G52CPP
C++ Programming
Lecture 16

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html

Last Lecture

- Casting
  - static cast
  - dynamic cast
  - const cast
  - reinterpret cast
- Implicit type conversion

How do we report errors?

1. Return an error value from function
   - Remember to check return value on each call
   - Must have a valid ‘error’ return value
   - How do we propagate the error? (return again?)

2. Set a global error code
   - Again, have to remember to check it after each call

3. Throw an exception (to report error)
   - Requires an exception handling mechanism
   - This lecture

Exceptions

- Exceptions are '**`throw`**n' to report exceptional circumstances
  - Similar to being able to return an error value of whatever type is desired
- You can **`throw`** any type of **object**, **fundamental type** or **pointer** as an exception in C++
  - These are different things (pointers != objects)
  - The standard class library provides some **standard exception types**, all derived from **`exception`** class
  - It is **good practice** to **`throw`** objects which are of sub-classes of **`exception`** rather than arbitrary types
- You add handler code to **`catch`** the exception
- The stack frame is unwound, **one function at a time** (as if the functions **`return`**ed immediately) **until** a **`catch`** which matches the type thrown is found
  - Like returning from the functions
  - Same problems/effects as returning from a function

Catching exceptions

- First specify that you want to check for exceptions (**`try`**)
- Then call the code which will may raise the exceptions
- Then specify which exceptions you will **`catch`**, and what to do (this can include re-**`throw`**ing them)
  - Use **`throw`** without arguments in a catch clause to rethrow them

```
try
{
   foo();
}
catch ( int& i )
{
   cout << "int was thrown by foo()" << endl;
}
catch ( … )
{
   cout << "Any other exception was thrown" << endl;
}
```

Assume that **`foo()`** throws an exception
e.g. **`throw 1;`**
or **`MyException ob; throw ob;`**

The `catch` clause

- A `catch` clause will match an exception **of the specified type**
- `catch` clauses are **checked in the order in which they are encountered**
  - The order of the catch clauses matters!
- Pointers and objects are different
- **Exceptions are thrown by value**
  - **Catch by reference or by value would work**
  - **Catch by reference avoids the copy**
- `char`s, `short`s, `int`s (etc) are different things
- `catch ( ... )` will match ANY exception

Exception `thrown` by `new`

```
void foo()
{
   while ( true )
   {
       new int[10000];
       cout << '.';
   }
}

int main()
{
   try
   { foo(); }
   catch ( bad_alloc )
   { cout << "bad_alloc exception thrown" << endl; }
   catch ( ... )
   { cout << “Other exception thrown " << endl; }
}
```

Loop forever – until it fails

If memory allocation fails, an exception of type `bad_alloc` is thrown

Catching this exception potentially allows handling the out of memory problem, e.g. ‘save and exit’

Multiple functions

```
#include <iostream>
using namespace std;
void bar2()
{
   switch( rand() % 3 )
   {
   case 0: throw 1.2f;
   case 1: throw "string";
   case 2: throw new
   string("String");
   }
}

void bar()
{
  try { bar2(); }
  catch( float f )
  {
    cout << "float" << endl;
  }
}
```

bar2() throws an exception of a random type

```
void foo()
{
   try { bar(); }
   catch( const char* sz )
   {
        cout << "char*" << endl;
   }
}

int main()
{
   for ( int i=0 ; i<20 ; i++ )
   {
        try { foo(); }
        catch( ... )
        {
        cout << "Other" << endl;
        }
   }
}
```

The `catch` clause and sub-classes

- Sub-class objects ARE base class objects
  - Because inheritance models the ‘is-a’ relationship
  - `catch` clauses will match sub-class objects
  - e.g.:

```
catch ( BaseClass& b ) { }
```

will also catch sub-class objects

```
catch ( BaseClass* b ) { }
```

will also catch sub-class pointers

- Reminder: Pointers and objects are different
  - First catch will NOT catch thrown pointers
  - Second catch will NOT catch thrown objects

