Lecture 16: exceptions, Virtual function table
Project 3E

• You will need to think about how to accomplish the data flow execution pattern and think about how to extend your implementation to make it work.

• This prompt is vaguer than some previous ones
  — ... not all of the details are there on how to do it
Project 3E

```c
blender.SetInput(tbconcat2.GetOutput());
blender.SetInput2(reader.GetOutput());

writer.SetInput(blender.GetOutput());

reader.Execute();
shrinker1.Execute();
lrconcat1.Execute();
tbconcat1.Execute();
shrinker2.Execute();
lrconcat2.Execute();
tbconcat2.Execute();
blender.Execute();

writer.Write(argv[2]);
```
Project 3E

• Worth 3% of your grade
• Assigned today, due May 23
New Stuff: Exceptions
Exceptions

• C++ mechanism for handling error conditions
• Three new keywords for exceptions
  – try: code that you “try” to execute and hope there is no exception
  – throw: how you invoke an exception
  – catch: catch an exception ... handle the exception and resume normal execution
Exceptions

```cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 105" << endl;
        throw 105;
        cout << "Done throwing 105" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
}
```

```bash
fawcett:330 childs$ cat exceptions.C
fawcett:330 childs$ g++ exceptions.C
```
Exceptions: catching multiple types

```cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 105" << endl;
        throw 105;
        cout << "Done throwing 105" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
}
```

```
fawcett:330 childs$ g++ exceptions2.C
fawcett:330 childs$ ./a.out
About to throw 105
Caught an int: 105
```
Exceptions: catching multiple types

```cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 10.5" << endl;
        throw 10.5;
        cout << "Done throwing 10.5" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
}
```

```bash
fawcett:330 childs$ cat exceptions3.C
fawcett:330 childs$ g++ exceptions3.C
fawcett:330 childs$ ./a.out
About to throw 10.5
terminate called after throwing an instance of 'double'
Abort trap
```
Exceptions: catching multiple types

```cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    try
    {
        cout << "About to throw 10.5" << endl;
        throw 10.5;
        cout << "Done throwing 10.5" << endl;
    }
    catch (int &theInt)
    {
        cout << "Caught an int: " << theInt << endl;
    }
    catch (float &theFloat)
    {
        cout << "Caught a float: " << theFloat << endl;
    }
    catch (double &theDouble)
    {
        cout << "Caught a double: " << theDouble << endl;
    }
}
```
Exceptions: throwing/catching complex types

```cpp
void Foo()
int main()
{
    try
    {
        Foo();
    }
    catch (MemoryException &e)
    {
        cout << "I give up" << endl;
    }
    catch (OverflowException &e)
    {
        cout << "I think it is OK" << endl;
    }
    catch (DivideByZeroException &e)
    {
        cout << "The answer is bogus" << endl;
    }
```
Exceptions: cleaning up before you return

```c
void Foo(int *arr);

int *
Foo2(void)
{
    int *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (MyExceptionType &e)
    {
        delete [] arr;
        return NULL;
    }

    return arr;
}
```
Exceptions: re-throwing

```c
void Foo(int *arr);

int *
Foo2(void)
{
    int *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (MyExceptionType &e)
    {
        delete [] arr;
        throw e;
    }

    return arr;
}
```
Exceptions: catch and re-throw anything

```c
void Foo(int *arr);

int *
Foo2(void)
{
    int *arr = new int[1000];
    try
    {
        Foo(arr);
    }
    catch (...)
    {
        delete [] arr;
        throw;
    }

    return arr;
}
```
Exceptions: declaring the exception types you can throw

```c
int * MyIntArrayMemoryAllocator(int num) throw(FloatingPointException) {
    int *arr = new int[num];
    if (arr == NULL)
        throw DivideByZeroException();

    return arr;
}
```
Exceptions: declaring the exception types you can throw ... not all it is cracked up to be

```cpp
int * MyIntArrayMemoryAllocator(int num) throw(FloatingPointException) {
  int *arr = new int[num];
  if (arr == NULL)
    throw MemoryException();

  return arr;
}
```

This will compile ... compiler can only enforce this as a run-time thing.

