<https://sites.google.com/dhyanahita.org/fdp-gitam-vskp-blr-2020/>

<https://docs.google.com/forms/d/e/1FAIpQLSfZxaO_cxnTHZyIrEQjM-dwvI1dD9-1bBKEO03qo-F1lJbWjw/viewform>

[**https://tinyurl.com/ycf863ju(short**](https://tinyurl.com/ycf863ju(short) **url for my site )**

[journal@sersc.org.(IJAST)](mailto:journal@sersc.org.(IJAST))

**editor@warse.org(IJATCSE)**

**C++ Data Types**

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

There are 4 types of data types in C++ language.

|  |  |
| --- | --- |
| **Types** | **Data Types** |
| Basic Data Type | int, char, float, double, etc |
| Derived Data Type | array, pointer, etc |
| Enumeration Data Type | enum |
| User Defined Data Type | Structure,union,class |

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

Let's see the basic data types. It size is given according to 32 bit OS.

|  |  |  |
| --- | --- | --- |
| **Data Types** | **Memory Size** | **Range** |
| char | 1 byte | -128 to 127 |
| signed char | 1 byte | -128 to 127 |
| unsigned char | 1 byte | 0 to 127 |
| short | 2 byte | -32,768 to 32,767 |
| signed short | 2 byte | -32,768 to 32,767 |
| unsigned short | 2 byte | 0 to 32,767 |
| int | 2 byte | -32,768 to 32,767 |
| signed int | 2 byte | -32,768 to 32,767 |
| unsigned int | 2 byte | 0 to 32,767 |
| short int | 2 byte | -32,768 to 32,767 |
| signed short int | 2 byte | -32,768 to 32,767 |
| unsigned short int | 2 byte | 0 to 32,767 |
| long int | 4 byte |  |
| signed long int | 4 byte |  |
| unsigned long int | 4 byte |  |
| float | 4 byte |  |
| double | 8 byte |  |
| long double | 10 byte |  |

**Enumeration Example**

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

1. #include <iostream>
2. **using** **namespace** std;
3. **enum** week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
4. **int** main()
5. {
6. week day;
7. day = Friday;
8. cout << "Day: " << day+1<<endl;
9. **return** 0;
10. }

Output:

Day: 4

# C++ Storage Classes

Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.

Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

There are four types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Storage Class** | **Keyword** | **Lifetime** | **Visibility** | **Initial Value** |
| Automatic | auto | Function Block | Local | Garbage |
| Register | register | Function Block | Local | Garbage |
| External | extern | Whole Program | Global | Zero |
| Static | static | Whole Program | Local | Zero |

**Automatic Storage Class**

It is the default storage class for all local variables. The auto keyword is applied to all local variables automatically.

1. {
2. auto **int** y;
3. **float** y = 3.45;
4. }

The above example defines two variables with a same storage class, auto can only be used within functions.

**Register Storage Class**

The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables.

It is recommended to use register variable only for quick access such as in counter.

Note: We can't get the address of register variable.

1. **register** **int** counter=0;

**Static Storage Class**

The static variable is initialized only once and exists till the end of a program. It retains its value between multiple functions call.

The static variable has the default value 0 which is provided by compiler.

1. #include <iostream>
2. **using** **namespace** std;
3. **void** func() {
4. **static** **int** i=0; //static variable
5. **int** j=0; //local variable
6. i++;
7. j++;
8. cout<<"i=" << i<<" and j=" <<j<<endl;
9. }
10. **int** main()
11. {
12. func();
13. func();
14. func();
15. }

Output:

i= 1 and j= 1

i= 2 and j= 1

i= 3 and j= 1

**External Storage Class**

The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

**extern** **int** counter=0;

Example:

|  |
| --- |
| #include <iostream>  using namespace std;    // declaring the variable which is to  // be made extern an intial value can  // also be initialized to x  int x;  void externStorageClass()  {        cout << "Demonstrating extern class\n";        // telling the compiler that the variable      // x is an extern variable and has been      // defined elsewhere (above the main      // function)      extern int x;        // printing the extern variables 'x'      cout << "Value of the variable 'x'"           << "declared, as extern: " << x << "\n";        // value of extern variable x modified      x = 2;        // printing the modified values of      // extern variables 'x'      cout          << "Modified value of the variable 'x'"          << " declared as extern: \n"          << x;  }    int main()  {        // To demonstrate extern Storage Class      externStorageClass();        return 0;  } |

1. **Output:**
2. Demonstrating extern class
3. Value of the variable 'x'declared, as extern: 0
4. Modified value of the variable 'x' declared as extern:
5. 2

**C++ References**

Till now, we have read that C++ supports two types of variables:

* An ordinary variable is a variable that contains the value of some type. For example, we create a variable of type int, which means that the variable can hold the value of type integer.
* A pointer is a variable that stores the address of another variable. It can be dereferenced to retrieve the value to which this pointer points to.
* There is another variable that C++ supports, i.e., references. It is a variable that behaves as an alias for another variable.

