An object is a variable of a user-defined type. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Polymorphism

Polymorphism is the quality that allows one name to be used for two or more related but technically different purposes.
Polymorphism allows one name to specify a general class of actions. Within a general class of actions, the specific action to be applied is determined by the type of data. For example, in C, the absolute value action requires three distinct function names: **abs( )** for integer, **labs( )** for long integer and **fabs( )** for floating-point value. However in C++, each function can be called by the same name, such as **abs( ).** The type of data used to call the function determines which specific version of the function is actually executed.
In C++ it is possible to use one function name for many different purposes. This type of polymorphism is called *function overloading*.
Polymorphism can also be applied to operators. In that case it is called *operator overloading*.
More generally the concept of polymorphism is characterised by the idea 'one interface, multiple methods'. The key point to remember about polymorphism is that it allows you to handle greater complexity by allowing the creation of standard interfaces to related activities.

Inheritance

Inheritance is the process by which one object can acquire the properties of another. An object can inherit a general set of properties to which it can add those features that are specific only to itself.
Inheritance is important because it allows an object to support the concept of *hierarchical classification.* Most information is made manageable by hierarchical classification.
The child class inherits all those qualities associated with the parent and adds to them its own defining characteristics.

Differences between C and C++

Although C++ is a subset of C, there are some small differences between the two, and few are worth knowing from the start.
First, in C, when a function takes no parameters, its prototype has the word **void** inside its function parameter list. For example if a function **f1( )** takes no parameters (and returns a **char**), its prototype will look like this:

```
char f1(void);  /* C version */
```

In C++, the **void** is optional. Therefore the prototype for **f1( )** is usually written as:

```
char f1( );  //C++ version
```

this means that the function *has no parameters*. The use of **void** in C++ is not illegal, it is just redundant. Remember these two declarations are equivalent.

Another difference between C and C++ is that in a C++ program, *all functions must be prototyped*. Remember in C prototypes are recommended but technically optional. As an example from the previous section show, a member function's prototype contained in a class also serves as its general prototype, and no other separate prototype is required.

A third difference between C and C++ is that in C++, if a function is declared as returning a value, it must return a value. That is, if a function has a return type other than **void**, any **return** statement within the function must contain a value. In C, a non **void** function is not required to actually return a value. If it doesn't, a garbage value is 'returned'.
In C++, you *must explicitly declare the return type* of all functions.

Another difference is that in C, local variables can be declared only at the start of a block, prior to any 'action' statement. In C++, local variables can be declared anywhere. Thus, local variables can be declared close to where they are first use to prevent unwanted side effects.

C++ defines the **bool** date type, which is used to store Boolean values. C++ also defines the keywords **true** and **false,** which are the only values that a value of type **bool** can have.

In C, a character constant is automatically elevated to an integer, whereas in C++ it is not.

In C, it is not an error to declare a global variable several times, even though it is bad programming practice. In C++, this is an error.

In C an identifier will have at least 31 significant characters. In C++, all characters are considered significant. However, from practical point of view, extremely long identifiers are unwieldy and seldom needed.

In C, you can call **main( )** from within the program. In C++, this is not allowed.

In C, you cannot take the address of a **register** variable. In C++, you can.

In C, the type **wchar_t** is defined with a **typedef**. In C++, **wchar_t** is a keyword.

Differences between C++ and Standard C++

The traditional C++ and the Standard C++ are very similar. The differences between the old-style and the modern style codes involve two new features: new-style headers and the **namespace** statement. Here an example of a do-nothing program that uses the old style,

```
/*   A traditional-style C++ program */
#include < iostream.h >
int main( ) {
   /* program code */
   return 0;
}
```


New headers

Since C++ is build on C, the skeleton should be familiar, but pay attention to the **#include** statement. This statement includes the file **iostream.h**, which provides support for C++'s I/O system. It is to C++ what **stdio.h** is to C.
Here the second version that uses the modern style,

```
/*
   A modern-style C++ program that uses
   the new-style headers and namespace
*/
#include < iostream>
using namespace std;
int main( ) {
   /* program code */
   return 0;
}
```

First in the **#include** statement, there is no **.h** after the name **iostream.** And second, the next line, specifying a namespace is new.

