CSE 1325: Object-Oriented Programming

Lecture 16 – Chapter 09

Operator Overloading, Inheritance, and Polymorphism

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

Or by appointment

Homework Questions?

- The surprising surge in teams
- The GUI design artifacts
  - Yes, they should be submitted to Blackboard
  - Update pending
- Thoughts on use cases and classes

Overview

- Classes
  - Data validation
  - Structs vs Classes
- Enum Classes
- Operator Overloading
- Inheritance
  - Single Inheritance (Review)
  - Polymorphism
  - Memory Layout

Definitions of OOP

- One perspective is by *capabilities* or *language features*

  Object-Oriented Programming =

  Encapsulation (covered it!)

  \+ **Inheritance** (covered it – and will review today!)

  \+ **Polymorphism** (introduced today!)

- Another perspective is by *structure*
  - A structured program is organized around actions and logic
    - "The algorithm's the thing", with apologies to The Bard
  - An object-oriented program is organized around **data** encapsulated in **classes** that also contain related **methods**
    - "The data's the thing"

Classes are Key

- The class is fundamental to object-oriented programming
  - A class directly represents a *concept* in a program
    - A physical item – candy bars in a vending machine, products on Amazon.com, players in a football simulation
    - A logical item – debits and credits on an accounting ledger, positions in 3D space in the solar system, mathematical concepts like complex numbers
  - A class is a *user-defined* type
    - You create variables of it just like ints and doubles
    - You may use it with templates such as vector<>
    - You may define operators for it, commonly
      - = (assignment operator)
      - + - * (binary arithmetic operators)
      - += -= *= (compound assignment operators)
      - == !=
  - A class' functionality may be extended via inheritance *without modifying the class itself*

Members and Member Access

- One way of looking at a class

```
class X {    // this class' name is X
    // data members or fields (they store information)
    // methods (they manipulate the information)
    // friend functions (they are NOT members and not part of the class)
};
```

- Yet Another Simple Class Example (YASCE)

```
class X {
public:
    int m;     // field
    int mf(int v) { int old = m; m=v; return old; }   // method
};

X var;           // var is a variable of type X
var.m = 7;       // directly access var's field m (!)
int x = var.mf(9);   // call var's method mf()
```

What does X do?

Classes Share Relationships

- A class may inherit the public and protected members of one or more other classes
- A class may include, or be built from, other classes
  - Composition is actually a different approach to inheritance, as we'll see later
- A class may have a dependency on a class
- A class may have a more general association with another class

Relationships are more easily understood in the UML

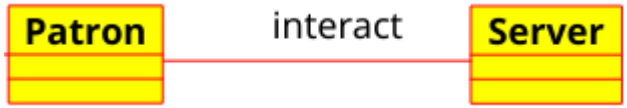
UML Class Relationships
Association

- A line simply indicates a general association
  - One class has a relationship to another class

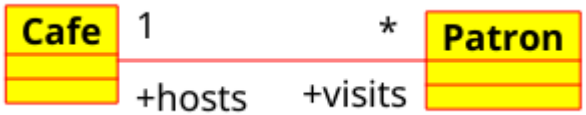
General

Directional

  - The association can be named



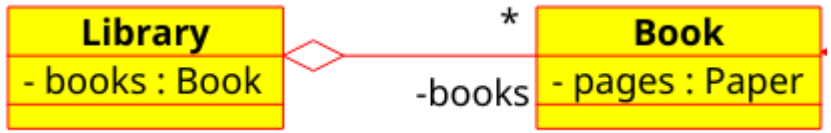
  - Each class may include multiplicity and role



UML Class Relationships

Aggregation and Composition

- Aggregation shows a class being aggregated or constructed of one or more other classes
  - The library includes many books



- Composition additionally shows that the existence of the aggregated classes depends on the aggregate
  - Books are composed of many pages

UML Class Relationships
Inheritance and Dependency

- Inheritance shows an “is a” relationship
  - A duck “is a” bird

- Dependency shows that one class depends in some way on another
  - Mainwin depends on the Gtkmm package

Review
Structs

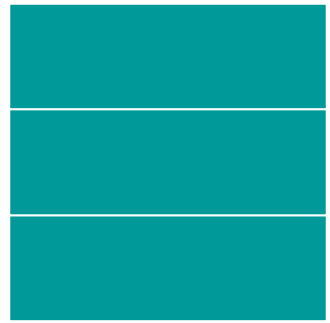
A struct is a class with public data by default and (*by convention only*) no methods

Date:

my_birthday: y

m

d



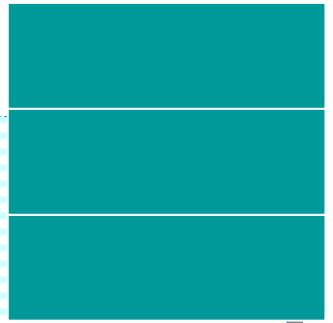
```
// simplest Date (just data)

struct Date {
    int y,m,d;   // year, month, day
};

Date my_birthday;// a Date variable (object)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950; // oops! (no day 1950 in month 30)

// later in the program, we'll have a problem!
```

