Lecture 12: more class, C++ memory management
Announcements

• Projects
  – 3B assigned Monday 5/1, due Tuesday 5/9
  – 3C posted tomorrow, due May 17
  – 3D posted Weds, also due May 17
    • 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
Announcements

• Valgrind on ix reports spurious leak with C++
  – Hank will investigate

• I no longer expect “pixel correct” for 3B
  – Floating point precision issues will derail us

• For Proj3, it is very important that you use my interface
  – Do not modify the files I tell you not to modify
  – If you do modify the files, it will be quite painful when I had you ~60 regression tests that assume the interface I have been providing
Project 3B

• Retrofit to use references
• Add useful routines for manipulating an image
  – Halve in size
  – Concatenate
  – Crop
  – Blend
• Assigned: May 1st
• Due: tomorrow
Review
3 Big changes to structs in C++

1) You can associate “methods” (functions) with structs
Methods vs Functions

• Methods and Functions are both regions of code that are called by name (“routines”)

• With functions:
  – the data it operates on (i.e., arguments) are explicitly passed
  – the data it generates (i.e., return value) is explicitly passed
  – stand-alone / no association with an object

• With methods:
  – associated with an object & can work on object’s data
  – still opportunity for explicit arguments and return value
Function vs Method

(left) function is separate from struct  
(right) function (method) is part of struct

C02LN00GFD58:330 hank$ cat function.c
typedef struct
{     int i;
 } Integer;

int doubler(int x) { return 2*x; };

int main()
{     Integer i;
     i.i = 3;
     i.i = doubler(i.i);
}

typedef struct
{     int i;
     void doubler(void) { i = 2*i; }
 } Integer;

int main()
{     Integer i;
     i.i = 3;
     i.doubler();
}

(left) arguments and return value are explicit  
(right) arguments and return value are not necessary, since they are associated with the object.
Tally Counter

3 Methods:
- Increment Count
- Get Count
- Reset
Methods & Tally Counter

• Methods and Functions are both regions of code that are called by name (“routines”)

• With functions:
  – the data it operates on (i.e., arguments) are explicitly passed
  – the data it generates (i.e., return value) is explicitly passed
  – stand-alone / no association with an object

• With methods:
  – associated with an object & can work on object’s data
  – still opportunity for explicit arguments and return value
C++-style implementation of TallyCounter

```c
#include <stdio.h>

typedef struct
{
    int    count;
    
    void   Reset() { count = 0; };
    int    GetCount() { return count; };
    void   IncrementCount() { count++; };
} TallyCounter;

int main()
{
    TallyCounter tc;
    tc.count = 0;
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is \%d\n", tc.GetCount());
}
```

```bash
hank$ cat tallycounter.C
hank$ g++ tallycounter.C
hank$ ./a.out
Count is 4
```
typedef struct
{
    int    count;
    void   Initialize() { count = 0; }
    void   Reset() { count = 0; }
    int    GetCount() { return count; }
    void   IncrementCount() { count++; }
} TallyCounter;

int main()
{
    TallyCounter tc;
    tc.Initialize();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is %d\n", tc.GetCount());
}
Constructors

• Constructor: method for constructing object.
  – Called automatically

• There are several flavors of constructors:
  – Parameterized constructors
  – Default constructors
  – Copy constructors
  – Conversion constructors
Note the typedef went away ... not needed with C++.  

