.h>  using namespace std;  class integer  {  private:  int n,m;  public:  integer();  void showdata()  {  cout<<"m= "<<m<<"\n";  cout<<"n= "<<n<<"\n";  }  };  integer::integer()  {  m=0;  n=0;  }  main()  {  integer i;  i.showdata();  } |

**Parameterized constructor:**

A constructor with arguments called parameterized constructor.

**Copy constructor:**

Initialization of an object through another object is called **copy constructor**. In other words, copying the values of one object into another object is called **copy constructor.**

**Dynamic Constructor:**

Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

**Dynamic constructor:**

#include<iostream>

#include<conio.h>

using namespace std;

class dynamic

{

private:

int \*ptr;

public:

dynamic()

{

ptr=new int;

\*ptr=100;

}

dynamic(int v)

{

ptr=new int;

\*ptr=v;

}

int showdata()

{

return \*ptr;

}

};

main()

{

dynamic d1;

dynamic d2(10);

cout<<" the value of d1 object = ";

cout<<d1.showdata();

cout<<"\n the value of d2 object =";

cout<<d2.showdata();

getch();

}

**Copy constructor:**

#include<iostream>

#include<conio.h>

using namespace std;

class integer

{

private:

int n,m;

public:

integer(int x, int y);

integer (integer &i);

void showdata()

{

cout<<"m= "<<m<<"\n";

cout<<"n= "<<n<<"\n";

}

};

integer::integer(int a, int b)

{

m=a;

n=b;

}

integer::integer(integer &i)

{

m=i.m;

n=i.n;

}

main()

{

integer i(4,5);

integer i1(i);

i.showdata();

i1.showdata();

}

**Constructor overloading:**

In c++ a class can have multiple constructors. All the constructors have the same name as the corresponding class, and they are different in argument list.

**Example:**

#include<iostream>

**Example: Destructor**

#include<iostream>

#include<conio.h>

using namespace std;

class test

{

public:

test()

{

cout<<" Constructor ";

}

~test()

{

cout<<" \n Destructor";

}

};

main()

{

test t;

}

#include<conio.h>

using namespace std;

class integer

{

int m,n;

public:

integer();

integer(int x);

integer(int x, int y);

integer(integer &i);

void showdata()

{

cout<<"m ="<<m<<endl;

cout<<"n ="<<n<<endl;

}

};

integer::integer()

{

m=n=0;

}

integer::integer(int x)

{

m=n=x;

}

integer::integer(int x, int y)

{

m=x;

n=y;

}

integer::integer(integer &i )

**Output:**

m =0

n =0

m =10

n =10

m =20

n =30

m =20

n =30

{

m=i.m;

n=i.n;

}

main()

{

integer i1;

integer i2(10);

integer i3(20,30);

integer i4(i3);

i1.showdata();

i2.showdata();

i3.showdata();

i4.showdata();

}

**Operator overloading:**

C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

**Syntax:**

returntype classname :: operator op(arglist)

{

//body of the function

}

**Example:**

#include<iostream>

#include<conio.h>

using namespace std;

class space

{

int x,y,z;

public:

void setdata(int a, int b, int c);

void showdata();

void operator -();

};

void space::setdata(int a, int b, int c)

{

x=a;

y=b;

z=c;

}

void space::showdata()

{

cout<<" X= "<<x<<"\n";

cout<<" Y= "<<y<<"\n";

cout<<" Z= "<<z<<"\n";

}

void space::operator -()

{

x= -x;

y= -y;

z= -z;

}

main()

{

space s;

s.setdata(10,-30,45);

s.showdata();

-s;

s.showdata();

}

A class can be derived from more than one class, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

**Single inheritance:**

Deriving a single class from a base class is called single inheritance.

**Example:**

#include <iostream>

using namespace std;

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

cout << "Total area: " << Rect.getArea() << endl;

}

**Multiple inheritance:** Deriving a class from multiple base classes.

**Example:**

#include <iostream>

using namespace std;

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

class PaintCost {

public:

int getCost(int area) {

return area \* 70;

}

};

class Rectangle: public Shape, public PaintCost {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

cout << "Total area: " << Rect.getArea() << endl;

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

}