The only difference is that in C or traditional C++, the **#include** statement *includes a file* (file-name**.h**). While the Standard C++ do not specify filenames. Instead the new style headers simply specify *standard identifiers* that might be map to files by the compiler, but they need not be. New headers are abstractions that simply guaranty that the appropriate prototypes and definitions required by the C++ library have been declared.
Since the *new-style header is not a filename*, it does not have a **.h** extension. Such header consists only of the header name between angle brackets:

```
< iostream >
< fstream >
< vector >
< string >
```

Standard C++ supports the entire C function library, it still supports the C-style header files associated with the library. That is, header files such as **stdio.h** and **ctype.h** are still available. However Standard C++ also defines new-style headers that you can use in place of these header files. For example,

Old style header files	*Standard C++ headers*
< math.h >	< cmath >
< string.h >	< cstring >

Remember, while still common in existing C++ code, old-style headers are obsolete.

Namespace

When you include a new-style header in your program, the contents of that header are contained in the **std** namespace. The namespace is simply a declarative region. The purpose of a namespace is to localise the names of identifiers to avoid name collision. Traditionally, the names of library functions and other such items were simply placed into the global namespace (as they are in C). However, the contents of new-style headers are place in the **std** namespace. Using the statement,

```
using namespace std;
```

brings the **std** namespace into visibility. After this statement has been compiled, there is no difference working with an old-style header and a new-style one.

Working with an old compiler

If you work with an old compiler that does not support new standards: simply use the old-style header and delete the **namespace** statement, i.e.

replace:	by:
`#include < iostream>`	`#include < iostream.h >`

```
      if (even(11)) cout << "11 is even\n";
      return 0;
}
```

In this example the function **even( )** which return **true** if its argument is even, is declared as being in-line. This means that the line

```
if (even(10)) cout << "10 is even\n";
```

is functionally equivalent to

```
if (!(10%2)) cout << "10 is even\n";
```

This example also points out another important feature of using **inline**: an in-line function must be define *before* it is first called. If it is not, the compiler has no way to know that it is supposed to be expanded in-line. This is why **even( )** was defined before **main( )**.

Depending upon the compiler, several restrictions to in-line functions may apply. If any in-line restriction is violated the compiler is free to generate a normal function.

Automatic in-lining

If a member function's definition is short enough, the definition can be included inside the class declaration. Doing so causes the function to automatically become an in-line function, if possible. When a function is defined within a class declaration, the **inline** keyword is no longer necessary. However, it is not an error to use it in this situation.

```
//example of the divisible function
#include < iostream >
using namespace std;

class samp {
      int i, j;
   public:
      samp(int a, int b);
         //divisible is defined here and
         //automatically in-lined
      int divisible( ) { return !(i%j); }
};

samp::samp(int a, int b){
      i = a;
      j = b;
}

int main( ) {
      samp ob1(10, 2), ob2(10, 3);
      //this is true
      if(ob1.divisible( )) cout<< "10 divisible by 2\n";
   //this is false
   if (ob2.divisible( )) cout << "10 divisible by 3\n";
   return 0;
}
```

Perhaps the most common use of in-line functions defined within a class is to define constructor and destructor functions. The samp class can more efficiently be defined like this:

```
//...
class samp {
      int i, j;
   public:
         //inline constructor
      samp(int a, int b) { i = a; j = b; }
      int divisible( ) { return !(i%j); }
};
//...
```

MORE ABOUT CLASSES

Assigning object

One object can be assigned to another provided that both are of the same type. By default, when one object is assigned to another, a bitwise copy of all the data members is made. For example, when an object called **o1** is assigned to an object called **o2** , the contents of all **o1'**s data are copied into the equivalent members of **o2.**

```
//an example of object assignment.
//...
class myclass {
      int a, b;
   public:
      void set(int i, int j) { a = i; b = j; };
      void show( ) { cout << a << " " << b << "\n"; }
};

int main( ) {
```

```
      samp(int a, int b) { i=a; j=b; }
      int get_product( ) { return i*j; }
};
int main( ) {
      samp *p;

      p = new samp(6, 5);  //allocate object
            // with initialisation
      if (!p) {
         cout << "Allocation error\n";
         return 1;
      }
      cout<< "product is: "<< p- >get_product( )<<
"\n";
      delete p;  // release memory
      return 0;
}
```

