C++ Classes

A class specification has two parts:
- Class declaration, which describes how certain class member functions are implemented.
- Class definition, which describes the data component, i.e., members and the public interface, in terms of member functions.

The principle is to separate the details of the implementation from the design.

The class method definitions, which describe how certain class member functions are implemented, should be changed without changing the interface.

The implementation of the data representation of the member functions can be changed without changing the interface.

The important distinction between function declaration and function definition is the inclusion of the keyword inline for function definitions that are to be embedded.

Recall the distinction between function declaration and function definition.
When defining a member function outside the class definition, use the scope resolution operator (::) to identify the class to which the function belongs. We can use the same name for a member function for a different class.

Class member functions can access the private components of the class. Examples:

```cpp
void Stack::push(int item)
{
    data[count] = item;
    ++count;
    pileTop = pilePtr;
}
```

```
void Pile::push(int item)
{
    pilePtr = new pileNode;
    pilePtr->value = item;
    pilePtr->next = pileTop;
    pileTop = pilePtr;
}
```

```cpp
class Stack
{
public:
    void init(); // Initialize the stack
    void push(int item); // Push an item onto the stack
    int pop(); // Pop an item from the stack
private:
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves
};
```

```
C++ Class Declaration Example
```

The scope resolution operator can be used to refer to member functions within a class.

```cpp
class X
{
public:
    void f(int N)
    {
        float X = 1.5; // Local x
    }

    ::X = 2.5; // Global X
};
```

```cpp
void glocal x;
```

```cpp
// C++ Class Declaration Example
```
Good Style: Use friend access only when absolutely necessary.

Friend declarations must be used in some cases which will be described later.

The second and third forms grant friend privileges to the entire class function.

The first form grants friend privileges to the entire class function.

{ friend (friendclass friendFunct( );
friend (friendclass :: friendFunct( );

classGrantee

++Friend declarations

++ Using a Class

Can create a class object by declaring a class variable or by using new to allocate.

Example:

Stack s;

//define a stack object
Stack *sp;

//point to a stack object
sp->init();

//invokes a method
sp->push(7);

//insert 7
sp->push(12);

//insert 12
sp->pop();

//remove and print
sp->data[0];

***Error: data private***

sp = new Stack[20];

//array of stacks

++ Strict information hiding provided by private may be too strong in some cases.

++ C++ friends

C++ provides a controlled way to grant access to private data to functions that are not member functions of the class.

++ const Member Functions

A program can declare a const member function of a const class.

Example:

const Stack

In the function body, it is an error to invoke a non-const member function of a class by passing the key word const between the function and argument list.

A non-const function can be declared to not change the internal data of a class by suffixing the keyword const.

Since the programmer manipulates data in the class by calling member functions, a program cannot declare a const member function of a class. For example:

++ Good Style:

417
More Class Constructors Examples

```
class Color
{
private:
    float red;
    float green;
    float blue;

public:
    Color(); // constructor with initial values
    Color(float r, float g, float b); // constructor used
    ...// other member functions

Color c1; // Color::Color() constructor used
Color *c2 = new Color; // Color::Color() constructor used
Color c3(1.0, 0.5, 0.0); // Color::Color(float r, float g, float b) constructor used
```

Class constructors are special method functions for constructing new objects.

The compiler ensures that the constructor is invoked whenever an object is created. A constructor's name is the class's name. The constructor takes no arguments if it is called the `default constructor`. If you do not define any constructors, the compiler provides a default constructor definition.

Once you define a constructor, a program must use it when creating an object. If you fail to define any constructors, the compiler provides a default constructor, one does nothing.

```
class Stack
{
public:
    Stack(); // stack constructor
    void push(int item); // Push an item on the stack
    int pop(); // Pop an item from the stack

private:
    int count; // Number of items in the stack
    int data[STACK_SIZE]; // The items themselves

    Stack::Stack()
    { count = 0; } // constructor definition
}
```

Friends Example

```
class C
{
private:
    int cdm; // Number of items in the stack

public:
    int m(void); // class method
    friend int t(void); // friend method
    friend int F::f(void); // method friend
}
```

Class Constructors Example

```
class F
{
private:
    int adm;

public:
    int f(void); // method friend
}
```
Other C++ Operators

Value construction operator:

- `type (expression)`

An alternative to the cast operator.

