UPE Tutoring:

CS 31 Final Review

Event Sign-in https://tinyurl.com/cs31finalreview

Table of Contents

Pointers

Pointers

- A **pointer** is the memory address of a variable.

- The **&** operator can be used to determine the **address** of a variable to be stored in the pointer.
- The ***** operator can be used to **dereference** a pointer and get the value stored in the variable that is being pointed to.

```
double d = 10.0;
double *dp;          // pointer that holds the address of a double
dp = &d;             // stores the address of d into dp
*dp = 20.0;          // changes the value of d from 10.0 to 20.0
```

Pointers

```
int var = 20;   // actual variable declaration
int *ip;        // pointer variable declaration

// store address of var in ip
ip = &var;

cout << "Value of var variable: ";
cout << var << endl;

// print the address stored in ip pointer
cout << "Address stored in ip variable: ";
cout << ip << endl;

// access the value at address stored in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;
```

```
> Value of var variable: 20
> Address stored in ip variable: 0xBFC601AC
> Value of *ip variable: 20
```

Pointer Arithmetic

```
int arr[] = {10, 20, 30, 40};

int *ptr = arr;                // arr == &arr[0]
cout << *ptr << endl;

ptr++;
cout << *ptr << endl;

ptr = ptr + 1;
cout << *ptr << endl;

ptr--;
cout << *(ptr + 2) << endl;
```

```
> 10
> 20
> 30
> 40
```

Pointers – new and delete

- The **new** operator can be used to create **dynamic** variables. These variables can be accessed using pointers.

```
string *p;
p = new string;
p = new string(“hello”);
```

- The **delete** operator eliminates dynamic variables.

```
delete p;
```

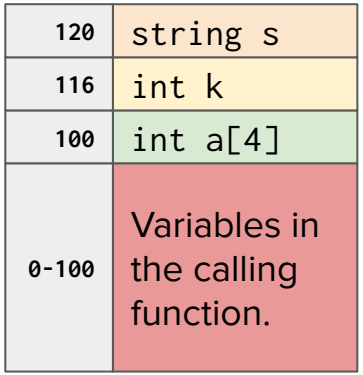
- Note: Pointer p is now a **dangling pointer**! Dereferencing it is dangerous and leads to undefined behavior. One way to avoid this is to set p to NULL after using delete.

Pointers – the Heap and the Stack

- As it turns out, there are **two** places where your variables live.
- The first is the **stack**, which is the place you're most familiar with. With **local variables**, the compiler is like a city planner who decides where each variable should live.

```
void foo() {
  int a[4];  // Stored at 100
  int k;     // Stored at 116
  string s;  // Stored at 120
}
```

When foo called



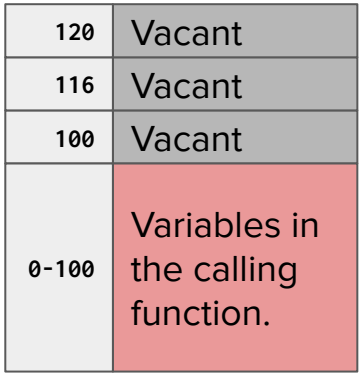
If the size isn't specified at compile time, how would the compiler know where to put k or s?

Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The first is the **stack**, which is the place you're most familiar with. With **local variables**, the compiler is like a city planner who decides where each variable should live.

```
void foo() {
  int a[4];  // Stored at 100
  int k;     // Stored at 116
  string s;  // Stored at 120
}
```