Example of array allocation

```
// Allocating dynamic objects
#include < iostream >
using namespace std;

class samp {
      int i, j;
   public:
      void set_ij(int a, int b) { i=a; j=b; }
      ~samp( ) { cout << "Destroying...\n"; }
      int get_product( ) { return i*j; }
};

int main( ) {
      samp *p;
      int i;

      p = new samp [10]; //allocate object array
      if (!p) {
         cout << "Allocation error\n";
         return 1;
      }
      for (i=0; i<10; i++)   p[i].set_ij(i, i);
      for (i=0; i<10; i++) {
           cout << "product [" << i << "] is: ";
           cout << p[i].get_product( ) << "\n";
         }
      delete [ ] p;// release memory the destructor
                   // should be called 10 times
      return 0;
}
```


References

C++ contains a feature that is related to pointer: the *reference*. A reference is an implicit pointer that for all intents and purposes acts like another name for a variable. There are three ways that a reference can be used: a reference can be passed to a function; a reference can be return by a function, an independent reference can be created.

The most important use of a reference is as a parameter to a function.
To help you understand what a reference parameter is and how it works, let's first start with a program the uses a pointer (not a reference) as parameter.

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

void f(int *n); // use a pointer parameter

int main( ) {
      int i=0;

      f(&i);
      cout << "Here is i's new value: " << i << "\n";
      return 0;
}

// function definition
void f(int *n) {
      *n = 100;  // put 100 into the argument
                 // pointed to by n
}
```

Here **f( )** loads the value 100 into the integer pointed to by **n.** In this program, **f( )** is called with the address of i in **main( ).** Thus, after **f( )** returns, **i** contains the value 100.

This program demonstrates how pointer is used as a parameter to manually create a call-by-reference parameter-passing mechanism.

In C++, you can completely automate this process by using a reference parameter. To see how, let's rework the previous program,

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

void f(int &n); // declare a reference parameter
```

A copy constructor only applies to initialisation. It does not apply to assignments.
By default, when an initialisation occurs, the compiler will automatically provide a bitwise copy (that is, C++ automatically provides a default copy constructor that simply duplicates the object.) However, it is possible to specify precisely how one object will initialise another by defining a copy constructor. Once defined, the copy constructor is called whenever an object is used to initialise another.

The most common form of copy constructor is shown here:

```
class-name(const class-name &obj) {
     // body of constructor
}
```

Here *obj* is a reference to an object that is being used to initialise another object. For example, assuming a class called **myclass** and that **y** is an object of type **myclass**, the following statements would invoke the **myclass** copy constructor:

```
myclass x = y;     // y explicitly initialising x
func1(y);          // y passed as a parameter
y = func2( );      // y receiving a returned object
```

In the first two cases, a reference to **y** would be passed to the copy constructor. In the third, a reference to the object returned by **func2( )** is passed to the copy constructor.

```
/* This program creates a 'safe' array class. Since
space for the array is dynamically allocated, a copy
constructor is provided to allocate memory when one
array object is used to initialise another
*/
#include < iostream >
#include < cstdlib >
using namespace std;

class array {
     int *p;
     int size;
   public:
     array(int sz) {    // constructor
          p = new int[sz];
          if (!p) exit(1);
          size = sz;
          cout << "Using normal constructor\n";
     }
     ~array( ) { delete [ ] p; }  //destructor

          // copy constructor
     array(const array &a);       //prototype
     void put(int i, int j) {
          if (i>=0 && i<size) p[i] = j;
     }
     int get(int i) {   return p[i]; }
};

// Copy constructor:
// In the following, memory is allocated specifically
// for the copy, and the address of this memory is
// assigned to p.Therefore, p is not pointing to the
// same dynamically allocated memory as the original
// object
array::array(const array &a) {
     int i;

     size = a.size;
     p = new int[a.size]; // allocate memory for copy
     if (!p) exit(1);
                              // copy content
     for(i=0; i<a.size; i++) p[i] = a.p[i];
     cout << "Using copy constructor\n";
}

int main( ) {
     array num(10);  // this call normal constructor
     int i;
     // put some value into the array
     for (i=0; i<10; i++) num.put(i, j);
     // display num
     for (i=9; i>=0; i--) cout << num.get(i);
     cout << "\n";
     // create another array and initialise with num
     array x = num; // this invokes the copy
constructor
     // display x
     for (i=0; i<10; i++) cout << x.get(i);
     return 0;
}
```