Pointer to class members must reference a specific class object. There are two special operators for doing this:

- `classObject* memberPtrVar` Dereferences `memberPtrVar` contained in class `Object`.
- `classObjectPtr->*memberPtrVar` Dereferences `classObjectPtr` to access some `classObject`, then dereferences `memberPtrVar` in that `classObject`.

Constructor/Destructor Example

class myString

private:

- `int len;` // length
- `char* s;` // string

public:

- two constructors

  - `myString(int size=255)` // String S1
    
    ```cpp
    len = size;
    s = new char[size+1];
    ```

  - `myString(char* si)` // String S2 = "initialValue"
    
    ```cpp
    assert(si);
    len = strlen(si);
    s = new char[len+1];
    strncpy(s, si, len+1);
    ```

- `˜myString()`

```cpp
if (s)
    delete[] s;
```
There are two ways to define overloaded (binary) operators

- The overloaded operator is defined as a member function of some class.
  - The operator will be invoked when it appears with a left operand
    that is an object of the class.
  - The this pointer will refer to the left operand.
  - A parameter of the right type will be required for the right operand.
  - The overloaded operator is defined as the member function of some class.
  - There are two ways to define overloaded (binary) operations

- The overloaded operator is defined as a non-member function
  - The standard rules for resolving references to overloaded functions will be used to
    determine when the operator function is invoked.
  - The function will require two parameters for the left and right operands.
  - You must use this form if the left operand cannot be an object of the class.

Examples:

```cpp
int operator << (const myString &sRight) const;
int operator <= (const myString &sLeft, const myString &sRight) const;
```

### C++ this Pointer Example

```cpp
class Stock {
private:
    double totalVal;
...
public:
    const Stock &topVal( const Stock &s) const {
        if (s.totalVal > totalVal) return s;
        else return *this;
    }
}
```

```cpp
Stock s1, s2;
// two class objects
Stock top = s1.topVal(s2);
// top = max(s1, s2)
```
To overload the `<<` operator to display an object of class `ClassName`, use a friend function with a definition:

```cpp
friend ostream & operator <<(ostream & os, const ClassName & obj)
```

Example:

```cpp
Pair V(4.0, 5.0);
```

```cpp
cout << V; // now supported
```

```cpp
Pair Q = V * 2.0; // now supported
```

```cpp
Pair Q = 2.0 * V; // Error, not supported
```

## Operator Overloading Restrictions

- You cannot overload the following operators:
  - `sizeof`
  - `operator /`
  - `operator %`
  - `operator ->`
  - `operator ->*`
  - `operator *=`
  - `operator +=`
  - `operator -=`
  - `operator &=`
  - `operator |=`
  - `operator >>=`
  - `operator <<=`
  - `operator +=`
  - `operator -=`
  - `operator &=`
  - `operator |=`
  - `operator >>=`
  - `operator <<=`

- You cannot change the type (arity) of the operator.
- You cannot change the prototype of the operator.
- You cannot use an operator in a manner that violates the syntax and semantic rules of the original operator.
- You cannot overload operators as operators of the type `operator...`.
- You cannot overload operators as operators of the type `friend operator...`.

**Good Style:**
- **Declaration:**
  ```cpp
  Pair operator*(double N, const Pair & A);
  ```
- **Definition:**
  ```cpp
  Pair operator*(double N, const Pair & A) { return A * N; }
  ```

## Friends and Operator Overloading

A friend function is a nonmember function that is allowed access to a member of a class. A class member function is not allowed to access a member of a class unless it is a friend of the class.

### Example

```cpp
class Pair {
  private:
    double X, Y;
  public:
    Pair(double X, double Y) : X(X), Y(Y) {
      // default constructor
    }
    Pair(double X) : Pair(X, 0) {
      // constructor
    }
    Pair() : X(0), Y(0) {
      // default constructor
    }
    Pair operator -(const Pair & B) const { return Pair(X - B.X, Y - B.Y); }
    Pair operator *(double N) const { return Pair(MX, MY); }
    Pair operator +(const Pair & B) const { return Pair(MX + B.X, MY + B.Y); }
    Pair operator -(double N) const { return Pair(MX - N, MY); }
    Pair operator +(double N) const { return Pair(MX + N, MY); }
    // additional operators...

    // Getters
    double X() const { return X; }
    double Y() const { return Y; }

    // Setters
    void X(double x) { X = x; }
    void Y(double y) { Y = y; }

    // Other methods...
};

int main() {
  Pair P1(1.0, 2.0);
  Pair P2(3.0, 4.0);
  P1 = P1 + P2; // operator overloaded
  return 0;
}
```
Conversion Function Example

```cpp
class Clock {
public:
    int hour, min, ampm;
    Clock();
    // hour = 12, min = 0, ampm = 0
private:
    Clock(int); // time is given as 24-hour time
};

int time = int (c);  // convert clock to int using conversion function
if (ampm == 1) return (hour+12)*100+min;
else return hour*100+min;

Clock c = 2249;  // convert int to clock using convert constructor

int time1 = c; // convert clock to int using conversion function
int time2 = int (c);  // convert int to clock using convert constructor
```