When foo returns

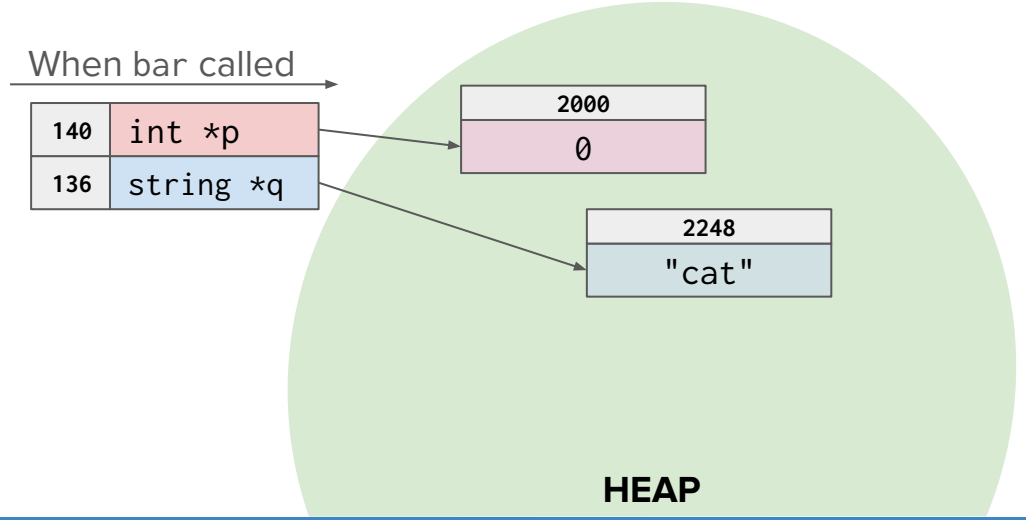


When the function returns, the variables are evicted from their addresses.

Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The second is the **heap**, which is the place where dynamic variables live. Dynamic variables essentially lease some part of the heap to live in.

```
void bar() {
  int *p = new int(0);
  string *q = new string("cat");
}
```

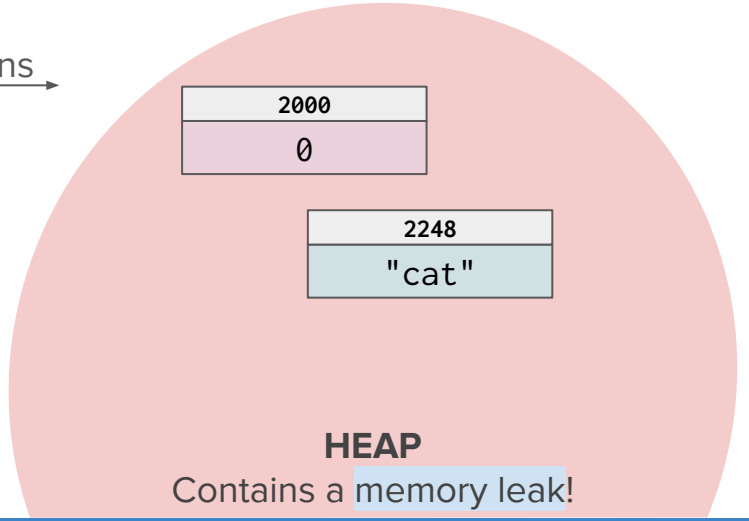


Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The second is the **heap**, which is the place where dynamic variables live. Dynamic variables essentially lease some part of the heap to live in.

```
void bar() {
  int *p = new int(0);
  string *q = new string("at");
}
```

