File Handling using File Streams in C++

File represents storage medium for storing data or information. Streams refer to sequence of bytes. In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data. So we use the term File Streams/File handling. We use the header file `<fstream>`

- **ofstream:** It represents output Stream and this is used for writing in files.
- **ifstream:** It represents input Stream and this is used for reading from files.
- **fstream:** It represents both output Stream and input Stream. So it can read from files and write to files.

Operations in File Handling:

- Creating a file: `open()`
- Reading data: `read()`
- Writing new data: `write()`
- Closing a file: `close()`

Creating/Opening a File

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating. Syntax for file creation: `FilePointer.open("Path",ios::mode);`

- Example of file opened for writing: `st.open("E:\studytonight.txt",ios::out);`
- Example of file opened for reading: `st.open("E:\studytonight.txt",ios::in);`
- Example of file opened for appending: `st.open("E:\studytonight.txt",ios::app);`
- Example of file opened for truncating: `st.open("E:\studytonight.txt",ios::trunc);`

```
#include<iostream>

#include<conio>

#include <fstream>



using namespace std;



int main()

{

    fstream st; // Step 1: Creating object of fstream class

    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new
file
```

```
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created";
        st.close(); // Step 4: Closing file
    }
    getch();
    return 0;
}
```

Copy

Writing to a File

```
#include <iostream>
#include<conio>
#include <fstream>


using namespace std;


int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new
file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
```

```
    {

        cout<<"New file created";

        st<<"Hello";     // Step 4: Writing to file

        st.close(); // Step 5: Closing file

    }

    getch();

    return 0;

}
```

Copy

Here we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after **"FilePointer <<"** will be passed to output file.

Reading from a File

```
#include <iostream>

#include<conio>

#include <fstream>



using namespace std;



int main()

{

    fstream st; // step 1: Creating object of fstream class

    st.open("E:\studytonight.txt",ios::in);   // Step 2: Creating new
file

    if(!st) // Step 3: Checking whether file exist

    {

        cout<<"No such file";

    }

    else

    {

        char ch;
```

```
        while (!st.eof())

        {

            st >>ch;  // Step 4: Reading from file

            cout << ch;   // Message Read from file

        }

        st.close(); // Step 5: Closing file

    }

    getch();

    return 0;

}
```

Copy
Here we are reading input from a file. So, we use ios::in. As given in the program, information from the output file is obtained with the help of following syntax **"FilePointer >>variable"**.

Close a File

It is done by FilePointer.close().

```
#include <iostream>

#include<conio>

#include <fstream>



using namespace std;



int main()

{

    fstream st; // Step 1: Creating object of fstream class

    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new
file

    st.close(); // Step 4: Closing file

    getch();

    return 0;

}
```

Copy

Special operations in a File

There are few important functions to be used with file streams like:

- tellp() - It tells the current position of the put pointer.

  Syntax: filepointer.tellp()

- tellg() - It tells the current position of the get pointer.

  Syntax: filepointer.tellg()

- seekp() - It moves the put pointer to mentioned location.

  Syntax: filepointer.seekp(no of bytes,reference mode)

- seekg() - It moves get pointer(input) to a specified location.

  Syntax: filepointer.seekg((no of bytes,reference point)

- put() - It writes a single character to file.
- get() - It reads a single character from file.

Note: *For seekp and seekg three reference points are passed:* ***ios::beg*** *- beginning of the file* ***ios::cur*** *- current position in the file* ***ios::end*** *- end of the file*