Type Conversions: Conversion Functions

Conversion function are used to convert a class object to some other type, i.e., `type-name()`. Conversion function is a class member function, it has no declared return type and no arguments.

The conversion function is automatically invoked when you assign a class object to a variable of that type or use the type cast operator to that type.

- The conversion function is automatically invoked when you assign a class object to a variable of that type.
- Conversion function are used to convert a class object to some other type.

Type Conversions: Conversion Constructors

A one-parameter constructor is called a convert constructor.

- When a function that is declared to return a `C` object tries to return a `T` value.
- When a function that is declared to return a `C` object expects a `T` object.
- When a `T` value is passed to a function expecting a `C` object.
- When a `T` value is assigned to a `C` object.
- When an object is initialized to a `T` type value.

In the following situations, convert constructors are used to convert a `T` type to another type.

- When a function that is declared to return a `C` object expects a `T` object.
- When an object is initialized to a `T` type value.
- A one-parameter constructor is called a convert constructor.

Type Conversions: Conversion Constructors
Overloading the Assignment Operator

The assignment operator (=) is used when one object is assigned to another existing object. If you don't define an assignment operator, the compiler provides one which performs a member-by-member copy of the non-static members.

Good Technique:
- If an overloaded assignment operator is defined, it should check for self-assignment, i.e., X = X;
- It should free memory formerly pointed to by member pointers
- It should copy the data, not just the address of the data
- It should perform a member-by-member copy of the non-static members
- If an overloaded assignment operator is defined, the assignment operator should be called
- If you don't define an assignment operator, the compiler provides one which
- The assignment operator (=) is used when one object is assigned to another

Copy Constructors

Class name (const Class name & st);

The copy constructor is invoked whenever a new object is created and initialized to an existing object of the same kind.

Good Technique:
- If the class contains a member whose value changes when new objects are created, you should provide an explicit copy constructor.
- If a class contains a member whose value changes when new objects are created, you should provide an explicit copy constructor.
- If you don't define a copy constructor, the compiler provides a default copy constructor.
- The copy constructor is used whenever a new object is created and initialized to an existing object of the same kind.

Class name (const Class name & st);

Copy Constructors
The overloaded `new` operator function must return a `void*`. The pointer `p` must point to the object to be created.

The first parameter in the overloaded operator must return a `void*`.

The second parameter of the overloaded operator must be the size in bytes of the object to be created.

The value of the parameter is the size in bytes of the object to be created.

The second parameter in the overloaded new operator must be of type `size_t`.

The overloaded new operators must return a `void*`.

Both forms can be either a member function or a non-member function.

```cpp
friend ostream & operator<<(ostream & os, const myStr & S);
myStr(int size=255);
...;
myStr(char * SI);
myStr(int size=255, char * SI);
int length();
int length(int size=255);
myStr(const myStr & SR);
myStr(const myStrPtr this, const myStr & SR);
myStr(const char * SI);
myStr(const myStrPtr this, const char * SI);
```

### Overloading Memory Management Operators

The overloaded `new` operators must return a `void*`.

The first parameter in the overloaded operator must be of type `size_t`.

The value of this parameter is the size in bytes of the object created.

The overloaded `delete` or `delete[]` operator function must return a `void`.

The first parameter in the overloaded operator must be of type `void*`.

The pointer `objectPtr` must point to the storage to be freed.

Object initialization with `new`:

```cpp
Class name* ptr = new Class name(val);
```

Invoke the `Class name(Type)` constructor, where `Type` is the type of `val`.

```cpp
Class name* ptr = new Class name;
```

Invoke the default constructor.

Initializer lists:

- Class name::Class name(...): member1(..), member2(..), ...
- Must use this form to initialize a non-static const data member.
- Must use this form to initialize a referenced data member.

