Intro To C++ And Algorithms

1. Intro (1.5 hrs)
    1. Introduction to C++ (hello world, comparison to Python)
    2. I/O with cin and cout
    3. Types
2. Loops and Number Systems
    1. Decimal, binary, and hexadecimal
    2. Binary fractions
    3. “for” and “while” loops in C++
3. Strings, Functions, and Pointers
    1. Writing our own functions
    2. Passing pointers as function parameters
    3. Using strings with cin and getline()
4. Arrays and Vectors
    1. Static arrays
    2. Dynamic arrays
    3. The vector class
5. Algorithms – Sorting
    1. Insertion Sort
    2. Bubble Sort
6. The Preprocessor
    1. #define and #include
    2. #define macro usage
    3. Words of warning
7. Structs and Classes
    1. Defining and using structs
    2. Defining and using classes
8. Operator Overloading
    1. Constructors
    2. Function overloading
    3. Operator overloading
9. Recursion
    1. The relationship between loops and recursion
    2. Solving a common recursive problem (factorial)
    3. Generating the Fibonacci series

Homework #1 - C++ Intro

Name:

1 Intro

When people need to write efficient, high speed programs, they often turn to languages like C or C++. This is true because these languages give you a lot of control over how data is stored and manipulated inside the computer. You decide where things will go in memory, how much memory you will need, and how things will be accessed, changed, and stored. This is also the reason that C++ and C are used in programming competitions and in algorithms classes: this flexibility gives you the power to write everything yourself, be it for the speed boost you need in competition, or just to prove you understand it all.

Python is an **interpreted** language, which means that Python programs cannot run by themselves...they have to be read and run line-by-line using a computer program called an **interpreter**. In Python, the interpreter is a program called - you guessed it - Python! C++ programs don't work this way. There's a program called a **compiler** that reads our whole program and turns it into something called **machine code**. This is a sequence of 0's and 1's which the computer can run by itself without any help from external programs (well...almost without any help). This helps to make C++ programs fast, but means the syntax (the way the language looks, and how we write it) is more complicated. Here is the C++ program to say "Hello world"

```
#include <iostream>

using namespace std;

int main(){
  cout << "Hello world" << endl;
  return 0;
}
```

You can see it's a lot longer than the Python "Hello world", which is just:

```
print "Hello world"
```

All that extra stuff helps the compiler decide how to turn the program into machine code (0's and 1's)...Python programs have the Python **interpreter** to help them when they are run, so they don't need to include as much detail.

2 Input and Output in C++

In C++, input and output is handled a little bit differently than in Python. Inputs come from the "input stream" which is called "cin". Outputs go to the "output stream" or "cout", like this:

```
1
// A demonstration of I/O
3
#include <iostream>
5 using namespace std;
int main(){
7     int x = 0;
      cout << "Enter a number: ";
9     cin >> x;
      cout << "Your number was: " << x << endl;
11    return 0;
}
```

ioTest.cpp

Anything the user types in will be sent to **cin**. You can get individual words, letters, or numbers from cin using the "»" operator. We do this in the program above to read in a single number with "cin » x;". Anything you want to put on the screen goes to **cout**. You can send things to cout using the "«" operator, as in "cout « x;" which will print the contents of variable x on the screen.

3 Data Types

In Python, the interpreter handled a lot of the "hard work" of representing numbers and other data for us, in a behind-the-scenes way. We didn't have to worry about if our numbers were floats or integers. We didn't need to know what would happen if a number was too big or whether it was positive or negative. We didn't worry about how long lists or strings were. The computer took care of all of that for us. With C++, this is not the case. I said before that computers treat all data - no matter what it is - as sequences of ones and zeros. What we should be telling the computer to **do** with that data depends on its **type**. Is our data signed numbers? Unsigned? The start of a string or array? When we want to use variables in C++, it's our job to **declare** them. We must tell the computer how we plan to use our variables and what kind of data they will store. This is important because choosing the wrong "type" can cause problems like **rollover** or **overflow** which we've talked about in class. For instance, we store numbers like "5" and "6" as **int**, numbers like "2.5" or "0.3" as **float**, single letters are **char** and whole words as **string**. We can store these different types like this:

```
1 int x = 5; // An integer
float y = 2.2252; // A float
3 char letter = 'x'; // A single letter
string word = "stuff"; // A whole word
```

