Ben Jacobs
PE8

SECTION 1

1. What is a reference? What is a pointer? How are they different?

A pointer is a variable that holds a memory address of another stored variable address. A reference is essentially copying the value at the memory address, but not directly effecting it. A pointer can directly change and modify the original value at a memory location while a reference is creating a new version of a variable that can be repurposed without effecting the original value.

2. Given the following class definition, how would you call the method CalculateMysteries?

Foo obj(5);

obj.CalculateMysteries();

```
class Foo {
public:
  Foo(int bar) : bar_(bar) {}

  int CalculateMysteries();

  static int TotalBars();

private:
  int bar_;

  static int baz_;
};
```

3. How would you call the method TotalBars()?

Foo::TotalBars();

4. Can you access the field bar_ from inside the method TotalBars()? Why/why not?

Yes, we can access the field bar_ from inside the method, but with specific conditions. You need to get an instance of the class first. Then once it is defined, the function can access the private variables.

5. Can you access the field baz_ from inside the method CalculateMysteries()? Why/why not?

Yes you can, non-static methods can access static variables such as baz_ in the same class.

6. Create a Foo reference, then call one of the Foo methods.

Foo obj(5);

Foo &ref = obj;

Ref.CalculateMysteries();

7. Create a Foo pointer, then call one of the Foo methods.

Foo *p = new Foo(5);

p->CalculateMysteries();

8. Why would you make a field a pointer?

You would make a field a pointer if you wanted direct access to change/access that original instance.

SECTION 2

1. How would you improve the following code block:

```
bool hasACat = some value from somewhere;

If (hasACat == true) {
  // Do nothing.
} else {
  returnItForADog();
}
```

if(!hasACat){

returnItForADog();

2. How would you improve the following code block:

```
int x = some value from somewhere;
int y = some value from somewhere;

if(y > x) {
  cout << "We're messing with numbers!" << endl;
  x += 1;
} else if(x > y) {
  cout << "We're messing with numbers!" << endl;
  y += 1;
} else if(x == y) {
  cout << "We're messing with numbers!" << endl;
  x = x + y;
}
```

cout<< “We’re messing with numbers!” << endl;

if(y>x){

x++;

}

else if(x>y){

y++;

}

else{

x+=y;

}

Section 3: keywords/const/overloading

1. Why would you use a const field? Why would you use a const method?

You would use a const field so that variable can’t be modifiable. You use a const method for the same logic, you don’t want to modify the functionality of that method.

2. Where do you instantiate a const field in a class?

You instantiate a const field in a class in the constructor, it follows the definition with a colon which is considered the initialize list.

3. What does inheritance give you?

Inheritance gives you the ability to create subclasses, known as child class, which can inherit the functionality of the parent class. The child class is able to modify its own variations that were initially derived from the parent class.

3. Where and why would you use the "virtual" keyword? What is this concept called and why is it important?

You would use the “virtual” keyword for inheritance related classes. This concept is called “derived class method”. This is important/useful in the case where two inherited classes have the same method with some alterations. Referencing an instance of a class object in a function might alter the intended outcome. Using virtual allows for proper functionality

4. Why would you need to overload a comparison operator?

We would need to overload a comparison operator for comparing classes/structs. This is because the system needs guidance on what to compare due to the fact that it isn’t initially implemented in the language.

Section 4: Version control & git

1. What is a branch? Why would you work on a non-master branch?

A branch is a derived workspace from the master branch that allows for someone to change/modify the code without making immediate changes to the master. Working on a non-master branch is to ensure that the original code’s functionality is preserved if mistakes are made.

2. If you are currently working on the branch my-fabulous-feature and your teammate merges another feature into master, what are the commands that you would run to switch to master, get the new code, switch back to your branch, and merge the new code from master into your branch?

Git checkout master

Git pull origin master

Git checkout my-fabulous-feature

Git merge origin/master

3. What is the difference between issuing a "git pull" and submitting a "pull request"?

A “git pull” allows you to get the changes from the remote repository onto you own. A “pull request” asks other branches to get your changes and put your code into theirs.

4. When you're reviewing a PR, what are 3 different specific things that you should look for in the code that you're reviewing?

You should look for merge conflicts, errors, style, effectiveness, can it be smaller/improved, etc.

Section 5: bash

1. If you are developing in an IDE (such as Sublime, CodeRunner, CodeBlocks, XCode, etc) and your project isn't compiling by pressing the "run/build" button, what are two things that you could do to fix the issue? (Imagine that a friend is having this issue and you are explaining a few things that they could do to figure out what is going on).

