Lecture 12:
C++ memory management,
Transitioning to Classes, Part 2
Outline

• Review
• Project 3B
• C++ memory management
• More on classes
• Data Flow Networks
• Project 3C
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
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
(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
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)
Constructors

• Constructor: method for constructing object.
  – Called automatically

• There are several flavors of constructors:
  – Parameterized constructors
  – Default constructors
  – Copy constructors
  – Conversion constructors
Access Control

- New keywords: public and private
  - public: accessible outside the struct
  - private: accessible only inside the struct
- Also “protected”

```cpp
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.
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$ make
make: *** [main.o] Error 1
```
The friend keyword can override access controls.

```cpp
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 allows “main” to have access to the private data member “count”.
class vs struct

• class is a new keyword in C++
• classes are very similar to structs
  – the only differences are in default 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 …
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!!)
Public, private, protected

<table>
<thead>
<tr>
<th>Accessed by derived types*</th>
<th>Accessed outside object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Yes</td>
</tr>
<tr>
<td>Protected</td>
<td>Yes</td>
</tr>
<tr>
<td>Private</td>
<td>No</td>
</tr>
</tbody>
</table>

* = with public inheritance
protected example (new slide)

```c
fawcett:330 child$ cat protected.C
class A
{
    protected:
        int x;
};

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

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

fawcett:330 child$ g++ protected.C
protected.C: In function ‘int main()’:  
protected.C:4: error: ‘int A::x’ is protected
protected.C:15: error: within this context
protected.C:16: error: ‘foo’ was not declared in this scope
public / private inheritance (new slide)

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

• For P, base class's public members will be P
• e.g.,
  – For public, base class's public members will be public

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

• 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

• class A : protected B
  – .... can’t find practical reasons to do this
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

You can specify the method you want to call by specifying it explicitly.
Outline

• Review
• Project 3B
• C++ memory management
• More on classes
• Data Flow Networks
• Project 3C
Project 3B

• Add useful routines for manipulating an image
  – Halve in size
  – Concatenate
  – Crop
  – Blend
• Assigned: yesterday
• Due: Tuesday
Outline

• Review
• Project 3B
• C++ memory management
• More on classes
• Data Flow Networks
• Project 3C
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”
• Allocate memory: “delete”
**new / delete syntax**

```
fawcett:330 child$ cat new.C
int main()
{
    int *oneInt = new int;
    *oneInt = 3;
    int *intArray = new int[3];

    delete oneInt;
    delete [] intArray;
}
```

No header necessary

Allocating array and single value is the same.

new knows the type and allocates the right amount.

Deleting array takes [], deleting single value doesn’t.

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");
}
```

```bash
fawcett:330 childsd$ ./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)
Outline

• Review
• Project 3B
• C++ memory management
• More on classes
• Data Flow Networks
• Project 3C
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;
    }
};
```

Defined like any other method, does cleanup

Class name with ~ prepended

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
Inheritance and Constructors/Destructors: Example

```cpp
#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;
  }
}```
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

```c
#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;
}
```

```
fawcett:330 childds$ ./a.out
Constructing C
Constructing D
Constructing C
Constructing D
Destructing C
Destructing D
Destructing C
```
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!
#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;
}

I didn’t need to put virtual there. If the base class has a virtual function, then the derived function is virtual, whether or not you put the keyword in.

I recommend you still put it in ... it is like a comment, reminding anyone who looks at the code.
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

```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:
        A a;
        B b;
};

int main()
{
    C 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;
}
```

```
fawcett:330 childs$ ./a.out
x is 5
```
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
  - Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class
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

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

class C : public A
{
public:
  C(int x, int y) : A(y), z(x) { }
private:
  int z;
};

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

```java
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; };
    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());
}
```

Shell command:
```
fawcett:330 child$ g++ diamond_inheritance.C
```

Error messages:
```
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-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 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()
Outline

• Review
• Project 3B
• C++ memory management
• More on classes
• Data Flow Networks
• Project 3C
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!).
Outline

• Review
• Project 3B
• C++ memory management
• More on classes
• Data Flow Networks
• Project 3C
Assignment: make your code base be data flow networks with OOP
Project 3C

CIS 330: Project #3B
Assigned: May 7th, 2014
Due May 14th, 2014
(which means submitted by 6am on May 15th, 2014)
Worth 5% of your grade

Please read this entire prompt!

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

== New code available on the website ==

=== main3C.C ===

Start with my main3C.C. It shows what the interfaces should be for the modules. Do not modify my main3C.C, aside from adding “#includes” and print statements (if you want).

=== Makefile ===

I added another Makefile. Note that I put all of my filters in a file called “filters.C” / “filters.h”. I didn’t think it was worth splitting them into separate files.
Bonus Slides
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!!
Unix and Windows difference

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

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

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

vi: “set ff=unix” solves this
memcpy

NAME
memcpy -- copy memory area

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <string.h>

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

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

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

I mostly use C++, and I still use memcpy all the time