CIS 330:

Lecture 15 & 16:
Potpourri: exceptions, const, globals & static, scope, overloading operators

May 19th & 21st, 2015  Hank Childs, University of Oregon
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
Outline

- Announcements
- Project 3E
- exceptions
- Project 3F
- const
- global, static
- scope
- overloading operators
Announcements

• I need Weds OH to end earlier
• My proposal:
  – Weds 3:25-4:30 in this classroom
CIS 410:
Introduction to Scientific Visualization
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
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

```csharp
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]);

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

writer.SetInput(blender.GetOutput());

blender.GetOutput() -> Update();

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

• Worth 5% of your grade
• Assigned last Friday, due May 22\textsuperscript{nd}
Outline

• Announcements
• Project 3E
• (review access control)
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
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
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
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;
    }
}
```

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

```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;
  }
}
```

fawcett:330 child$ g++ exceptions3.C
fawcett:330 child$ ./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

```java
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;
    }
    return 0;
}
```
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
// using standard exceptions
#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 new on allocation failure</td>
</tr>
<tr>
<td>bad_cast</td>
<td>thrown by dynamic_cast 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 typeid</td>
</tr>
<tr>
<td>bad_function_call</td>
<td>thrown by empty function objects</td>
</tr>
<tr>
<td>bad_weak_ptr</td>
<td>thrown by shared_ptr when passed a bad weak_ptr</td>
</tr>
</tbody>
</table>
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
Project 3F (old)

CIS 330: Project #3F
Assigned: May 21st, 2014
Due May 27th, 2014
(which means submitted by 6am on May 28th, 2014)
Worth 6% of your grade

*Please read this entire prompt!*

Assignment:

Add logging and exception handling to your code.

Turnin: a tarball of your source code, plus a Makefile

Please note: a new main3F.C and a new Makefile are on the class page. I also added a header file called “logger.h” that has the base class for logging and exceptions.
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
Outline

- Announcements
- Project 3E
- exceptions
- Project 3F
- const
- global, static
- scope
- overloading operators
• **const:**
  – is a keyword in C and C++
  – qualifies variables
  – is a mechanism for preventing write access to variables
const example

```c
int main()
{
    const int X = 5;
}
```

The compiler enforces const ... just like public/private access controls
Efficiency

```c
int NumIterations() { return 10; }

int main()
{
    int count = 0;
    int i;
    const int X = 10;
    int Y = 10;
    for (i = 0; i < X; i++)
        count++;
    for (i = 0; i < Y; i++)
        count++;
    for (i = 0; i < NumIterations(); i++)
        count++;
}
```

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.

```c
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

\[
\text{int } X = 4; \\
\text{int } *P = &X;
\]

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 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

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

```cpp
// C++ code snippet

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.
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
globals

- You can create global variables that exist outside functions.

```c
#include <stdio.h>
int X = 5;

int main()
{
    printf("X is %d\n", X);
}
```

```
fawcett:Documents childs$ g++ global1.C
fawcett:Documents childs$ ./a.out
X is 5
fawcett:Documents childs$ 
```
global variables

- global variables are initialized before you enter main
Storage of global variables...

• global variables are stored in a special part of memory
  – “data segment” (not heap, not stack)

• If you re-use global names, you can have collisions
Externs: mechanism for unifying global variables across multiple files

```c
#include <stdio.h>

int count = 0;

int doubler(int Y)
{
    count++;
    return 2*Y;
}
```

extern int count;

```c
int doubler(int);

int main()
{
    count++;
    doubler(3);
    printf("count is \%d\n", count);
}
```

eextern: there’s a global variable, and it lives in a different file.
static

• static memory: third kind of memory allocation
  – reserved at compile time
• contrasts with dynamic (heap) and automatic (stack) memory allocations
• accomplished via keyword that modifies variables

There are three distinct usages of statics
static usage #1: persistency within a function

```c
#include <stdio.h>

int fibonacci()
{
    static int last2 = 0;
    static int last1 = 1;
    int rv = last1+last2;
    last2 = last1;
    last1 = rv;
    return rv;
}

int main()
{
    int i;
    for (int i = 0 ; i < 10 ; i++)
        printf("%d\n", fibonacci());
}
```
static usage #2: making global variables be local to a file

