Lecture 11
Object Oriented Programming

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov, Clayton Price (Missouri S&T), Hal Perkins (University of Washington)

Administrivia

- Homework 3 Part 2 due tonight

- Lab2 is also due tonight (70% of class is done)
  - The point is to get a working C++ setup before homework starts
  - Reach out if you're having problems with it
    - See debugging post on Piazza first though!

- Homework 4 will be released tonight
  - Breakout game!

Today's Goals

- Introduce Classes and Objects in C++
  - Why are they an important concept?
  - How do we use them?

- Understand special functions useful for objects
  - Constructors
  - Overloaded operators

- Walk through GE211 to discuss how it works

Getting the code for today

- Download code in a zip file from here: https://nu-cs211.github.io/cs211-files/lec/11_objects.zip

- Extract code wherever

- Open with CLion
  - Make sure you open the folder with the CMakeLists.txt

  - Details on CLion in Lab05

Outline

- **Object Oriented Programming**
  - Writing code with objects
  - Constructors
  - Example Object: Vectors

- Tour of GE211

Object Oriented Programming

- Basic idea
  - Combine data and code that modifies the data together

- In code this takes the form of structs (or classes)
  - Which contain various fields (data)
  - And have various methods (functions)

- When you create one of these, you're create an "object"
  - Unit of data and interaction
  - Big chunk of memory that holds all the fields
    - But also with functions that you can run on it

How we handled this idea in C

- Created a file for dealing with a single "object"
  - i.e. a `ballot_t`

- Functions inside the file operate on that object
  - Each function takes a `ballot_t` as the first argument
  - Functions are named ballot_<action>()
    - `ballot_create`, `ballot_destroy`, `ballot_count`, etc.

- All access to the data must go through the functions
  - Other files couldn't access the ballot fields directly
  - Otherwise they could screw up the rules of the `ballot_t`

What would a ballot_t look like in C++?

- Create a ballot struct
  - With length and entries fields just like the C version

- Add functions to the struct
  - (Couldn’t do this in C)

  - Each function will modify the struct it’s called on

Why do this?

- Keep concepts located together
  - One object for VC, one for ballot, one for ballot_box
  - Could have written it all as one big thing
    - But it would be easy to get lost in the complexity
    - Separating things into smaller parts meant each was easier to write
- Access control
  - Later, we'll see that there are ways to control which data/functions can be publicly accessed versus privately accessed
  - Often there are public functions but private data

Outline

- **Object Oriented Programming**
  - **Writing code with objects**
  - Constructors
  - Example Object: Vectors

- Tour of GE211

Code organization

- Header files (.hxx)
  - struct definitions, including member functions
  - You can inline simple one-liner functions in the definition

- Source files (.cxx)
  - Implementations of member functions

- Usually a set of cxx/hxx files for each struct/class you make
  - Classes are nearly the same as structs, we'll talk about them next week

Implementing member functions

src/position.hxx
src/position.cxx

```
struct Position {
  double x; // fields, now called data members
  double y;
  void print(); // member functions, also called methods
};

void Position::print() { // method implementation
  std::cout << "{" << x << " , " << y << " }\n";
}
```

Accessing data members in member functions

- Within member functions, you can just use the name of any data member
  - Make sure not to make local variables with the same name as data members!!

- The `this` pointer can also be used inside member functions
  - It's a pointer to the object itself
  - `this->member` can access the data member directly
    - Means the same thing as just `member` generally

  - You will almost never need to use `this` in C++

Live coding example: positions

src/position.hxx
src/position.cxx

- Data
  - Doubles for x and y coordinate

- Methods
  - print()
  - set_location()
  - distance_to()

const is used everywhere in C++

- `const` keyword means that the thing cannot be modified
  - Used significantly more in C++ than it was in C
  - Signals intent to the compiler to keep you from making mistakes!

  - `const int x = 0;`
    - Integer `x` cannot be modified

  - `const int& x = y;`
  - `int const& x = y;`
    - Reference to an int now named `x`. You cannot modify `x`
    - These two are identical! Either way is fine

  - `print() const;`
    - There will be a `print()` member function doesn't modify its object

Defining operators for our objects

- One strength of C++ is that we can define how normal operators work on our objects
  - +, -, +=, ==, <<, etc.

- Most of these are not defined for you
  - How would the compiler know what they mean for a `Position`?
  - An exception is assignment (=), which is defined as a copy of all fields

  - We can implement the operators ourselves though!
  - Can be implemented as standalone functions or member functions

Example overloaded operator

src/position.hxx
src/position.cxx

- Define == as a standalone function that takes two Position arguments

Note: lhs - left-hand side, rhs - right-hand side

```
bool operator==(Position const& lhs, Position const& rhs){
    return (lhs.x == rhs.x) && (lhs.y == rhs.y);
}
```

- Future code can now use == on positions!
  - Don’t have to (and shouldn’t) type out the full function name
  - More on operator overloading next lecture

Break + Open Question

- How would you have written libvc using C++ objects?

