Lecture 16: potpourri
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]);
```
Project 3E

• Worth 3% of your grade
• Assigned May 17, due yesterday
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
Review
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

```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;
    }
}
```

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

```c++
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 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: 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;
}
```
New Material
const

• 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.
• 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.
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

```c
int X = 4;
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 goes before type
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**

```
int X = 4;
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 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

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;
}
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.
globals
globals

• You can create global variables that exist outside functions.

```
#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

```c
#include <stdio.h>

intInitializer()
{
    printf("In initializer\n");
    return 6;
};

intX = Initializer();

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

fawcett:Documents child$ cat global2.C
fawcett:Documents child$ g++ global2.C
fawcett:Documents child$ ./a.out
In initializer
In main
X is 6
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

```c
int main()
{
    int X = 6;
    int Y;
    return 2*Y;
}
```

```console
fawcett:Documents child$ cat file1.C
int X = 6;
int main()
{
    int Y;
    return 2*Y;
}
```
Externs: mechanism for unifying global variables across multiple files

```c
#include <stdio.h>

int count = 0;

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

```bash
fawcett:330 childs$ cat file2.C
extern int count;
```

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

```bash
fawcett:330 childs$ g++ -c file1.C
fawcett:330 childs$ g++ -c file2.C
fawcett:330 childs$ g++ file1.o file2.o
fawcett:330 childs$ ./a.out
count is 2
```

extern: there’s a global variable, and it lives in a different file.
There are three distinct usages of statics

- 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
static usage #1: persistency within a function

fawcett:330 childs$ cat static1.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.

```c
#include <stdio.h>

static int count = 0;

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

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

```
fawcett:330 childs$ cat file2.C
static int count = 0;

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

int main()
{
    count++;
    doubler(3);
    printf("count is %d\n", count);
}
```
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
fawcett:Downloads childs$  
```
static usage #3: making a singleton for a class

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

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

class MyClass
{
    public:
        MyClass();
        virtual ~MyClass();

    int GetNumInstances();

    private:
        static int numInstances;
};

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

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
fawcett:Downloads child$ 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:
    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 data members and static methods are useful and they are definitely used in practice.
Scope
• 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;
}
```
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);
}
This one will compile ... the compiler thinks that you made a new scope on purpose.

So what does it print?

Answer: 3
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);
}
```cpp
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);
}
Scope Rules

• The compiler looks for variables:
  – inside a function or block
  – function arguments
  – data members (methods only)
  – globals
Pitfall #8

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

```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;
}
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
Overloading Operators

• NOTE: I lectured on this some, but it was informal. These slides formally capture the ideas we discussed.
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 childs$ ./a.out
Value is 8
More operator overloading

```cpp
#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;
}

fawcett:330 childs$ g++ oostream.C
fawcett:330 childs$ ./a.out
100x100
No buffer allocated!
```
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

```
int main()
{
    Image img;
    cerr << img;
}
```

```
int main()
{
    Image img;
    ofstream ofile("output_file");
    ofile << img;
}
```

fawcett:330 childs$ ./a.out
100x100
No buffer allocated!

fawcett:330 childs$ g++ oostream.C
fawcett:330 childs$ ./a.out
fawcett:330 childs$ cat output_file
100x100
No buffer allocated!
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;
};

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

Image & Image::operator=(const Image &rhs)
{
    if (buffer != NULL)
        delete [] buffer;
    buffer = NULL;

    width = rhs.width;
    height = rhs.height;
    if (rhs.buffer != NULL)
    {
        buffer = new unsigned char[3*width*height];
        memcpy(buffer, rhs.buffer, 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 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 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?

```
fawcett:330 child$ ./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
Project 3G

• Will add new filters.
• Likely assigned tomorrow.
Stress Test Project (3H)

- We will have ~60 stress tests
- We can’t check in 60 baseline images and difference them all
  - Will slow ix to a grind
- Solution:
  - We commit “essence of the solution”
  - We also complement that all images posted if needed.
Checksums

Most useful when input is very large and checksum is very small

<table>
<thead>
<tr>
<th>Input</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox</td>
<td>1582054665</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>2367213558</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>3043859473</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>1321115126</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>1685473544</td>
</tr>
</tbody>
</table>

From Wikipedia
Our “checksum”

• Three integers:
  – Sum of red channel
  – Sum of green channel
  – Sum of blue channel

• When you create a stress test, you register these three integers

• When you test against others stress tests, you compare against their integers
  – If they match, you got it right

This will be done with a derived type of Sink.
Should Checksums Match?

- On ix, everything should match
- On different architectures, floating point math won’t match
- Blender: has floating point math
- → no blender
Bonus Topics
Upcasting and Downcasting

• Upcast: treat an object as the base type
  – We do this all the time!
  – Treat a Rectangle as a Shape

• Downcast: treat a base type as its derived type
  – We don’t do this one often
  – Treat a Shape as a Rectangle
    • You better know that Shape really is a Rectangle!!
Upcasting and Downcasting

class A {
};

class B : public A {
    public:
        B() { myInt = 5; };
        void Printer(void) { cout << myInt << endl; };

    private:
        int myInt;
};

void Downcaster(A *a) {
    B *b = (B *) a;
    b->Printer();
}

int main()
{
    A a;
    B b;

    Downcaster(&b); // no problem
    Downcaster(&a); // no good
}

fawcett:330 childs$ g++ downcaster.C
fawcett:330 childs$ ./a.out
5
-1074118656

what do we get?
Upcasting and Downcasting

• C++ has a built in facility to assist with downcasting: dynamic_cast
• I personally haven’t used it a lot, but it is used in practice
• Ties in to std::exception
Default Arguments

```c
void Foo(int X, int Y = 2) {
    cout << "X = " << X << " Y = " << Y << endl;
}

int main() {
    Foo(5);
    Foo(5, 4);
}
```

default arguments: compiler pushes values on the stack for you if you choose not to enter them.
Booleans

- New simple data type: bool (Boolean)
- New keywords: true and false

```c
int main()
{
    bool b = true;
    cout << "Size of boolean is " << sizeof(bool) << endl;
}
```

fawcett:330 childs$ g++ Boolean.C
fawcett:330 childs$ ./a.out