Data Validation - ensuring that a program operates on clean, correct and useful data.
Validation Rules – Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data

Structs

Helper functions *could* provide code to manipulate structs

- Don't need access to private members
- Often in the same namespace



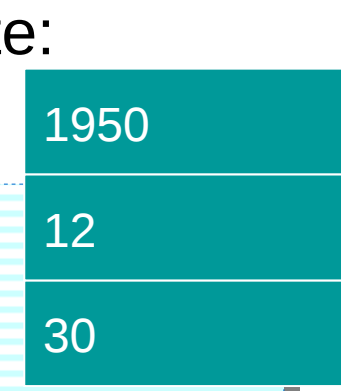

```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d;   // year, month, day
};

Date my_birthday;// a Date variable (object)


// helper functions:

void init_day(Date& dd, int y, int m, int d); // check valid date and initialize
                                              // Note: this y, m, and d are local

void add_day(Date& dd, int n);    // increase the Date by n days

init_day(my_birthday, 12, 30, 1950); // run time error: no day 1950 in month 30
```

My Take

A helper function is a “poor man’s” method *in my opinion*, and classes with methods should be preferred. I have read arguments that they have value even with classes in reducing the number of formal methods in a class. Not buying it.

Date as Struct

The C approach...

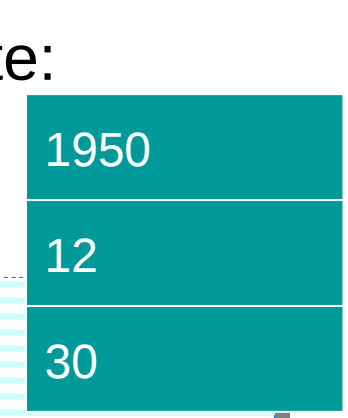



```
// simple Date
//       guarantee initialization with constructor
//       provide some notational convenience
struct Date {
    int y,m,d;                    // year, month, day
    Date(int y, int m, int d);    // constructor: check for valid date and initialize
    void add_day(int n);          // increase the Date by n days
};

// …
Date my_birthday;                  // error: my_birthday not initialized
Date my_birthday {12, 30, 1950};   // oops! Runtime error
Date my_day {1950, 12, 30};        // ok
my_day.add_day(2);                 // January 1, 1951
my_day.m = 14;                     // ouch! (now my_day is a bad date)
```

Public fields require great discipline to use without errors (don’t try this at home – or ESPECIALLY at work!).

Date as Class

The C++ approach...




```
// simple Date (control access)
class Date {
  public:
    Date(int y, int m, int d) : _year{y}, _month{m}, _day{d} { }

    // access methods
    void add_day(int n);        // increase the Date by n days
    int month() { return _month; }   // “getter”
    int day() { return _day; }
    int year() { return _year; }
  private:
    int _year;
    int _month;
    int _day;
};

// …
Date my_birthday {1950, 12, 30};      // ok
cout << my_birthday.month() << endl;  // we can read
my_birthday._month = 14;              // error: Date::_month is private
                                      //      (this is a feature)
```

Classes can Ensure Data Validity

- The notion of a “valid Date” is an important special case of the idea of a valid value
- Design your classes so that fields are guaranteed to be valid
  - Otherwise we have to check for validity all the time
  - Remember that users of your class are exceptionally clever, so program defensively
- Invariant classes, once instanced with valid data, remain valid
  - Variant classes must validate every time data changes
  - Validation may be provided via (sometimes public) methods

Enumeration Classes

- An **enum** (enumeration) **class** is a *managed* user-defined type, specifying its set of values (its enumerators)
  - Unlike simple enums, class enums cannot be interchanged with ints or other class enums

```
enum Color { red, green, blue };                    // plain enum
enum Currency { dollar, pound, yen };               // plain enum
enum class Horse { pony, clydesdale, mustang};      // enum class
enum class Car { mustang, tesla, model_t};          // enum class

void test_enums() {

    Color color = Color::red;  // OK
    int num = color;           // OK but bad (color is not an int!)
    cout << (color == Currency::dollar) << endl  // OK but REALLY bad

    Horse h = Horse::mustang;
    Car c = Car::mustang;

    int i = h;                 // error – h is not an int, but a Horse
    cout << (c == h) << endl;  // error – c and h are mustangs of diff types
}
```

Enum Classes Help Validate the Date

```
enum class Month {jan, feb, mar,
                  apr, may, jun,
                  jul, aug, sep,
                  oct, nov, dec};

class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday(1950, 30, Month::dec);  // error: 2nd argument not a Month
    Date my_birthday(1950, Month::dec, 30);  // OK
}
```

Great! Enum classes are not integers, so they are more accurately type checked. Let's try printing out some dates!