Break + Open Question

- How would you have written libvc using C++ objects?
    - Add the vc_ functions to the struct vote_count
    - Maybe make a few operators to make your life easier

Outline

- **Object Oriented Programming**
  - Writing code with objects
  - **Constructors**
  - Example Object: Vectors

- Tour of GE211

Contructors initialize newly-created objects

- Written with the class name as the method name, no return value!

```
Position(double x, double y);
```

- Allow us to define how data is initialized
  - Might use inputs as values for some data members
  - Might give default values to some data members
  - Might do some computation to decide what data members should be

  - Any and all of the above

Default constructor

- If you do not create a constructor, C++ will attempt a default
  - Leave all basic types uninitialized
  - Call the default constructor on all data members that are objects

- This is how we've been using Position so far

- C++ notation
  - Basic data types: plain old data (POD)
  - Object data types: non-POD

Writing our own constructor

src/position.hxx
src/position.cxx

```
struct Position {
  double x;
  double y;
  Position(double in_x, double in_y);
}
```

Note: doesn't return `void`
Has no return at all!

```
Position::Position(double in_x, double in_y) {
  x = in_x;
  y = in_y;
}
```

Initialization lists

- C++ lets you optionally declare an initialization list as part of your constructor definition
  - Lists fields and initializes them, one-by-one
  - **MUST** be in same order as the data members are in the struct

```
Position::Position(double in_x, double in_y)
    : x(in_x),
      y(in_y)
{ } // must have function body, even if empty
```

Initialization lists

- **Always** write initializer lists for constructors
  - *Nearly* identical to doing it manually
  - But the word nearly hides a lot of pain there

- Examples:
  - Data members that don’t have a default constructor need to be created in the initializer list

  - Data members that are references can never be NULL, so they don’t have a default! But the initializer list can still set them

Must use exclusively default constructors or defined ones

- Once you create a single constructor, C++ will no longer allow default ones
  - So if you want more options, you'll need to make them!

- Remember: C++ allows multiple functions with the same name, as long as their input arguments are different
  - We can create multiple constructors!

Multiple constructors make objects easier to use

src/position.hxx
src/position.cxx

- Default constructor

```
Position::Position()
    : x(0),
      y(0)
{ }
```

- Constructor with arguments

```
Position::Position(double in_x, double in_y)
    : x(in_x),
      y(in_y)
{ }
```

Copy constructor

src/position.hxx
src/position.cxx

- Makes a copy of an existing object

```
Position::Position(const Position& orig)
    : x(orig.x),
      y(orig.y)
{ }
```

- Can be called automatically or used via assignment

```
Position x;

Position y(x);

Position z = x;
```

When do copies happen?

- The copy constructor is invoked if:

1. You *initialize* an object from another object of the same type

```
Position x;     // default constructor
Position y(x); // copy constructor
Position z = y;// copy constructor
```

2. You pass a non-reference object as a value parameter to a function

```
void foo(Position x) { ... }

Position y; // default constructor
foo(y);     // copy constructor
```

3. You return a non-reference object value from a function

```
Position foo() {
  Position y; // default constructor
  return y;   // copy constructor
}
```

Destructors

src/position.hxx
src/position.cxx

- Same concept as constructors: used to clean up an object
  - Automatically called when the object goes out of scope
  - Note: you **never** call the destructor yourself!

- Handles any cleanup, including freeing necessary resources

```
Position::~Position() {
  // nothing to clean here since we don't use
  // dynamic memory
}
```

Break + Question

- Why make a constructor instead of having users set individual fields?

Break + Question

- Why make a constructor instead of having users set individual fields?
  - Constructor can ensure that everything is initialized
  - Constructor knows what the rules are!
    - Can check that the inputs are valid
  - Generally: harder to make mistakes when using someone else's code

Outline

- **Object Oriented Programming**
  - Writing code with objects
  - Constructors
  - **Example Object: Vectors**

- Tour of GE211

C++ libraries provide various useful structures for you

- C libraries had some functions that would let you interact with things like files or the user

- C++ has those, but also has libraries with data structures and with various algorithms (such as sorting)
  - C++ data structures (containers): https://cplusplus.com/reference/stl/
  - C++ algorithms: https://cplusplus.com/reference/algorithm/

C++ Vectors

- One example C++ library: Vector
  - An automatically expanding "array" capable of holding any type
  - `std::vector<TYPE>` to choose what type it should hold
    - `std::vector<int>`, `std::vector<double>`, etc.
    - This idea is known as "generics". We'll discuss in a later lecture

- Example vector types
  - std::vector<int> - holds ints
  - std::vector<char> - holds chars
  - std::vector<Position> - holds Positions