I have no idea why the static keyword is used in this way.
static usage #3: making a singleton for a class

```cpp
fawcett:Downloads childs$ cat static3.C
#include <iostream>

using std::cout;
using std::endl;

class MyClass
{
  public:
    MyClass() { numInstances++; }
    virtual ~MyClass() { numInstances--; }

    int GetNumInstances(void) { return numInstances; }

  private:
    int numInstances;
};

int main()
{
  MyClass *p = new MyClass[10];
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
  delete [] p;
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
```

fawcett:Downloads childs$ g++ static3.C
fawcett:Downloads childs$ ./a.out
Num instances = 1
Num instances = 0
static usage #3: making a singleton for a class

We have to tell the compiler where to store this static.

What do we get?
static usage #3: making a singleton for a class

```cpp
#include <iostream>

using namespace std;

class MyClass
{
public:
    MyClass() { numInstances++; }
    virtual ~MyClass() { numInstances--; }
    int GetNumInstances() { return numInstances; }

private:
    static int numInstances;
};

int MyClass::numInstances = 0;

int main()
{
    MyClass *p = new MyClass[10];
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
    delete [] p;
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
static methods

Static data members and static methods are useful and they are definitely used in practice.
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
I saw this bug quite a few times...

The compiler will sometimes have multiple choices as to which variable you mean. It has rules to make a decision about which one to use.

This topic is referred to as "scope".

```cpp
class MyClass
{
    public:
        void SetValue(int);

    private:
        int X;
};

void MyClass::SetValue(int X)
{
    X = X;
}
```
This one won’t compile.

The compiler notices that you have a variable called X that “shadows” the argument called X.
int X = 0;

class MyClass
{
    public:
        MyClass() { X = 1; };

    void SetValue(int);

    private:
        int X;
};

void MyClass::SetValue(int X)
{
    { int X = 3;
        cout << "X is " << X << endl;
    }
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
int X = 0;

class MyClass
{
    public:
        MyClass() { X = 1; };

    void SetValue(int);

    private:
        int X;
};

void MyClass::SetValue(int X)
{
    
    int X = 3;
    cout << "X is " << X << endl;

}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
int X = 0;

class MyClass
{
    public:
    MyClass() { X = 1; }

    void SetValue(int);

private:
    int X;
};

void MyClass::SetValue(int X)
{
    
    int X = 3;
    cout << "X is " << X << endl;

}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}

What does this one print?

Answer: 1
int X = 0;
class MyClass
{
  public:
    MyClass() { X = 1; };
    void SetValue(int);
  
  private:
    int X;
};

void MyClass::SetValue(int X)
{
  int X = 3;
  cout << "X is " << X << endl;
}

int main()
{
  MyClass mc;
  mc.SetValue(2);
}
Scope Rules

• The compiler looks for variables:
  – inside a function or block
  – function arguments
  – data members (methods only)
  – globals
Shadowing

- Shadowing is a term used to describe a “subtle” scope issue.
  - ... i.e., you have created a situation where it is confusing which variable you are referring to

```cpp
class Sink
{
  public:
    void SetInput(Image *i) { input = i; };
  protected:
    Image *input;
};

class Writer : public Sink
{
  public:
    void Write(void) { /* write input */ };
  protected:
    Image *input;
};

int main()
{
  Writer writer;
  writer.SetInput(image);
  writer.Write();
}
Outline

• Announcements
• Project 3E
• exceptions
• Project 3F
• const
• global, static
• scope
• overloading operators
C++ lets you define operators

• You declare a method that uses an operator in conjunction with a class
  – +, -, /, !, ++, etc.