Enum Classes Help Validate

```
#include <iostream>
using namespace std;

enum class Month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

class Date {
  public:
    void set_date(int y, Month m, int d) {year=y; month=m; day=d;}
    void print_date() {cout << month << " " << day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday;
    my_birthday.print_date();
    my_birthday.set_date(1950, Month::dec, 30);  // OK
    my_birthday.print_date();
}
```

Uh oh – enum classes really are NOT just ints!

```
ricegf@pluto:~/dev/cpp/201701/16$ g++ -std=c++11 enum_class_no_constructor.cpp
enum_class_no_constructor.cpp: In member function ‘void Date::print_date()’:
enum_class_no_constructor.cpp:9:32: error: cannot bind ‘std::ostream {aka std::basic_ostream<char>}’ l
value to ‘std::basic_ostream<char>&&’
     void print_date() {cout << month << " " << day << ", " << year << endl;}
                                ^
```

Enum Classes Require String Conversion (like other classes)

```
enum class Month {jan, feb, mar,
                  apr, may, jun,
                  jul, aug, sep,
                  oct, nov, dec};

string month_to_string(Month m) {
  switch(m) {
    case Month::jan: return "January"; break;
    case Month::feb: return "February"; break;
    case Month::mar: return "March"; break;
    case Month::apr: return "April"; break;
    case Month::may: return "May"; break;
    case Month::jun: return "June"; break;
    case Month::jul: return "July"; break;
    case Month::aug: return "August"; break;
    case Month::sep: return "September"; break;
    case Month::oct: return "October"; break;
    case Month::nov: return "November"; break;
    case Month::dec: return "December"; break;
    default: return "Unknown"; break; // Or thrown a runtime exception
  }
}
```

Better, we could define the << operator for our enum class (though that doesn't obviate the basic problem addressed here). We'll get to that improvement shortly.

With much effort and arcane library calls, you can cast an enum class into an integer. If you do, then I must ask: Why were you using an enum class to begin with?

Now We Can Print Dates
(Crudely)

```
#include <iostream>
using namespace std;

// enum class code goes here

class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    void print_date() {cout << month_to_string(month) << " " <<
                          day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday{1950, Month::dec, 30};  // OK
    my_birthday.print_date();
}
```

```
ricegf@pluto:~/dev/cpp/201701/16$ g++ -std=c++11 enum_class_io.cpp
ricegf@pluto:~/dev/cpp/201701/16$ ./a.out
December 30, 1950
```

Class Constructors and Destructors

- C++ classes require (up to) 4 essential operations
  - Default and Non-default constructors (defaults to: nothing)
    - No default if any non-default constructors are declared
  - Copy constructor (defaults to: copy the member)
    - Needed if the class includes pointers or dynamic memory allocation
  - Copy assignment (defaults to: copy the members)
    - Used when an instance is to the left of the “=” operator
  - Destructor (defaults to: nothing)
    - Used to “clean up” e.g., dynamically allocated memory when an object is deleted (destroyed)

Constructors and destructors are described in detail in chapters 17-19

Default Constructor Only

Review

```
#include <iostream>
using namespace std;

// previous slide's code goes here

class Date {
  public:
    void set_date(int y, Month m, int d) {year=y; month=m; day=d;}
    void print_date() {cout << month_to_string(month) << " " <<
                         day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday;
    my_birthday.print_date();  // Oops – we haven't provided any data yet!
    my_birthday.set_date(1950, Month::dec, 30);  // OK
    my_birthday.print_date();  // Better
}
```

```
ricegf@pluto:~/dev/cpp/201701/16$ c++ -std=c++11 enum_class_io_no_constructor.cpp
ricegf@pluto:~/dev/cpp/201701/16$ ./a.out
Unknown 0, -1162798832
December 30, 1950
ricegf@pluto:~/dev/cpp/201701/16$
```

Non-Default Constructors

Review

```
#include <iostream>
using namespace std;

// previous slide's code goes here

class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    Date() : Date(1970, Month::jan, 1) { }
    void set_date(int y, Month m, int d) {year=y; month=m; day=d;}
    void print_date() {cout << month_to_string(month) << " " <<
                        day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date unix_epoch;  // or unix_epoch{} - Default constructor
    unix_epoch.print_date();
    Date my_birthday{1950, Month::dec, 30};  // Non-default constructor
    my_birthday.print_date();
}
```

Multiple constructors (with or without a default) is fine and quite common

```
ricegf@pluto:~/dev/cpp/201701/16$ g++ -std=c++11 enum_class_io_mult_constructors.cpp
ricegf@pluto:~/dev/cpp/201701/16$ ./a.out
January 1, 1970
December 30, 1950
ricegf@pluto:~/dev/cpp/201701/16$
```

Initialization vs Assignment

This is important for understanding copy constructors vs copy assignment

- Initialization causes *a new object* to have the same value as an existing object
  - `Robot r1 = r2; // initialization`
  - `Robot r1{r2};  // initialization – exactly the same`
- Assignment causes *an existing object* to have the same value as an existing object
  - `Robot r1;`
  - `r1 = r2; // assignment`
- These are two distinct operations in C++

Default Copy Constructors and Copy Assignment

```
#include <iostream>
using namespace std;

class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {} // Non-default constructor
    Foo() : Foo(0) {}           // Default constructor
    int val() {return _val;}
};

int main() {
  Foo bar0;           // Default constructor
  cout << "bar0 = " << bar0.val() << endl;
  Foo bar1{1};        // Non-default constructor
  cout << "bar1 = " << bar1.val() << endl;
  Foo bar2{bar1};     // Default copy constructor for initialization
  // Foo bar2 = bar1; // Exactly the same thing: bar2 has same values as bar1
  cout << "bar2 = " << bar2.val() << endl;
  bar0 = bar1;        // Default copy assignment for assignment
  cout << "bar0 now = " << bar0.val() << endl;
}
```

