Lecture 14:
pointer stuff
C++ streams
Announcements

• Projects
  – 3C posted yesterday, due May 19
  – 3D posted today, also due May 19
    • 3D is not required to do 3E, etc.
    • So you can skip it if you are behind, although you will lose points.

• Other
  – Final on Dead Week?
    • Hank may have to go to Barcelona to cover for a visa issue
More on Pointers
(Poor) Analogy

• Safe deposit box
(Poor) Analogy

• You go to the bank
• You ask for a safe deposit box
  – The key to the box is a pointer
• You get access to a space in the vault
  – The box in the vault is the memory on the heap
int main()
{
    float *buffer = new float[1000];
    // ...
    buffer = new float[100];
}

• You go to teller and request a safe deposit box
• Teller gives you a key, to box #105
• Then you go back the next day and request another safe deposit box, to box #107
• And you throw out the key to box #105 and only keep the key to box #107
• This is a memory leak
  — No one will ever be able to access to box #105
Now let’s think about stack/heap

```c
int main()
{
    float *buffer = new float[1000];
    // ...
    buffer = new float[100];
}
```
Analogy Continued

```c
int main()
{
    float *buffer = new float[1000];
    // ...
    float *buffer2 = buffer;
}
```

- You go to teller and request a safe deposit box
- Teller gives you a key, to box #105 (buffer)
- You make a copy of the key for your friend (buffer2)
- Now you and your friend have access to box #105
- If your friend changes the contents, then it affects you
- Terminology: this is called a “shallow copy”
Now let’s think about stack/heap

```c
int main()
{
    float *buffer = new float[1000];
    // ... 
    float *buffer2 = buffer;
}
```
Analogy Continued (But Starting to Break Down)

```c
int main()
{
    float *buffer = new float[1000];
    // ...
    float *buffer2 = new float[1000];
    for (int i = 0; i < 1000; i++)
        buffer2[i] = buffer[i];
}
```

- You go to teller and request a safe deposit box
- Teller gives you a key, to box #105 (buffer)
- You fill the box
- You later request a second safe deposit box
- Teller gives you a key, to box #107 (buffer2)
- You examine box #105. Whatever is in 105, you put in 107
  - Example: $10K in 105. So put an additional $10K in 107. ($20K total)
- This is called a “deep copy”
Now let’s think about stack/heap

```c
int main()
{
    float *buffer = new float[1000];
    // ...
    float *buffer2 = new float[1000];
    for (int i = 0; i < 1000; i++)
        buffer2[i] = buffer[i];
}
```
Arrays on the stack are different

```c
int main()
{
    float A[3] = { 0.5, 1.5, 2.5 };
}
```

- You cannot re-assign A to another value.
  - A is bound to its stack location
  - But you can assign a pointer to point at A’s location.
  - And compiler can do this automatically (int *A_ptr = A;)

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff0</td>
<td>0.5</td>
</tr>
<tr>
<td>0x7ffec</td>
<td>1.5</td>
</tr>
<tr>
<td>0x7ffe8</td>
<td>2.5</td>
</tr>
</tbody>
</table>
Review
Inheritance and Constructors/Destructors: Example

• Constructors from base class called [first], then next derived type second, and so on.
• Destructor from base class called [last], then next derived type second to last, and so on.

• Derived type always assumes base class exists and is set up
  – ... base class never needs to know anything about derived types
#include <stdio.h>

class C
{
    public:
        C() { printf("Constructing C\n"); }
        ~C() { printf("Destructing C\n"); }
};

class D : public C
{
    public:
        D() { printf("Constructing D\n"); }
        ~D() { printf("Destructing D\n"); }
};

int main()
{
    printf("Making a D\n");
    {
        D b;
    }

    printf("Making another D\n");
    {
        D b;
    }

    Making a D
    Constructing C
    Constructing D
    Destructing D
    Destructing C
    Making another D
    Constructing C
    Constructing D
    Destructing D
    Destructing C
Possible to get the wrong destructor

