CS 142
Objects/Classes in C++

Rhodes College

Announcements

- Program 7 has been assigned - due Sunday, April 19th by 11:55pm

Definitions

- A ***class*** is a struct plus some associated functions that act upon variables of that struct type.
  - class = struct + functions
- An ***object*** is a variable of some struct type
  - aka "an ***instance*** of a class."
- In a class, the variables of that class are called ***fields***; the functions are called ***methods.***
  - Together, the fields and methods are called data members (book uses ***data members*** and ***member functions***).


```
class dog {
  public:
  string name;
  int age;
  void bark();
};

void dog::bark() {
  cout << name << "says woof!";
}
```

Name of the class

Every dog has a name

Every dog has an age

Every dog has the ability to bark

```
class dog {
  public:
  string name;
  int age;
  void bark();
};

void dog::bark() {
  cout << name << "says woof!";
}
```

A class's methods are allowed to use the fields defined within that class as local variables.

A method (normally) only has access to the fields for its own object.


```
void dog::bark() {
  cout << name << "says woof!";
}
main
dog regan;
regan.name = "Regan";
regan.age = 3;

dog jack;
jack.name = "Jack";
jack.age = 8;

regan.bark();
jack.bark();
```

regan: name: "Regan" age: 3

jack: name: "Jack" age: 8

When regan.bark() is called, in the dog::bark() function, name is automatically set to "Regan" and age is set to 3.


```
void dog::bark() {
  cout << name << "says woof!";
}
main
dog regan;
regan.name = "Regan";
regan.age = 3;

dog jack;
jack.name = "Jack";
jack.age = 8;

regan.bark();
jack.bark();
```

regan: name: "Regan" age: 3

jack: name: "Jack" age: 8

When jack.bark() is called, in the dog::bark() function, name is automatically set to "Jack" and age is set to 8.

- Most object-oriented (OO) programming languages allow us to specify fields and methods as ***public*** or ***private***.
- ***Private*** members can be used only by the person writing the class (i.e., inside methods).
- ***Public*** members can be used by the person writing the class, or the person using the class.

```
class A {
  public:
  int x;
  void f();

private:
  int y;
  void g();
}
```

```
int main()
{
  A obj1, obj2;

  obj1.x = 4;  // ok
  obj1.y = 2;  // error

  obj2.f();    // ok
  obj2.g();    // error
}
```


Why have public and private?

- Sometimes we need to ***hide*** certain variables or functions from the user of a class so the user doesn't accidentally screw things up.
- This is called ***information hiding***.
- Used to protect the members of an object that should only be used by the person writing the class.


```
class dog {
  public:
  string name;
  int age;
  void bark();
};

void dog::bark() {
  cout << name << "says woof!";
}
```

What could go wrong with age or name being public?


```
class dog {
  public:
  void bark();

  private:
  string name;
  int age;
};

void dog::bark() {
  cout << name << "says woof!";
}
```

Good rule of thumb to make all fields (variables) private unless you have a very good reason not to.

main

```
dog regan;
regan.name = "Regan";
regan.age = 3;

dog jack;
jack.name = "Jack";
jack.age = 8;

regan.bark();
jack.bark();
cout << "Jack is " << jack.age << endl;
```

What is wrong with this code now?

main

```
dog regan;
regan.name = "Regan";
regan.age = 3;

dog jack;
jack.name = "Jack";
jack.age = 8;

regan.bark();
jack.bark();
cout << "Jack is " << jack.age << endl;
```

What is wrong with this code now?

Red fields are private; cannot be used outside of the class now.


```
class dog {
  public:
  void bark();
  void setName(string newName);
  string getName();
  void setAge(int newAge);
  int getAge();

  private:
  string name;
  int age;
};  // rest of code on computer
```

Add setters and getters.

- The public members of a class are known as the class's ***interface***.
  - These members are what the users of your class see.
  - Generally describes ***what*** a class does.
- The private members of a class are known as the class's ***implementation***.
  - These are hidden from the user.
  - Generally describe how a class works.
- We strive to keep a class's interface consistent over time. We can change the implementation any time we want.