Note that in C++ the single quotes (' ') are used **only** when you want to save a single letter. Words **must** use double quotes (" "). Compare this to Python where you can use single and double quotes wherever you like. To see what can happen when you choose the **wrong** type for your data, have a look at the program on the next page.

```
// Demonstrates the concept of rollover (tested on 64-bit Intel machine)

#include <iostream>
#include <stdio.h>

using namespace std;

int main(){
    unsigned char x = 0;
    int y = 0;

    cout << "int has size " << sizeof(y) << " byte(s)" << endl;
    cout << "unsigned char has size " << sizeof(x) << " byte(s)" << endl;

    for (y = 0; y < 500; y++) {
        printf("y = %d, x = %u\n", y, x);
        x++;
    }
}
```

rollover.cpp

Try running this on your computer. What happens?

4 If/else

If/else works pretty much the way we are used to. The only difference is that the conditional has to have parentheses around it, as in $(x > 5)$ or $(x == 2)$, like this:

```

// A demonstration of branching

#include <iostream>

using namespace std;

int main(){
    int x = 5;

    if (x > 1) {
        cout << "STUFF" << endl;
    }
    else {
        cout << "THINGS" << endl;
    }
}
```

conditionals.cpp

Also, the if/else statements in C++ don't use whitespace to indicate where they start and end, instead they use curly braces ({}).

5 Now You Try

Doing Simple Math

Write a program that can ask for two numbers, add them, and print the answer. Like this:

```
Enter two numbers: 2 5
Sum: 7
```

Making Choices

Write a program that asks you to enter a number. If the number is bigger than 100 it says "That's big!". If the number is smaller than or equal to 100, it says "Wow, that's small!"

```
Enter a number: 100
Wow, that's small!
```

Or...

```
Enter a number: 101
Wow, that's big!
```

Homework #2 - Loops

Name:

1 For loops

Just like Python, C++ has both ”while” and ”for” loops. A ”for” loop in C++ is written like this:

```
#include <iostream>

using namespace std;

int main() {

  for (int i = 0; i < 10; i++) {
    cout << i << endl;
  }

  return 0;
}
```

This code will print out the numbers from 0 - 9. In Python, you'd print the numbers from 0 - 9 like this:

```
for i in range(0,10):
  print i
```

What if we wanted to go through a list of both numbers and words? In Python, we could easily do that like this:

```
for i in ['banana', 'apple', 36]
  print i
```

Unfortunately, there is nothing like this in C++. For loops can only count up or down, they can't actually go ”item by item” through a list containing different types of data. This is one of the big differences between C++ and Python.

2 While loops

While loops in C++ are a lot closer to their Python counterparts. A while loop looks like this:

```
#include <iostream>

using namespace std;

int main() {

  int i = 0;
  while (i < 10) {
    cout << i << endl;
    i++;
  }

  return 0;
}
```

One new thing here is the "i++". In C++ this means "add 1 to i". It's a bit like doing "i += 1" in Python...it's just shorter than writing "i = i + 1" and programmers love keeping things short!

3 Now You Try

3.1 Summing up numbers

Write a short C++ program which asks for a number, then adds up all the numbers from 1 up to that number. The output should be something like this:

```
Enter a number: 5
The sum is: 15
```

3.2 Countdown

Try writing a short program which counts from -1 to -99 by twos, like this:

```
-1
-3
-5
-7
...
-99
```

When should the loop stop? How much should you add or subtract from "i" each time the loop runs?