Copy Constructor

- For initialization, C++ actually invokes a special constructor – the copy constructor
  - If you have not specified one, you'll get the default – all of your variables will be copied directly across
- The copy constructor is thus invoked when:
  - You initialize a new object to an existing object
    - `Foo bar2 = bar1;  // Identical to Foo bar2{bar1};`
  - You pass an object as a non-reference parameter
    - `analyze(bar1); // A copy of bar1 is created for analyze`
  - You return an object from a function
    - `return bar1;  // A copy of bar1 is created and returned`

Declaring a Copy Constructor

- A copy constructor is just a constructor that accepts a const reference to the object to be copied
  - I *know* this isn't useful here – we'll come back to when writing a copy constructor is useful shortly

```
#include <iostream>
using namespace std;

class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                 // Non-default constructor
    Foo() : Foo(0) {}                           // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}    // Copy constructor
    int val() {return _val;}
};
```

Copy Assignment

- For assignment, C++ invokes the assignment operator to overwrite the left-hand object's values
  - If you have not specified one, you'll get the default – all of your variables will be copied directly across
- The copy assignment is thus invoked when:
  - You assign to an existing object the value of another (*or the same*) existing object
    - `bar2 = bar1;  // bar2 was existing, so we overwrite it`

Declaring a Copy Assignment

- A copy assignment is a definition of the = operator to handle copying members as needed
  - Not useful here, either – almost there!

```
class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}                 // Non-default constructor
    Foo() : Foo(0) {}                           // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {}    // Copy constructor
    Foo& operator=(const Foo &rhs) {            // Copy assignment
      if (this != &rhs) _val = rhs.val();
      return *this;
    }
    int val() const {return _val;}
};
```

Destructors

- The destructor is invoked when the object itself is being deleted
  - *Really* useless here! Next slide, promise!

```
class Foo {
    int _val;
  public:
    Foo(int val) : _val{val} {}              // Non-default constructor
    Foo() : Foo(0) {}                        // Default constructor
    Foo(const Foo &rhs) : _val{rhs.val()} {} // Copy constructor
    Foo& operator=(const Foo &rhs) {         // Copy assignment
      if (this != &rhs) _val = rhs.val();
      return *this;
    }
    ~Foo() {}                                // Destructor
    int val() const {return _val;}
};
```

Practical Use and the Rule of Three

- So when would copy constructors, copy assignment, and destructors actualy be *useful*?
  - Typically, when your class allocates memory from the heap, keeping a pointer to its address
    - To copy the object, you also need to “deep copy” the allocated heap memory to another heap memory area
    - To assign the object, you need to deallocate the existing heap memory before doing the “deep copy” to allocate the new heap memory
    - To delete the object, you also need to delete the heap memory

Rule of Three: If you define one of the above, you probably need to define all three of the above!

For more detail, I recommend the following paper. This isn’t required for the exam.
http://www.keithschwarz.com/cs106l/winter20072008/handouts/170_Copy_Constructor_Assignment_Operator.pdf

Operator Overloading

- Copy Assignment is one example of operator overloading

  Operator Overloading - Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class).

- Most C++ operators can be overloaded
  - The key is to know the method or function signature, e.g., the parameter types and return values
  - Let's teach Date (and Month) from earlier in this lecture about streams

Date / Month Streams

```
class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    Date() : Date(1970, Month::jan, 1) { }
    friend ostream& operator<<(ostream& os, const Date& date);
  private:
    int year;
    Month month;
    int day;
};
```

Here we're streaming an enum class...

```
ostream& operator<<(ostream& os, const Month& month) {
    os << month_to_string(month);
    return os;
}
```

...and here we're streaming our Date class

```
ostream& operator<<(ostream& os, const Date& date) {
    os << date.month << " " << date.day << ", " << date.year;
    return os;
}
```

This really cleans up our main code!

```
int main() {
    Date unix_epoch;  // or unix_epoch{}
    cout << unix_epoch << endl;
    Date my_birthday{1950, Month::dec, 30};
    cout << my_birthday << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201701/16$ g++ -std=c++11 enum_class
ricegf@pluto:~/dev/cpp/201701/16$ ./a.out
January 1, 1970
December 30, 1950
ricegf@pluto:~/dev/cpp/201701/16$
```

Operator overloading

- You can overload other operators, too

```
Month operator++(Month& m) {
  switch(m) {
    case Month::jan: m = Month::feb; break;
    case Month::feb: m = Month::mar; break;
    case Month::mar: m = Month::apr; break;
    case Month::apr: m = Month::may; break;
    case Month::may: m = Month::jun; break;
    case Month::jun: m = Month::jul; break;
    case Month::jul: m = Month::aug; break;
    case Month::aug: m = Month::sep; break;
    case Month::sep: m = Month::oct; break;
    case Month::oct: m = Month::nov; break;
    case Month::nov: m = Month::dec; break;
    case Month::dec: m = Month::jan; break;
  }
  return m;
}

int main() {
    Month m{Month::oct};
    for (int i; i<5; ++i)
      cout << ++m << endl;
}
```