Catching **Base** class objects

```
struct Base
{
   virtual void temp() {}
};

struct Sub1 : public Base
{
   void temp() {}
};

struct Sub2 : public Base
{
   void temp() {}
};
```

Base class object is thrown

Which catch clause will be used?

```
int test1()
{
   try
   {
        Base b;
        throw b;
   }

A  catch ( Sub1& b )
   { cout << "Sub1" << endl; }

B  catch ( Base& b )
   { cout << “Base" << endl; }

C  catch ( Sub2& b )
   { cout << "Sub2" << endl; }

D  catch ( ... )
   { cout << "Other" << endl; }
}
```

Answer

- B

- Check catches in order:
  - It is NOT a Sub1
  - It is a Base

Catching **Sub1** class objects

```
struct Base
{
   virtual void temp() {}
};

struct Sub1 : public Base
{
   void temp() {}
};

struct Sub2 : public Base
{
   void temp() {}
};
```

Sub-class Sub1 object is thrown

Which catch clause will be used?

```
int test1()
{
   try
   {
        Sub1 s1;
        throw s1;
   }
```

A
```
catch ( Sub1& b )
{ cout << "Sub1" << endl; }
```

B
```
catch ( Base& b )
{ cout << “Base" << endl; }
```

C
```
catch ( Sub2& b )
{ cout << "Sub2" << endl; }
```

D
```
catch ( ... )
{ cout << "Other" << endl; }
}
```

Answer

- A
- Check catches in order:
  - It is a Sub1

Catching **Sub2** class objects

```
struct Base
{
   virtual void temp() {}
};

struct Sub1 : public Base
{
   void temp() {}
};

struct Sub2 : public Base
{
   void temp() {}
};
```

Sub-class Sub2 object is thrown

Which catch clause will be used?

```
int test1()
{
   try
   {
        Sub2 s2;
        throw s2;
   }

A  catch ( Sub1& b )
   { cout << "Sub1" << endl; }

B  catch ( Base& b )
   { cout << “Base" << endl; }

C  catch ( Sub2& b )
   { cout << "Sub2" << endl; }

D  catch ( ... )
   { cout << "Other" << endl; }
}
```

Answer

- B
- Check catches in order:
  - It is not a Sub1
  - It **is** a Base (Sub2 objects ***are*** Base objects)
- Note: The order here matters
  - It gets caught by the Base catch before it gets to the Sub2 catch
  - The compiler may give you a warning here about the sub-class type exception being caught by the base class catch
  - gcc / g++ will

Catching **Sub2** class objects

```
struct Base
{
   virtual void temp() {}
};

struct Sub1 : public Base
{
   void temp() {}
};

struct Sub2 : public Base
{
   void temp() {}
};
```

Sub-class Sub2 object is thrown

Which catch clause will be used?

```
int test1()
{
   try
   {
        Sub2* ps2 = new Sub2;
        throw ps2;
   }

A  catch ( Sub1& b )
   { cout << "Sub1" << endl; }

B  catch ( Base& b )
   { cout << “Base" << endl; }

C  catch ( Sub2& b )
   { cout << "Sub2" << endl; }

D  catch ( ... )
   { cout << "Other" << endl; }
}
```

Answer

- D
- Check catches in order:
  - It is not a Sub1
  - It is not a Base
  - It is not a Sub2
  - … catches all exceptions
- Pointers are not objects
- Objects are not pointers
- Note: References and objects will match
  - **&** Just says whether a copy is made or not

Aside: exceptions and `throw()`

- `throw()` at the end of the function declaration limits the exceptions which can be thrown
  - It is **optional**
  - In Java, `throws <types>` is mandatory
- **If** specified, then all exception types which can be thrown by function must be specified
  - Throwing a different type will terminate the program
- Examples:

```
void MyFunction(int i) throw();
```

  - Function will not throw exceptions

```
void MyFunction(int i) throw(int);
```

  - Function will only throw `int`s as exceptions

```
void MyFunction(int i) throw(...);
```

  - Function could throw ANY exception

Why does C++ not need ‘finally’?

