* **Namespace in C++**
* Most naming collisions occur in two cases:  
  1) Two (or more) definitions for a function (or global variable) are introduced into separate files that are compiled into the same program. This will result in a linker error, as shown above.  
  2) Two (or more) definitions for a function (or global variable) are introduced into the same file (often via an #include). This will result in a compiler error.
* As programs get larger and use more identifiers, the odds of a naming collision being introduced increases significantly. The good news is that C++ provides plenty of mechanisms for avoiding naming collisions. Local scope, which keeps local variables defined inside functions from conflicting with each other, is one such mechanism. But local scope doesn’t work for functions names.
* A **namespace** is a region that allows you to declare names inside of it for the purpose of disambiguation. The namespace provides a scope (called **namespace scope**) to the names declared inside of it -- which simply means that any name declared inside the namespace won’t be mistaken for identical names in other scopes.
* Within a namespace, all names must be unique, otherwise a naming collision will result.
* Namespaces are often used to group related identifiers in a large project to help ensure they don’t inadvertently collide with other identifiers. For example, if you put all your math functions in a namespace called math, then your math functions won’t collide with identically named functions outside the math namespace.
* **The global namespace**

In C++, any name that is not defined inside a class, function, or a namespace is considered to be part of an implicitly defined namespace called the **global namespace** (sometimes also called **the global scope**).

* **The std namespace**
* When C++ was originally designed, all of the identifiers in the C++ standard library (including std::cin and std::cout) were available to be used without the std:: prefix (they were part of the global namespace). However, this meant that any identifier in the standard library could potentially conflict with any name you picked for your own identifiers (also defined in the global namespace). Code that was working might suddenly have a naming conflict when you #included a new file from the standard library. Or worse, programs that would compile under one version of C++ might not compile under a future version of C++, as new identifiers introduced into the standard library could have a naming conflict with already written code. So C++ moved all of the functionality in the standard library into a namespace named “std” (short for standard).
* It turns out that std::cout‘s name isn’t really std::cout. It’s actually just cout, and std is the name of the namespace that identifier cout is part of. Because cout is defined in the std namespace, the name cout won’t conflict with any objects or functions named cout that we create in the global namespace.
* Similarly, when accessing an identifier that is defined in a namespace (e.g. std::cout) , you need to tell the compiler that we’re looking for an identifier defined inside the namespace (std).
* The most straightforward way to tell the compiler that we want to use *cout* from the *std* namespace is by explicitly using the *std::* prefix. For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include <iostream>    int main()  {      std::cout << "Hello world!"; // when we say cout, we mean the cout defined in the std namespace      return 0;  } |

* The :: symbol is an operator called the **scope resolution operator**. The identifier to the left of the :: symbol identifies the namespace that the name to the right of the :: symbol is contained within. If no identifier to the left of the :: symbol is provided, the global namespace is assumed.

So when we say *std::cout*, we’re saying “the *cout* that lives in namespace *std*“.

This is the safest way to use *cout*, because there’s no ambiguity about which *cout* we’re referencing (the one in the *std* namespace).

**Using namespace std (and why to avoid it)**

Another way to access identifiers inside a namespace is to use a *using directive* statement. Here’s our original “Hello world” program with a *using directive*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>    using namespace std; // this is a using directive telling the compiler to check the std namespace when resolving identifiers with no prefix    int main()  {      cout << "Hello world!"; // cout has no prefix, so the compiler will check to see if cout is defined locally or in namespace std      return 0;  } |

A **using directive** tells the compiler to check a specified namespace when trying to resolve an identifier that has no namespace prefix. So in the above example, when the compiler goes to determine what identifier *cout* is, it will check both locally (where it is undefined) and in the *std* namespace (where it will match to *std::cout*).

Many texts, tutorials, and even some compilers recommend or use a *using directive* at the top of the program. However, used in this way, this is a bad practice, and highly discouraged.

Consider the following program:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <iostream> // imports the declaration of std::cout    using namespace std; // makes std::cout accessible as "cout"    int cout() // declares our own "cout" function  {      return 5;  }    int main()  {      cout << "Hello, world!"; // Compile error!  Which cout do we want here?  The one in the std namespace or the one we defined above?        return 0;  } |

The above program doesn’t compile, because the compiler now can’t tell whether we want the *cout* function that we defined, or the *cout* that is defined inside the *std* namespace.

When using a using directive in this manner, *any* identifier we define may conflict with *any* identically named identifier in the *std* namespace. Even worse, while an identifier name may not conflict today, it may conflict with new identifiers added to the std namespace in future language revisions. This was the whole point of moving all of the identifiers in the standard library into the *std* namespace in the first place!

**Uninitialized variable in C++:**

“One of the things that has kept C++ viable is the zero-overhead rule: What you don’t use, you don’t pay for.” -Stroustrup. The overhead of initializing a stack variable is costly as it hampers the speed of execution, therefore these variables can contain indeterminate values. It is considered a best practice to initialize a primitive data type variable before using it in code.