As a result, this is mostly unused (I had to read up on it)

But: “standard” exceptions have a “throw” in their declaration.
std::exception

• c++ provides a base type called “std::exception”
• It provides a method called “what”
## Exceptions generator by C++ standard library

<table>
<thead>
<tr>
<th>exception</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad_alloc</td>
<td>thrown by <code>new</code> on allocation failure</td>
</tr>
<tr>
<td>bad_cast</td>
<td>thrown by <code>dynamic_cast</code> when it fails in a dynamic cast</td>
</tr>
<tr>
<td>bad_exception</td>
<td>thrown by certain dynamic exception specifiers</td>
</tr>
<tr>
<td>bad_typeid</td>
<td>thrown by <code>typeid</code></td>
</tr>
<tr>
<td>bad_function_call</td>
<td>thrown by empty <code>function</code> objects</td>
</tr>
<tr>
<td>bad_weak_ptr</td>
<td>thrown by <code>shared_ptr</code> when passed a bad <code>weak_ptr</code></td>
</tr>
</tbody>
</table>
Project 3F in a nutshell

• Logging:
  – infrastructure for logging
  – making your data flow code use that infrastructure

• Exceptions:
  – infrastructure for exceptions
  – making your data flow code use that infrastructure

The webpage has a head start at the infrastructure pieces for you.
Warning about 3F

• My driver program only tests a few exception conditions

• Your stress tests later will test a lot more.
  – Be thorough, even if I’m not testing it
3F timeline

• Assigned tonight, due Saturday May 26th
3F: warning

- 3F will almost certainly crash your code
  - It uses your modules wrong!
- You will need to figure out why, and add exceptions
  - gdb will be helpful
Review: Access Control
Two contexts for access control

class A : public B {
    public:
        A() { x=0; y=0; }
        int foo() { x++; return foo2(); }
    private:
        int x, y;
        int foo2() { return x+y; }
};
Inheritance ("class A : public B")

• public $\rightarrow$ “is a”
  – (I never used anything but public)

• private $\rightarrow$ “implemented using”
  – (I have never used this, but see how it could be useful)

• protected $\rightarrow$ the internet can not think of any useful examples for this
Access Control

class Hank{
    public/private/protected:
        BankAccount hanksId;
};

<table>
<thead>
<tr>
<th>Access control type</th>
<th>Who can read it</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Only Hank class</td>
</tr>
<tr>
<td>public</td>
<td>Anyone</td>
</tr>
<tr>
<td>protected</td>
<td>Those who inherit from Hank</td>
</tr>
</tbody>
</table>
Class Vs Struct

• Class:
  – Default inheritance is private
    • That’s why you add public (class A : public B)
  – Default access control is private

• Struct:
  – Default inheritance is public
    • That’s why you don’t have to add public (struct A : B)
  – Default access control is public
Example of data flow (image processing)
How C++ Does Methods
“this”: pointer to current object

• From within any struct’s method, you can refer to the current object using “this”
How methods work under the covers (1/4)

class MyIntClass
{
    public:
        MyIntClass(int x) { myInt = x; }

    friend void FriendIncrementFunction(MyIntClass *); 
    int GetMyInt() { return myInt; }

    protected:
        int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++; 
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (2/4)

class MyIntClass
{
    public:
        MyIntClass(int x) { myInt = x; }

    friend void FriendIncrementFunction(MyIntClass *mic);
    int GetMyInt() { return myInt; }

    protected:
        int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;  // Annotation
}

int main()
{
    MyIntClass MIC(12);  // Annotation
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (3/4)

class MyIntClass
{
public:
    MyIntClass(int x) { myInt = x; }

    friend void FriendIncrementFunction(MyIntClass *);
    void IncrementMethod(void);
    int GetMyInt() { return myInt; }

protected:
    int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

void MyIntClass::IncrementMethod(void)
{
    this->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (4/4)

The compiler secretly slips “this” onto the stack whenever you make a method call.