- Constructor:

  ```cpp
  Class name::Class name();
  ```

### More on Constructors

To help you understand how C++ implements various C++ constructs:

456

Looking Under the Hood

This example doesn't necessarily correspond to any actual implementation of the C++ compiler.

- Identifiers that start with `#` are internal names that are created by the compiler.
- The right column shows the declarations and statements that the C++ compiler generates internally to implement the C++ constructs.
- The left column shows the C++ program as you would write it.
- To help you understand how C++ classes actually work, Slides 448 through 452 describe how the C++ compiler implements various C++ constructs in terms of C code.

448
The class template construct defines a generic class with substitutable type and constant expression parameters. The templateParameters is a list of items:

class typeName
type-name

The class parameter defines a type parameter that can be substituted in the body of the class anywherethatatype is required.

The second form of parameter defines an expression parameter that can be substituted in the class anywherethat an expression of type-type-name is required.

Function Templates

The function definition uses the angle brackets to indicate that the function is parameterized.

The typeList is a comma-separated list of items of the form:

class identifier
type-name

The identifiers may be used in the function definition any place that a class reference is a component-separated list of items of the form:

template and the angle brackets are required.

The first use of each type argument in a call determines the type used in the call to instantiate an instance of the function with an appropriate body.

When the function is called the compiler uses the type of the arguments to the function template to determine the type of the arguments to the function.

The elements of the typeList may be used in the function definition any place that a class reference is used.

Function Template Example

```cpp
template <Type>

void swap(Type &A, Type &B)
{
    Typetemp = A;
    // Swap any A and B
    A = B;
    B = temp;
}

int J, K;
double X, Y;
swap(J, K); // swaps integers
swap(X, Y); // swap doubles
```
In general, inheritance can never increase the visibility of a member:

- In a derived class, the local member hides the inherited member.
- If a derived class adds a member with the same name as a member in the base class, the derived class can access the inherited member.
- A derived class can access its public and protected members and methods of the base class, except for constructors, the destructor, and the overloaded assignment operator.

Member Accessibility

Inheritance is the mechanism that is used to build a hierarchy of classes to perform

```
template<class T, int size>
class Stack {
    private:
        T items[size];
        int top;

    public:
        Stack();
        void push(T newVal);
        T pop();
        bool isEmpty();
        bool isFull();

    // Stack of 1000 integers
    Stack<int, 1000> S1;
    // Stack of 100 myStrings
    Stack<std::string, 100> S2;
};
```
Constructor Under Inheritance: Example

```cpp
const int MaxLen = 100;

class B
// base class
protected:
  char *name;
  int maxlen;
public:
  B() // B's default constructor
    maxlen = MaxLen;
    name = new char[maxlen];

class D:
// derived class
public:
  // invoke B's default constructor
  D(char *n): B()
    strcpy(name, n);

D foo("foo");
```

Inheritance Example

```cpp
class B
// base class
...

class D:
// derived class
...
```

Constructor Under Inheritance

Constructors have to invoke a constructor explicitly:
- If D has no constructors but B has no default constructor, then D will be used instead of the default constructor.
- If D has no constructors but B has a default constructor, then B's default constructor will be used.
- If D has constructors and B has a default constructor, then B's default constructor will be used unless the D constructor explicitly invokes some other B constructor.
- If D has constructors and B has no constructors, then the appropriate D constructor will be used.

Let B be a base class and D be a derived class from B. When a D object is created:
- Derived class constructors may invoke a base class constructor (if exists) explicitly.
- Derived class constructor initializes fields added by the derived class.
- Base class constructor initializes fields assigned from the base class.
- Derived class constructor overrides the "from the base class" part of the derived class.

Inheritance Example

```cpp
class B
// base class
...

class D:
access-specifier is one of public, protected, or private
...;

class D:
access-specifier is one of public, protected, or private
...

class D:
access-specifier is one of public, protected, or private
...
```

Inheritance
In multiple inheritance, a derived class has multiple base classes.

Derived class typically represents a combination of its base classes.

```cpp
class iostream:
public
  istream,
public
  ostream
...
```

Name conflicts are resolved using scope resolution operator.

In multiple inheritance, a derived class has multiple base classes.

