Namespace

```
int a = 2;
int main () {
    int a = 4;
    cout << a;          // 4
    cout << "\n";
    cout << ::a;        // 2
    return 0;
}
```

redeclaration creates name collision.

Redeclaration of a variable in C++ shows error due to name collision of variable names.

used to create two same functions with No parameters and use them in function Overloading.

```
namespace ns {
    int a = 1;
}
int main () {
    int a = 2;
    cout << a << endl;        // 2
    cout << ns::a << endl;    // 1
}
/* ns:: a  → fully qualified name
    one way of calling the namespace.
namespace ns1 {
    int a = 1;
}
namespace ns2 {
    int a = 2;
```

```
int main () {
    cout << ns1 :: a << endl ;    // 1
    cout << ns2 :: a << endl;     // 2
}
```

features

→ Namespace do NOT have any access specifier like private / public .

→ Declarations of name space must be done in global scope, that it, namespace can NOT exist inside a class or a function.

But they can have a class or a function inside it.

→ can be nested within one another.

→ No semicolon will be put at the end of a namespace declaration

→ definition of namespace can be splitted over several units.

```
namespace ns1 {
    // code //
}
namespace ns1 {
    // code //
}
```

This is continuation of the same namespace, that is extension.

be created.

a name space can NOT

```
namespace ns1 {
    void fun() {
        cout << "ns1::fun";
    }
}

namespace ns2 {
    void fun() {
        cout << "ns2::fun";
    }
}

int main() {
    ns1::fun();    // ns1::fun
    ns2::fun();    // ns2::fun
}

namespace ns {
    void fun();
}

void ns::fun() {
    cout << "ns::fun";
}
```

```
const double x = 2.5;
```

This is correct.

```
const double x;
x = 2.5;
// error obtains :: initially, x has a
                    garbage value assigned to it.
                    That garbage value can NOT be changed
                    to 2.5 //.
```

```
namespace ns1 {
    int value() {
        return 10;
    }
}
namespace ns2 {
    const double x = 2.5;
    double value {
        cout << x;
    }
}
```

```
int main () {
    cout << ns1::value() ; // 10
    cout << ns2:: x ; // 2.5
    cout << ns2:: value() ; // 2.5
}

namespace ns {
    class A {
        public :
            void fun () {
                cout << " A : fun ";
            }
    };
}
int main () {
    ns:: A ob ;
    ob. fun (); // A: fun
}

namespace ns {
    class A {
        public:
            void fun() {
                cout << "A: fun ";
            }
    };
```

```
        void fun() {
            cout << "G: fun";
        }
}
int main() {
    ns:: A ob;
    ob.fun();      // A: fun
    ns::fun();     // G: fun
}

namespace ns {
    class A {
        public:
            static void fun() {
                cout << "A: fun";
            }
    };
    void fun() {
        cout << "G: fun";
    }
}
int main() {
    ns::A::fun();  // A: fun
    ns::fun();     // G: fun
}

namespace ns {
    class A {
        public:
            void fun();
    };
    void fun();
}
void ns::A::fun() {
    // code //
}
void ns::fun() {
    // code //
}
```

Nested namespace

```
namespace ns1 {
    int a = 1;
    int b = 3;
    int sum(int x, int y) {
        a = x;
        b = y;
        return (a+b);
    }
    namespace ns2 {
        int a = 10;
        int b = 20;
```

```
        int sum() {
            return (a+b);
        }
    }
}
int main() {
    cout << ns1 :: sum (5, 6);   // 11
    cout << ns1 :: ns2 :: sum ();  // 30
}
```

Unnamed namespace

```
namespace {
    void fun () {
        cout << "value = 2 " << endl;
    }
}
int main() {
    fun ();  // value = 2
}
```

They can be declared in any module with NO full qualification needed.

```
void fun () {
    cout << 3 ;
}
namespace {
    void fun () {
        cout << 5;
    }
}
int main () {
    fun ();  // Error is created
}
```

∵ Ambiguity takes place. Compiler can NOT decide which fun() to call //.