When **num** is used to initialise **x** the copy constructor is called, memory for the new array is allocated and store in **x.p** and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that have the same values, but each array is separated and distinct. That is, **num.p** and **x.p** do not point to the same piece of memory.

```
// friend.
#include <iostream>
using namespace std;

class coord {
     int x, y;  // coordinate values
   public:
     coord( ) { x = 0; y = 0; }
     coord(int i, int j) { x = i; y = j; }
     void get_xy(int &i, int &j) { i = x; j = y; }
     friend coord operator++(coord &ob);
};

// Overload ++ using a friend.
coord operator++(coord &ob) {
     ob.x++;
     ob.y++;
     return ob;
}

int main( ) {
     coord o1(10, 10);
     int x, y;

     ++o1;        //o1 is passed by reference
     o1.get_xy(x, y);
     cout << "(++o1) X: " << x << ", Y: " << y << "\n";
     return 0;
}
```


With modern compiler, you can also distinguish between the prefix and the postfix form of the increment or decrement operators when using a friend operator function in much the same way you did when using member functions. For example, here are the prototypes for both versions of the increment operator relative to **coord** class:

```
coord operator++(coord &ob);              // prefix
coord operator++(coord &ob, int notused); // postfix
```

A closer look at the assignment operator

As you have seen, it is possible to overload the assignment operator relative to a class. By default, when the assignment operator is applied to an object, a bitwise copy of the object on the right side is put into the object on the left. If this is what you want there is no reason to provide your own **operator=( )** function. However, there are cases in which a strict bitwise copy is not desirable (e.g. cases in which object allocates memory). In these types of situations, you will want to provide a special assignment operator. Here is another version of **strtype** class that overload the = operator so that the point **p** is not overwritten by the assignment operation.

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

class strtype {
     char *p;
     int len;
   public:
     strtype(char *s);    // constructor
     ~strtype( ) {        // destructor
        cout << "Freeing " << (unsigned) p << "\n";
         delete [ ] p;
     }
     char *get( ) { return p; }
     strtype &operator=(strtype &ob);
};

// Constructor
strtype::strtype(char *s) {
     int l;

     l = strlen(s) + 1;
     p = new char [l];
     if (!p) {
        cout << "Allocation error\n";
        exit(1);
     }
     len = 1;
     strcpy(p, s);
}

// Assign an object
strtype &strtype::operator=(strtype &ob) {
     // see if more memory is needed
     if (len < ob.len) {// need to allocate more memory
        delete [ ] p;
        p = new char [ob.len];
        if (!p) {
           cout << "Allocation error\n";
           exit(1);
        }
     }
     len = ob.len;
     strcpy(p, ob.p);
```

```
        return *this;
}

int main( ) {
        strtype a("Hello"), b("there");

        cout << a.get( ) << "\n";
        cout << b.get( ) << "\n";

        a = b;                  // now p is not
overwritten
        cout << a.get( ) << "\n";
        cout << b.get( ) << "\n";
        return 0;
}
```

Notice two important features about the **operator=( )** function:

- It takes a reference parameter (prevent a copy of the object on the right side from being made).
- It returns a reference, not an object (prevent a temporary object from being created).

Overloading the [ ] *subscript operator*

The last operator that we will overload is the **[ ]** array subscript operator. In C++, the **[ ]** is considered a binary operator for the overloading purposes. The **[ ]** *can be overloaded only by a member function*. Therefore the general form of a member operator**[ ]( )** function is as shown here

```
type class-name::operator[ ](int index)
{
   // body ...
}
```

Technically, the parameter does not have to be of type **int,** but **operator[ ]( )** function is typically used to provide array subscript and as such an integer value is generally used.

To understand how the **[ ]** operator works, assume that an object colled **O** is indexed as shown here:

```
O[9]
```

This index will translate into the following call to the **operator[ ]( )** function:

```
O.operator[ ](9)
```

That is, the value of the expression within the subscript operator is passed to the **operator[ ]( )** function in its explicit parameter. The **this** pointer will point to **O,** the object that generates the call.

In the following program, **arraytype** declares an array of five integers. Its constructor function initialises each member of the array. The overloaded **operator[ ]( )** function returns the value of the element specified by its parameter.

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

const int SIZE = 5;

class arraytype {
      int a[SIZE];
   public:
      arraytype( ) {
         int i;
         for (i=0;i<SIZE; i++) a[i] = i;
      }
      int operator[ ] (int i) { return a[i]; }
};

int main( ) {
      arraytype ob;
      int i;

      for (i=0; i<SIZE; i++) cout << ob[i] << "  ";
      return 0;
}
```

This program displays the following output:

```
0  1  2  3  4
```

It is possible to design the **operator[ ]( )** function in such a way that the **[ ]** can be used on both the left and right sides of an assignment statement. To do this return a reference to the element being indexed,

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

const int SIZE = 5;

class arraytype {
      int a[SIZE];
   public:
```

```
   public:
      int k;
};

// Here Derived3 inherits both Derived1 and Derived2.
// However, only one copy of base is inherited.
class Derived3 : public Derived1, public Derived2 {
   public:
      int product( ) { return i*j*k; }
};

int main( ) {
      Derived3 ob;

      ob.i = 10; // unambiguous because virtual Base
      ob.j = 3;
      ob.k = 5;
      cout << "Product is: " << ob.product( ) << "\n";
      return 0;
}
```


If **Derived1** and **Derived2** had not inherited **Base** as virtual, the statement **ob.i=10** would have been ambiguous and a compile-time error would have resulted.

It is important to understand that when a base class is inherited as virtual by a derived class, that base class still exists within that derived class. For example, assuming the preceding program, this fragment is perfectly valid:

```
Derived1 ob;
ob.i = 100;
```

The only difference between a normal base class and a virtual one occurs when an object inherits the base more than once. If virtual base classes are used, only one base class is present in the object. Otherwise, multiple copies found.

VIRTUAL FUNCTIONS

Pointers to derived class

Although we have discussed pointers at some length, one special aspect relates specifically to virtual functions. This feature is: a pointer declared as a pointer to a base class can also be used to point to any derived from that base. For example, assume two classes called **base** and **derived,** where **derived** inherits **base**.

Given this situation, the following statements are correct:

```
base *p;              // base class pointer

base base_ob;         // object of type base
derived derived_ob;   // object of type derived

// p can, of course, points to base objects
p = &base_ob;         // p points to base object

// p can also points to derived objects without error
p = &derived_ob;          // p points to derived object
```

Although you can use a base pointer to point to a derived object, you can access only those members of the derived object that were inherited from the base. This is because the base pointer has knowledge only of the base class. It knows nothing about the members added by the derived class.

While it is permissible for a base pointer to point to a derived object, the reverse is not true.

One final point: remember that pointer arithmetic is relative to the data type the pointer is declared as pointing to. Thus, if you point a base pointer to a derived object and then increment that pointer, it will not be pointing to the next derived object. It will be pointing to (what it thinks is) the next base object. Be careful about this.

```
// Demonstrate pointer to derived class
#include <iostream>
using namespace std;

class base {
      int x;
   public:
      void setx(int i) { x = i; }
      int getx( ) { return x; }
};

class derived : public base {
      int y;
   public:
      void sety(int i) { y = i; }
      int gety( ) { return y; }
};

int main( ) {
      base *p;        // pointer to base type
      base b_ob;      // object of base
```

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

class area {
      double dim1, dim2;   // dimensions of figure
public:
      void setarea(double d1, double d2) {
            dim1 = d1;
            dim2 = d2;
      }
      void getdim(double &d1, double &d2) {
            d1 = dim1;
            d2 = dim2;
      }
      virtual double getarea( ) = 0;
                        // pure virtual function
};
class rectangle : public area {
public:
      double getarea( ) {
            double d1, d2;
            getdim(d1, d2);
            return d1*d2,
      }
};

class triangle : public area {
public:
      double getarea( ) {
            double d1, d2;
            getdim(d1, d2);
            return 0.5*d1*d2;
      }
};

int main( ) {
      area *p;
      rectangle r;
      triangle t;

      r.setarea(3.3, 4.5);
      t.setarea(4.0, 5.0);

      p = &r;
      cout << "Rectangle area: "<< p- >getarea( )
<<"\n";

      p = &t;
      cout << "Triangle area: "<< t- >getarea( ) <<
"\n";
      return 0;
}
```

Now that **getarea( )** is pure, it ensures that each derived class will override it.