(This is the flavor called “default constructor”)

typedef struct
{
    int count;
    void Initialize() { count = 0; };
    void Reset() { count = 0; };
    int GetCount() { return count; };
    void IncrementCount() { count++; };
} TallyCounter;

int main()
{
    TallyCounter tc;
    tc.Initialize();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is %d\n", tc.GetCount());
}

#include <stdio.h>

struct TallyCounter
{
    int count;
    TallyCounter(void) { count = 0; };
    void Reset() { count = 0; };
    int GetCount() { return count; };
    void IncrementCount() { count++; };
};

int main()
{
    TallyCounter tc;
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    tc.IncrementCount();
    printf("Count is %d\n", tc.GetCount());
}
Argument can be passed to constructor.
(This is the flavor called “parameterized constructor”)

```c
#include <stdio.h>

struct TallyCounter
{
  int count;

  TallyCounter(void) { count = 0; };
  TallyCounter(int c) { count = c; };

  void Reset() { count = 0; };
  int GetCount() { return count; };
  void IncrementCount() { count++; };
};

int main()
{
  TallyCounter tc(10);
  tc.IncrementCount();
  tc.IncrementCount();
  tc.IncrementCount();
  tc.IncrementCount();
  printf("Count is %d\n", tc.GetCount());
}
```
More traditional file organization

• struct definition is in .h file
  – #ifndef / #define
• method definitions in .C file
• driver file includes headers for all structs it needs
More traditional file organization

Methods can be defined outside the struct definition. They use C++’s namespace concept, which is automatically in place. (e.g., TallyCounter::IncrementCount)
“this”: pointer to current object

- From within any struct’s method, you can refer to the current object using “this”
Copy Constructor

- Copy constructor: a constructor that takes an instance as an argument
  - It is a way of making a new instance of an object that is identical to an existing one.

```c
struct TallyCounter
{
    int count;

    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);

    void Reset();
    int GetCount();
    void IncrementCount();
};
```
Constructor Types

```c
struct TallyCounter
{
    int count;
    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);
    void Reset();
    int GetCount();
    void IncrementCount();
};
```

- Default constructor
- Parameterized constructor
- Copy constructor
Conversion Constructor

```cpp
struct ImperialDistance {
    double miles;
};

struct MetricDistance {
    double kilometers;

    MetricDistance() { kilometers = 0; }
    MetricDistance(ImperialDistance &id) {
        kilometers = id.miles * 1.609;
    }
};
```
3 big changes to structs in C++

1) You can associate “methods” (functions) with structs

2) You can control access to data members and methods
• New keywords: public and private
  – public: accessible outside the struct
  – private: accessible only inside the struct
• Also “protected” … we will talk about that later

```c
struct TallyCounter
{
    private:
        int       count;

    public:
        TallyCounter(void);
        TallyCounter(int c);
        TallyCounter(TallyCounter &);
        void     Reset();
        int      GetCount();
        void     IncrementCount();
};
```

Everything following is private. Only will change when new access control keyword is encountered.

Everything following is now public. Only will change when new access control keyword is encountered.
public / private

struct TallyCounter
{
    public:
        TallyCounter(void);
        TallyCounter(int c);
        TallyCounter(TallyCounter &);

    private:
        int count;

    public:
        void Reset();
        int GetCount();
        void IncrementCount();
};

You can issue public and private as many times as you wish...
The compiler prevents violations of access controls.

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

int main()
{
    TallyCounter tc;
    tc.count = 10;
}
```

```
128-223-223-72-wireless:TC hank$ cat main.C
#include <stdio.h>
#include <TallyCounter.h>

int main()
{
    TallyCounter tc;
    tc.count = 10;
}
```

```
128-223-223-72-wireless:TC hank$ make
g++ -I. -c main.C
main.C:7:8: error: 'count' is a private member of 'TallyCounter'
    tc.count = 10;
    ^
./TallyCounter.h:12:12: note: declared private here
    int    count;
    ^
1 error generated.
make: *** [main.o] Error 1
```
The friend keyword can override access controls.