It also automatically changes “myInt” to this->myInt in methods.

```cpp
#include <iostream>

class MyIntClass {
public:

    int myInt;

private:

    int this->myInt;

    static int MIC;

public:

    MyIntClass(int intVal) : MIC(intVal) {
        this->myInt = intVal;
    }

    void increment() {
        this->myInt++;
    }

private:

    int MIC;

};

int main() {

    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.incrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;

    return 0;
}
```

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>MIC/myInt</td>
<td>12</td>
</tr>
<tr>
<td>0x8004</td>
<td>mic</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>MIC/myInt</td>
<td>13</td>
</tr>
<tr>
<td>0x8004</td>
<td>this</td>
<td>0x8000</td>
</tr>
</tbody>
</table>
Virtual Function Tables
Virtual functions

• Virtual function: function defined in the base type, but can be re-defined in derived type.
• When you call a virtual function, you get the version defined by the derived type
```c
#include <stdio.h>

struct SimpleID {
    int id;
    virtual int GetIdentifier() { return id; };
};

struct ComplexID : SimpleID {
    int extraId;
    virtual int GetIdentifier() { return extraId*128+id; };
};

int main()
{
    ComplexID cid;
    cid.id = 3;
    cid.extraId = 3;
    printf("ID = %d\n", cid.GetIdentifier());
}
```

Virtual functions: example
Picking the right virtual function

class A
{
 public:
  virtual const char *GetType() { return "A"; }
};

class B : public A
{
 public:
  virtual const char *GetType() { return "B"; }
};

int main()
{
  A a;
  B b;

  cout << "a is " << a.GetType() << endl;
  cout << "b is " << b.GetType() << endl;
}

It seems like the compiler should be able to figure this out ...
it knows that a is of type A and
it knows that b is of type B
Picking the right virtual function

```cpp
class A
{
    public:
    virtual const char *GetType() { return "A"; };
};

class B : public A
{
    public:
    virtual const char *GetType() { return "B"; };
};

void ClassPrinter(A *ptrToA)
{
    cout << "ptr points to a " << ptrToA->GetType() << endl;
}

int main()
{
    A a;
    B b;

    ClassPrinter(&a);
    ClassPrinter(&b);
}
```

So how to does the compiler know?

How does it get “B” for “b” and “A” for “a”??
Virtual Function Table

• Let C be a class and X be an instance of C.
• Let C have 3 virtual functions & 4 non-virtual functions
• C has a hidden data member called the “virtual function table”
• This table has 3 rows
  – Each row has the correct definition of the virtual function to call for a “C”.
• When you call a virtual function, this table is consulted to locate the correct definition.
Showing the existence of the virtual function pointer with sizeof()

```cpp
class A {
public:
    virtual
};
class B : public A {
public:
    virtual
};
class C {
public:
    const char *GetType() { return "C"; }
};
int main() {
    A a;
    B b;
    cout << "Size of A is " << sizeof(A) << endl;
    cout << "Size of a pointer is " << sizeof(int *) << endl;
    cout << "Size of C is " << sizeof(C) << endl;
}
```

empty objects have size of 1? why?!?

Answer: so every object has a unique address.

```
fawcett:330 childs$ ./a.out
Size of A is 8
Size of a pointer is 8
Size of C is 1
```
Virtual Function Table

• Let C be a class and X be an instance of C.
• Let C have 3 virtual functions & 4 non-virtual functions
• Let D be a class that inherits from C and Y be an instance of D.
  – Let D add a new virtual function
• D’s virtual function table has 4 rows
  – Each row has the correct definition of the virtual function to call for a “D”.
More notes on virtual function tables

• There is one instance of a virtual function table for each class
  – Each instance of a class shares the same virtual function table

• Easy to overwrite (i.e., with a memory error)
  – And then all your virtual function calls will be corrupted
  – Don’t do this! ;)

Virtual function table: example

CIS 330: Project #2C
Assigned: April 17th, 2014
Due April 24th, 2014
(which means submitted by 6am on April 25th, 2014)
Worth 6% of your grade

Please read this entire prompt!

Assignment: You will implement subtypes with C.