Below is a program to show importance of tellp, tellg, seekp and seekg:

```
#include <iostream>

#include<conio>

#include <fstream>



using namespace std;



int main()

{

    fstream st; // Creating object of fstream class

    st.open("E:\studytonight.txt",ios::out);  // Creating new file

    if(!st) // Checking whether file exist

    {

        cout<<"File creation failed";

    }

    else
```

```
    {
        cout<<"New file created"<<endl;
        st<<"Hello Friends"; //Writing to file

        // Checking the file pointer position
        cout<<"File Pointer Position is "<<st.tellp()<<endl;

        st.seekp(-1, ios::cur); // Go one position back from current position

        //Checking the file pointer position
        cout<<"As per tellp File Pointer Position is "<<st.tellp()<<endl;

        st.close(); // closing file
    }
    st.open("E:\studytonight.txt",ios::in);   // Opening file in read mode
    if(!st) //Checking whether file exist
    {
        cout<<"No such file";
    }
    else
    {
        char ch;
        st.seekg(5, ios::beg);  // Go to position 5 from begning.
        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer position
        cout<<endl;
        st.seekg(1, ios::cur); //Go to position 1 from beginning.
        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer position
        st.close(); //Closing file
```

```
    }

    getch();

    return 0;

}
```

Copy

```
New file created

File Pointer Position is 13

As per tellp File Pointer Position is 12

As per tellg File Pointer Position is 5

As per tellg File Pointer Position is 6
```

Exception Handling in C++

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

Run Time Errors - They are also known as exceptions. An exception caught during run time creates serious issues.

Errors hinder normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling technics in our code.

In C++, Error handling is done using three keywords:

- try
- catch
- throw

Syntax:

```
try

{

    //code

    throw parameter;

}

catch(exceptionname ex)
```

```
{

    //code to handle exception

}
```

Copy

---

try block

The code which can throw any exception is kept inside(or enclosed in) a`try` block. Then, when the code will lead to any error, that error/exception will get caught inside the `catch` block.

catch block

`catch` block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur. For example, we can display descriptive messages to explain why any particular excpetion occured.

throw statement

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A `throw` expression accepts one parameter and that parameter is passed to handler.

`throw` statement is used when we explicitly want an exception to occur, then we can use `throw` statement to throw or generate that exception.

Understanding Need of Exception Handling

Let's take a simple example to understand the usage of try, catch and throw.

Below program compiles successfully but the program fails at runtime, leading to an exception.

```
#include <iostream>#include<conio.h>

using namespace std;

int main()

{

    int a=10,b=0,c;

    c=a/b;

    return 0;

}
```

Copy

The above program will not run, and will show **runtime error** on screen, because we are trying to divide a number with **0**, which is not possible.

How to handle this situation? We can handle such situations using exception handling and can inform the user that you cannot divide a number by zero, by displaying a message.

Using try, catch and throw Statement

Now we will update the above program and include exception handling in it.

```
#include <iostream>

#include<conio.h>

using namespace std;

int main()

{

    int a=10, b=0, c;

    // try block activates exception handling

    try

    {

        if(b == 0)

        {

            // throw custom exception

            throw "Division by zero not possible";

            c = a/b;

        }

    }

    catch(char* ex) // catches exception

    {

        cout<<ex;

    }

    return 0;

}
```

Copy

```
Division by zero not possible
```

In the code above, we are checking the divisor, if it is zero, we are throwing an exception message, then the catch block catches that exception and prints the message.

Doing so, the user will never know that our program failed at runtime, he/she will only see the message "Division by zero not possible".

This is **gracefully handling** the exception condition which is why exception handling is used.

Using Multiple `catch` blocks

Below program contains multiple `catch` blocks to handle different types of exception in different way.

```
#include <iostream>

#include<conio.h>

using namespace std;


int main()

{

    int x[3] = {-1,2};

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

    {

        int ex = x[i];

        try

        {

            if (ex > 0)

                // throwing numeric value as exception

                throw ex;

            else

                // throwing a character as exception

                throw 'ex';

        }

        catch (int ex)  // to catch numeric exceptions

        {

            cout << "Integer exception\n";

        }

        catch (char ex) // to catch character/string exceptions
```

```
            {
                cout << "Character exception\n";
            }
        }
}
```

Copy

```
Integer exception
Character exception
```

The above program is self-explanatory, if the value of integer in the array `x` is less than 0, we are throwing a numeric value as exception and if the value is greater than 0, then we are throwing a character value as exception. And we have two different `catch` blocks to catch those exceptions.