• You can then use operator in your code, since the compiler now understands how to use the operator with your class

• This is called “operator overloading”
  – ... we are overloading the use of the operator for more than just the simple types.
Example of operator overloading

class MyInt
{
    public:
        MyInt(int x) { myInt = x; }
    MyInt& operator++();
    int GetValue(void) { return myInt; }

    protected:
        int myInt;
};

MyInt &
MyInt::operator++()
{
    myInt++; return *this;
}

int main()
{
    MyInt mi(6);
    ++mi;
    ++mi;
    printf("Value is %d\n", mi.GetValue());
}

fawcett:330 childd$ ./a.out
Value is 8
fawcett:330 child$ cat oostream.C
#include <iostream>

using std::ostream;
using std::cout;
using std::endl;

class Image
{
  public:

    Image();

  friend ostream& operator<<(ostream &os, const Image &);

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

Image::Image()
{
  width = 100;
  height = 100;
  buffer = NULL;
}

ostream &
operator<<(ostream &out, const Image &img)
{
  out << img.width << "x" << img.height << endl;
  if (img.buffer == NULL)
    out << "No buffer allocated!" << endl;
  else
    out << "Buffer is allocated!" << endl;
}

int main()
{
  Image img;
  cout << img;
}
Beauty of inheritance

• `ostream` provides an abstraction
  – That’s all `Image` needs to know
    • it is a stream that is an output
  – You code to that interface
  – All `ostream`’s work with it
class Image {
    public:
        Image();
        Image(int w, int h);
        void SetSize(int w, int h);

        friend ostream& operator<<(ostream &os, const Image &);

        Image & operator=(const Image &);
    
    private:
        int width, height;
        unsigned char *buffer;
};

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

int main() {
    Image img1, img2;
    img1.SetSize(200, 200);
    cout << "Image 1: " << img1;
    cout << "Image 2: " << img2;
    img2 = img1;
    cout << "Image 1: " << img1;
    cout << "Image 2: " << img2;
}
let’s do this again...

```cpp
ostream &
operator<<(ostream &out, const Image &img)
{
    out << img.width << "x" << img.height << endl;
    if (img.buffer == NULL)
        out << "No buffer allocated!" << endl;
    else
        out << "Buffer is allocated, and value is "
            << (void *) img.buffer << endl;

    return out;
}
```

```
fawcett:330 child$ ./a.out
Image 1:200x200
Buffer is allocated, and value is 0x10080000
Image 2:0x0
No buffer allocated!
Image 1:200x200
Buffer is allocated, and value is 0x10080000
Image 2:200x200
Buffer is allocated, and value is 0x10081e600
```
let’s do this again...

class Image
{
    public:
        Image();
        void SetSize(int w, int h);
    friend ostream& operator<<(ostream &os, const Image &);
    // Image & operator=(const Image &);

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

int main()
{
    Image img1, img2;
    img1.SetSize(200, 200);
    cout << "Image 1:" << img1;
    cout << "Image 2:" << img2;
    img2 = img1;
    cout << "Image 1:" << img1;
    cout << "Image 2:" << img2;
}

fawcett:330 childs$ g++ assignment_op.C
fawcett:330 childs$
C++ defines a default assignment operator for you

• This assignment operator does a bitwise copy from one object to the other.

• Does anyone see a problem with this?

```bash
fawcett:330 childs$ ./a.out
Image 1:200x200
Buffer is allocated, and value is 0x100800000
Image 2:0x0
No buffer allocated!
Image 1:200x200
Buffer is allocated, and value is 0x100800000
Image 2:200x200
Buffer is allocated, and value is 0x100800000
```

This behavior is sometimes OK and sometimes disastrous.
Copy constructors: same deal

• C++ automatically defines a copy constructor that does bitwise copying.

• Solutions for copy constructor and assignment operators:
  – Re-define them yourself to do “the right thing”
  – Re-define them yourself to throw exceptions
  – Make them private so they can’t be called
Backup slides
Unix and Windows difference

• Unix:
  – “\n”: goes to next line, and sets cursor to far left

• Windows:
  – “\n”: goes to next line (cursor does not go to left)
  – “\m”: sets cursor to far left

• Text files written in Windows often don’t run well on Unix, and vice-versa
  – There are more differences than just newlines

  vi: “set ff=unix” solves this
I mostly use C++, and I still use memcpy all the time

memcpy

MEMCPY(3) BSD Library Functions Manual MEMCPY(3)

NAME
memcpy -- copy memory area

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <string.h>

void *
memcpy(void *restrict dst, const void *restrict src, size_t n);

DESCRIPTION
The memcpy() function copies n bytes from memory area src to memory area
dst. If dst and src overlap, behavior is undefined. Applications in
which dst and src might overlap should use memmove(3) instead.

RETURN VALUES
The memcpy() function returns the original value of dst.
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!!