1) Make a union called ShapeUnion with the three types (Circle, Rectangle, Triangle).
2) Make a struct called FunctionTable that has pointers to functions.
3) Make an enum called ShapeType that identifies the three types.
4) Make a struct called Shape that has a ShapeUnion, a ShapeType, and a FunctionTable.
5) Modify your 9 functions to deal with Shapes.
6) Integrate with the new driver function. Test that it produces the correct output.
Virtual function table: example

class Shape
{
    virtual double GetArea() = 0;
    virtual void GetBoundingBox(double *) = 0;
};

class Rectangle : public Shape
{
    public:
        Rectangle(double, double, double, double, double);
        virtual double GetArea();
        virtual void GetBoundingBox(double *);
    protected:
        double minX, maxX, minY, maxY;
};

class Triangle : public Shape
{
    public:
        Triangle(double, double, double, double, double);
        virtual double GetArea();
        virtual void GetBoundingBox(double *);
    protected:
        double pt1X, pt2X, minY, maxY;
};
Questions

• What does the virtual function table look like for a Shape?
  ```c
  typedef struct
  {
    double (*GetArea)(Shape *);
    void (*GetBoundingBox)(Shape *, double *);
  } VirtualFunctionTable;
  ```

• What does Shape’s virtual function table look like?
  – Trick question: Shape can’t be instantiated, precisely because you can’t make a virtual function table
    • abstract type due to pure virtual functions
Questions

• What is the virtual function table for Rectangle?

```c
    c->ft.GetArea = GetRectangleArea;
    c->ft.GetBoundingBox = GetRectangleBoundingBox;
```

• (this is a code fragment from my 2C solution)
Calling a virtual function

• Let $X$ be an instance of class $C$.
• Assume you want to call the $4^{th}$ virtual function
• Let the arguments to the virtual function be an integer $Y$ and a float $Z$.
• Then call:

$$ (X.vptr[3])(&X, Y, Z); $$

- The pointer to the virtual function pointer (often called a vptr) is a data member of $X$
- The $4^{th}$ virtual function has index 3 (0-indexing)
- Secretly pass “this” as first argument to method
Inheritance and Virtual Function Tables

This whole scheme gets much harder with multiple inheritance, and you have to carry around multiple virtual function tables.

```cpp
class A
{
    public:
        virtual void Foo2();
};

class C : public B
{
    public:
        virtual void Foo1();
        virtual void Foo2();
        virtual void Foo3();
};
```

<table>
<thead>
<tr>
<th></th>
<th>Location of Foo1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Location of Foo1</td>
</tr>
<tr>
<td></td>
<td>Location of Foo2</td>
</tr>
<tr>
<td>B</td>
<td>Location of Foo2</td>
</tr>
<tr>
<td></td>
<td>Location of Foo3</td>
</tr>
</tbody>
</table>

Same as B’s Foo1
This is how you can treat a C as a B
Virtual Function Table: Summary

• Virtual functions require machinery to ensure the correct form of a virtual function is called
• This is implemented through a virtual function table
• Every instance of a class that has virtual functions has a pointer to its class’s virtual function table
• The virtual function is called via following pointers
  – Performance issue
Now show Project 2D in C++

• Comment:
  – C/C++ great because of performance
  – Performance partially comes because of a philosophy of not adding “magic” to make programmer’s life easier
  – C has very little pixie dust sprinkled in
    • Exception: ‘\0’ to terminate strings
  – C++ has more
    • Hopefully this will demystify one of those things (virtual functions)
#include <iostream>
using std::cerr;
using std::endl;

class Shape
{
    public:
        int s;
        virtual double GetArea() = 0;
        virtual void GetBoundingBox(double *) = 0;
};

class Triangle : public Shape
{
    public:
        virtual double GetArea() { cerr << "In GetArea for Triangle" << endl; return 1; }; 
        virtual void GetBoundingBox(double *) { cerr << "In GetBBox for Triangle" << endl; };
};

class Rectangle : public Shape
{
    public:
        virtual double GetArea() { cerr << "In GetArea for Rectangle" << endl; return 2; }; 
        virtual void GetBoundingBox(double *) { cerr << "In GetBBox for Rectangle" << endl; };
};

struct VirtualFunctionTable
{
    double (*GetArea)(Shape *);
    void (*GetBoundingBox)(Shape *, double *);
};

int main()
{
    Rectangle r;
    cerr << "Size of rectangle is " << sizeof(r) << endl;