Generalized `catch` block in C++

Below program contains a generalized `catch` block to catch any uncaught errors/exceptions. `catch(...)` block takes care of all type of exceptions.

```
#include <iostream>
#include<conio.h>
using namespace std;


int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex=x[i];
        try
        {
            if (ex > 0)
                throw ex;
            else
                throw 'ex';
        }
```

```
        // generalised catch block
        catch (...)
        {
            cout << "Special exception\n";
        }
    }
return 0;
}
```

Copy

```
Special exception
Special exception
```

In the case above, both the exceptions are being catched by a single `catch` block. We can even have separate `catch` blocks to handle integer and character exception along with th generalised `catch` block.

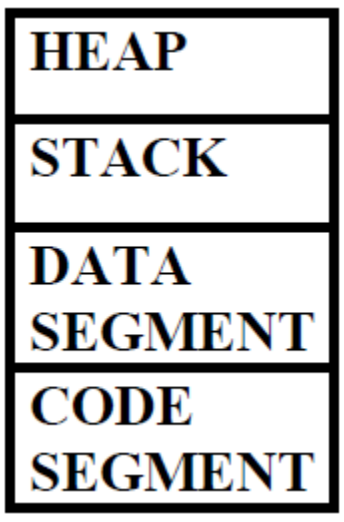
Standard Exceptions in C++

There are some standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:

- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic_error** - Exception happens in the internal logical of a program.
    - **domain_error** - Exception due to use of invalid domain.
    - **invalid argument** - Exception due to invalid argument.
    - **out_of_range** - Exception due to out of range i.e. size requirement exceeds allocation.
    - **length_error** - Exception due to length error.
- **runtime_error** - Exception happens during runtime.
    - **range_error** - Exception due to range errors in internal computations.
    - **overflow_error** - Exception due to arithmetic overflow errors.
    - **underflow_error** - Exception due to arithmetic underflow errors
- **bad_alloc** - Exception happens when memory allocation with new() fails.
- **bad_cast** - Exception happens when dynamic cast fails.
- **bad_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
- **bad_typeid** - Exception thrown by typeid.

Dynamic Memory Allocation in C++

Below is a basic memory architecture used by any C++ program:



- **Code Segment**: Compiled program with executive instructions are kept in code segment. It is read only. In order to avoid over writing of stack and heap, code segment is kept below stack and heap.
- **Data Segment**: Global variables and static variables are kept in data segment. It is not read only.
- **Stack**: A stack is usually pre-allocated memory. The stack is a LIFO data structure. Each new variable is pushed onto the stack. Once variable goes out of scope, memory is freed. Once a stack variable is freed, that region of memory becomes available for other variables. The stack grows and shrinks as functions push and pop local variables. It stores local data, return addresses, arguments passed to functions and current status of memory.
- **Heap**: Memory is allocated during program execution. Memory is allocated using new operator and deallocating memory using delete operator.

Allocation of Heap Memory using `new` Keyword

Here we will learn how to allocate heap memory to a variable or class object using the `new` keyword.

Syntax:

```
datatype pointername = new datatype
```

Copy
For example:

```
int *new_op = new int;

// allocating block of memory

int *new_op = new int[10];
```

Copy
If enough memory is not available in the **heap** it is indicated by throwing an exception of type `std::bad_alloc` and a pointer is returned.

Deallocation of memory using `delete` Keyword

Once heap memory is allocated to a variable or class object using the `new` keyword, we can deallocate that memory space using the `delete` keyword.

Syntax:

```
delete pointer variable
```

Copy
For example:

```
delete new_op;
```

Copy
The object's extent or the object's lifetime is the time for which the object remains in the memory during the program execution. Heap Memory allocation is slower than a **stack**. In heap there is no particular order in which you can allocate memory as in stack.