Constructors and Destructors

- A ***constructor*** is a method that is run automatically when an object is created.
- A ***destructor*** is a method that is run automatically when an object is "destroyed."
  - For objects on the stack, destroyed == goes out of scope.
  - For objects on the heap, destroyed == is deleted.

Constructors

- Constructors are commonly used to initialize the fields (variables) in a class to appropriate values.
- Without constructors, the user would have to set all the fields in a class by hand after each object creation.
- The name of a constructor is always the same name as the class itself.

Multiple Constructors

- Classes can have multiple constructors.
  - This is different than Python!!!
- The *default constructor* never takes any arguments, but other constructors can.
- These arguments are typically used to set the fields of the class.

Destructors

- The name of a destructor is always the same name as the class, prefaced with a ~ (tilde).
  - Destructors never have any arguments, and there can be only one per class.

Dynamic Memory with Objects

```
dog lassie;
lassie.setAge(4);
dog rowlf = lassie;
// copies all of lassie's fields to rowlf.
// The two dogs are still 100% separate.

dog* toto = &lassie;
toto->setAge(6);
// sets lassie's age (toto is just a pointer,
// not a separate standalone dog)

dog* cujo = new dog;
cujo->setAge(3);
delete cujo;
```

Use dot operator when left side is an ***object***.

Use arrow operator when left side is a ***pointer to an object.***

Syntactic sugar

- Syntax in a programming language that makes something easier to express.

Example of syntactic sugar

```
int x = 1, y = 2;
int z = x + y;
```

```
int x = 1, y = 2;
int z = add(x, y)
```

Many operators are syntactic sugar because usually they are unnecessary in the language; we could get by with just functions.

Example of syntactic sugar

```
vector<int> vec(3);
vec[0] = 100;
cout << vec[0];
```

```
vector<int> vec(3);
vec.set(0, 100);
cout << vec.get(0);
```

Many operators are syntactic sugar because usually they are unnecessary in the language; we could get by with just functions.

Operator Overloading

- ***Function overloading***: Allowing different functions with the same name, distinguished by argument number or data type(s).
  - Allowed in C++
  - Had to use default values for parameters in Python
- ***Operator overloading***: Adding new meanings for operators when used with different data types.

As simple as defining a function

- Define a function called:

  operator+ operator- operator* operator/
  operator+= operator< operator++ operator==
- Number of arguments is determined by the operator name.
  - i.e., operator+ always takes two arguments.
- Return type can be anything you want.

Overloading +

```
vector<int> vec1, vec2, vec3;
vec1.push_back(1);
vec1.push_back(2);
vec2.push_back(10);
vec2.push_back(20);
vec3 = vec1 + vec2;
```


Overloading +

```
vector<int> vec1, vec2, vec3;
vec1.push_back(1);
vec1.push_back(2);
vec2.push_back(10);
vec2.push_back(20);
vec3 = vec1 + vec2;
cout << vec3;
```

Overload these operators

```
vector<int> vec, vec2;
vec += 1;  // overload += so it does push_back
vec += 2;
vec += 1;
vec += 3;
vec2 = vec – 1;        // vec2 is now [2, 3]
                       // overload minus so it removes all
                       // all instances of an item from a vector
```


Overloading with classes

```
class rational {
 public: …
 private:
   int num, den;
};

rational operator* (const rational & a, const rational & b)
{
 rational ans;
 ans.num = a.num * b.num;
 ans.den = a.den * b.den;
 return ans;
}
```


Solution 1

```
class rational {
 public:
   rational operator*(const rational & b);
 private:
   int num, den;
};

rational rational::operator* (const rational & b)
{
 rational ans;
 ans.num = num * b.num;
 ans.den = den * b.den;
 return ans;
}
```


Solution 2

```
class rational {
 public:
   friend rational operator* (const rational & a, const rational & b)
 private:
   int num, den;
};

rational operator* (const rational & a, const rational & b)
{
 rational ans;
 ans.num = a.num * b.num;
 ans.den = a.den * b.den;
 return ans;
}
```

Solution 2

```
class rational {
 public:
  friend rational operator* (const rational & a, const rational & b)
 private:
   int num, den;
};

rational operator* (const rational & a, const rational & b)
{
 rational ans;
 ans.num = a.num * b.num;
 ans.den = a.den * b.den;
 return ans;
}
```