Throwing an Exception

throw indicates that an exception has occurred…

- Usually has one operand of any type…
  - The operand can be an object… ***exception object***
  - Conditional expression can be thrown…
    throw *condition* **?** *expression_1* **:** *expression_2*
- Code referenced in a **try** block can throw an exception…
- Exception caught by ***closest*** exception handler…
- Control exits current try block and goes to catch handler (if it exists)

Exception does not terminate the program…
- However, it terminates the block where the exception occurred…

Catching an Exception

Exception handlers are defined in **catch** blocks…
- Format:

```
catch( exceptionType parameterName){
    Exception handling code...
}
```

- Caught if argument type matches **throw** type…
- If not caught then ***terminate*** called which (by default) calls ***abort***…

Catch all exceptions… **catch(...**[JAN1]**)**

```
...
  try {
     // Code to be tried...
     throw exception_type;
  }
  catch (type  parameter_name) {
     // Code to be executed in case of exception 1...
  }
  catch (...) {
     // Code to be executed for all exception ...
  }
...
```

- You do not know what type of exception occurred…
- There is no parameter name, thus cannot reference an object…
- Not to be placed in front of other **catch** blocks… None of the other blocks will ever be executed…

If no handler matches thrown object
- Searches next enclosing **try** block…
  - If none found, ***terminate*** called…

```
...
try {
    if ( number1 == 0 )
       throw 0 ;
    try {
       if ( number1 < 0 )
          throw -1 ;
    }
    catch ( char * str ) {
       cout << "Exception: " << str << endl;
    }
}
catch ( int i) {
    cout << " Exception: " << i << endl;
}
...
```

  - If found, control resumes after last **catch** block…
  - If several handlers match thrown object, first one found is executed…

catch parameter matches thrown object when:
- They are of the same type… Exact match required, ***no promotions/conversions allowed***…

```
...
try {
 if ( number1 == 0 )
   throw 1.0[JAN2] ;
}
catch ( int i ) {
    cout << " Exception: " << i << endl;
}
...
```

- The **catch** parameter is a class of the thrown object…

```
class ABC {
 public:
   ABC() : baseMsg ("Class Exception...") {}
   const char *who() const {return baseMsg;}
 private:
   const char *baseMsg;
};
```

```
...
try {
   if ( number1 == 0 )
     throw ABC();
}
catch ( ABC obj1 ) {
   cout << "Exception occurred: "
        << obj1.who() << endl;
}
...
```


- The **catch** parameter is a base class of the thrown object…

```
class ABase {
 public:
   ABase() : baseMsg ("Base Exception...") {}
   const char *who() const {return baseMsg;}
 private:
   const char *baseMsg;
};

class ADerived : public Abase {
 public:
   ADerived() : drvMsg ("Derived Exception...") {}
   const char *who() const {return drvMsg;}
 private:
   const char *drvMsg;
};
```

```
...
ADerived dObj;
...
try {
    if ( number1 == 0 )
       throw ADerived();
    if ( number1 == 1 )
       throw dObj;
}
catch ( ABase[JAN3] bObj) {
    cout << "Exception occurred: "
          << bObj.who() << endl;
}
...
```


- The **catch** parameter is a base-class pointer/ reference type and the thrown object is a derived-class pointer/ reference type…

```
...
ADerived *dPtr=&dObj, &dRef=dObj;
...
try {
   if ( number1 == 1 )
     throw dPtr;
   if ( number1 == 2 )
     throw dRef;
}
catch ( ABase *bPtr) {
   cout << "Exception 1 occurred: "
        << bPtr->who() << endl;
}
catch ( ABase &bRef) {
   cout << "Exception 2 occurred: "
        << bRef.who() << endl;
}
...
```

- The **catch** handler is **catch(...)**

```
...
try {
   if ( number1 == 0 )
     throw 1.0 ;
}
catch ( ... ) {
     cout << " Exception occurred...\n";
}
...
```

catch handler should delete space allocated by **new** and **close** any opened files…

- Resources may have been allocated when exception thrown…

catch handlers can **throw** exceptions…

- It can only be processed by outer **try** blocks…

```
...
try {
   if ( number1 == 0 )     throw 0  ;
   try {
     if ( number1 < 0 )   throw -1 ;
   }
   catch ( int i) {
        cout << "Exception 1: " << i << endl;
        throw "Inner exception...";
   }
   catch ( char * str ) {
        cout << "Exception 2: " << str << endl;
   }
}
catch ( int i) {
     cout << "Exception 3: " << i << endl;
}
catch ( char * str ) {
     cout << "Exception 4: " << str << endl;
}
...
```


Rethrowing an Exception

It is possible that the handler that caches the exception may decide that it cannot process it…

Or it may simply want to release some resources before letting someone else handle the exception…

The handler can simply rethrow the exception with
  Throw ;

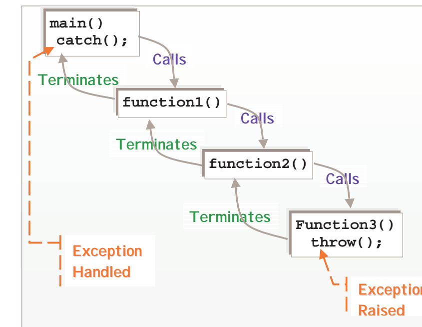
Rethrown exception detected by next enclosing try block…

```
...
try {
   if ( number1 == 0 )    throw 0 ;
   try {
     if ( number1 < 0 )   throw -1 ;
   }
   catch ( int i) {
        cout << "Exception 1: " << i << endl;
        throw;[JAN4]
   }
   catch ( char * str ) {
        cout << "Exception 2: " << str << endl;
   }
}
catch ( char * str ) {
     cout << "Exception 4: " << str << endl;
}
catch[JAN5] ( int i) {
     cout << "Exception 3: " << i << endl;
}
...[JAN6]
```