```
struct TallyCounter
{
    friend int main();

public:
    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);

private:
    int count;
```

- Note that the struct declares who its friends are, not vice-versa
  - You can’t declare yourself a friend and start accessing data members.
- friend is used most often to allow objects to access other objects.

This will compile, since main now has access to the private data member “count”.

class vs struct

• class is new keyword in C++
• classes are very similar to structs
  – the only differences are in access control
    • primary difference: struct has public access by default, class has private access by default
• Almost all C++ developers use classes and not structs
  – C++ developers tend to use structs when they want to collect data types together (i.e., C-style usage)
  – C++ developers use classes for objects … which is most of the time

You should use classes!
Even though there isn’t much difference …
3 big changes to structs in C++

1) You can associate “methods” (functions) with structs
2) You can control access to data members and methods
3) Inheritance
New Stuff
Simple inheritance example

```c
struct A {
    int x;
};

struct B : A {
    int y;
};

int main() {
    B b;
    b.x = 3;
    b.y = 4;
}
```

- **Terminology**
  - B inherits from A
  - A is a base type for B
  - B is a derived type of A

- **Noteworthy**
  - "::" (during struct definition) → inherits from
    - Everything from A is accessible in B
      - (b.x is valid!!)
Object sizes

#include <stdio.h>

struct A {
  int x;
};

struct B : A {
  int y;
};

int main() {
  B b;
  b.x = 3;
  b.y = 4;
  printf("Size of A = %lu, size of B = %lu\n", sizeof(A), sizeof(B));
}

128-223-223-72-wireless:330 hank$ g++ simple_inheritance.C
128-223-223-72-wireless:330 hank$ .a.out
Size of A = 4, size of B = 8
struct TallyCounter
{
    friend    int main();

public:
    TallyCounter(void);
    TallyCounter(int c);
    TallyCounter(TallyCounter &);

private:
    int    count;

public:
    void    Reset();
    int    GetCount();
    void    IncrementCount();
};

struct FancyTallyCounter : TallyCounter
{
    void    DecrementCount() { count--; }
}
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
```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());
}
```

```
Virtual functions: example
```

```
#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());
}
```

```
Virtual functions: example
```

```
128-223-223-72-wireless:330 hank$ g++ virtual.C
128-223-223-72-wireless:330 hank$ ./a.out
ID = 387
```
Virtual functions: example

You get the method furthest down in the inheritance hierarchy

```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; };
};

struct C3 : ComplexID
{
    int extraExtraId;
};

int main()
{
    C3 cid;
    cid.id = 3;
    cid.extraId = 3;
    cid.extraExtraId = 4;
    printf("ID = %d\n", cid.GetIdentifier());
}
```
Virtual functions: example

You can specify the method you want to call by specifying it explicitly
public / private inheritance

• class A : [public|private] B
  – → class A : public B
  – → class A : private B

• So:
  – For public, base class's public members will be public
  – For private, base class's public members will be private

• Public common
  – I’ve never personally used anything else
public / private inheritance

• class A : public B
  – A “is a” B

• class A : private B
  – A “is implemented using” B
    • And: !(A “is a” B)
    • ... you can’t treat A as a B
Access controls and inheritance

B and C are the same. public is the default inheritance for structs.

Public inheritance: derived types gets access to base type’s data members and methods

Private inheritance: derived types don’t get access.
One more access control word: protected

• Protected means:
  – It cannot be accessed outside the object
    • Modulo “friend”
  – But it can be accessed by derived types
    • (assuming public inheritance)
<table>
<thead>
<tr>
<th>Access Level</th>
<th>Accessed by derived types*</th>
<th>Accessed outside object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Protected</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Private</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

* = with public inheritance
protected example