    VirtualFunctionTable *vft = *((VirtualFunctionTable**) &r);
    cerr << "Vptr = " << vft << endl;
    double d = vft->GetArea(&r);
    cerr << "Value = " << d << endl;

    double bbox[4];
    vft->GetBoundingBox(&r, bbox);
}
Pitfalls
This is using call-by-value, not call-by-reference.
struct Image
{
    int width;
    int height;
    unsigned char *buffer;
};

Image *ReadFromFile(char *filename)
{
    Image *rv = NULL;

    /* OPEN FILE, descriptor = f */
    /* ... */
    /* set up width w, and height h */
    /* ... */

    rv = malloc(sizeof(Image));
    rv->width = w;
    rv->height = h;
    fread(rv->buffer, sizeof(unsigned char), w*h, f);
}
Pitfall #3

• int *s = new int[6*sizeof(int)];
Pitfall #4

```cpp
int main()
{
    // new black image
    int height = 1000, width = 1000;
    unsigned char *buffer = new unsigned char[3*width*height];
    for (int i = 0 ; i < sizeof(buffer) ; i++)
    {
        buffer[i] = 0;
    }
}
```

- Assume:
  ```cpp```
  ```cpp
  int *X = new int[100];
  ```
- What is `sizeof(X)`?
- What is `sizeof(*X)`?
/* struct definition */
struct Image
{
    /* data members */
};

/* prototypes */
void WriteImage(Image *, const char *);

/* main */
int main()
{
    Image *img = NULL;
    /* set up Image */
    const char *filename = "out.pnm";
    WriteImage(img, filename);
}

/* WriteImage function */
void WriteImage(char *filename, Image *img)
{
    /* code to write img to filename */
}
(not-a-)Pitfall #6

```cpp
unsigned char* Image::getPixel(int i, int j) {
    int pixStart = 3*i*this->width+3+j;
    unsigned char *pixel = new unsigned char[3];
    pixel[0] = this->data[pixStart];
    pixel[1] = this->data[pixStart + 1];
    pixel[2] = this->data[pixStart + 2];
    return pixel;
}
```

Top requires memory allocation / deletion, and does extra copy.
Pitfall #7

- For objects on the stack, the destructors are called when a function goes out of scope
  - You may have a perfectly good function, but it seg-faults on return
- Especially tricky for main
  - program ran to completion, and crashed at the very end
#include <stdlib.h>

class Image {
    public:
        Image() { width = 0; height = 0; buffer = NULL; }
        virtual ~Image() { delete [] buffer; }

    void ResetSize(int width, int height);
    unsigned char *GetBuffer(void) { return buffer; }

    private:
        int width, height;
        unsigned char *buffer;
};

void Image::ResetSize(int w, int h) {
    width = w;
    height = h;
    if (buffer != NULL)
        delete [] buffer;
    buffer = new unsigned char[3*width*height];
}

int main() {
    Image img;
    unsigned char *buffer = img.GetBuffer();
    img.ResetSize(1000, 1000);
    for (int i = 0; i < 1000; i++)
        for (int j = 0; j < 1000; j++)
            for (int k = 0; k < 1000; k++)
                buffer[3*(i*1000+j)+k] = 0;
}
Bonus Topics
Backgrounding

• “&”: tell shell to run a job in the background
  – Background means that the shell acts as normal, but the command you invoke is running at the same time.

• “sleep 60” vs “sleep 60 &”

When would backgrounding be useful?
Suspending Jobs

• You can suspend a job that is running
  Press “Ctrl-Z”

• The OS will then stop job from running and not schedule it to run.

• You can then:
  – make the job run in the background.
    • Type “bg”
  – make the job run in the foreground.
    • Type “fg”
      – like you never suspended it at all!!
Web pages

- ssh –l <user name> ix.cs.uoregon.edu
- cd public_html
- put something in index.html
- → it will show up as
  
  http://ix.cs.uoregon.edu/~<username>
Web pages

- You can also exchange files this way
  - `scp file.pdf <username>@ix.cs.uoregon.edu:~/public_html`
  - point people to `http://ix.cs.uoregon.edu/~<username>/file.pdf`

Note that `~/public_html/dir1` shows up as `http://ix.cs.uoregon.edu/~<username>/dir1`

(`~/dir1" is not accessible via web)