Lecture 16: exceptions
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

```cpp
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
Example of data flow (image processing)

- FileReader
- Crop
- Transpose
- Invert
- Color
- Concatenate
- FileWriter
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
Review: Lecture 15
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.
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
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 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()

empty objects have size of 1? why?!?

Answer: so every object has a unique address.

```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;
}
```

```
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”.

Questions

• What does the virtual function table look like for a Shape?

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

• What goes in Shape’s virtual function table?
  – 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.
• Let the virtual function be the 4th 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 4th 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 virtual inheritance, and you have to carry around multiple virtual function tables.

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

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

Same as B’s
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
#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);
}
New Material
Pitfalls
Pitfall #1

```c
void AllocateBuffer(int w, int h, unsigned char **buffer) {
    *buffer = new unsigned char[3*w*h];
}

int main() {
    int w = 1000, h = 1000;
    unsigned char *buffer = NULL;
    AllocateBuffer(w, h, &buffer);
}
```

This is using call-by-value, not call-by-reference.
Pitfall #2

```c
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

```c
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:
  ```c
  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
# Pitfall #8

```c++
#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;
    return 0;
}
```
New Stuff: Misc. from OH
Make it easy on yourself to run...

128-223-223-73-wireless:330 hank$ cat r ./proj3C 3C_input.pnm 3C_output.pnm
128-223-223-73-wireless:330 hank$ ./r
Other ways to make life easier

- tab from shell: auto-completes
- Ctrl-R: searches backwards in your shell history
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)
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

#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;
    }
}

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

```cpp
fawcett:330 childs$ cat exceptions2.C
#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

```
#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;
  }
}
```

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

```cpp
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

```cpp
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

```c++
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"

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

class myexception: public exception
{
    virtual const char* what() const throw()
    {
        return "My exception happened";
    }
} myex;

int main () {
    try {
        throw myex;
    }
    catch (exception& e) {
        cout << e.what() << '\n';
    }
    return 0;
}
```
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 <code>dynamic_cast</code></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 tomorrow, due May 26th
const
const

• const:
  – is a keyword in C and C++
  – qualifies variables
  – is a mechanism for preventing write access to variables
The compiler enforces const ... just like public/private access controls
Efficiency

Are any of the three for loops faster than the others? Why or why not?

Answer: NumIterations is slowest … overhead for function calls.

Answer: X is probably faster than Y … compiler can do optimizations where it doesn’t have to do “i < X“ comparisons (loop unrolling)
const arguments to functions

- Functions can use const to guarantee to the calling function that they won’t modify the arguments passed in.

```
struct Image
{
    int width, height;
    unsigned char *buffer;
};

ReadImage(char *filename, Image &);
WriteImage(char *filename, const Image &);
```

read function can’t make the same guarantee

guarantees function won’t modify the Image
const pointers

- Assume a pointer named “P”
- Two distinct ideas:
  - P points to something that is constant
    - P may change, but you cannot modify what it points to via P
  - P must always point to the same thing, but the thing P points to may change.
const pointers

```
int X = 4;
int *P = &X;
```

Idea #1:
violates const:
```
*P = 3;
```
OK:
```
int Y = 5; P = &Y;
```

pointer can change, but you can’t modify the thing it points to

Idea #2:
violates const:
```
int Y = 5; P = &Y;
```
OK:
```
*P = 3;
```

pointer can’t change, but you can modify the thing it points to
const pointers

int X = 4;
int *P = &X;

Idea #3: violates const:
  "*P = 3;"
  "int Y = 5; P = &Y;"

OK: none
const pointers

int X = 4;
int *P = &X;

Idea #1:
violates const:
“*P = 3;”

OK:
“int Y = 5; P = &Y;”

pointer can change, but you can’t modify the thing it points to
const pointers

```
int X = 4;
int *P = &X;
```

Idea #2: violates const:
```
int Y = 5; P = &Y;
```

OK:
```
*P = 3;
```

pointer can't change, but you can modify the thing it points to
const pointers

Idea #3: violates const:
   "*P = 3;"
   "int Y = 5; P = &Y;"

OK:
   none

pointer can’t change, and you can’t modify the thing it points to

const in both places
const usage

• class Image;

• const Image *ptr;
  – Used a lot: offering the guarantee that the function won’t change the Image ptr points to

• Image * const ptr;
  – Helps with efficiency. Rarely need to worry about this.

• const Image * const ptr;
  – Interview question!!
Very common issue with const and objects

How does compiler know GetNumberOfPixels doesn’t modify an Image?

We know, because we can see the implementation.

But, in large projects, compiler can’t see implementation for everything.
**const functions with objects**

```c
class Image {
  public:
    int GetNumberOfPixels() const { return width*height; };

  private:
    int width, height;
};

unsigned char * Allocator(const Image *img) {
  int npixels = img->GetNumberOfPixels();
  unsigned char *rv = new unsigned char[3*npixels];
  return rv;
}
```

- **const after method name**
- **If a class method is declared as const, then you can call those methods with pointers.**
mutable

• mutable: special keyword for modifying data members of a class
  – If a data member is mutable, then it can be modified in a const method of the class.
  – Comes up rarely in practice.
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!!