```c

class A
{
  protected:
    int x;
};

class B : public A
{
  public:
    int foo() { return x; }
};

int main()
{
  B b;
  b.x = 2;
  int y = b.foo();
}
```

128-223-223-73-wireless:CV hank$ g++ protected.C
protected.C:16:7: error: 'x' is a protected member of 'A'
b.x = 2;
^

protected.C:4:9: note: declared protected here
int x;
^
1 error generated.
protected inheritance

• class A : [public | protected | private] B

• class A : protected B
  – .... can’t find practical reasons to do this
More on virtual functions upcoming

• “Is A”
• Multiple inheritance
• Virtual function table
• Examples
  – (Shape)
Memory Management
C memory management

• **Malloc**: request memory manager for memory from heap
• **Free**: tell memory manager that previously allocated memory can be returned

• All operations are in bytes
  
  ```c
  Struct *image = malloc(sizeof(image)*1);
  ```
C++ memory management

• C++ provides new constructs for requesting heap memory from the memory manager
  – stack memory management is not changed
    • (automatic before, automatic now)

• Allocate memory: “new”

• Deallocate memory: “delete”
new / delete syntax

No header necessary

Allocating array and single value is the same.

new knows the type and allocates the right amount.

new int → 4 bytes
new int[3] → 12 bytes
new calls constructors for your classes

• Declare variable in the stack: constructor called
• Declare variable with “malloc”: constructor not called  
  – C knows nothing about C++!
• Declare variable with “new”: constructor called
new calls constructors for your classes

```c
#include <stdio.h>

int counter = 0;
class Counter
{
    public:
        Counter() { counter++; }
}

void PrintCount(char *location)
{
    printf("Count at %s is %d\n", location, counter);
}

int main()
{
    PrintCount("beginning");
    Counter c;
    PrintCount("after one");
    Counter *c2 = new Counter;
    PrintCount("after heap one");
    Counter *c3 = new Counter[10];
    PrintCount("after heap ten");
    Counter **c4 = new Counter*[10];
    PrintCount("after heap-pointer-ten");
    for (int i = 0 ; i < 10 ; i++)
    {
        c4[i] = new Counter;
    }
    PrintCount("after allocating heap-pointer-ten");
}
```

fawcett:330 childs$ ./a.out
Count at beginning is 0
Count at after one is 1
Count at after heap one is 2
Count at after heap ten is 12
Count at after heap-pointer-ten is 12
Count at after allocating heap-pointer-ten is 22
new & malloc

• Never mix new/free & malloc/delete.
• They are different & have separate accesses to heap.
• New error code: FMM (Freeing mismatched memory)
More on Classes
Destructors

• A destructor is called automatically when an object goes out of scope (via stack or delete)
• A destructor’s job is to clean up before the object disappears
  – Deleting memory
  – Other cleanup (e.g., linked lists)
• Same naming convention as a constructor, but with a prepended ~ (tilde)
Destructors example

```cpp
struct Pixel
{
    unsigned char R, G, B;
};

class Image
{
    public:
        Image(int w, int h);
        ~Image();

    private:
        int width, height;
        Pixel *buffer;
};

Image::Image(int w, int h)
{
    width = w; height = h;
    buffer = new Pixel[width*height];
}

Image::~Image()
{
    delete [] buffer;
}
```

Class name with ~ prepended

Defined like any other method, does cleanup

If Pixel had a constructor or destructor, it would be getting called (a bunch) by the new’s and delete’s.
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

• With a constructor, you always know what type you are constructing.
• With a destructor, you don’t always know what type you are destructing.
• This can sometimes lead to the wrong destructor getting called.
Getting the wrong destructor

#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"); }
};

D* D_as_D_Creator() { return new D; }
C* D_as_C_Creator() { return new D; }

int main()
{
  C* c = D_as_C_Creator();
  D* d = D_as_D_Creator();

  delete c;
  delete d;
}
Virtual destructors

• Solution to this problem:
  Make the destructor be declared virtual
• Then existing infrastructure will solve the problem
  – ... this is what virtual functions do!
Virtual destructors

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

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

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

D* D_as_D_Creator() { return new D; }
C* D_as_C_Creator() { return new D; }

int main()
{
  C* c = D_as_C_Creator();
  D* d = D_as_D_Creator();

  delete c;
  delete d;
}```
Virtual inheritance is forever

#include <stdio.h>

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

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

D* D_as_D_Creator() { return new D; }
C* D_as_C_Creator() { return new D; }

int main()
{
  C* c = D_as_C_Creator();
  D* d = D_as_D_Creator();

delete c;
delete d;
}
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;
    }
}
```