Since enum classes aren't ints, incrementing is a bit of a pain...

```
ricegf@pluto:~/dev/cpp/201701/16$ g++ -std=c++11 enum_class_increment.c
ricegf@pluto:~/dev/cpp/201701/16$ ./a.out
November
December
January
February
March
ricegf@pluto:~/dev/cpp/201701/16$
```

Another Example: A Box
More Detail This Time

- Let's write a Box class with 2 operators
  - + adds the boxes' linear dimensions
  - << streams (h,w,d) to represent the box

```
// Test the Box class
int main( ) {
   Box box1(6.0, 7.0, 5.0);     // Declare box1 of type Box
   Box box2(12.0, 13.0, 10.0);  // Declare box2 of type Box
   Box box3 = box1 + box2;

   cout << "Box 1: " << box1 << " volume " << box1.getVolume() <<endl;
   cout << "Box 2: " << box2 << " volume " << box2.getVolume() <<endl;
   cout << "Box 3: " << box3 << " volume " << box3.getVolume() <<endl;

   return 0;
}
```

Example: The Box Interface

```
#include <iostream>
using namespace std;

class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }

     double getVolume(void) {
        return length * breadth * height;
     }

     // Overload + operator to add two Box objects.
     Box operator+(const Box& b);

     // Overload << operator to stream a string representation
     friend ostream& operator<<(ostream& os, const Box& b);

  private:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};
```

The operator+ is a member method. “This” is the left operand, b is the right operand, and we return a new Box representing the sum.

The operator<< is NOT a member method, but is a “friend” of this class – hence, it can access Box private variables. The stream os is the left operand, b is the right operand, and we return a reference to the updated stream.

Example: The Box Implementation

```
Box Box::operator+(const Box& b) {
   return Box(length + b.length, breadth + b.breadth, height + b.height);
}
```

For operator+, we simply instance a new Box object, passing the constructor the sum of our and the right operand's lengths, breadths, and heights, respectively.

```
ostream& operator<<(ostream& os, const Box& b) {
   os << '(' << b.length << ',' << b.breadth << ',' << b.height << ')';
   return os;
}
```

Note the LACK of “Box::” preceding operator<< - this is NOT a member method, but rather a “friend” of the Box class. Thus, operator<< must be declared in the Box interface with the keyword “friend” (granting permission to access its private variables), but it is NOT part of the Box implementation.

The implementation is straightforward. We simply use the same syntax as we would use with cout, but with os instead. Then we return os.

```
ricegf@pluto:~/dev/cpp/08$ g++ box.cpp
ricegf@pluto:~/dev/cpp/08$ ./a.out
Box 1: (6,7,5) volume 210
Box 2: (12,13,10) volume 1560
Box 3: (18,20,15) volume 5400
ricegf@pluto:~/dev/cpp/08$
```

Operator Overloading

- You can define only existing operators
  - You can't create your own custom operator such as $$ or @
- You can define operators only with the usual number of operands
  - *E.g.*, no unary **<=** (less than or equal) and no binary **!** (not)
- Overloaded operators must have *at least* one user-defined type as an operand
  - **int operator+(int,int);** *//* error: you can't overload built-in +
  - **Vector operator+(const Vector&, const Vector &);** *//* ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - **+** should be addition, ***** be multiplication, **[]** be access, **()** be call, etc.
  - You must determine what is "conventional" with the classes you define
- Advice (not language rule):
  - Don't overload unless you really have to

Permissible Operator Overloading

- The following operators **<u>can</u>** be overloaded:
  - + - * / % ^
  - & | ~ ! , =
  - < > <= >= ++ --
  - << >> == != && ||
  - += -= /= %= ^= &=
  - |= *= <<= >>= [ ] ( )
  - → ->* new new [ ] delete delete [ ]
- The following operators **<u>cannot</u>** be overloaded:

  :: .* . ?:

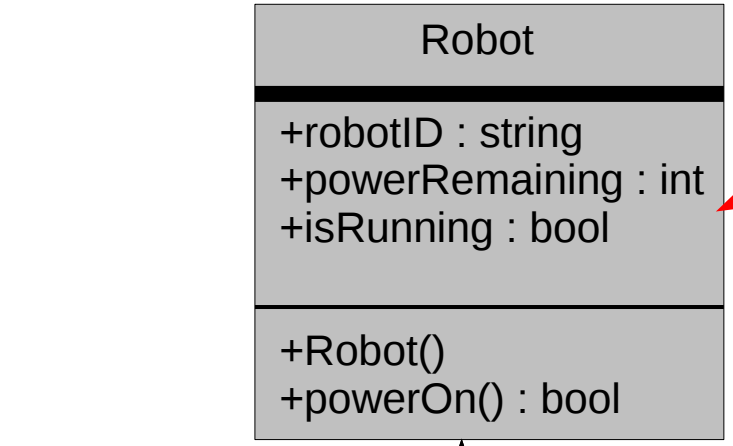
Inheritance
(Review)