Virtual Methods Example

```cpp
class B
public:
  virtual
  void g();//virtual method
  int h();

class D:
public B
  void g();//virtual method
  int h();

main()
  D d;
  B* ptr = &d;
  ptr->h();//B::h invoked
  ptr->g();//D::g invoked
```

Abstract Class and Pure Virtual Functions

Abstract class is a base class which is required to have a derived class.

Abstract class is not allowed to have objects that belong to it.

Abstract class is a base class which is required to have a derived class.

The purpose of declaring a pure virtual function is to have derived classes inherit a function interface only.

```cpp
virtual void f(int) = 0; // pure virtual function
```

Abstract class is specified by declaring a pure virtual function in the class.

```cpp
class AC //abstract class... public:
  virtual
    void f(int) = 0;//pure virtual function
```

The purpose of declaring a pure virtual function is to have derived classes inherit a function interface only.

```cpp
virtual methods are declared with the keyword virtual.
```

It is a derived class which has a virtual method, the declaration method must be compatible with the keyword virtual.

For non-virtual functions with the same name, the system determines at compile-time which of the functions to invoke.

Virtual methods are declared with the keyword virtual.

```cpp
virtual
  void f(int) = 0;//virtual method
  public
  class B
    {;
      virtual
        void g(){
          //virtual method
        }
    }
```

Polymorphism and Virtual Methods

Polymorphism refers to the runtime binding of a pointer to a method.

```cpp
virtual
  void f(int) = 0; // pure virtual function
```

In a derived class there is a virtual function that has the same name.

```cpp
virtual void f(int) = 0;
```

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

```cpp
virtual
  void f(int) = 0;
```

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.

Virtual methods are declared with the keyword virtual.

For virtual methods with the same name, the system determines at runtime which of the methods to invoke.
The exception handling mechanism in C++ provides a cleaner way to deal with error and exceptional conditions that arise during normal processing. Use exception mechanisms only for true exceptions, not for general processing.

General mechanism:
- User-defined exception classes
- `throw` statement to signal an exception
- `try` and `catch` to handle exceptions
- Use defined exception classes

The exception handling mechanism in C++ provides a cleaner way to deal with error.

C++ Exception Handling

Destructors Under Inheritance

Destructors Under Inheritance Example

```cpp
class B
public:
    B()
    { cout << "B's constructor \n"; }
    ~B()
    { cout << "B's destructor \n"; }

main()
{ D d; // Printed output
    // B's constructor
    // D's constructor
    // D's destructor
    // B's destructor
}
```

Virtual Base Class Example

```cpp
class A
protected:
    int x;
...

class B : public virtual A
{ ...
    int x;
    protected:
        B();
...
}

class C : public virtual A
{ ...
    int x;
    protected:
        C();
...
}

class D : public B,
     public C
{ ...
    int x;
    protected:
        D();
...
}
```
Namespaces

Namespaces can be used to resolve name conflicts, put global declarations in namespaces.

```
namespace lib1
{
    void clr_screen();
    ...
}
```

Namespaces members can be referred to using scope resolution operator, `lib1::clr_screen()` or using-declaration:

```
using lib1::clr_screen;//put lib1's clr_screen() in local namespace
```

Using namespaces to resolve name conflicts and global declarations in namespaces can be distinguished from global names. Two libraries may have the same name.

```
namespace lib2
{
    void clr_screen();
    ...
}
```

```
Namespaces
```

```
Try Statement
```

Try statement associates a collection of catchers with a block of statements.

```
try
{
    statements
}
catch ( .. )
{
    statements
}
```

C++ Try Statement

```
Exception Handling Example
```

```
class MyString
{
    char * S;
public:
    enum minSize = 1, maxSize = 1000;
    MyString();
    MyString(int);
    ...
};
```

```
MyString::MyString(int size)
{
    if ( size <= minSize || size >= maxSize )
        throw ( size );
    S = new char[size];
    if ( S == NULL )
        throw ( "Out of Memory" );
}
```

```
void f(int N)
{
    try
    {
        MyString str(N);
    }
    catch ( char * errMsg )
    {
        cerr << errMsg << endl;
        abort();
    }
    catch ( int k )
    {
        cerr << "Out of range error : " << k << endl;
    }
    ...}
```

C++ Throw Statement

```
throw expression;
```

Throw statement stops sequential execution and starts search for exception handler. The type of the expression is used to determine the handler that is invoked. The value of the expression is passed as a parameter to the handler that is invoked.

```
try
{
    ...}
throw expression;
```

```
C++ Try Statement
```