UNIVERSITY OF OREGON

CIS 330:

Unix and C/C++

Lecture 17:
Virtual function table, potpourri

May 21st, 2018

Hank Childs, University of Oregon

Schedule (lectures)

- Week 8
  - Mon & Weds: Hank lectures
  - Fri: Brent lab on debugging
- Week 9
  - Mon: Memorial Day
  - Weds: live code of project 3
  - Fri: Brent lectures on templates
- Week 10
  - Mon & Weds: Brent holds his OH in MCK125 during class time
  - Fri: Hank does review for final

Schedule (projects)

- 3E: due Weds
- 3F: “due” May 27
- 3G: assigned Weds May 23, “due” Weds May 30
- 3T: assigned Weds May 30, due Friday June 2
  - No late on this project
- 3H, 4A, 4B: “due” Friday June 9th
- AND: all work must be submitted by Weds June 13. No work will be accepted after this time.

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

```
–
    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]);
}
```

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

    writer.SetInput(blender.GetOutput());

    blender.GetOutput()->Update();
    writer.Write(argv[2]);
}
```

Project 3E

- Worth 3% of your grade
- Assigned today, due May 23

3F

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: 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; };
};
```

defines how a class inherits from another class

defines access controls for data members and methods

Inheritance ("class A : public B")

- public → "is a"
  - (I never used anything but public)
- private → "implemented using"
  - (I have never used this, but see how it could be useful)
- protected → the internet can not think of any useful examples for this

Access Control

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

Access control type	Who can read it
private	Only Hank class
public	Anyone
protected	Those who inherit from Hank

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

How C++ Does Methods

"this": pointer to current object

- From within any struct's method, you can refer to the current object using "this"

```
TallyCounter::TallyCounter(int c)
{
    count = c;
}

     <------->

TallyCounter::TallyCounter(int c)
{
    this->count = c;
}
```

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;
}
```

```
fawcett:330 childs$ g++ this.C
fawcett:330 childs$ ./a.out
My int is 14
fawcett:330 childs$
```

How methods work under the covers (2/4)

```
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;
}
```

Addr.	Variable	Value
0x8000	MIC/myInt	12

Addr.	Variable	Value
0x8000	MIC/myInt	12
0x8004	mic	0x8000

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;
}
```

```
fawcett:330 childs$ g++ this.C
fawcett:330 childs$ ./a.out
My int is 14
fawcett:330 childs$
```

How methods work under the covers (4/4)

```
class  MyIntClass
{
}
v
Fr
{
    mic->myInt++;
}

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

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}
```

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

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

Addr.	Variable	Value
0x8000	MIC/myInt	12

Addr.	Variable	Value
0x8000	MIC/myInt	12
0x8004	mic	0x8000

Addr.	Variable	Value
0x8000	MIC/myInt	13
0x8004	this	0x8000

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

Virtual functions: example

```
128-223-223-72-wireless:330 hank$ cat virtual.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());
}
128-223-223-72-wireless:330 hank$ g++ virtual.C
128-223-223-72-wireless:330 hank$ ./a.out
ID = 387
```

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;
}

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

??????

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

```
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);
}

fawcett:330 childs$ g++ virtual2.C
fawcett:330 childs$ ./a.out
```

??????

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()

```
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
```

what will this print?

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);
    virtual double GetArea();
    virtual void   GetBoundingBox(double *);
  protected:
    double minX, maxX, minY, maxY;
};

class Triangle : public Shape
{
  public:
                   Triangle(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?

```
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->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 4th 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 4th virtual function has index 3 (0-indexing)

The pointer to the virtual function pointer (often called a vptr) is a data member of X

Secretly pass “this” as first argument to method

Inheritance and Virtual Function Tables

```
class A
{
  public:
```

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

```
    virtual void Foo2();
};
```

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

A	
	Location of Foo1

	Location of Foo1
[illegible]	Location of Foo2

Same as B's
This is how you can treat a C as a B

C	
Foo1	Location of Foo1
Foo2	Location of Foo2
Foo3	Location of Foo3

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)

vptr.C

```
fawcett:vptr childs$ cat vptr.C
#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

Pitfall #1

```
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

```
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

```
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:

  int *X = new int[100];

- What is sizeof(X)?
- What is sizeof(*X)?

Pitfall #5

```
/* 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 */
}
```

```
fawcett:330 childs$ g++ write_image.c
Undefined symbols:
  "WriteImage(Image*, char const*)", referenced from:
      _main in ccSjC6w2.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

(not-a-)Pitfall #6

```
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;
}
```

```
unsigned char* Image::getPixel(int i, int j) {
    int pixStart = 3*i*this->width+3+j;
    return this->data+pixStart;
}
```

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

```
#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;
}
```

const

const

- const:
  - is a keyword in C and C++
  - qualifies variables
  - is a mechanism for preventing write access to variables

const example

```
fawcett:330 childs$ cat const1.C
int main()
{
    const int X = 5;

}
```

const keyword modifies int

The compiler enforces const … just like public/private access controls

Efficiency

```
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.

