CSCI 104
Classes
Mark Redekopp
David Kempe
CLASSES
C Structs

• Needed a way to group values that are related, but have different data types

• NOTE: struct has changed in C++!
  - C
    • Only data members
    • Some declaration nuances
  - C++
    • Like a class (data + member functions)
    • Default access is public

```c
struct Person{
    char name[20];
    int age;
};

int main()
{
    // Anyone can modify
    // b/c members are public
    Person p1;
    p1.age = -34;
    // probably not correct
    return 0;
}
```
Classes & OO Ideas

• In object-oriented programming languages (C++) classes are used as the primary way to organize code

• Encapsulation
  – Place data and operations on data into one code unit
  – Keep state hidden/separate from other programmers via private members

• Abstraction
  – Depend only on an interface!
    • Ex. a microwave...Do you know how it works? But can you use it?
  – Hide implementation details to create low degree of coupling between different components

• Polymorphism & Inheritance
  – More on this later...

struct Machine{
    Piece* pieces;
    Engine* engine;
};

int main()
{
    Machine m;
    init_subsystemA(&m);
    change_subsystemB(&m);
    replace_subsystemC(&m);
    m.start();
    // Seg. Fault!! Why?
}

Protect yourself from users & protect your users from themselves
Coupling

• Coupling refers to how much components depend on each other's implementation details (i.e. how much work it is to remove one component and drop in a new implementation of it)
  – Placing a new battery in your car vs. a new engine
  – Adding a USB device vs. a new video card to your laptop

• OO Design seeks to reduce coupling as much as possible by
  – Creating well-defined interfaces to change (write) or access (read) the state of an object
  – Enforcing those interfaces are adhered to
    • Private vs. public
  – Allow alternate implementations that may be more appropriate for different cases
C++ Classes

• A composition mechanism
  – Create really large and powerful software systems from tiny components
  – Split things up into manageable pieces
    • Somewhat of a bottom up approach (define little pieces that can be used to compose larger pieces)
  – Delegation of responsibility

• An abstraction and encapsulation mechanism
  – Make functionality publicly available, but hide data & implementation details

• A mechanism for polymorphism
  – More on this later
C++ Classes: Overview

• What are the main parts of a class?
  – Member variables
    • What data must be stored?
  – Constructor(s)
    • How do you build an instance?
  – Member functions
    • How does the user need to interact with the stored data?
  – Destructor
    • How do you clean up an after an instance?
C++ Classes: Overview

• Member data can be public or private (for now)
  – Defaults is private (only class functions can access)
  – Must explicitly declare something public

• Most common C++ operators will not work by default (e.g. ==, +, <<, >>, etc.)
  – You can't cout an object (cout << myobject; won't work)
  – The only one you get for free is '=' and even that may not work the way you want (more on this soon)

• Classes may be used just like any other data type (e.g. int)
  – Get pointers/references to them
  – Pass them to functions (by copy, reference or pointer)
  – Dynamically allocate them
  – Return them from functions
this Pointer

- How do member functions know which object’s data to be operating on?
- d1 is implicitly passed via a special pointer call the ‘this’ pointer

```cpp
#include <iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
    Deck d1, d2;
    d1.shuffle();
    d1.shuffle();
    ...
}
```

```cpp
#include <iostream>
#include "deck.h"

void Deck::shuffle() {
    cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++) {
        int r = rand() % (52-i);
        int temp = cards[r];
        cards[r] = cards[i];
        cards[i] = temp;
    }
}
```

```
#include <iostream>
#include "deck.h"

void Deck::shuffle(Deck *this) {
    this->cut(); // calls cut()
    // for this object
    for(i=0; i < 52; i++) {
        int r = rand() % (52-i);
        int temp = this->cards[r];
        this->cards[r] = this->cards[i];
        this->cards[i] = temp;
    }
}
```
Exercises

• cpp/cs104/classes/this_scope
Another Use of 'this'