What is wrong with this function?

```
void foo()
{
  int* iarray = new int[100];
  for (int i=0;i<100;i++)
  {
    iarray[i] = rand();

    if ( (iarray[i]%5) == 0 )
    {
      cout << "end " << i;
      return;
    }

    cout << iarray[i] << " ";
  }
  delete [] iarray;
}
```

Allocate memory

Set each element to a random value

End function if random number gives specific values

Free memory

Note: This example has nothing directly to do with exceptions/exception handling, yet

Prematurely ending functions

```
#include <iostream>

using namespace std;

struct MyClass
{
   MyClass()
        { cout << "C"; }
   ~MyClass()
        { cout << "D"; }
};

void bar()
{
   throw 1;
}
```

Throwing an uncaught exception will terminate the function, as if return was used. Objects on the stack will be destroyed correctly.

```
void foo()
{
   MyClass* pOb = new MyClass;
   bar();
   delete pOb;
}

int main()
{
   try
   {
        foo();
   }
   catch( ... )
   {
        cout << "E";
   }
   cout << endl;
}
```

bar() throwing an exception will mean delete is not called for **pOb**

The problem

```
// Function which may throw an exception
void bar()
{
   throw 1;
}

// This function throws an exception so the
// objects are not destroyed
void foo()
{
   // Create objects
   MyClass* pOb1 = new MyClass;

   // Call function which may throw an exception
   bar();

   delete pOb1;
}
```

Note: no '**throws**'/ '**throw**' on the function
If you add a **throw()** on a function then you are guaranteeing that it **ONLY** throws those exceptions (throw any others and program ends)

Function ends before here
The **delete** never gets called
Objects not deleted
Memory not freed

Code to open and use a file

```
void version1()
{
   FILE* f = fopen( "out1.txt", "w" );
   fprintf( f, "Output text" );

   // Do something which throws exception or returns?
   throw 1;

   // Never gets to the close, so file possibly
   // not flushed until process ends
   printf("Closing file manually\n" );
   fclose(f);
}
```

In Java we may put a ‘finally’ clause in for the close, to ensure that the code to close the file is called, regardless of how the function exits. This is more tricky in C++ than Java because we don’t know what will throw an exception

RAII : Resource Acquisition Is Initialisation

A useful concept to understand

When a function ends…

- Remember back to the discussion of the stack…
  - When a function ends, its stack frame is removed
  - ALL stack objects (local variables) are destroyed
    - **Destructors are called for each**
  - This applies even if the function is ended due to an exception!
- RAII takes advantage of this
  - My opinion (only): may be better named in this case: "Resource Release On Object Destruction" (RROOD?)
- **On initialisation, get the resource**
- **On destruction, release the resource**
- Example/summary:
  - Create stack object to 'wrap' the thing you need to release
  - When stack object is destroyed, the thing gets released (e.g. file closed)
- **Note:** Java has no stack objects and no proper destructors
  - only has: "protected void finalize()" : "*Before reclaiming the memory occupied by an object that has a finalizer, the garbage collector will invoke that object's finalizer.*"

Simplest(?) file ‘wrapper’ class

```
class Wrapper
{
public:
   FILE* pFile;

   // No constructor -
   default created

   // Key part is the
   destructor!
   ~Wrapper()
   {
       fclose(pFile);
   }
};
```

```
void version2a()
{
   Wrapper w;
   w.pFile = fopen(
       "out2a.txt", "w" );

   fprintf( w.pFile,
       "Output text" );

   // Do something which
   // e.g. throw exception
   throw 1;

   // Never gets to close
   // but we don't care
   //fclose(w.pFile);
}
```