- **Inheritance** – Reuse and extension of fields and method implementations from another class

- The original class is called the base class (e.g., Robot)
- The extended class is called the derived class (e.g., Drone)

```
class Drone : public Robot {
```

Terminology
(Review)



Base class

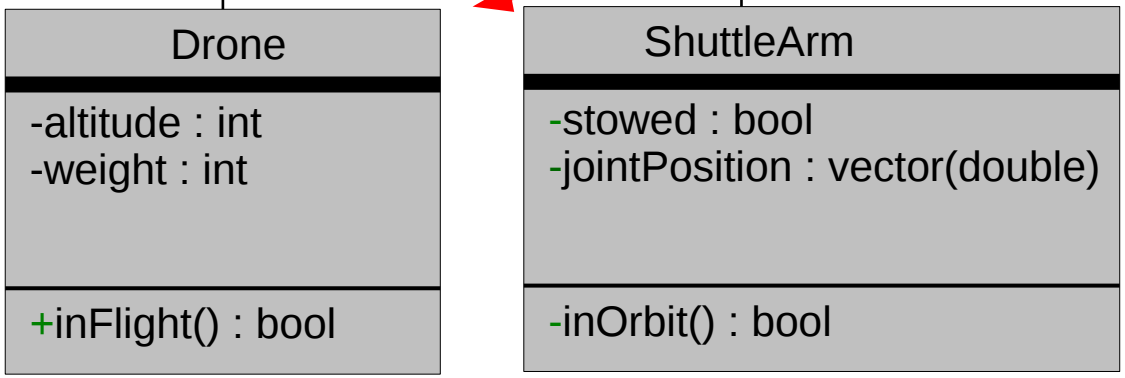
```
class Robot {
```

all have robotID, isRunning, powerRemaining, and powerOn() (they are public, not private)

Robot() is NOT inherited (constructors don't inherit!)

Derived class

"is-a"

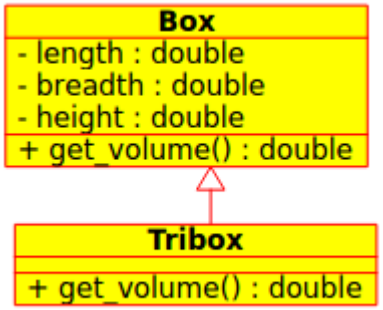



```
class Drone :
    public Robot {

class ShuttleArm :
    public Robot {
```

"public" means that all public fields in Robot will be public in ShuttleArm. "private" would make public fields in Robot private in ShuttleArm.

Example: Inheriting the Box Interface

```
#include <iostream>
using namespace std;

class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }

     double getVolume(void) {
        return length * breadth * height;
     }

  protected:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};

class Tribox : public Box {
   public:
     Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }

      double getVolume(void) {
         return length * breadth * height / 2;
      }
};
```



Protected is scoped between public (everyone sees it) and private (only this class and its friends see it) – useful for inheritance.

A “Tribox” is a triangular rather than rectangular box. Its volume is simply half that of a rectangular box of the same dimensions.

Since constructors don't inherit, we explicitly call the base constructor. We then overload getVolume.

Example: Inheriting the Box Interface

```
// Test the Box base class and Tribox derived class
int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Box box2(12.0, 13.0, 10.0);

   cout << "Box 1: volume " << box1.getVolume() <<endl;
   cout << "Box 2: volume " << box2.getVolume() <<endl;

   Tribox tbox1(6.0, 7.0, 5.0);
   Tribox tbox2(12.0, 13.0, 10.0);

   cout << "Tri-Box 1: volume " << tbox1.getVolume() <<endl;
   cout << "Tri-Box 2: volume " << tbox2.getVolume() <<endl;

   return 0;
}
```

```
ricegf@pluto:~/dev/cpp/08$ g++ box2.cpp
ricegf@pluto:~/dev/cpp/08$ ./a.out
Box 1: volume 210
Box 2: volume 1560
Tri-Box 1: volume 105
Tri-Box 2: volume 780
```

Here we create objects of the derived class and use them directly. Some of the features of the base class are inherited – the protected data, for example – and some are overridden – the get_volume() method. We could also add additional fields and methods to the derived class as needed.

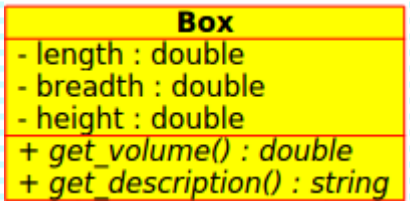
Polymorphism

- In the above example, we created variables of the derived class. Simple.
- Better is to create variables that reference objects of the base class or *any of its derived classes*.

- **Polymorphism** – The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results.