Namespace alias

syntax → namespace new_name = old_name;

```
namespace ns {
    void fun(int x) {
        cout << " value = " << x;
    }
}
```

```
int main () {
    namespace  Ns_new = ns ;
    Ns_new :: fun (5) ;  // value = 5
    ns :: fun (5) ;  // value = 5
}

void temp () {
    Ns_new :: fun(5);  // Error given.
                       // ∵ Ns_new is NOT
                       recognized outside
                       main() //
    ns :: fun (5) ;  // value = 5 ;
}
```

Namespace extension

```
namespace ns1 {
    int a = 1 ;
}
namespace ns2 {
    int b = 2;
    cout << ns1:: a ;
}                    //error
namespace ns1 {
    int c = 3;
}
```

```
int main () {
    [illegible]
    cout << ns1 :: a ;
    // 1
    cout << ns2 :: b ;
    // 2
    cout << ns1 :: c ;
    // 3
}
```

// namespace gets continued //.

Using declaration and using directives

Using directives allows us to import the entire namespace within the present scope.

```
namespace ns1 {
    int a = 1;
    void fun () {
        cout << "ns1 : fun " ;
    }
}
namespace ns2 {
    int a = 2;
    Void fun () {
        cout << "ns2 : fun " ;
    }
}
int main () {
    using namespace ns1 ;  // using directive
    fun () ;  // ns1 : fun
    cout << a ;  // 1
}
```

```
int main (){
    using namespace ns1;
    using namespace ns2;
    fun();          // error
    cout << a;      : ambiguity occours.
}
```

fun is for both ns1 and ns2.
∴ ambigious occours //.

// disadvantage of using 'using' directive //

Using declaration

```
namespace ns1 {
    int a = 1;
    void fun (){
        cout << 2;
    }
}
namespace ns2 {
    int a=3;
    void fun(){
        cout <<4;
    }
}
```

```
int main(){
    using namespace ns1;
    Using namespace ns2 :: fun ;
                    // using declaration //

    // The previous fun() gets overwritten
        if declaration is used. ∴ No
        more ambigious situation occours //.

    fun();      // "ns2 : fun"
                // 4
}
```

⟹ If declaration comes after directive, then ambiguity will not occour.

But,
if directive comes after declaration, then overwriting will NOT occour and ambiguity is occoured.

```
int main (){
    using namespace ns1 :: fun;
    usint namespace ns2;
    fun();      // error
}
```

```
namespace ns1 {
    int a = 1;
    void fun () {
        cout << "ns1 : fun"
    }
}
namespace ns2 {
    using namespace ns1;
    // int a = 2  → error : redeclaration //
    // entire ns1 gets copied to ns2
       like inline function //
    int b = 2;
    void show () {
        cout << "ns2 : show";
    }
}
int main () {
    using namespace ns2;
    show ();  // ns2 : show
    fun ();  // ns1 : show
}
```

In using declaration, never mention the argument list of function while inputing it. If the namespace has overloaded function, then it will result in ambiguity.

To create header file

// NS1.h :- //

vi NS1.h // file created, header file //

```
namespace X {
    int x;
    class sample {
        int i;
    };
}
```

// content of file NS1.h :- //

// NS2.h //

vi NS2.h

```
# include "NS1.h"
```

// If file is at same folder or same location, # include "NS1.h" . else # include < NS1.h > //

```
namespace Y {
    using namespace X;
    sample ob;
    int y;
}
```

vi Prog.cpp

```
#include "NS2.h"
void test (){
    using namespace Y;
    sample obj2;
}
```

vi add.h

```
namespace BasicMath {
    int add (int x, int y) {
        return (x+y);
    }
}
```

vi substract.h

```
namespace BasicMath {
    int substract (int x, int y) {
        return (x-y);
    }
}
```

vi Prog1.cpp

```
#include "add.h"
#include "substract.h"
using namespace BasicMath;
// ∵ Both namespaces has the same name,
//     ∴ namespace extension takes
//     place inside programm.
//  ∴ namespace extension occours //
int main (){
    add (5, 6); // 11
    substract (11, 10); // 1
}
```

Exception Handeling

Compile - Time Errors
(Syntax Errors)

Errors caught during compile time is basically Syntax error.

Run - Time Errors
(Exceptions)

An exception caught during runtime may abruptly terminate a program.

Exmp: division by 0.

It is syntactically correct, but Run Time error caught.

3 keywords of exception handling

→ try
→ catch
→ throw

try → it is intended to throw the exceptions

This block identifies a block of code which causes exceptions.

catch → A catch block is intended to handle the exception.

throw → It is used to communicate information about the error to the exception handler.

The throw statement accepts one parameter and that parameter is passed to handler.