Understanding Memory Leak in C++

Memory leak happens due to the mismanagement of memory allocations and deallocations. It mostly happens in case of **dynamic memory allocation**. There is no automatic **garbage collection** in C++ as in Java, so programmer is responsible for deallocating the memory used by pointers.

Misuse of an elevator in a building in real life is an example of memory leak. Suppose you stay in an apartment building which has 19 floors. You wanted to go to 10th floor so you pressed the button to call the lift. The status of elevator is displaying as basement for 20 minutes. Then you realize that something is wrong, and upon investigating you find out that kids were playing in basement and they had blocked the lift door.

Similarly once a pointer is done with its operations it should free the memory used by it. So that other variables can use the memory and memory can be managed effectively.

By using the `delete` keyword we can delete the memory allocated:

For example:

```
*ex= new Example();

delete ex;
```

Copy
But in the above example **dangling pointer** issue can happen. Wait! what is a dangling pointer?

What is a Dangling Pointer?

A pointer pointing to a memory location of already deleted object is known as a dangling pointer.

- In the first figure pointer points to a memory location 1100 which contains a value 25.
- In the second figure pointer points to a memory location where object is deleted.

Dangling pointers arise due to object destruction, when an object reference is deleted or deallocated, without modifying the value of the pointer, so the pointer will keep on pointing to the same memory location. This problem can be avoided by initializing the pointer to NULL.

For example:

```
*ex = new Example();

Delete ex;

// assigning the pointer to NULL

ex = NULL;
```

Copy

What is a Smart Pointer?

Smart Pointer is used to manage the lifetimes of dynamically allocated objects. They ensure proper destruction of dynamically allocated objects. Smart pointers are defined in the memory header file.

Smart pointers are built-in pointers, we don't have to worry about deleting them, they are automatically deleted.

Here is an example of a smart pointer:

```
S_ptr *ptr = new S_ptr();

ptr->action();

delete ptr;
```

Copy

Multithreading in C++

Multithreading means two or more threads running concurrently where each thread is handling a different task. When you login to you Facebook profile, on your news feed, you can see live videos, you can comment or hit a like button, everything simultaneously. This is the best example of multithreading. Multithreading environment allows you to run many activities simultaneously; where different threads are responsible for different activities.

There are various uses of Multithreading, some of them are:

- Better resource utilization.
- Simpler program design.
- More responsive programs.

What is a Thread?

Thread is generally referred as a light weight process. Each thread executes different parts of a program. Each thread shares memory, file descriptors and other system resources. In Linux, all thread functions are declared in **<pthread.h>** header file. But it is not available in standard C++ library.

Creating Threads in Linux(C++)

1. `pthread_create()`: It creates a new thread. Below is the syntax:

```
pthread_create(threadID, attr, start_routine, arg)
```

Copy
In the code above:

threadID: Is a unique identifier for each thread. ThreadID of threads are compared using `pthread_equal()` function.

attr: Attribute object that may be used to set various thread attributes. It controls interaction of thread with rest of the program.

start_routine: The C++ routine that the thread will execute once it is created.

arg: Single argument must be passed by reference as a pointer of type void. If no argument is to be passed, null can be used.

2. `pthread_exit()`: It is used to terminate any thread.

Below is a simple program on creating threads in C++:

```
#include <iostream>

#include <pthread.h>

using namespace std;



char* str = "Child thread";



void* func(void *str)

{
```

```
    cout << "Child thread Created: " << (char*)str;

}

// main function

int main()

{

    s = ctime(&Time);

    // Step 1: Declaring thread

    pthread_t t;

    // Step 2: Calling create thread function

    pthread_create(&t, NULL, &func, (void*)str);

    /*

        Syntax for pthread_create is:

        pthread_create(threadID,attr,start_routine,arg)

        Here,

        threadID = t, arg = (void*)str, atrr = Null, start_routine =
func

    */

    cout << "Main thread created" << endl;

    pthread_join(t, NULL);

    //Exiting after completion

    exit(EXIT_SUCCESS);

    return 0;

}
```

Copy

```
Main thread created

Child thread created: Child thread
```

Joining and Detaching Threads

There are two methods which we can use to join or detach threads:

`join()` function

Joining of a thread is done by using the `join()` function of the thread class. It makes main thread and child thread inter dependent. Main thread terminates only after child thread terminates i.e. main thread waits for child thread to complete execution.