The following program illustrates how the virtual nature of a function is preserved when it is inherited:

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

class base {
public:
      virtual void func( ) {
            cout << "Using base version of func()\n";
      }
}

class derived1 : public base {
public:
      void func( ) {
         cout << "Using derived1's version of func()\n";
      }
};

// derived2 inherits derived1
class derived2 : public derived1 {
public:
      void func( ) {
         cout << "Using derived2's version of func()\n";
      }

};

int main( ) {
      base *p;
      base ob;
      derived1 d_ob1;
      derived2 d_ob2;

      p = &ob;
      p- >func( );    // use base's func( )

      p = &d_ob1;
      p- >func( );    // use derived1's func( )

      p = &d_ob2;
      p- >func( );    // use derived2's func( )
      return 0;
}
```

badbit	A fatal error has occurred

For older compilers, the I/O status flags are held in an **int** rather than an object of type **iostate**.

There are two ways in which you can obtain the I/O status information. First, you call the **rdstate( )** function, which is a member of **ios**. It has this prototype:

```
iostate rdstate( );
```

It returns the current status of the error flags. **rdstate( )** returns **goodbit** when no error has occurred. Otherwise, an error flag is returned.

The other way you can determine whether an error has occurred is by using one of these **ios** member functions:

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

The **eof( )** function was discussed earlier. The **bad( )** function returns true if **badbit** is set. The **fail( )** function returns true if **failbit** is set. The **good( )** function returns true if there are no errors. Otherwise, they return false.

Once an error has occurred, it might need to be cleared before your program continues. To do this, use the **ios** member function **clear( )**, whose prototype is

```
void clear(iostate flags = ios::goodbit);
```

If *flags* is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set *flags* to the settings you desire.

Customised I/O and files

As stated in the previously, overloaded inserters and extractors, as well as I/O manipulators, can be used with any stream as long as they are written in a general manner. If you 'hard-code' a specific stream into an I/O function, its use is, of course, limited to only that stream. This is why you were recommended to generalised your I/O functions whenever possible.

In the following program, the **coord** class overloads the << and >> operators. Notice you can use the operator functions to write both to the screen and to file.


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

class coord {
     int x, y;
public:
     coord(int i, int j) { x=i; y=j; }
     friend ostream &operator<<(ostream &stream,
                                          coord ob);
     friend istream &operator>>(istream &stream,
                                          coord &ob);
};
ostream &operator<<(ostream &stream, coord ob) {
     stream << ob.x << " " << ob.y << "\n";
     return stream;
}
istream &operator>>(istream &stream, coord &ob) {
     stream >> ob.x >> ob.y;
     return stream;
}

int main( ) {
     coord o1(1, 2) o2(3, 4);
     ofstream out( "test" );

     if( !out ) {
          cout << "Cannot open file\n";
          return 1;
     }
     cout << o1 << o2;
     out.close( );

     ifstream in( "test" );
     if ( !in ) {
          cout << "Cannot open file\n";
          return 1;
     }

     coord o3(0, 0), o4(0, 0);
     in >> o3 >> o4;

     cout << o3 << o4;

     in.close( );
     return 0;
}
```

Remember that all the I/O manipulators can be used with files.

TEMPLATES AND EXCEPTION HANDLING

Two of C++ most important high-level features are the *templates* and *exception handling*. They are supported by all modern compilers.

Using templates, it is possible to create generic functions and classes. In generic functions or classes, the type of data that is operated upon is specified as a parameter. This allows you to use one function or class with several different types of data without having to explicitly recode a specific version for each type of data type. Thus, templates allow you to create reusable code.

Exception handling is a subsystem of C++ that allows you to handle errors that occur at run time in a structured and controlled manner. With C++ exception handling, your program can automatically invoke an error handling routine when an error occurs. The principle advantage of exception handling is that it automates much of the error handling code that previously had to be coded 'by hand' in any large program. The proper use of exception handling helps you to create resilient code.