- This can be used to resolve scoping issues with similar named variables

```cpp
class Student {
public:
    Student(string name, int id, double gpa);
~Student(); // Destructor
private:
    string name;
    int id;
    double gpa;
};

Student::Student(string name, int id, double gpa)
{ // which is the member and which is the arg?
    name = name; id = id; gpa = gpa;
}

Student::Student(string name, int id, double gpa)
{ // Now it's clear
    this->name = name;
    this->id = id;
    this->gpa = gpa;
}
```
C++ Classes: Constructors

• Called when a class is instantiated
  – C++ won't automatically initialize member variables
  – No return value

• Default Constructor
  – Can have one or none in a class
  – Basic no-argument constructor
  – Has the name ClassName()
  – If class has no constructors, C++ will make a default
    • But it is just an empty constructor (e.g. Item::Item() { } )

• Overloaded Constructors
  – Can have zero or more
  – These constructors take in arguments
  – Appropriate version is called based on how many and what type of arguments are passed when a particular object is created
  – If you define a constructor with arguments you should also define a default constructor

```cpp
class Item {
  int val;
public:
  Item(); // default const.
  Item(int v); // overloaded
};```
Identify that Constructor

- Prototype what constructors are being called here

```cpp
#include <string>
#include <vector>
using namespace std;

int main()
{
    string s1;
    string s2("abc");
    vector<int> dat(30);
    return 0;
}
```
Identify that Constructor

• Prototype what constructors are being called here

• s1
  – string::string()
    // default constructor

• s2
  – string::string(const char*)

• dat
  – vector<int>::vector<int>( int );

```cpp
#include <string>
#include <vector>
using namespace std;

int main()
{
    string s1;
    string s2("abc");
    vector<int> dat(30);
    return 0;
}
```
Exercises

• cpp/cs104/classes/constructor_init
Consider this Struct/Class

- Examine this struct/class definition...

```cpp
#include <string>
#include <vector>
using namespace std;

struct Student
{
    string name;
    int id;
    vector<double> scores;
    // say I want 10 test scores per student
};

int main()
{
    Student s1;
}
```
Composite Objects

- Fun Fact: Memory for an object comes alive before the code for the constructor starts at the first curly brace '{'

```cpp
#include <string>
#include <vector>
using namespace std;

struct Student
{
    string name;
    int id;
    vector<double> scores;
    // say I want 10 test scores per student

    Student() /* mem allocated here */
    {
        // Can I do this to init. members?
        name("Tommy Trojan");
        id = 12313;
        scores(10);
    }
};

int main()
{
    Student s1;
}
```
Composite Objects

• You cannot call constructors on data members once the constructor has started (i.e. passed the open curly '{' )
  – So what can we do??? Use initialization lists!

```cpp
#include <string>
#include <vector>
using namespace std;

struct Student
{
    string name;
    int id;
    vector<double> scores;
    // say I want 10 test scores per student

    Student() /* mem allocated here */
    {
        // Can I do this to init. members?
        name("Tommy Trojan");
        id = 12313;
        scores(10);
    }
};

int main()
{
    Student s1;
}
```

This would be "constructing" name twice. It's too late to do it in the {…}
Constructor Initialization Lists

```cpp
Student::Student()
{
    name = "Tommy Trojan";
    id = 12313
    scores.resize(10);
}
```

```cpp
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
    name = "Tommy Trojan";
    id = 12313
    scores.resize(10);
}
```

If you write this...
The compiler will still generate this.

- Though you do not see it, realize that the **default constructors** are implicitly called for each data member before entering the {...}

- You can then assign values but this is a **2-step** process
Constructor Initialization Lists

Rather than writing many assignment statements we can use a special initialization list technique for C++ constructors:

- Constructor(param_list) : member1(param/val), ..., memberN(param/val)
  { ... }

We are really calling the respective constructors for each data member.
Constructor Initialization Lists

You can still assign data members in the {...}

But any member not in the initialization list will have its default constructor invoked before the {...}

- You can still assign values in the constructor but realize that the default constructors will have been called already

- So generally if you know what value you want to assign a data member it's good practice to do it in the initialization list

```cpp
Student::Student() {
    name = "Tommy Trojan";
    id = 12313
    scores.resize(10);
}
```
Exercises

• cpp/cs104/classes/constructor_init2
Member Functions

- Have access to all member variables of class
- Use “const” keyword if it won't change member data
- Normal member access uses dot (.) operator
- Pointer member access uses arrow (->) operator