Exception handling mechanism has four steps

→ find the problem (Hit the exception).
→ Inform about its occourance (throw the exception)
→ Receive error information (catch the exception).
→ Take proper action (Handle the exception).

```
int main() {
    int a=10, b=0;
    int c;
    try {
        if (b==0) {
            throw "division not possible";
        }
        c = a/b;
    }
    catch (const char *ex) {
        cout << ex;
    }
}
```

// If throw condition is satisfied, all the code written under else condition inside try block will be unreachable and will not be executed.

If throw condition is NOT executed then catch blocks remain unreachable. and catch block remains unexecuted.

```
int main() {
    int arr[3] = {-1, 0, 2};
    for (int i=0; i<3; i++) {
        int x = arr[i];
        try {
            if (x<0) {
                throw x;          // integer
            }
            if (x==0) {
                throw "zero";     // string
            }
            if (x>0) {
                throw 'x';        // character
            }
        }
        catch (char ch) {
            cout << ch;
        }
        catch (int x) {
            cout << x;
        }
        catch (const char *ex) {
            cout << ch;
        }
    }
    return 0;
}
```

For each and every iteration of the loop. the try - catch block gets executed.

try - catch is executed for each iteration.

output

-1
zero
"2"

```
int main () {
    try {
        throw 'A';
    }
    catch (int x) {
        cout << x;
    }
}
```

// Run Time error. output is NOT 65. typecasting never takes place //

// Implicit type conversion does NOT happen for primitive types.

'A' will NEVER be converted to 65.

Implicite type conversion do NOT happen for primitive types.

```
int main () {
    try {
        throw 'A';
    }
    Catch (...) {
        cout << "Caught";
    }
}
```

// generalised exception can handle any type of exception //

// generalised catch block, can take in every type of exceptions of try block //

⇒ generalised catch block must be the last catch block in case of multiple catch blocks.

After generalised catch block NO more Catch blocks can be placed. If placed, syntax error is reported.

```
int main() {
    try {
        throw 'A';
    }
    catch (char ch) {
        cout << ch;
    }
    catch (...) {
        cout << "caught";
    }
}
```

There may be some scenarios where an exception may be partially handled in the catch block. In those cases we can again throw (rethrow) an exception from the first catch block.

```
int main () {
    try {
        try {
            throw 5;
        }
        catch (int x) {
            cout << x;          // 5
            throw (x * 2);
        }
    }
    catch (int y) {
        cout << y;              // 10
    }
}

double div (int x, int y) {
    if (y == 0) {
        throw "div not possible";
    }
    return double (x) / y;
}

int main () {
    int x = 10, y = 0;
    try {
        double z = div (x, y);
        cout << z;
    }
    catch (const char * ex) {
        cout << ex;
    }
    return 0;
}
```

```
void fun (int *ptr, int x) {
        if (ptr == NULL) {
                throw ptr;
        } if (x == 0) { throw 'x'; }
}
int main () {
        try {
                fun (NULL, 0);
        }
        catch (...) {
                cout << "Caught ";
        }
        cout << "out";
}

Class Test {
        public:
                Test () {
                        cout << "const call";
                }
                ~Test () {
                        cout << "dest call";
                }
};
```

```
int main () {
        Test ob;
        try {
                throw ob;
        }
        catch (...) {
                cout << "Caught";
        }
}
```

// output :

```
const call
Caught
dest call
```

∵ The object of class is created outside the try block and is not within scope of try block, ∴ the object is destroyed after the end of the catch block.

//

```
int main() {
    try {
        Test ob;
        throw 5;
    }
    catch (...) {
        cout << "caught";
    }
}
```

// output:

const call
dest call
Caught

∵ The object of the class is created within the try block. ∴ it is in the scope of the try block. ∴ object is destroyed inside the try block, and then catch is executed.

→ When an exception is thrown, all objects created inside the closing try block are destructed before the control is transferred to the catch block. try block will destroy the object.

Exception Handling on Inheritance

```
class Base {
    // code;
};
class child : public Base {
    // code;
};
int main () {
    child ob;
    try {
        throw ob;
    }
    Catch (child ob) {
        cout << "child ob";
    }
    Catch ( Base ob) {
        cout << "Base ob";
    }
}
```

// child ob is output

child ob

Child class exception will NEVER be executed. If Base class exception catch is first in main function. child class is unreachable.

Super Class exception can handle all the subclass exceptions.