Generic functions

A generic function defines a general set of operations that will be applied to various types of data. A generic function has the type of data that it will operate upon passed to it as a parameter. Using this mechanism, the same general procedure can be applied to a wide range of data. For example the Quicksort algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of data being sorted is different. By creating a generic function, you can define, independent of any data, the nature of the algorithm. Once this is done, the compiler automatically generates the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

A generic function is created by using the keyword **template**. The general form of a template function definition is as

```
template <class Ttype> ret-type-name(parameter list)
{
   // body of function
}
```

Here *Ttype* is a placeholder name for a data type used by the function. This name can be used within the function definition. However, it is only a placeholder; the compiler will automatically replace this placeholder with an actual data type when it creates a specific version of the function.
Although the use of the keyword **class** to specify a generic type in a template declaration is traditional, you can also use the keyword **typename**.

The following example creates a generic function the swaps the values of the two variables it is called with.

```
// Function template example
#include <iostream>
using namespace std;

// This is a function template
template <class X> void swapargs(X &a, X &b) {
      X temp;

      temp = a;
      a = b;
      b = temp;
}

int main( ) {
   int i=10, j=20;
   float x=10.1, y=23.3;

      cout << "Original i, j: " << i << j <<endl;
      cout << "Original x, y: " << x << y <<endl;

swapargs(i, j);    // swap integers
swapargs(x, y);    // swap floats

      cout << "Swapped i, j: " << i << j <<endl;
      cout << "Swapped x, y: " << x << y <<endl;

      return 0;
}
```

The keyword **template** is used to define a generic function. The line

```
template <class X> void swapargs(X &a, X &b)
```

tells the compiler two things: that a template is being creates and that a generic function is beginning.. Here **X** is a generic type that is used as a placeholder. After the **template** the template portion, the function **swapargs( )** is declared, using **X** as a data type of the values that will be swapped. In **main( ),** the

associated with a **try**. The **catch** statement that is used is determined by the type of the exception. That is, if the data type specified by a **catch,** matches that of the exception, that **catch** statement is executed (all other are bypassed). When an exception is caught, *arg* will receive its value. If you don't need access to the exception itself, specify only *type* in the **catch** clause (*arg* is optional). Any type of data can be caught, including classes that you create. In fact, class types are frequently used as exceptions.

The general form of a **throw** statement is

```
throw exception;
```

throw must be executed either from within the **try** block or from any function that the code within the block calls (directly or indirectly). *exception* is the value thrown.

If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination might occur. If your compiler complies with Standard C++, throwing an unhandled exception causes the standard library function **terminate( )** to be invoked. By default, **terminate( )** calls **abort( )** to stop your program, but you can specify your own termination handler, if you like. You will need to refer to your compiler's library reference for details.

```
// A simple exception handling example
#include <iostream>
using namespace std;

int main( ) {
     cout << "Start\n";

     try {   // start a try block
          cout << "Inside try block\n";
          throw 10;          // throw an error
          cout << "This will not execute\n";
     }

     catch( int i) {    // catch an error
          cout << "Caught One! Number is: ";
          cout << i << "\n";
     }

     cout << "end";
     return 0;
}
```

This program displays the following:

```
start
Inside try block
Caught One! Number is: 10
end
```

As you can see, once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is **catch** is not called. Rather, program execution is transferred to it. (The stack is automatically reset as needed to accomplish this) Thus, the **cout** statement following the **throw** will never execute.

After the **catch** statement executes, program control continues with the statements following the **catch.** Often, however, a **catch** block will end with a call to **exit( )** or **abort( )**, or some other function that causes program termination because exception handling is frequently used to handle catastrophic errors.
Remember that the type of the exception must match the type specified in a **catch** statement.

An exception can be thrown from a statement that is outside the **try** block as long as the statement is within a function that is called from within the **try** block.

```
// Throwing an exception from a function outside
// the try block
#include <iostream>
using namespace std;

void Xtest(int test) {
     cout << "Inside Xtest, test is: " << test << \n";
     if (test) throw test;
}

int main( ) {
     cout << "start\n";

     try {     // start a try block
          cout << "Inside try block\n";
          Xtest(0);
          Xtest(1);
          Xtest(2);
     }

     catch (int i) {    // catch an error
          cout << "Caught one! Number is: ";
          cout << i << "\n";
     }
```