Example: Inheriting Outside the Box

```
#include <iostream>
using namespace std;

class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }

     virtual double get_volume(void) {
        return length * breadth * height;
     }

     virtual string get_description() {return "Rectangular box";}

  protected:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};
```

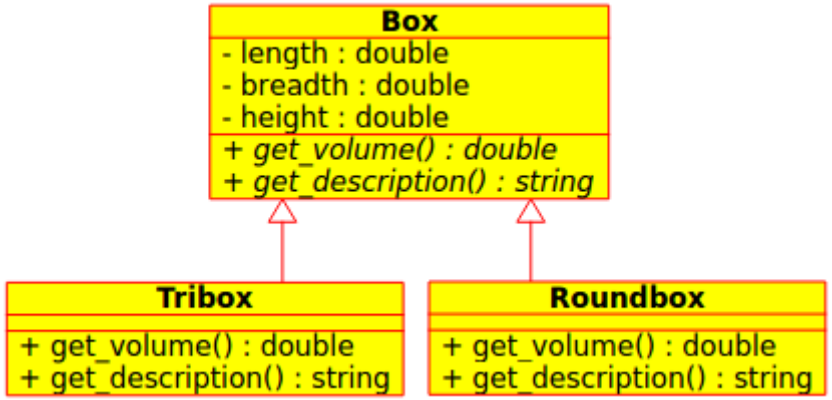


Virtual methods are depicted in UML using italics.

We declare get_volume() virtual, meaning it can be overridden even for Box *references*. That is, we don't need a variable of the derived type to access the overridden methods specified in the derived type; a simple reference variable of the base type will do.

Example: Inheriting Outside the Box

```
class Tribox : public Box {
   public:
      Tribox (double len, double bre, double hei)
        : Box(len, bre, hei) { }

      double getVolume(void) {
         return length * breadth * height / 2;
      }

     string get_description() {return "Triangular box";}

};

class Roundbox : public Box {
   const static double PI = 3.14159265;
   public:
      Roundbox (double dia, double hei)
        : Box(dia, dia, hei) { }

      double getVolume(void) {
         return PI * (length * breadth / 4) * height;
      }

     string get_description() {return "Round box";}

};
```



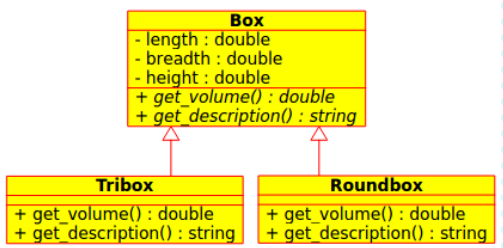
We derive two classes from Box, one for triangular boxes and one for round boxes (cylinders), overriding get_volume and get_description for each.

Example: Inheriting Outside the Box

```
void print_volume(Box b) {
   cout << b.get_description() << " volume is " << b.getVolume() <<endl;
}
```

A direct variable and reference variable version for printing volumes.

```
void print_volume_ref(Box& b) {
   cout << b.get_description() << " volume is " << b.getVolume() <<endl;
}

// Test the Box class
int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Tribox tbox1(6.0, 7.0, 5.0);
   Roundbox rbox1(6.5, 5.0);
```




```
   cout << endl << "Access derived class methods directly:" << endl;
   print_volume(box1);
   print_volume(tbox1);
   print_volume(rbox1);
```

Direct variable version is first.

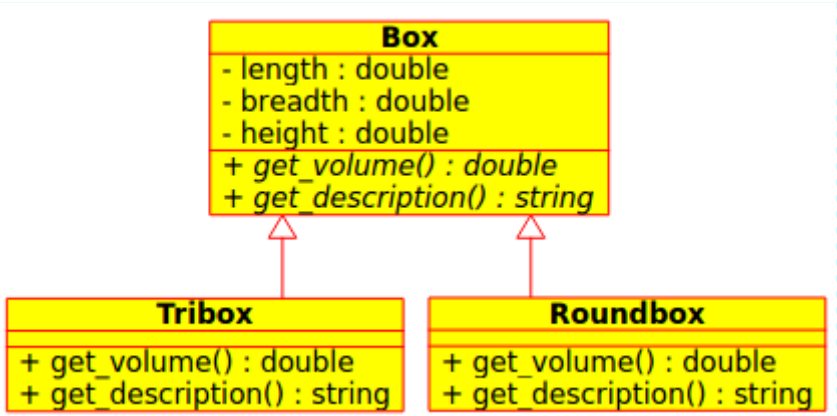
```
   cout << endl << "Access derived class methods via reference:" << endl;
   print_volume_ref(box1);
   print_volume_ref(tbox1);
   print_volume_ref(rbox1);
```

Reference variable version is second.

```
   return 0;
}
```

Example: Inheriting Outside the Box

```
void print_volume(Box b) {
   cout << b.get_description() << " v
}

void print_volume_ref(Box& b) {
   cout << b.get_description() << " v
}

// Test the Box class
int main( ) {
   Box box1(6.0, 7.0, 5.0);
   Tribox tbox1(6.0, 7.0, 5.0);
   Roundbox rbox1(6.5, 5.0);

   cout << endl << "Access derived class met
   print_volume(box1);
   print_volume(tbox1);
   print_volume(rbox1);

   cout << endl << "Access derived class met
   print_volume_ref(box1);
   print_volume_ref(tbox1);
   print_volume_ref(rbox1);

   return 0;
}
```




```
ricegf@pluto:~/dev/cpp/08$ g++ box_ref_polymorph.cpp
ricegf@pluto:~/dev/cpp/08$ ./a.out

Access derived class methods directly:
Rectangular box volume is 210
Rectangular box volume is 210
Rectangular box volume is 211.25

Access derived class methods via reference:
Rectangular box volume is 210
Triangular box volume is 105
Round box volume is 165.915
ricegf@pluto:~/dev/cpp/08$
```