• → Use “virtual” keyword to make destructors virtual
Objects in objects

```c++
#include <stdio.h>

class A {
  public:
    A() { printf("Constructing A\n"); }
    ~A() { printf("Destructing A\n"); }
};

class B {
  public:
    B() { printf("Constructing B\n"); }
    ~B() { printf("Destructing B\n"); }
  private:
    A a1, a2;
};

int main() {
  printf("Making a B\n");
  B b;
  printf("Making another B\n");
  B b;
```
#include <stdio.h>

class A
{
   public:
      A() { printf("Constructing A\n"); };
      ~A() { printf("Destructing A\n"); };
};

class B
{
   public:
      B() { printf("Constructing B\n"); };
      ~B() { printf("Destructing B\n"); };
};

class C
{
   public:
      C() { printf("Constructing C\n"); };
      ~C() { printf("Destructing C\n"); };
   private:
      A a;
      B b;
};

int main()
{
   C c;
}
Objects in objects: order is important

```c
#include <stdio.h>

class A
{
    public:
        A() { printf("Constructing A\n"); }
        ~A() { printf("Destructing A\n"); }
};

class B
{
    public:
        B() { printf("Constructing B\n"); }
        ~B() { printf("Destructing B\n"); }
};

class C
{
    public:
        C() { printf("Constructing C\n"); }
        ~C() { printf("Destructing C\n"); }
    private:
        B b;
        A a;
};

int main()
{
    C c;
}
```
Initializers

• New syntax to have variables initialized before even entering the constructor

```c
#include <stdio.h>

class A
{
    public:
        A() : x(5)
        {
            printf("x is %d\n", x);
        }
    private:
        int x;
};

int main()
{
    A a;
}
```
Initializers

- Initializers are a mechanism to have a constructor pass arguments to another constructor
- Needed because
  - Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class
  - Constructors for objects contained in a class are called before the container class & need to pass arguments in container class’s destructor
Initializers

• Needed because
  – Constructors for objects contained in a class are called before the
    container class & need to pass arguments in container class’s
    destructor

```c++
#include <stdio.h>

class A {
  public:
    A(int x) { v = x; }
  private:
    int v;
};

class B {
  public:
    B(int x) { v = x; }
  private:
    int v;
};

class C {
  public:
    C(int x, int y) : b(x), a(y) { }
  private:
    B b;
    A a;
};

int main()
{
  C c(3,5);
}
```
Initializers

- Needed because
  - Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class
Multiple inheritance

• A class can inherit from more than one base type
• This happens when it “is a” for each of the base types
  – Inherits data members and methods of both base types
Pure Virtual Functions

• Pure Virtual Function: define a function to be part of the interface for a class, but do not provide a definition.
• Syntax: add “=0” after the function definition.
• This makes the class be “abstract”
  – It cannot be instantiated
• When derived types define the function, then are “concrete”
  – They can be instantiated
class Shape
{
    public:
    virtual double GetArea(void) = 0;
};

class Rectangle : public Shape
{
    public:
    virtual double GetArea() { return 4; }
};

int main()
{
    Shape s;
    Rectangle r;
}

fawcett:330 childs$ g++ pure_virtual.C
pure_virtual.C: In function ‘int main()’:  
pure_virtual.C:15: error: cannot declare variable ‘s’ to be of abstract type ‘Shape’
pure_virtual.C:2: note: because the following virtual functions are pure within ‘Shape’:  
pure_virtual.C:4: note: virtual double Shape::GetArea()
Project 3C

CIS 330: Project #3C
Assigned: May 7th, 2017
Due May 17th, 2017
(which means submitted by 6am on May 18th, 2017)
Worth 7% of your grade

Please read this entire prompt!

Assignment: Change your 3B project to be object-oriented.

3D will be due on May 19th as well.
BUT: you can skip 3D.
You get 0/3 points.
But you don’t need 3D to do 3E-3I.
Assignment: make your code base data flow networks with OOP
New Stuff
C++ lets you define operators

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

• You can then use your 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.

You can also do this with functions.
Example of operator overloading

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

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

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

Define operator ++ for MyInt

Declare operator ++ will be overloaded for MyInt

We will learn more about operator overloading later in the quarter.

Call operator ++ on MyInt.
New operators: << and >>

- “<<”: Insertion operator
- “>>>”: Extraction operator
  - Operator overloading: you can define what it means to insert or extract your object.

- Often used in conjunction with “streams”
  - Recall our earlier experience with C streams
    - stderr, stdout, stdin
  - Streams are communication channels
cout: the C++ way of accessing stdout

New header file (and no ".h"!)

New way of accessing stdout stream.