Stack unwinding

Function-call stack unwound when exception thrown and not caught in a particular scope…

- Tries to catch exception in next *outer* **try/catch** block…
- Function in which exception was not caught terminates…
  - Local variables destroyed…
  - Control returns to place where function was called…
- If control returns to a **try** block, attempt made to catch exception…
  - Otherwise, further unwinds stack…
- If exception not caught, terminate called…




```
#include <iostream.h>

void function3(int m) {
   if (m < 0)
     throw "Exception thrown in function 3";
   cout << "Function Three\n";
}

void function2(int k) {
   function3(k);
   cout << "Function Two\n";
}

void function1(int i) {
   function2(i);
   cout << "Function One\n";
}

void main() {
   try {
     function1(10);
     cout << "------------------" << endl;

     function1(-10);
   }
   catch ( char * str) {
        cout << "Exception: " << str << endl;
   }
}
```

```
Function Three
Function Two
Function One
------------------
Exception: Exception thrown in function 3
```


Constructors and Exception Handling

Constructors don't have a return type, thus it's not possible to use return codes…

So what happens when an error is detected in the constructor?

➔ Partially constructed object or Zombie object ➔

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

class XYZ {
 public:
   XYZ(int i=10) {iVal=i;}

 private:
   int iVal;
};

class ABC {
 public:
   ABC(char *str) {[JAN7]
     msg = new char[strlen(str) +1];
     if (msg != 0)
        strcpy(msg,str);
   }
   void sayIt() { cout << msg << endl; }

 private:
   XYZ x1[J8];
   char *msg;
};

void main() {
   ABC abc("I Love C++");
   abc.sayIt();
}
```


Put the object into a "***deffect***" state by setting an internal status…

```
class XYZ {
 public:
   XYZ(int i=10) {iVal=i;}

 private:
   int iVal;
};

class ABC {
 public:
   ABC(char *str) {
     status = 0;         // Fully constructed.
     msg = new char[strlen(str) +1];
     if (msg != 0)
        strcpy(msg,str);
     else
        status = -1; // Partially constructed.
   }

   int isZombie() {return status;}

 private:
   XYZ x1;
   char *msg;
   int status;
};

void main() {
   ABC abc("Hello World");

   if (abc.isZombie())
     cout << "Failed to construct\n";
   ...
   ...
}
```

Throw an exception from the constructor passing some information… ***Destructors are called for automatic objects***…

```
class XYZ {
 public:
  XYZ(int i=10) {iVal=i;}
  ~XYZ[JAN9]() {
    cout << "XYZ destructor\n";
  }
 private:
  int iVal;
};
class ABC {
 public:
  ABC(char *str) {
    msg = new char[strlen(str) +1];
    if (msg == 0)
       throw[JAN10] "No Sapce Left\n";
    strcpy(msg,str);
  }
 private:
  XYZ x1;
  char *msg;
};
void main() {
  try {
    ABC abc("Hello World");
    ...
  }
  catch (char *exMsg) {
    cout << exMsg << endl;
  }
}
```



```
class XYZ {
 public:
  XYZ(int i=10) {iVal=i;}
  ~XYZ() {
    cout << "XYZ destructor\n";
  }
 private:
  int iVal;
};
class ABC {
 public:
  ABC(char *str) {
    msg = new char[strlen(str) +1];
    if (msg == 0)
       throw this;
    strcpy(msg,str);

  }
  void what() {
    cout << "Error: No space left..." << endl;
  }
 private:
  XYZ x1;
  char *msg;
};

void main() {
  try {
    ABC abc("Hello World");
    ...
  }
  catch (ABC *obj) {
    obj->what();
  }
}
```


Destructors and Exception Handling

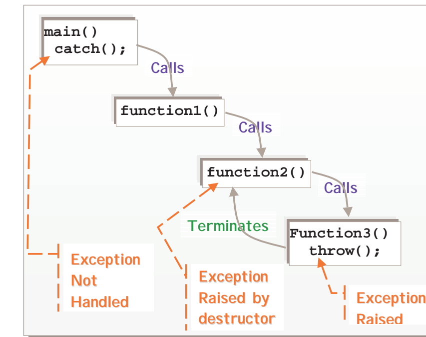
The C++ rule is that you must be careful throwing an exception from a destructor …

During stack unwinding, all the local objects are destructed...

If one of *those* destructors throws an exception, it confuses the C++ runtime system…

- Should it ignore the "last thrown exception" and continue where it was originally headed? Or vice versa…

So the C++ language guarantees that it will call terminate() at this point, and terminate() kills the process.




```
class XYZ {
 public:
  XYZ(int i=10) {iVal=i;}
  ~XYZ() {
    cout << "Destructor: " << iVal << endl;
    if (iVal == 20)
       throw "XYZ Destructor Exception\n";
  }
 private:
  int iVal;
};
void function3() {
  XYZ x3(30);
  throw "Exception thrown in function 3";
}
void function2() {
  XYZ x2(20);
  function3();
}
void function1() {
  XYZ x1(10);
  function2();
}
void main() {
  try {
    function1();
  }
  catch ( char * str) {
       cout << "Exception: " << str << endl;
  }
}
```