Super class exception can handle all subclass exceptions.

```
int main () {
    Child ob;
    try {
        throw ob;
    }
    Catch ( Base ob) {
        cout << "Base ob";
    }
    catch (Child ob) {
        Cout << "Child ob";
    }
}
```

// Base ob // Base ob

Output is Base ob

Custom Exception classes

```
Class Stack {
    int * ptr;
    int size;
    int top;
    public:
        Stack (int n) {
            size = n;
            top = -1;
            ptr = new int[size];
        }
        Void push (int x) {
            if ( top == size - 1) {
                throw overflow ();
            }
            top ++;
            ptr [top] = x;
        }
        int pop ( ) {
            if (top == -1) {
                throw Underflow ();
            }
            int x = ptr [top];
            top --;
            return x;
    } };
```

```
Class Overflow {
    public:
        overflow () {
            cout << "Stack Full" ;
        }
        void display(){ cout << "\n" ; }
};

Class Underflow {
    public:
        Underflow () {
            cout << "Stack empty " ;
        }
        void display () { cout << "\n" ; }
};

int main () {
    Stack ob(5);
    try {
        ob.push (1) ;
        ob. push (2);
        ob.push (3);
        ob.push (4);
        ob.push (5);
        ob.push (6);
        Cout << ob. pop () << ob.pop () << ob.pop ;
        cout << ob.pop () << ob.pop ;
        Cout << ob.pop () ;
    }
    Catch ( overflow ov ) {
        cout << " overflow caught";
        ov. display () ;
    }
    Catch ( Underflow Un) {
        cout << " Underflow caught " ;
        Un. display () ;
    }
    return 0;
}
```

Exception Handling with operator overloading

```
Class Number {
    int n ;
    public:
        Number (int x = 0) {
            n = x;
        }
        Void operator -- ( ) {      // pre decrement
            if (n = = 0) {
                throw Number() ;
            }
            n -- ;
        }
        void show () {
            cout << n ;
        }
};
```

```
int main () {
 try { Number N(5);
       cout << "Before decrement ";
        N. show();
        while (1) {
            -- N;
             N.show();
        }
    }
    catch (Number ob) {
          ob.show();
    }
    return 0;
}
```

// output: 5 4 3 2 1 destructor called → 0

```
class A {
   public:
      A ( ) {
        cout << "default constructor ";
      }
      ~A () {
          cout << " destructor called";
      }
      A ( A& ob) { cout << "copy constructor " ; }
};
```

```
int main () {
      A ob ;
   try {
      throw ob;
   }
   catch (...) {
        cout << "Caught ";
   }
   return 0;
}
```

// output

constructor called
copy constructor
caught
destructor called
destructor called

for the ob thrown, another object is created and destroyed. The original ob is NOT thrown. new object ob is made.

```
int main () {
    A ob;
    try {
        throw 5;
    }
    catch (...) {
        cout << "Caught " ;
    }
    return 0;
}
```

// output:
constructor called
5
destructor called.

```
class ZeroDivide {

    public:
        void show() {
            cout << " zero divide error ";
        }

};

int main () {
    int x, y;
    x = 3;
    y = 0;

    try {
        // double z = double x / y ; //
        if (y == 0) {
            throw ZeroDivide ();
        }
        double z = double x / y;
        cout << z;
    }

    catch (ZeroDivide ob) {
        ob.show();
    }

    return 0;

}
```

```
double div (int x, int y) {
    if (y == 0) {
        throw ZeroDivide ();
    }
    return (double) x / y ;
}
```

Exceptions and Inheritence

```
class Person {
    protected:
        char name [50];
        int age;
};

class Employee : public Persons {
    float height ;
    float salary ;
    public :
        void get_Data () {
            cout << " Enter age, name ";
            cin >> name;
            cin >> age;
            if (age <= 0) {
                throw Employee ();
            }
            cin >> id ;
            cin >> salary;
        }
```

```
    void show() {
        cout << name << age << endl;
        cout << id << salary << endl;
    }
};
int main() {
    try {
        Employee ob;
        ob.get_Data();
        ob.show();
    }
    Catch ( Person ob) {
        Cout << "Caught";
    }
    catch (Employee ob) {
        cout << "Caught";
    }
    return 0;
}
```

// Both the catch ( ) statements will run.

```
class X {
    int x;
    public:
        X(int a=0) {
            x=a;
        }
        friend class Y;
}
class Y {
    int y;
    public:
        Y(int a=0) {
            y=a;
        }
        void set_a (X ob) {
            ob.x = 100;
        }
}
```

⇒ A constructor of a class can take any types of parameters except the photocopies of objects of that particular class. ∴ copy constructors must have reference passed and not photocopies.