How to create a reference?

Reference can be created by simply using an ampersand (&) operator. When we create a variable, then it occupies some memory location. We can create a reference of the variable; therefore, we can access the original variable by using either name of the variable or reference. For example,

1. **int** a=10;

Now, we create the reference variable of the above variable.

The above statement means that 'ref' is a reference variable of 'a', i.e., we can use the 'ref' variable in place of 'a' variable.

int &ref=a;

C++ provides two types of references:

* References to non-const values
* References as aliases

**References to non-const values**

It can be declared by using & operator with the reference type variable.

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=10;
6. **int** &value=a;
7. cout << value << endl;
8. **return** 0;
9. }

**Output**

10

**References as aliases**

References as aliases is another name of the variable which is being referenced.

**For example,**

1. **int** a=10;   // 'a' is a variable.
2. **int** &b=a; // 'b' reference to a.
3. **int** &c=a; // 'c' reference to a.

**Let's look at a simple example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=70; // variable initialization
6. **int** &b=a;
7. **int** &c=a;
8. cout << "Value of a is :" <<a<< endl;
9. cout << "Value of b is :" <<b<< endl;
10. cout << "Value of c is :" <<c<< endl;
11. **return** 0;}

In the above code, we create a variable 'a' which contains a value '70'. We have declared two reference variables, i.e., b and c, and both are referring to the same variable 'a'. Therefore, we can say that 'a' variable can be accessed by 'b' and 'c' variable.

**Output**

Value of a is :70

Value of b is :70

Value of c is :70

Properties of References

**The following are the properties of references:**

**Initializátion**

It must be initialized at the time of the declaration.

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=10; // variable initialization
6. **int** &b=a; // b reference to a
7. std::cout << "value of a is " <<b<< std::endl;
8. **return** 0;
9. }

**In the above code, we have created a reference variable, i.e., 'b'. At the time of declaration, 'a' variable is assigned to 'b'. If we do not assign at the time of declaration, then the code would look like:**

1. **int** &b;
2. &b=a;

The above code will throw a compile-time error as 'a' is not assigned at the time of declaration.

**Output**

value of a is 10

**Reassignment**

It cannot be reassigned means that the reference variable cannot be modified.

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** x=11; // variable initialization
6. **int** z=67;
7. **int** &y=x; // y reference to x
8. **int** &y=z; // y reference to z, but throws a compile-time error.
9. **return** 0;}

In the above code, 'y' reference variable is referring to 'x' variable, and then 'z' is assigned to 'y'. But this reassignment is not possible with the reference variable, so it throws a compile-time error.

**Compile-time error**

1. main.cpp: In function 'int main()':
2. main.cpp:18:9: error: redeclaration of 'int& y'
3. **int** &y=z; // y reference to z, but throws a compile-time error.
4. ^
5. main.cpp:17:9: note: 'int& y' previously declared here
6. **int** &y=x; // y reference to x
7. ^

**Function Parameters**

References can also be passed as a function parameter. It does not create a copy of the argument and behaves as an alias for a parameter. It enhances the performance as it does not create a copy of the argument.

**Let's understand through a simple example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=9; // variable initialization
6. **int** b=10; // variable initialization
7. swap(a, b); // function calling
8. std::cout << "value of a is :" <<a<< std::endl;
9. std::cout << "value of b is :" <<b<< std::endl;
10. **return** 0;
11. }
12. **void** swap(**int** &p, **int** &q) // function definition
13. {
14. **int** temp; // variable declaration
15. temp=p;
16. p=q;
17. q=temp;
18. }

In the above code, we are swapping the values of 'a' and 'b'. We have passed the variables 'a' and 'b' to the swap() function. In swap() function, 'p' is referring to 'a' and 'q' is referring to 'b'. When we swap the values of 'p' and 'q' means that the values of 'a' and 'b' are also swapped.

**Output**

value of a is :10

value of b is :9