By the time you enter B’s constructor, a1 and a2 are already valid.
Objects in objects

```cpp
#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;
}
```

```
fawcett:330 childs$ ./a.out
Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A
```
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
Quiz

#include <stdio.h>

int doubler(int X)
{
    printf("In doubler\n");
    return 2*X;
}

class A
{
    public:
        A(int x) { printf("In A's constructor\n"); }
};

class B : public A
{
    public:
        B(int x) : A(doubler(x)) { printf("In B's constructor\n"); }
};

int main()
{
    B b(3);
}

What’s the output?
The “is a” test

- Inheritance should be used when the “is a” test is true
- Base class: Shape
- Derived types: Triangle, Rectangle, Circle
  - A triangle “is a” shape
  - A rectangle “is a” shape
  - A circle “is a” shape

You can define an interface for Shapes, and the derived types can fill out that interface.

I will do a live coding example of this next week, and will discuss how C++ implements virtual functions.
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
Multiple inheritance

class Professor
{
    void Teach();
    void Grade();
    void Research();
};

class Father
{
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor
{
};
Diamond-Shaped Inheritance

• Base A, has derived types B and C, and D inherits from both B and C.
  – Which A is D dealing with??

• Diamond-shaped inheritance is controversial & really only for experts
  – (For what it is worth, we make heavy use of diamond-shaped inheritance in my project)
Diamond-Shaped Inheritance Example

class Person
{
    int X;
};

class Professor : public Person
{
    void Teach();
    void Grade();
    void Research();
};

class Father : public Person
{
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor
{ 
};
Diamond-Shaped Inheritance Pitfalls

```c
#include <stdio.h>

class Person
{
    public:
        Person(int h) { hoursPerWeek = h; };
    protected:
        int hoursPerWeek;
};

class Professor : public Person
{
    public:
        Professor() : Person(90) { ; };
    void Teach();
    void Grade();
};

class Hank : public Father, public Professor
{
    public:
        int GetHoursPerWeek() { return hoursPerWeek; };
};

int main()
{
    Hank hrc;
    printf("HPW = %d\n", hrc.GetHoursPerWeek());
}
```

```
fawcett:330 childs$ g++ diamond_inheritance.C
diamond_inheritance.C: In member function ‘int Hank::GetHoursPerWeek()’: 
diamond_inheritance.C:31: error: reference to ‘hoursPerWeek’ is ambiguous
diamond_inheritance.C:8: error: candidates are: int Person::hoursPerWeek
diamond_inheritance.C:8: error: int Person::hoursPerWeek
```

Diamond-Shaped Inheritance Pitfalls

This can get stickier with virtual functions.

You should avoid diamond-shaped inheritance until you feel really comfortable with OOP.
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
Pure Virtual Functions Example

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 child$ 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()
Data Flow Networks
Data Flow Networks

• Idea:
  – Many modules that manipulate data
    • Called filters
  – Dynamically compose filters together to create “networks” that do useful things
  – Instances of networks are also called “pipelines”
    • Data flows through pipelines
  – There are multiple techniques to make a network “execute” ... we won’t worry about those yet
Data Flow Network: the players

- **Source**: produces data
- **Sink**: accepts data
  - Never modifies the data it accepts, since that data might be used elsewhere
- **Filter**: accepts data and produces data
  - A filter “is a” sink and it “is a” source

Source, Sink, and Filter are abstract types. The code associated with them facilitates the data flow.

There are concrete types derived from them, and they do the real work (and don’t need to worry about data flow!).
Project 2C
Assignment: make your code base be data flow networks with OOP
More on virtual functions upcoming

- “Is A”
- Multiple inheritance
- Virtual function table
- Examples
  - (Shape)
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)