When bar returns

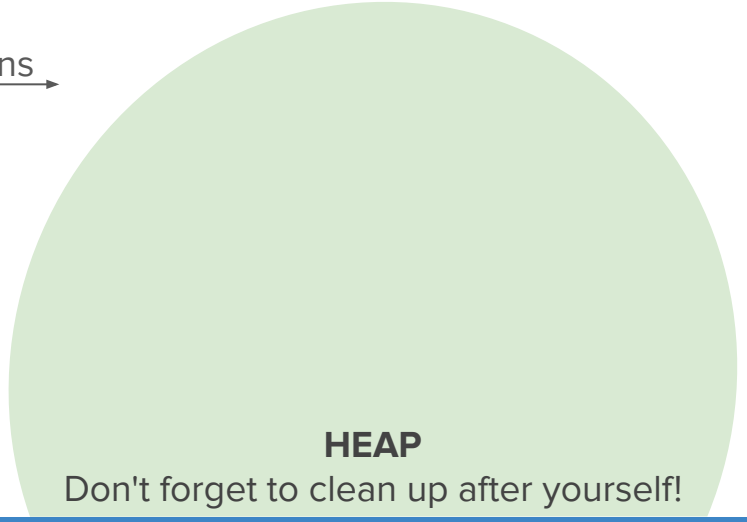


Pointers – the Heap and the Stack (cont.)

- As it turns out, there are **two** places where your variables live.
- The second is the **heap**, which is the place where dynamic variables live. Dynamic variables essentially lease some part of the heap to live in.

```
void bar() {
  int *p = new int(0);
  string *q = new string("cat");
  delete p;
  delete q;
}
```

When bar returns



Practice Question: Array Traversal w/ Pointers

Write a function that sums the items of an array of n integers using only pointers to traverse the array.

Solution: Array Traversal w/ Pointers

Simple array traversal, but with pointers. To get to item `i`, add `i` to your head pointer then dereference. This works because the compiler knows the size of an item in your array in C++. Sum values as you go by adding them to a `total` value created outside of the for loop.

```
int sum(int *head, int n) {
  int total = 0;
  for (int i = 0; i < n; i++) {
    total += *(head + i);
  }
  return total;
}
```

```
sum(arr, 5);
```

Practice Question: C String Reversal with Pointers

Implement the function `reverse`, which takes a C String as an argument and **prints out** the characters in reverse order. You are not allowed to use the `strlen` function, and you must use pointers in any traversal of the C String.

```
void reverse(const char s[]);

int main() {
    char str[] = “stressed”
    reverse(str);
    // OUTPUT: desserts
}
```

Solution: C String Reversal with Pointers

```
void reverse(const char s[]) {
    const char *p = s; // create a new pointer for our traversal
    while (*p != '\0') { // move the pointer to the end of the C String
        p++;
    }
    p--; // set p to point at the last char in the C String
    while (p >= s) { // print out chars as we traverse back to the beginning
        cout << *p;
        p--;
    }
    cout << endl;
}
```

Practice Question: strcat

Implement the C string concatenation function. The function takes two C strings and copies the chars from the source C string to the end of the destination C string. The original null byte of the destination is overwritten when copying the source. Return the destination pointer at the end of the function. You do not know the size of the destination and source C strings (so you can't create a temporary C string to store all of the characters!)

```
char* strcat(char* destination, const char* source);
```

Solution: strcat

```
char* strcat(char* destination, const char* source) {
    char* d = destination;
    while (*d) // this loop sets d to point at the null byte of destination
        d++;
    const char* s = source;
    while (*s) { // this loop copies the source C string to where d is pointing
        *d = *s;
        d++;
        s++;
    }
    *d = '\0'
    return destination; }
```

Structs & Classes

Structs

- A **struct** is a collection of data that is treated as its own special data type. We use them to organize data that belongs together.

```
struct Person {
  int age;
  string name;
};  // Must end with semicolon!
```

- Structs can store any number of any other data type, accessed using `.` (the **dot operator**). These stored values are called **member variables**.