**Access Modifiers in C++:**

Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding. Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

**Header files available in C++ for Input/Output operations are:**

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.

Compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.

1. **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
2. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

* extraction operator(**>>**) and insertion operator(<<)

1. #include <iostream>

using namespace std;

int main()

{

    char sample[] = "GeeksforGeeks";

    cout << sample << " - A computer science portal for geeks";

    return 0;

}

Output:

GeeksforGeeks - A computer science portal for geeks

2. #include <iostream>

using namespace std;

int main()

{

    int age;

    cout << "Enter your age:";

    cin >> age;

    cout << "\nYour age is: " << age;

    return 0;

}

* **Why C/C++ compiler does not initialize variables with default values?**

“One of the things that has kept C++ viable is the zero-overhead rule: What you don’t use, you don’t pay for.” -Stroustrup.

The overhead of initializing a stack variable is costly as it hampers the speed of execution, therefore these variables can contain indeterminate values. It is considered a best practice to initialize a primitive data type variable before using it in code.

Default values:

Double / float : Garbage

Int/ long : 0

Char: Null

* **Instance Variables**:

Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Initilisation of Instance Variable is not Mandatory.

Instance Variable can be accessed only by creating objects.

Sample Program:

filter\_none

edit

play\_arrow

brightness\_4

// C++ program to demonstrate Local variables

#include <iostream>

using namespace std;

class Marks {

public:

// This is a class variable

static int studentNumber;

// These variables are instance variables.

// These variables are in a class

// and are not inside any function

int engMarks;

int mathsMarks;

int phyMarks;

public:

Marks()

{

// Modify the class variable

++studentNumber;

};

};

// Setting the class variable of Marks

int Marks::studentNumber = 0;

// Driver code

int main()

{

// first object

Marks obj1;

obj1.engMarks = 50;

obj1.mathsMarks = 80;

obj1.phyMarks = 90;

// second object

Marks obj2;

obj2.engMarks = 80;

obj2.mathsMarks = 60;

obj2.phyMarks = 85;

// displaying marks for first object

cout << "Marks for first object:\n";

cout << Marks::studentNumber << endl;

cout << obj1.engMarks << endl;

cout << obj1.mathsMarks << endl;

cout << obj1.phyMarks << endl;

// displaying marks for second object

cout << "Marks for second object:\n";

cout << Marks::studentNumber << endl;

cout << obj2.engMarks << endl;

cout << obj2.mathsMarks << endl;

cout << obj2.phyMarks << endl;

}

OUTPUT:

Marks for first object:

2

50

80

90

Marks for second object:

2

80

60

85

As you can see in the above program the variables, engMarks , mathsMarks , phyMarksare instance variables. In case we have multiple objects as in the above program, each object will have its own copies of instance variables. It is clear from the above output that each object will have its own copy of instance variable.

**Static Variables:**

Static variables are also known as Class variables.

These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.

Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.

Static variables are created at the start of program execution and destroyed automatically when execution ends.

Initialization of Static Variable is not Mandatory. Its default value is 0.

If we access the static variable like Instance variable (through an object), the compiler will show the warning message and it won’t halt the program. The compiler will replace the object name to class name automatically.

If we access the static variable without the class name, Compiler will automatically append the class name.

To access static variables, we need not create an object of that class, we can simply access the variable as

class\_name::variable\_name;

|  |
| --- |
| // C++ program to demonstrate Static variables    #include <iostream>  using namespace std;    class Marks {    public:      // This is a class variable      static int studentNumber;        // These variables are instance variables.      // These variables are in a class      // and are not inside any function      int engMarks;      int mathsMarks;      int phyMarks;        Marks()      {            // Modify the class variable          ++studentNumber;      };  };    // Setting the class variable of Marks  int Marks::studentNumber = 0;    // Driver code  int main()  {        // object of Marks      Marks obj1;      obj1.engMarks = 50;      obj1.mathsMarks = 80;      obj1.phyMarks = 90;        // displaying marks for first object      cout << "Marks for object:\n";        // Now to display the static variable,      // it can be directly done      // using the class name      cout << Marks::studentNumber << endl;        // But same is not the case      // with instance variables      cout << obj1.engMarks << endl;      cout << obj1.mathsMarks << endl;      cout << obj1.phyMarks << endl;  } |

Output:

1

50

80

90

* **Instance variable Vs Static variable**

Each object will have its own copy of instance variable whereas We can only have one copy of a static variable per class irrespective of how many objects we create.

Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of instance variable. In case of static, changes will be reflected in other objects as static variables are common to all object of a class.

We can access instance variables through object references and Static Variables can be accessed directly using class name.

Syntax for static and instance variables:

class Example

{

static int a; // static variable

int b; // instance variable

}