Direct variab

Base methods

Reference v

Derived methods

Bottom line: Polymorphism relies on references (or pointers) in C++

Object Layout in Memory

```
class Parent{
public:
    int a, b;
    virtual void foo(){cout << "parent" << endl;}
};

class Child : public Parent{
public:
    int c, d;
    virtual void foo()
      {cout << "child" << endl;}
};

int main(){
    Parent p; p.foo();
    Child c; c.foo();

    cout << "Parent Offset a = " << (size_t)&p.a - (size_t)&p << endl;
    cout << "Parent Offset b = " << (size_t)&p.b - (size_t)&p << endl;

    cout << "Child Offset a = " << (size_t)&c.a - (size_t)&c << endl;
    cout << "Child Offset b = " << (size_t)&c.b - (size_t)&c << endl;
    cout << "Child Offset c = " << (size_t)&c.c - (size_t)&c << endl;
    cout << "Child Offset d = " << (size_t)&c.d - (size_t)&c << endl;
}
```

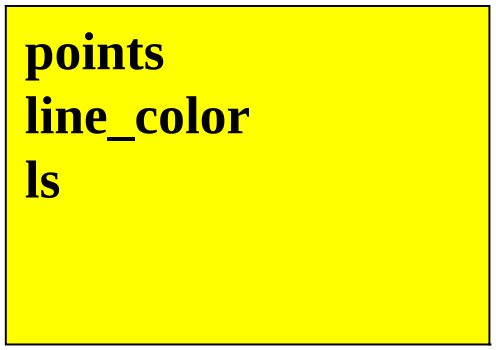
```
ricegf@pluto:~/dev/cpp/19$ g++ -o mem_layout mem_layout.cpp
ricegf@pluto:~/dev/cpp/19$ ./mem_layout
parent
child
Parent Offset a = 8
Parent Offset b = 12
Child Offset a = 8
Child Offset b = 12
Child Offset c = 16
Child Offset d = 20
ricegf@pluto:~/dev/cpp/19$
```

http://stackoverflow.com/questions/8672218/memory-layout-of-inherited-class

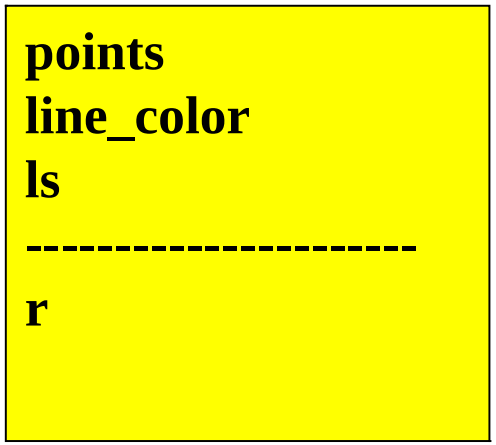
Object layout

- The data members of a derived class are simply added at the end of its base class (a Circle is a Shape with a radius)

Shape:



Circle:



Benefits of inheritance

- Interface inheritance
  - A function expecting a shape (a **Shape&**) can accept any object of a class derived from Shape.
  - Simplifies use
    - sometimes dramatically
  - We can add classes derived from Shape to a program without rewriting user code
    - Adding without touching old code is one of the “holy grails” of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another “holy grail”

Quick Review

- True or False: A class (type) may not be used to instantiate templates such as vector
- True or False: The only difference in C++ between a struct and a class is the default visibility of its members.
- Ensuring that a program operates on clean, correct and useful data is _____ _____.
- Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data is _____ _____.
- Which of the following help with that: (1) Clever programming (2) Invariant classes (3) Validation methods (4) Validation in the constructor (5) Limiting members to primitives
- The following are good reasons to make class fields private:
  (1) To provide a clean interface (2) To hide proprietary data
  (3) To maintain an invariant (4) To enforce HIPPA regulations
  (5) To ease debugging (6) To change representation or algorithm without impacts

Quick Review

- Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class) is _____ _____.
- Which of the following operators can NOT be overloaded: (1) :: (2) new (3) ?: (4) &&
- To overload the + operator for a class, define the _____ method.
- Reuse and extension of fields and method implementations from one class to other classes is called _____. The original class is called the _____ class, and the other classes are called _____ classes.
- A ____ _____ defines the inheritance relationships between classes.
- In specifying a base class in the derived class, the keyword “public”, “protected”, or “private” is specified. What are the implications of this?

For Next Class

- (Optional) Read Chapter 9 in Stroustrop
  - Do the Drills!
- Skim Chapter 10 for next Tuesday
  - Multiple Inheritance Considerations
  - File I/O – beyond the bare basics
- Sprint #1 due Thursday at 8 am
  - UML diagrams (Use Case, Class, Sequence)
  - User Interface Design
  - Features with Test Code (varies by team size)