Structs

You can declare a struct outside of any functions and treat it as a normal data type, for the most part. A struct can even contain another struct!

```
struct Date {
  int day, month, year;
};

struct Person {
  Date birthday;
  string name;
  double money;
};
```

```
void doubleMoney(Person& guy) {
  guy.money *= 2;
}

int main() {
  Person p1;
  p1.name = “Smelborp”;
  p1.money = 3.50;
  p1.birthday.day = p1.birthday.month = 1;
  Person p2 = p1;  // Perfectly legal
  doubleMoney(p2);
  cout << p2.money; // 7
  cout << p1.money; // 3.5
  Person p3 = { p1.birthday, “Jimbo”, 3.5 };
  p2 += p1;  // ERROR! How do you add people?!
}
```

Structs – Tips

- Structs are a good way to keep code looking organized and readable
- When declaring a struct, member variables with primitive types will be left uninitialized. Classes will be constructed with their default constructor.
  - Long story short: you must assign them yourself!
- **Always remember the semicolon!** It's there so that you can declare struct variables at the same time that you define the struct.

```
struct Circle {
  int radius;
} ring, hoop;    // This creates two Circle structs named ring and hoop
```

Classes

- A **class** is like a struct, but with private default access (more on this soon). Classes are at the core of Object-oriented Programming (OOP).
- Classes often have **member functions** in addition to member variables

```
class Person {
  int age;
  string name;          // Strings are actually objects from the string class!
  double money;
  void doubleMoney();   // Member function, declared but not yet implemented.
};
```

- We call an instance of a class an **object**. In this way, OOP involves different types of objects interacting with each other.

Classes – Member Functions

Now, let's fill in the `doubleMoney` function of the `Person` class from the previous slide. There are two ways to do this:

```
// 1. Inside the class definition.
class Person {
  int age;
  string name;
  double money;
  void doubleMoney() { money *= 2; }
};

// 2. Outside of the class definition.
Person::doubleMoney() {
  money *= 2;
}
```

The **scope resolution operator** (e.g. `Person::`) tells the compiler that we are defining the `doubleMoney` function of the `Person` class.

Note: we don't need the dot operator to refer to `Person`'s `money` variable when we are in one of `Person`'s member functions!

Classes – Access Specifiers

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;
  bobby.name = “Bobby”;
  bobby.money = 3.49;
  bobby.doubleMoney();
}
```

Classes – Access Specifiers (cont.)

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;  //ERROR!
  bobby.name = "Bobby";  //ERROR!
  bobby.money = 3.49;  //ERROR!
  bobby.doubleMoney();  //ERROR!
}
```

Classes – Access Specifiers (cont.)

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;  // Good to go :^)
  bobby.name = "Bobby";
  bobby.money = 3.49;
  bobby.doubleMoney();
}
```

The compiler doesn't let us access any of bobby's members! This is because class members have the **private** access specifier by default and cannot be accessed by an outside class or function. To fix this, we adjust our class:

```
class Person {
public:
  int age;
  string name;
  double money;
  void doubleMoney() { money *= 2; }
};
```

Classes – Access Specifiers (cont.)

- The reason we were able to access the members of a struct earlier is because they are **public** by default.
- One big problem with our `Person` class so far is that if we make its member variables `age`, `name`, and `money` private, we have no way to change them. If we make them public, they can be set to an invalid state by any code that instantiates a `Person` object!

```
int main() {
  Person p;
  p.age = -5;   // This doesn't make sense (unless we're Benjamin Button).
}
```

Classes – Encapsulation

To fix this, we make Person's member variables private and add public **accessor** (getter) and **mutator** (setter) functions that set rules on how to access and change them. This is called **encapsulation**!

```
class Person {
public:
  void setAge(int yrs);
  int getAge();
  void setName(string nm);
  string getName();
  void doubleMoney();
  double getMoney();
private:  // Same member variables!
};
```

```
void Person::setAge(int yrs) {
  if (yrs >= 0)
    age = yrs;
}
int Person::getAge() { return age; }

void Person::setName(string nm) {
  if (nm.length > 0)
    name = nm;
}
string Person::getName() { return name; }

void Person::doubleMoney() { money *= 2; }
double Person::getMoney() { return money; }
```

Classes – Encapsulation (cont.)

- Generally, we want to make our member variables private. Therefore, we make them accessible through public member functions that access or mutate them in ways that are reasonable towards our implementation.
  - This keeps objects in a valid state and hides the “nitty gritty” details of our implementation from anyone who wants to use our class
- Now, you just have to call `doubleMoney` on a `Person` and you know that their money will be doubled.

Encapsulation Example