Creating C++ Vectors

- Creating a vector (there are many ways)

```
std::vector<TYPE> myvector(); //empty vector of with no size

std::vector<TYPE> myvector(len); //vector of size len with uninitialized values

std::vector<TYPE> myvector(len, val); //vector of size len with values set to val

std::vector<TYPE> myvector{val1, val2, val3, ...};
//vector with initial values, set to the correct size to hold them all
```

Other useful Vector operations

- `vec[n]` is used to get the value at index `n`
  - Works just like a C array
  - Still **UNDEFINED BEHAVIOR** if `n` is out of bounds for the Vector

- `vec.at(n)` accesses value at index `n`
  - Just like square brackets, but throws an exception if out-of-bounds
  - Exceptions: new way of signaling errors. Will talk about in later lecture

- `vec.size()` returns the length of the Vector

- `vec.push_back()` and `vec.pop_back()` add/remove items
  - And resize the Vector automatically as needed

Example vector code

test/vector_examples.cxx

- Play around with vectors

C++ allows for simpler iteration (like Python)

```
double sum_vec(std::vector<double> const& vec){
  double result = 0;
  for (double val : vec) {
    result += val;
  }
  return result;
}
```

Iterates over elements in the vector, not indices!

Modifying elements inside the vector

- Warning: make sure you're modifying the actual vector element

```
void dec_vec_wrong(std::vector<int>& vec){
  for (int val : vec){
    --val;
  }
}
```

Each `val` is a copy of the value in the vector

Modifying elements inside the vector

- Warning: make sure you're modifying the actual vector element

```
void dec_vec_wrong(std::vector<int>& vec){
  for (int val : vec){
    --val;
  }
}
```

Each `val` is a copy of the value in the vector

```
void dec_vec_right(std::vector<int>& vec){
  for (int& val : vec){
    --val;
  }
}
```

Each `val` is a reference to the value in the vector.

So modifying it works!

Break + Practice

- What does the following code print?

```
std::vector<double> values{1.5, 2.0, 3.0};
values.push_back(5.75);
values.at(1) = -37.8;

std::cout << "Vector values<double>\n";
std::cout << "size=" << values.size() << "\n";

for (double val : values) {
  std::cout << "\t" << val << "\n";
}
```

Break + Practice

- Printed results

```
Vector values<double>
size=4
     1.5
     -37.8
     3
     5.75
```

Outline

- Object Oriented Programming
  - Writing code with objects
  - Constructors
  - Example Object: Vectors

- **Tour of GE211**

GE211

- A simple game engine designed by Jesse Tov at Northwestern!
  - Game Engine for CS211

- Source:
  - https://github.com/tov/ge211

- Docs:
  - https://tov.github.io/ge211/

High-level overview

- GE211 has a big while loop that runs 60 times per second

- Each time through the loop:
  - Checks for user inputs (mouse and keyboard)
    - Calls functions in your code providing you those details

  - Draws everything on screen
    - Calls the `draw()` function in your code to get the sprites to draw
    - Starts from scratch each time

- All of this works through C++ objects
  - Some details rely on inheritance, which we'll discuss later

Game application code structure

- Model
  - Keeps track of “game” state
  - Might have multiple helper files for various objects it needs
- Controller
  - Reads inputs from user and changes the model
- View
  - Reads from model and sets the drawing
- Lab2 combined Controller and View into a single UI
  - Homeworks will not

GE211 data structures

- GE211 also provides a bunch of data structures that you can (and will have to) use in your games

- Controls or keeps track of:
  - Time
  - Audio
  - Sprites
  - Geometry

ge211::geometry::Posn

- Docs: https://tov.github.io/ge211/structge211_1_1geometry_1_1_posn.html

- Keeps track of a 2D position!
  - **X and Y coordinates**
  - Defines various constructors
  - Methods that shift the coordinate
  - Operators for comparison and modification

  - Generic over a type `ge211::Posn<TYPE>`

ge211::geometry::Dims

- Docs: https://tov.github.io/ge211/structge211_1_1geometry_1_1_dims.html

- Keeps track of the dimensions of an object
  - **Width and height**
  - Returned as the difference between two Posn
  - Defines constructors and operators

  - Generic over a type `ge211::Dims<TYPE>`

ge211::geometry::Rect

- Docs: https://tov.github.io/ge211/structge211_1_1geometry_1_1_rect.html

- Rectangular area
  - **Posn and Dims**
  - Defines constructors and operators

  - Generic over a type `ge211::Rect<TYPE>`

Live coding: open up Lab2

- https://nu-cs211.github.io/cs211-files/lab/lab02.pdf

- https://nu-cs211.github.io/cs211-files/lab/lab02.zip

Outline

- Object Oriented Programming
  - Writing code with objects
  - Constructors
  - Example Object: Vectors

- Tour of GE211