Insertion operation (<<)
cout is in the “standard” namespace

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

using std::cout;

int main()
{
    cout << "The answer is: ";
    cout << 8;
    cout << "\n";
}
```

“using” command puts the “cout” portion of the standard namespace (“std”) in the global namespace.

Don’t need “std::cout” any more…
endl: the C++ `endl` mechanism

- prints a newline
- flushes the stream
  - C version: `fflush(stdout)`
  - This is because `printf` doesn’t always print when you ask it to.
    - It buffers the requests when you make them.
    - This is a problem for debugging!!
endl in action

```cpp
fawcett:330 childs$ cat printCPP.C
#include <iostream>

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

int main()
{
    cout << "The answer is: ";
    cout << 8;
    cout << endl;
}

fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$ 
```
<< and >> have a return value

• ostream & ostream::operator<<(int);
  – (The signature for a function that prints an integer)

• The return value is itself
  – i.e., the cout object returns “cout”

• This allows you to combine many insertions (or extractions) in a single line.
  – This is called “cascading”.
Cascading in action

```cpp
fawcett:330 childs$ cat printCPP.C
#include <iostream>

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

int main()
{
    cout << "The answer is: " << 8 << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$  
```
Putting it all together

```c
fawcett:330 childls$ cat print.c
#include <stdio.h>

int main()
{
    printf("The answer is: ");
    printf("%d", 8);
    printf("\n");
}
fawcett:330 childls$ gcc print.c
fawcett:330 childls$ ./a.out
The answer is: 8
```

```cpp
fawcett:330 childls$ cat printCPP.C
#include <iostream>

int main()
{
    std::cout << "The answer is: ";
    std::cout << 8;
    std::cout << "\n";
}
fawcett:330 childls$ g++ printCPP.C
fawcett:330 childls$ ./a.out
The answer is: 8
```

```c
fawcett:330 childls$ cat print.C
#include <stdio.h>

int main()
{
    printf("The answer is: %d\n", 8);
}
fawcett:330 childls$ g++ print.C
fawcett:330 childls$
```

```cpp
fawcett:330 childls$ cat printCPP.C
#include <iostream>

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

int main()
{
    cout << "The answer is: " << 8 << endl;
}
fawcett:330 childls$ g++ printCPP.C
fawcett:330 childls$
```
Three pre-defined streams

• `cout <= => fprintf(stdout, ...`
• `cerr <= => fprintf(stderr, ...`
• `cin <= => fscanf(stdin, ...`
cin in action

```cpp
fawcett:330 childs$ cat cin.C
#include <iostream>

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

int main()
{
    int X, Y, Z;
    cin >> X >> Y >> Z;
    cout << Z << "", " << Y << "", " << X << endl;
}
fawcett:330 childs$ ./a.out
3 5
4
4, 5, 3
```
cerr

• Works like cout, but prints to stderr
• Always flushes everything immediately!

“See the error”
fstream

- ifstream: input stream that does file I/O
- ofstream: output stream that does file I/O

- Not lecturing on this, since it follows from:
  - C file I/O
  - C++ streams

http://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm
Project 3D

• Assigned: today, 5/12
• Due: Friday May 19
• Important: if you skip this project, you will still be able to do future projects (3E, 3F, etc)
• Assignment:
  – Write PNMreaderCPP and PNMwriterCPP ... new version of the file reader and writer that use fstream.
Default Methods

• C++ makes 4 methods for you by default:
  – Default constructor
  – Copy constructor
  – Assignment operator
  – Destructor

```cpp
class A
{
    A() { ; };
    A(A &) { ; };
    ~A() { ; };
    A & operator=(A &){ ; };
};
```
What if there are data members?

class A
{
  public:
    A() { };
    A(A &a) { x = a.x; };
    A & operator=(A &a) { x = a.x; return a; };
  private:
    int x;
};
For Image

class Image
{
public:
    Image() { buffer = NULL; };
~Image() { if (buffer != NULL) delete [] buffer; };
    Resize() { ... };
private:
    Pixel *buffer;
};

int main()
{
    Image i;
i.Resize(1000, 1000);
    Image i2 = i;
}

THIS WILL CRASH
Solution

class Image
{
    public:
        Image() { buffer = NULL; };
    ~Image() { if (buffer != NULL) delete [] buffer; };
        Resetsize() { ... };
    private:
        Pixel *buffer;
    Image(Image &i) { ; };
    Image &operator(Image &i) { ; };
};

• This will prevent you from accidentally calling copy constructor or assignment operator
• (You should add this to your Image class)
And you may be using assignment operators right now without knowing it…

• ... so “=” is doing more work than you might expect