A better wrapper class

```
class MyFile
{
   FILE* pFile;

public:
   // Constructor
   MyFile(
   const char* szFileName,
   const char* szType = "r" )
       : pFile(NULL)
   {
      pFile = fopen(
         szFileName, szType );
   }

   // Conversion operator
   operator FILE*()
       { return pFile; }

   // Is file open?
   bool isopen()
   { return pFile != NULL; }

   // Close file if open
   void close()
   {
      if ( pFile != NULL )
         fclose( pFile );
      pFile = NULL;
   }

   // Destructor!!!
   ~MyFile()
   {
       close();
   }
};
```

Using the wrapper

```
void version2b()
{
   // FILE* f=fopen("out2.txt","w");
   MyFile file( "out2.txt", "w" );

   fprintf( file, "Output text" );

   // Do something which throws
   // exception or returns?
   throw 1;

   // Never gets to the close below
   // but we don't care
   file.close();
}
```

```
class MyFile
{
public:
   MyFile( ... )
   {
       … fopen(…);
   }

   operator FILE*()
   { return pFile; }

   void close()
   { … }

   ~MyFile()
   { close(); }
};
```

Wrapping pointers

Wrapping pointers : `int*`

```
class Deleter
{
public:
  int* pOb; // wrapped ptr

  // construct from pointer
  Deleter(int *pOb = NULL)
  : pOb(pOb)
  { }

  // destroy the object
  ~Deleter()
  {
    if ( pOb )
      delete pOb;
  }
};
```

```
class ArrayDeleter
{
public:
  int* pArray;

  // construct from pointer
  ArrayDeleter(
       int* pArray = NULL)
  : pArray(pArray)
  { }

  // destroy the array
  ~ArrayDeleter()
  {
    if ( pArray )
      delete [] pArray;
  }
};
```

Wrapping pointers : templates

```
template<class T>
class Deleter
{
public:
  T* pOb; // wrapped pointer

  // construct from pointer
  Deleter(T *pOb = NULL)
  : pOb(pOb)
  { }

  // destroy the object
  ~Deleter()
  {
    if ( pOb )
      delete pOb;
  }
};
```

```
template<class T>
class ArrayDeleter
{
public:
  T* pArray;

  // construct from pointer
  ArrayDeleter(
       T* pArray = NULL)
  : pArray(pArray)
  { }

  // destroy the array
  ~ArrayDeleter()
  {
    if ( pArray )
      delete [] pArray;
  }
};
```

Summary

Other Exception Comments

- The destructor is guaranteed to be called for a stack object when the stack frame is destroyed
  - It is the **only** function which we can guarantee will be called when an exception occurs

1. Throwing an exception while there is an uncaught exception will end the program
   - Ensure that exceptions cannot be `thrown` from within a destructor because the destructor could be called as a result of an exception, e.g. to destroy objects on the stack
2. Not `catch`ing a `throw`n exception will end the program

The problem of pointers

- Throwing an exception is similar to a **`return`**
  - Except that you get the value in a different way
    - And it will keep ‘returning’ from functions until caught
- When exceptions are **`throw`**n:
  - Objects on the stack are destroyed (destructor called)
  - Memory allocated dynamically will **not** be freed
  - You need to **either create objects on the stack**, or **free them yourself** – in **every** return and **whenever** an exception is **`throw`**n
    - e.g. **`catch`** exception, **`delete`** object, re-**`throw`**
- You could **wrap** the pointers in stack objects
  - Destructor for stack object should then call **`delete`**/**`free()`** on the wrapped pointer to **`delete`** the object/**`free()`** the memory
  - The **`auto_ptr`** template class is designed for this purpose – see standard class library

Exceptions Advice

- Try to **`catch`** (and handle) an exception as close as possible to the place it was generated
- Do not **`catch`** an exception if you cannot do something with it (leave it to your caller)
- If you **`throw`** exceptions, prefer to **`throw`** standard class library exceptions, or sub-classes of these
  - Choose a meaningful exception
- My suggestion – and **ONLY** a suggestion:
  - There is a risk involved in using exceptions – i.e. less control over the flow of control, like an **implicit return**, so, use exceptions only for ***exceptional*** circumstances

Next Lecture

- Operator overloading
- Strings and streams
  - Short comments/examples about file access