Syntax:

```
threadname.join();
```

Copy

It returns once all functions are completed. A thread is not joinable when it is assigned to another thread or when `join()` or `detach()` is called.

Syntax:

```
/*
    It checks whether a thread is joinable.
    It returns bool value.
*/
threadname.joinable();
```

Copy

`detach()` function

The `detach()` function detaches a thread from the parent thread. It allows both main thread and child thread to execute independently.

Syntax:

```
threadname.detach();
```

Copy

Program Example for using `join()` method

Let's have a simple example to demonstrate the use of `join()` function to join two threads:

```
#include <iostream>
#include <unistd.h>   // To include sleep function
#include<ctime>   // To get system time
#include <pthread.h>
using namespace std;


string s;
time_t Time = time(0);
```

```
void* func(void*)
{
    s = ctime(&Time);
    sleep(1);   //C alls sleep function
    cout << "Child thread Created " << s << endl;
}

// main function
int main()
{
    s = ctime(&Time);
    //Step 1: Declaring thread
    pthread_t t1[5];
    for(int i=0; i<5; i++)
    {
        cout << "Thread T[" << i << "] is Created " << s << endl;
        // Step 2: calling create thread function
        pthread_create(&t1[i], NULL, &func, NULL);
        // Joining threads, main thread waits for child thread to
complete
        pthread_join(t1[i], NULL);
}
//Exiting after completion
exit(EXIT_SUCCESS);
return 0;
}
```

Copy

```
Thread T[0] is Created Wed Nov 1 02:30:57 2017
Child thread Created Wed 1 02:30:57 2017]
Thread T[1] is Created Wed Nov 1 02:30:57 2017
```

```
Child thread Created Wed 1 02:30:57 2017

Thread T[2] is Created Wed Nov 1 02:30:57 2017

Child thread Created Wed 1 02:30:57 2017

Thread T[3] is Created Wed Nov 1 02:30:57 2017

Child thread Created Wed 1 02:30:57 2017

Thread T[4] is Created Wed Nov 1 02:30:57 2017

Child thread Created Wed 1 02:30:57 2017
```

Program Example for using `detach()` method

Let's have a simple example to demonstrate the use of `join()` function to detach two threads:

```
#include <iostream>

#include <unistd.h>   // To include sleep function

#include<ctime>   // To get system time

#include <pthread.h>

using namespace std;


string s;

time_t Time = time(0);


void* func(void*)

{

    s = ctime(&Time);

    sleep(1);   // Calls sleep function

    cout << "Child thread Created " << s << endl;

}


// main function

int main()

{

    s = ctime(&Time);

    // Step 1: Declaring thread

    pthread_t t1[5];

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

```
    {
        cout << "Thread T[" << i << "] is Created " << s << endl;
        // Step 2: Calling create thread function
        pthread_create(&t1[i], NULL, &func, NULL);
        // Step 3: main thread doesn't waits for child thread to
complete
        pthread_detach(t1[i]);
}
// Exiting after completion
exit(EXIT_SUCCESS);
return 0;
}
```

Copy

```
Thread T[0] is Created Wed Nov 1 02:38:14 2017
Thread T[1] is Created Wed Nov 1 02:38:14 2017
Thread T[2] is Created Wed Nov 1 02:38:14 2017
Thread T[3] is Created Wed Nov 1 02:38:14 2017
Thread T[4] is Created Wed Nov 1 02:38:14 2017
```

Hoope you have understood the concept of thread creation in C++.

Initializer List in C++

Initializer list is used to initialize data members. The syntax begins with a colon(:) and then each variable along with its value separated by a comma. The initializer list does not end in a semicolon.

Syntax:

```
Constructorname(datatype value1, datatype
value2):datamember(value1),datamember(value2)
{
    ...
}
```

Copy

For example:

```
#include<iostream>
using namespace std;
```

```
class Base
{
    private:
    int value;
    public:
    // default constructor
    Base(int value):value(value)
    {
        cout << "Value is " << value;
    }
};


int main()
{
    Base il(10);
    return 0;
}
```

Copy

```
Value is 10
```

The above code is just an example to understand the syntax of Initializer list. In the above code, `value` can easily be initialized inside the constructor as well, hence we do not have to use initializer list.

Uses of Initializer List in C++

There are situations where initialization of data members inside constructor doesn't work and Initializer List must be used. Following are such cases:

1) When no Base class default constructor is present

In Inheritance base class constructor is called first(Order of Constructor call), followed by the child class constructor.

Therefore, in the example below `Base_` class constructor will be called before `InitilizerList_` class constructor due to which the below program will throw compilation error: **"No default constructor exists for class Base_"**.

```
#include<iostream>

using namespace std;



class Base_

{

    public:

    // parameterized constructor

    Base_(int x)

    {

        cout << "Base Class Constructor. Value is: " << x << endl;

    }

};



class InitilizerList_:public Base_

{

    public:

    // default constructor

    InitilizerList_()

    {

        Base_ b(10);

        cout << "InitilizerList_'s Constructor" << endl;

    }

};



int main()

{

    InitilizerList_ il;

    return 0;
```

```
}
```

Copy

The above code example can be rewritten using initializer list, and will execute smoothly without any error.

Here is the new code:

```
#include<iostream>

using namespace std;



class Base_

{

    public:

    // parameterized constructor

    Base_(int x)

    {

        cout << "Base Class Constructor. Value is: " << x << endl;

    }

};



class InitilizerList_:public Base_

{

    public:

    // default constructor using initializer list

    InitilizerList_():Base_(10)

    {

        cout << "InitilizerList_'s Constructor" << endl;

    }

};



int main()

{

    InitilizerList_ il;
```

```
    return 0;

}
```

Copy

```
Base Class Constructor value is 10

InitilizerList_'s constructor
```

---

2) When reference type is used

If you have a data member as reference type, you must initialize it in the initialization list. References are immutable hence they can be initialized only once.

```
#include<iostream>

using namespace std;


class Base

{

    private:

    int &ref;

    public:

    Base(int &ref):ref(ref)

    {

        cout << "Value is " << ref;

    }

};


int main()

{

    int ref=10;

    Base il(ref);

    return 0;

}
```

Copy

```
Value is 10
```

---

3) For initializing const data member

const data members can be initialized only once, so it must be initialized in the initialization list.

```
#include<iostream>

using namespace std;



class Base

{

    private:

    const int c_var;

    public:

    Base(int c_var):c_var(c_var)

    {

        cout << "Value is " << c_var;

    }

};



int main()

{

    Base il(10);

}
```

Copy

```

Value is 10
```

---

4) When data member and parameter have same name

```
#include<iostream>

using namespace std;



class Base
```

```
{

    private:

    int value;

    public:

    Base(int value):value(value)

    {

        cout << "Value is " << value;

    }

};


int main()

{

    Base il(10);

    return 0;

}
```

Copy

```
Value is 10
```

---

5) For improving performance

If you are assigning the values inside the body of the constructor, then a temporary object would be created which will be provided to the assignment operator. The temporary object will be destroyed at the end of the assignment statement. Creation of temporary object can be avoided by using initializer list.

Creating and Using Namespace in C++

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

Creating a Namespace

Creating a namespace is similar to creation of a class.