```
int main() {
  string name;
  cout << "What is my name? " << endl;
  getline(cin, name);

  Person p;
  p.setName(name);
  p.setName("");
  cout << "I Am " <<
      p.getName() << "\n";
  p.setAge(49);
  p.setAge(-1);
  cout << "I am " << p.getAge() <<
      " years old.\n";
}
```

```
> What is my name? the Walrus
> I Am the Walrus
> I am 49 years old.
```

Classes vs. Structs

- Technically, the only difference between a class and a struct is the default access specifier.
- By convention, we use structs to represent simple collections of data and classes to represent complex objects.
    - This comes from C, which has only structs and no member functions.

Constructors

Constructors

- There is yet another problem with our `Person` class: initializing its members one-by-one is annoying but if we don't do it, our `Person` starts in an invalid state!
- A **constructor** is a member function that has the **same name as the class**, **no return type**, and **automatically performs initialization** when we **declare an object**:

```
class Person {
public:
  Person();
  // Same stuff as before!
};
```

Constructors (cont.)

- Constructors can be defined **inside or outside of a class**, just like normal functions.
- Like normal functions, constructors **can be (and usually are) overloaded** with different numbers and types of parameters to suit different purposes.
- Unlike normal functions, they **cannot be called with the dot operator**.
- A **default constructor** is one with **no arguments**; the compiler generates an empty one by default.
    - The default constructor leaves primitive member variables (int, double, etc.) uninitialized and calls the default constructors of class members
    - Example: any string members will be created with the default string constructor

Constructors – Basic Syntax

Let's add constructors to our Person class!

```
class Person {
public:
  Person();                                  // 1
  Person(int yrs, string nm, double cash); // 2
  // Same stuff as before!
};

Person::Person() { // implementation of c'tor 1
  age = 0;
  name = "";
  money = 0.0;
}
```

```
Person::Person(int yrs, string nm, double cash) {
  // implementation of c'tor 2
  setAge(yrs);
  setName(nm);
  money = cash;
}

int main() {
  Person p1;  // c'tor 1 is called.
  Person p2(44, "Elon Musk", 13000000000.0);
  // Constructor 2 is called.
  p1 = Person(19, "Freshman at UCLA", -100000.0);
  Person p3(); /* Constructor 1 NOT called:
                  The compiler thinks we're
                  defining a function!   */
  p1.Person(); // Illegal!
}
```

Constructors – Pitfalls

- If we declare a constructor, the compiler will not longer generate a default constructor!

```
int main() {
  Person p1;  // Illegal if no defined default constructor!
  Person p2(5, "Squam", 3.51);
}
```

Practice Question: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }
};
```

```
int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

What is the output of this code?

Practice Question: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};


int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }
};
```

WAIT!! There’s a memory leak!!

How do we fix it?

Solution: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};


int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }

    ~Person() {
      delete m_cat;
    }
};
```

Destructors

Destructors

- A **destructor** is a member function that is **called automatically** when an object of a class passes out of scope
  - The destructor should **use `delete` to eliminate any dynamically allocated variables** created by the object
- For example, suppose that our `Person` class creates a **dynamically allocated** `Pet` type object. This memory would need to be freed in the destructor!

Destructors (cont.)

Let's add a destructor to our class!

```
class Pet { ... };

class Person {
public:
  Person();
  ~Person();
  // Same stuff as before ...
private:
  Pet* fluffy;
};
```

```
Person::Person() {
  // Same stuff as before ...
  fluffy = new Pet("Steve");
}

Person::~Person() {
  delete fluffy;
}
```

Practice Question: Destruction

What is the output of the following code snippet?

