Client code can "catch" a **throw**n exception:

```
void crash( Array& x )
{
   x[ x.getSize()+2 ];  // a deliberate range error
}
void f( Array& v )
{
   // ...
   try
   {
      Array u;
      ... g(v) ... crash(u) ... h(v) ...
   }
catch( Array::Range )
   {
      // This block catches objects of type Range that
      // are thrown while preceding try block is active.
      // That is, it "handles" Array::Range
      // exceptions.
      // If Array::operator[] is called with
      // an invalid index while the try block is active,
      // control passes directly to this block.
      // Otherwise this block is skipped.
   }
}
```


When an exception is thrown,

- the call stack is unwound to the activation record of the catching function
- intervening functions are forcibly shut down
- intervening functions' local variables are deallocated —and their destructors, if any, are called.

The **first valid handler** encountered by an exception catches it and ends the process.

Hence

```
void g( )
{
   try
   {
      Array z;
      f( z );      // catches Array::Range
   }
   catch ( Array::Range )
   {  /* can never be executed */  }
}
```

If a program throws an exception for which **no handler** is waiting, the **program terminates**.

Different kinds of errors can throw different exceptions:

```
class Array
{
public:
   class Range { };  // range exception
   class Size { };   // size exception
   Array( int s = ArraySize );
   // ...
   int& operator[]( int );
   int getSize();
protected:
   int *data;
   int size;
};
inline int& Array::operator[]( int index )
{
   if( 0 <= index && index < size )
       return data[ index ];
   throw Range();  // constructs and throws
}
Array::Array( int s )
{
   if( s < 0 || 32767 < s ) throw Size();
   // ...
}
```


Different exceptions can be handled differently:

```
void f( Array& v )
{
   // ...

   try
   {
      Array u;
      ... g(v) ... crash(u) ...
   }

   catch( Array::Range )
   {  // This block handles Array::Range
      // exceptions.
   }

   catch( Array::Size )
   {  // This block handles Array::Size
      // exceptions.
   }
}
```

An exception is an object, and can carry information from throw to catch:

```
class Array
{
public:
   class Range   // exception class
   { public:
       int index;
       Range( int i ){ index = i; }
   };
   Array( int sz = ArraySize );
   // ...
   int& operator[]( int );
   int getSize();
protected:
   int *data;
   int size;
};

inline int& Array::operator[]( int index )
{
   if( 0<=index && index<size )
       return data[ index ];
   throw Range( index );
}
```


Extracting an exception object's information:

```
void f( Array& v )
{
   // ...
   try
   {
      Array u;
      ... g(v) ... crash(u) ...
   }
   catch( Array::Range r )
   {
     cerr << "bad index " << r.index << '\n';
      // ...
   }
}
```

"Catch-all" exception handlers are also allowed:

```
void m()
{
   try
   {
      // possible throw in here
   }
   catch( ... )  // catches all exceptions
   {
      // do some tidying up…
      throw;      // …and pass the buck
   }
}
```

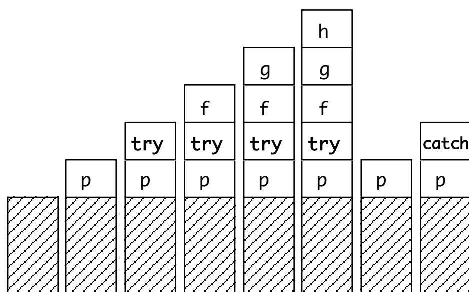

When an exception is thrown, the stack "unwinds":

```
void h()
{ ...
  throw bomb();
  ...
}
```

```
void g()
{ ...
  h();
  ...
}
```

```
void f()
{ ...
  g();
  ...
}
```

```
void p()
{ ...
  try
  {   ... f(); ... }
  catch( bomb b )
  {
      // handle bomb exceptions
  }
}
```



☞ As each block's activation record is discarded, all of its local variables' destructors are called.

How are exception handlers used?

A typical example (composite):

```
int f( int arg )
{
   int success = 0;
   int result;
   while( !success )
   {
      try
      {
         result = g( arg );
         success = 1;
      }
      catch( x0 )
      {  // fix something and try again }
      catch( x1 )
      {  // use another method and return a result
         return 1;
      }
      catch( x2 )
      {  throw;  // pass the bug
      }
      catch( x3 )
      {  // transform x3 into some other exception
         throw xxii;
      }
      catch( ... )
      {  // all other exceptions: give up
         terminate();
      }
   }
   return result;
}
```