```
namespace MySpace

{
```

```
    // declarations

}



int main()

{

    // main function

}
```

Copy

This will create a new namespace called **MySpace**, inside which we can put our member declarations.

Rules to create Namespaces

1. The namespace definition must be done at **global scope**, or nested inside another namespace.
2. Namespace definition doesn't terminates with a semicolon like in class definition.
3. You can use an alias name for your namespace name, for ease of use.

   Example for Alias:

   ```
   namespace StudyTonightDotCom

   {

       void study();

       class Learn

       {

           // class defintion

       };

   }



   // St is now alias for StudyTonightDotCom

   namespace St = StudyTonightDotCom;
   ```

   Copy

4. You cannot create instance of namespace.
5. There can be **unnamed** namespaces too. Unnamed namespace is unique for each translation unit. They act exactly like named namespaces.

   Example for Unnamed namespace:

```
namespace

{

    class Head

    {

        // class defintion

    };

    // another class

    class Tail

    {

        // class defintion

    };

    int i,j,k;

}



int main()

{

    // main function

}
```

Copy

6. A namespace definition can be continued and extended over multiple files, they are not redefined or overriden.

   For example, below is some **header1.h** header file, where we define a namespace:

```
namespace MySpace

{

    int x;

    void f();

}
```

Copy

We can then include the **header1.h** header file in some other **header2.h** header file and add more to an existing namespace:

```
#include "header1.h";
```

```
namespace MySpace

{

    int y;

    void g();

}
```

Copy

Using a Namespace in C++

There are three ways to use a namespace in program,

1. Scope resolution operator (::)
2. The using directive
3. The using declaration

With Scope resolution operator (::)

Any name (identifier) declared in a namespace can be explicitly specified using the namespace's name and the scope resolution :: operator with the identifier.

```
namespace MySpace

{

    class A

    {

        static int i;

        public:

        void f();

    };


    // class name declaration

    class B;


    //gobal function declaration

    void func();

}
```

```
// Initializing static class variable
int MySpace::A::i=9;


class MySpace::B
{
    int x;
    public:
    int getdata()
    {
        cout << x;
    }
    // Constructor declaration
    B();
}


// Constructor definition
MySpace::B::B()
{
    x=0;
}
```

Copy

The `using` directive

`using` keyword allows you to import an entire namespace into your program with a global scope. It can be used to import a namespace into another namespace or any program.

Conside a header file **Namespace1.h**:

```
namespace X
{
    int x;
    class Check
```

```
    {

        int i;

    };

}
```

Copy

Including the above namespace header file in **Namespace2.h** file:

```
include "Namespace1.h";



namespace Y

{

    using namespace X;

    Check obj;

    int y;

}
```

Copy

We imported the namespace `X` into namespace `Y`, hence class `Check` will now be available in the namespace `Y`.

Hence we can write the following program in a separate file, let's say **program1.cpp**

```
#include "Namespace2.h";



void test()

{

    using Namespace Y;

    // creating object of class Check

    Check obj2;

}
```

Copy

Hence, the `using` directive makes it a lot easier to use namespace, wherever you want.

The `using` declaration

When we use `using` directive, we import all the names in the namespace and they are available throughout the program, that is they have a global scope.

But with using declaration, we import one specific name at a time which is available only inside the current scope.

NOTE: The name imported with using declaration can override the name imported with using directive

Consider a file **Namespace.h**:

```
namespace X

{

    void f()

    {

        cout << "f of X namespace\n";

    }

    void g()

    {

        cout << "g of X namespace\n";

    }

}


namespace Y

{

    void f()

    {

        cout << "f of Y namespace\n";

    }

    void g()

    {

        cout << "g of Y namespace\n";

    }

}
```

Copy

Now let's create a new program file with name **program2.cpp** with below code:

```
#include "Namespace.h";
```

```
void h()

{

    using namespace X;  // using directive

    using Y::f; // using declaration

    f();    // calls f() of Y namespace

    X::f(); // class f() of X namespace

}
```

Copy

```
f of Y namespace

f of X namespace
```

In using declaration, we never mention the argument list of a function while importing it, hence if a namespace has overloaded function, it will lead to ambiguity.