```
class Cat {
public:
  Cat(string name) {
    cout << "I am a cat: " << name << endl;
    m_name = name;
  }
  ~Cat() { cout << "Farewell, meow." << endl; }
private:
  string m_name;
};
```

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = new Cat("Alfred");
    m_age = age;
  }
  ~Person() { cout << "Goodbye!" << endl; }
private:
  int m_age;
  Cat *m_cat;
};

int main() {
  Person p(21);
}
```

Solution: Destruction

We would expect the following output:

```
I am 21 years old. I am a cat: Alfred
Goodbye!
```

Notice that the **destructor for `m_cat` is never called**: this is a memory leak, which should be **addressed by adding `delete` in the `Person` destructor.**

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = new Cat("Alfred");
    m_age = age;
  }
  ~Person() {
    cout << "Goodbye!" << endl;
    delete m_cat; // memory leak fixed!
  }
private:
  int m_age;
  Cat *m_cat;
};
```

Practice Question: Destruction (cont.)

```
class Cat {
public:
  Cat(string name) {
    cout << "I am a cat: " << name << endl;
    m_name = name;
  }
  ~Cat() { cout << "Farewell, meow." << endl; }
private:
  string m_name;
};
```

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = new Cat("Alfred");
    m_age = age;
  }
  ~Person() { cout << "Goodbye!" << endl;
    delete m_cat;}
private:
  int m_age;
  Cat *m_cat;
};
```

```
int main() {
  Person p(21);
}
```

Pointers to Objects – Arrow Operator

Like the dot operator, the **arrow operator** `->` can be used to access an object's member variables and functions. The arrow operator is used when we have a pointer to the object whose members we are trying to reference.

```
int main() {
  Person* p = new Person;
  p->setAge(20);
  p->setName("Bob");
  double money = p->getMoney();
  p->age = 10;        // ERROR: age is not a public variable!
}
```

Note: `p->setAge(20)` and `(*p).setAge(20)` are equivalent statements!

Pointers to Objects – The `this` Pointer

When defining member functions for a class, we sometimes want to refer to the calling object. The `this` pointer is a predefined pointer that points to the calling object.

```
int Person::getAge() {
  return age;
}
```

```
int Person::getAge() {
  return this->age;
}
```

Pointers to Objects – The `this` Pointer

- The `this` pointer allows us to access member variables even when they are shadowed by local variables.

```
Person::setAge(int age) {
    this->age = age;
}
```

- The `this` pointer also allows us to pass the current object into a function that takes an argument of its class.

```
void printPerson(Person *p);
Person::print() {
    printPerson(this);
}
p1.print()
```

Function Overloading

- You can have multiple definitions for the same function name in the same scope. However, the definition of the functions must differ from each other by the types and/or the number of arguments in the argument list.

```
class printData {
   public:
      void print(int i) {
         cout << "Printing int: " << i << endl;
      }

      void print(int i, double f) {
         cout << "Printing numbers: " << f << ' ' << i << endl;
      }

      void print(char* c) {
         cout << "Printing character: " << c << endl;
      }
};
```

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print numbers
   pd.print(42, 500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Operator Overloading

- You can also overload basic operators in C++ (such as +, -, <<, etc.) so they work with user-defined structs and classes as well.

```
class Vector {
   public:
      double getX() {
         return m_x;
      }

      double getY() {
         return m_y;
      }

      void setX(int x) {
         m_x = x;
      }

      void setY(int y) {
         m_y = y;
      }
```

```
Vector operator+(const Vector& vec) {
         Vector newVec;
         newVec.setX(this->m_x + vec.getX());
         newVec.setY(this->m_y + vec.getY());
         return newVec;
      }
  private:
      double m_x;      // x component of vector
      double m_y;      // y component of vector
};
```

Good luck!

Sign-in/feedback	https://tinyurl.com/cs31finalreview
Slides	https://tinyurl.com/cs31finalreviewslides
Practice	https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.