```
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 pointe

- 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.

nst pointer

```
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 p

int X

int *P

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;

<u>Idea #1:</u>

violates const:

"*P = 3;"

OK:

"int Y = 5; P = &Y;"

pointer <u>can</u> change, but you <u>can't</u> modify the thing it points to

```
fawcett:330 childs$ cat const3.C
int main()
{
     int X = 5;
     int Y = 6;
     const int *P;
     P  = &X;   // compiles
     P  = &Y;   // compiles
     *P = 7;    // won't compiles
}
fawcett:330 childs$ g++ const3.C
const3.C: In function 'int main()':
const3.C:8: error: assignment of read-only location
```

const goes before type

const pointers

int X = 4;

int *P = &X;

```
fawcett:330 childs$ cat const4.C
int main()
{
    int X = 5;
    int Y = 6;
    int * const P = &X; // must initialize
    *P = 7;     // compiles
    P  = &Y;    // won't compile
}
fawcett:330 childs$ g++ const4.C
const4.C: In function 'int main()':
const4.C:7: error: assignment of read-only variable 'P'
```

const goes after *

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 in both places

```
fawcett:330 childs$ cat const5.C
int main()
{
    int X = 5;
    int Y = 6;
    const int * const P = &X; // must initialize
    *P = 7;     // won't compile
    P  = &Y;    // won't compile
}
fawcett:330 childs$ g++ const5.C
const5.C: In function 'int main()':
const5.C:6: error: assignment of read-only location
const5.C:7: error: assignment of read-only variable 'P'
```

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

```
fawcett:330 childs$ cat const6.C
class Image
{
  public
    int
  
  priva
    int
};

unsigne
Allocat
{
    int
    unsi
    return rv;
}
```

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

```
fawcett:330 childs$ cat const7.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;
}
fawcett:330 childs$ g++ -c const7.C
fawcett:330 childs$ 
```

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.

globals

globals

- You can create global variables that exist outside functions.

```
fawcett:Documents childs$ cat global1.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

```
fawcett:Documents childs$ cat global2.C

#include <stdio.h>

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

int X = Initializer();

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

fawcett:Documents childs$ g++ global2.C
fawcett:Documents childs$ ./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

```
fawcett:Documents childs$ cat file1.C
int X = 6;

int main()
{
}
fawcett:Documents childs$ g++ -c file1.C
fawcett:Documents childs$ cat file2.C
int X = 7;

int doubler(int Y)
{
   return 2*Y;
}
fawcett:Documents childs$ g++ -c file2.C
fawcett:Documents childs$ g++ file1.o file2.o
ld: duplicate symbol _X in file2.o and file1.o
collect2: ld returned 1 exit status
```

Externs: mechanism for unifying global variables across multiple files

```
fawcett:330 childs$ cat file1.C

#include <stdio.h>

int count = 0;

int doubler(int);

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

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

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

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

```
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());
}
```

```
fawcett:330 childs$ g++ static1.C
fawcett:330 childs$ ./a.out
1
2
3
5
8
13
21
34
55
89
```

static usage #2: making global variables be local to a file

I have no idea why the static keyword is used in this way.

```
faw

#include <stdio.h>

static int count = 0;

int doubler(int);

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;
}
```

```
fawcett:330 childs$ g++ -c file2.C
fawcett:330 childs$ g++ file1.o file2.o
fawcett:330 childs$ ./a.out
count is 1
```

static usage #3: making a singleton for a class

```
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

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

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

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

   int       GetNumInstar

  private:
   static int       numInstances;
};

i
{
```

```
    delete [] p;
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
```

```
fawcett:Downloads childs$ g++ static3.C
Undefined symbols:
  "MyClass::numInstances", referenced from:
      MyClass::MyClass()in ccoao8Hf.o
      MyClass::MyClass()in ccoao8Hf.o
      MyClass::GetNumInstances()      in ccoao8Hf.o
      MyClass::~MyClass()in ccoao8Hf.o
      MyClass::~MyClass()in ccoao8Hf.o
      MyClass::~MyClass()in ccoao8Hf.o
      MyClass::~MyClass()in ccoao8Hf.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

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

What do we get?

static usage #3: making a singleton for a class

```
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:
   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

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

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

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

   static int       GetNumInstances(void) { return numInstances; };

  private:
   static int       numInstances;
};

int MyClass::numInstances = 0;

int main()
{
    MyClass *p = new MyClass[10];
    cout << "Num instances = " << MyClass::GetNumInstances() << endl;
    delete [] p;
    cout << "Num instances = " << MyClass::GetNumInstances() << endl;
}
fawcett:Downloads childs$ g++ static3.C
fawcett:Downloads childs$ ./a.out
Num instances = 10
Num instances = 0
```

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

Scope

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".

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

  private:
    int   X;
};

void MyClass::SetValue(int X)
{
   X = X;
}
```

scope

```
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 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);
}
```

scope

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);
}
```

scope

What does this one print?

Answer: 2

scope