What I have the tendency to do is incrementally test and run the code. I will construct one feature at a time to ensure that the code is running. If I am having issues getting the code to compile, I will use print statements throughout my new portion of code to see where it fails to run. If it is having errors on code that I believe should be working, I will comment out individual portions of code to narrow down where the error is.

Section 6: Unit testing/Catch2

1. What is a TEST_CASE? What is a SECTION? What should each be used for? How many methods should be tested in each TEST_CASE? what about each SECTION?

A TEST_CASE is a “test” that is run against the code you developed. This is to ensure that the code’s output matches the intended output. A SECTION allows for several TEST_CASEs to be grouped together. This is to thoroughly test the code for several components such as test cases. A TEST_CASE can have however many methods it needs to be tested to run against the expected output. A SECTION should have all cases that need to be matched with the expected output (ie edge, null, out of index,etc).

Section 7: Design Patterns

1. For each of the following design patterns, briefly describe what problem they solve **and** how they are implemented in c++.

a. Singleton – a design pattern that is used to guarantee that only one instance is created. It can be used across all the code and solves the issue of resource connection. They are implemented at the very start of a program. It's described sometimes a fancy global variable that's can coordinate actions across the system.

b. Flyweight – its an object that is useful for memory minimalization. This data is commonly shared across a large quantity of similar objects to reduce the total amount of data stored. They are implemented when creating subclass objects that could reference the same data with no need for individual object to be created.

c. Prototype – is a precursive declaration of a method to tell the system of the return type, to tell the number of arguments passed into the function, types of data that's passed in, and the order. Often the function is declared below the main() and could have conflicts, prototype is defined before main() and to cast over to ensure there is no errors.

d. Factory – this is a utility class that creates a class instance from a group of derived/child classes. This can be used to have an unknown parameter type be passes to create an instance of a class that could be of one of several variable subclasses.

e. Iterator – iterator is used to traverse elements of a collection/container. Every loop goes to the next value in that collection of data. Once the iterator is created, it can reference specific datatypes if the collection/container has more variables assigned to it. We commonly use it for maps in our programs to access the first and second value, which could be of foreign datatypes to the language (ie Party for hw3).

2. In class, we went over how you will frequently interact with the design pattern Iterator but will rarely implement it yourselves. Why is this?

We will hardly use iterators because there are better methods of traversal of a data collection that are already implemented in the language. We can only traverse that one type of

collection while other iterators (for, while) are able to move from one data type to a different data type.

Section 8: templating/writing generalizable code

1. Write a templated function `void Swap(T & a, T & b)`.

Template <typename T>

Swap(T & a, T & b){

T temp = a;

a = b;

b = temp;

}

2. Write a `main()` in which you make at least two calls to `Swap` that do work and two calls to `Swap` that you would expect to work but that don't.

```
Int main(){
        int a = 1;
        int b = 2;
        char c = 'x';
        char d = 'y';
        string e = "Hello";
        string f = "World";
        //WORKS
        Swap(a,b);
```

```
Swap(c,d);


//DOESN'T WORK
Swap(a,c);
Swap(a,f);
}
```

3. Adjust the non-working function calls so that they do work and write comments about why they did not initially work and what changes you made.

Instead of:

Template <typename T>

The change should be:

Template <struct T>

Since you cannot template two different variable types, you would need to cast it as one of the types or create a struct that could handle the thrown error. You can implement several data types into that struct and once instantiated can be passed through properly.

5. Why/why shouldn't you write all functions in c++ as templated functions?

There are a lot of issues associated with templated functions that can be easily dismissed if you were to instead use a normal method construction. The best way to utilize these functions is through structs to guarantee no type mismatch or other issues that may arise with the parameters/outcomes.

Section 9: GUIs/Qt

1. What happens behind the scenes for a GUI to respond to a user interaction with the user interface?

If a user interacts with the UI, a specific slot is triggered from that signal. A slot is connected to a method which provides the interface with an intended response.

2. In Qt, how many signals can a single object emit? How many slots can respond to a single signal?

A single object can emit several signals and several slots can respond to a single signal.

3. Give example signatures for a custom signal that has at least one parameter and the slot that you would connect it to. Give an example call to `connect` that you would use to link the to signal to the slot. Describe when you would call `emit` for this signal. You may need to do some research on your own to answer this question!

Answer on next page:

Signal
void PlotWindow::Clicked(int x);

Slot
void PlotWindow::SlotTest(int x){
qDebug() << x;
qDebug() << “Is the x value”;
}

```
connect(ui->randomButton, &QPlotWindow::Clicked(int x), this, &PlotWindow::SlotTest);
```

Emit will literally emit the signal, all objects whose slots have been connected to that type of object will be emitted.