```cpp
class Item
{
    int val;
    public:
        void foo();
        void bar() const;
};

void Item::foo() // not Foo()
{
    val = 5; }

void Item::bar() const
{
    }

int main()
{
    Item x;
    x.foo();
    Item *y = &x;
    (*y).bar();
    y->bar(); // equivalent
    return 0;
}
```
Exercises

• cpp/cs104/classes/const_members
• cpp/cs104/classes/const_members2
• cpp/cs104/classes/const_return
C++ Classes: Destructors

• Called when a class goes out of scope or is freed from the heap (by “delete”)
• Why use it?
  – Not necessary in simple cases
  – Clean up resources that won't go away automatically (e.g. stuff you used “new” to create in your class member functions or constructors
• Destructor
  – Has the name ~ClassName()
  – Can have one or none
  – No return value
  – Destructor (without you writing any code) will automatically call destructor of any data member objects...but NOT what data members point to!
    • You only need to define a destructor if you need to do more than that (i.e. if you need to release resources, close files, deallocate what pointers are point to, etc.)
C++ Classes: Other Notes

• Classes are generally split across two files
  – ClassName.h – Contains interface description
  – ClassName.cpp – Contains implementation details

• Make sure you remember to prevent multiple inclusion errors with your header file by using #ifndef, #define, and #endif

```cpp
#ifndef CLASSNAME_H
#define CLASSNAME_H

class ClassName { ... };
#endif
```

```cpp
#ifndef ITEM_H
#define ITEM_H

class Item
{ int val;
 public:
  void foo();
  void bar() const;
};
#endif
```

```cpp
#include "item.h"

void Item::foo()
{ val = 5; }

void Item::bar() const
{ }
```
CONDITIONAL COMPILATION
Multiple Inclusion

- Often separate files may include's of the same header file
- This may cause compiling errors when a duplicate declaration is encountered
  - See example
- Would like a way to include only once and if another attempt to include is encountered, ignore it
Conditional Compiler Directives

- Compiler directives start with '#'
  - `#define XXX`
    - Sets a flag named XXX in the compiler
  - `#ifdef, #ifndef XXX … #endif`
    - Continue compiling code below until #endif, if XXX is (is not) defined
- Encapsulate header declarations inside a
  - `#ifndef XX
    #define XX
    …
    #endif`

```c
#include "string.h"
class Widget{
  public:
    string s;
};
```

```c
#include "string.h"
#include "string.h"
```

```c
#include "string.h"
#define STRING_H
class string{
  ... }
#include "string.h"
```

```c
#include "string.h"
class Widget{
  // inc. from widget.h
  ... 
  class string{ // inc. from string.h 
    // inc. from string.h }
};
```

String.h

Character.h

main.cpp after preprocessing
Conditional Compilation

- Often used to compile additional DEBUG code
  - Place code that is only needed for debugging and that you would not want to execute in a release version
- Place code in a `#ifdef` XX...`#endif` bracket
- Compiler will only compile if a `#define XX` is found
- Can specify `#define` in:
  - source code
  - At compiler command line with (-Dxx) flag
    - `g++ -o stuff -DDEBUG stuff.cpp`

```cpp
int main()
{
    int x, sum=0, data[10];
    ...
    for(int i=0; i < 10; i++){
        sum += data[i];
    #ifdef DEBUG
        cout << "Current sum is ";
        cout << sum << endl;
    #endif
    }
    cout << "Total sum is ";
    cout << sum << endl;
}
```

```
g++ -o stuff -DDEBUG stuff.cpp
```
PRE SUMMER 2016
Example Code

• Login to your VM, start a terminal
• Best approach – Clone Lecture Code Repo
  – $ git clone git@github.com:usc-csci104-fall2015/r_lecture_code.git lecture_code
  – $ cd lecture_code/coninit
  – $ make coninit
• Alternate Approach – Download just this example
  – Create an 'lecture_code' directory
  – $ wget http://ee.usc.edu/~redekopp/ee355/code/coninit.cpp
  – $ make coninit