```
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

scope

```
int X = 0;

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

    void SetValue(int);

  private:
    int    X;                     // struck through

};

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


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

What does this one print?

Answer: 0

Scope Rules

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

Pitfall #8

```
#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];
}
```

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

```
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

```
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();
}
```

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;
}
```

Declare operator ++ will be overloaded for MyInt

Define operator ++ for MyInt

```
int main()
{
   MyInt mi(6);
   ++mi;
   ++mi;
   printf("Value is %d\n", mi.GetValue());
}
fawcett:330 childs$ ./a.out
Value is 8
```

Call operator ++ on MyInt.

More operator overloading

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

```
int main()
{
    Image img;
    ofstream ofile("output_file");
    ofile << img;
}
fawcett:330 childs$ g++ oostream.C
fawcett:330 childs$ ./a.out
fawcett:330 childs$ cat output_file
100x100
No buffer allocated!
```

assignment operator

```
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];
}
```

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

```
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…

```
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 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 0x10081e600
```

(ok, fine)

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$
```

it still compiled … why?

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 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

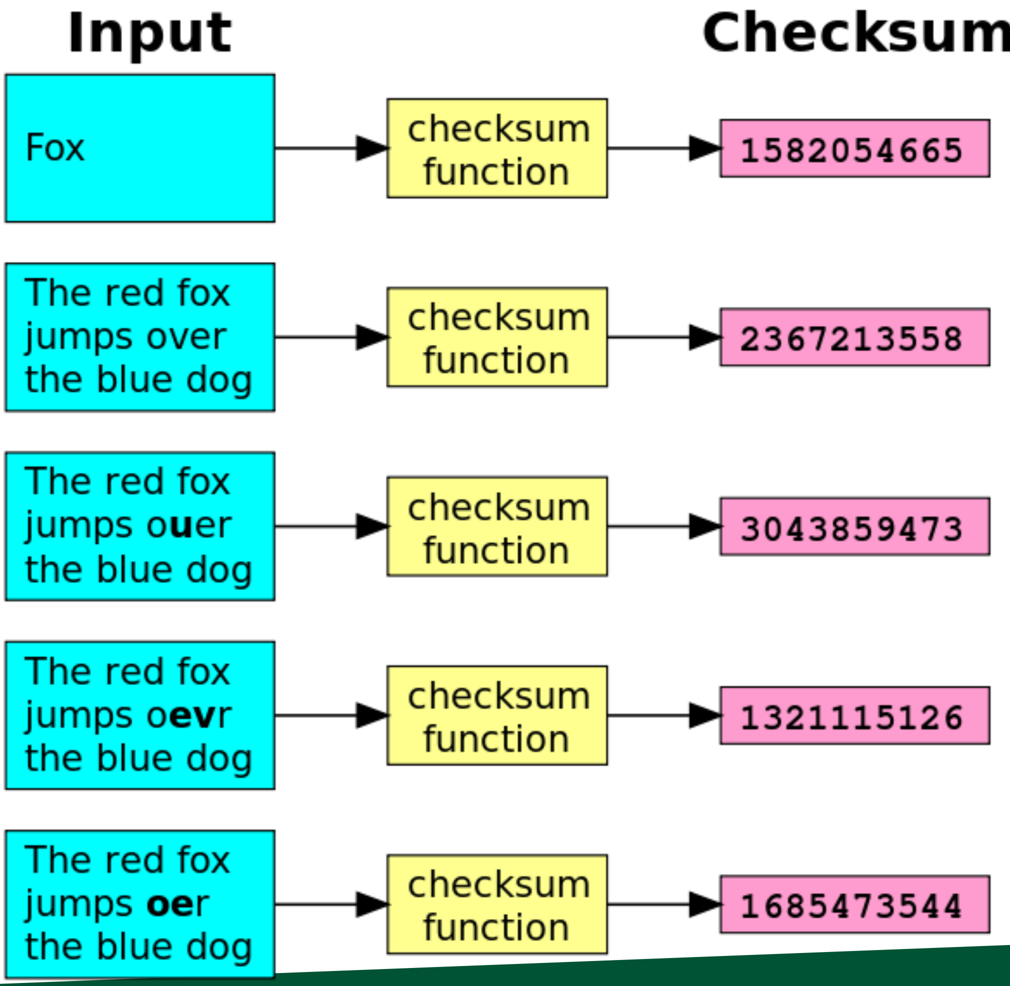
Project 3G

- 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

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

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

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

```
fawcett:330 childs$ g++ default.C
fawcett:330 childs$ ./a.out
X = 5, Y = 2
X = 5, Y = 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

```
int main()
{
    bool b = true;
    cout << "Size of boolean is " << sizeof(bool) << endl;
}
fawcett:330 childs$ g++ Boolean.C
fawcett:330 childs$ ./a.out
```

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)