Homework #2 - Binary Numbers

Name:

1 Intro

If we want to write good computer programs, we need to understand a little bit about the inner workings of computers. All modern computers use electronic devices to store and manipulate data and programs. To help make them simpler, these devices are built to work with only two voltages, an "ON" and an "OFF" voltage (usually 0V and 3.3V or 0V and 5V). Because of this, all data in the computer is represented using the numbers "0" and "1" (the 0 is the "OFF" state and the 1 is the "ON" state). Everything, and I mean **everything** that the computer can do is done using only these two numbers. Videos, text, pictures, and computer programs themselves are all stored inside the computer as sequences of ones and zeros. We have to understand how these numbers work (how to do math with them, compare them etc...) in order to write good computer programs. I have introduced you to binary, decimal, and hexadecimal numbers in class before. This homework tests your ability to understand and convert between these three types of numbers.

2 Binary Conversion

Convert the following numbers into binary (**hint**: use the repeated division method we talked about in class):

1. $45_{10} =$
2. $255_{10} =$
3. $0_{10} =$
4. $36_{10} =$

What would happen if you tried to convert 256 into an 8-bit binary number? Can you do this or do you need more than 8 bits?

3 Binary to Decimal Conversion

Convert the following binary numbers into decimal:

5. $101_2 =$
6. $00001111_2 =$
7. $10000000_2 =$
8. $10101010_2 =$

4 Addition for positive numbers

Add these binary numbers together:

9. $1010_2 + 0010_2 =$
10. $0010_2 + 1110_2 =$
11. $11110010_2 + 00101110_2 =$

Think about what happens if you try to add two 8-bit numbers together and the result uses more than 8-bits. What will the computer do?

5 Forming the two's complement

In the computer, positive and negative numbers are **not** stored using the "sign magnitude" form we are used to, where a number has a + or - in front to show its sign. Instead, negative numbers in binary are represented by forming the "two's complement" of a positive number with the correct magnitude (size). For instance, in binary the number "5" is represented by "101" or if our computer uses 8-bit memory, "00000101". That is the same as "+5". How do we make "-5"? We have to change all of the "1"s to "0"s and then add an additional "1" to the number. Like this:

$$00000101_2 \rightarrow 11111010_2 + 1_2 \rightarrow 11110101_2$$

So "-5" is actually represented by "11110101". You try! Convert the following **negative** decimal numbers to their negative binary equivalents:

12. $-45_{10} =$
13. $-255_{10} =$
14. $-0_{10} =$
15. $-36_{10} =$

Hint: Since these are the same decimal numbers you converted to binary at the beginning of the homework, you can use the binary conversions you already did to form the two's complement. What happens when you form the two's complement of 0? Does this seem like the right behavior?

6 Representing fractions

In decimal or base 10 numbers, a fraction like 0.255 is represented as:

$$2 \cdot 10^{-1} + 5 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

You can see that it's the same system we use for representing integers (225, 34) but with negative exponents. We can do this with binary numbers as well. For instance 0000.1000 in binary would convert to the following decimal number:

$$1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} =$$

$$0.5 + 0 + 0 + 0 = 0.5$$

So $0.1_2 = 0.5_{10}$. Try it yourself for the following numbers:

16. $0000.1010_2 =$
17. $0000.0010_2 =$
18. $1010.0101_2 =$
19. $1100.0011_2 =$

Now try to go the other way!

20. $0.5_{10} =$
21. $0.1_{10} =$
22. $2.5_{10} =$
23. $45.25_{10} =$

Notice that some numbers do not have an exact representation when you convert them from one base to another. 0.1 in decimal has no exact representation in base 2 because it repeats forever!

7 Understanding Hexadecimal

Converting between bases is easier when they share the relationship $R_1 = R_2^K$. This is the case for base 2 and base 16 where $16 = 2^4$, and because of this base 16 is a popular way to represent numbers when talking about computer systems. Hexadecimal numbers are shorter than their binary equivalents which makes them easier to read and write, and there's a lower chance of making mistakes when writing them. In hexadecimal, there are actually 16 different symbols instead of the 10 (0 to 9) we use in decimal. In hexadecimal, the first 10 digits are represented by the numbers 0 to 9 just like in decimal, but the last 6 digits are represented by the letters A to F, where A = 10, B = 11, ...up to F = 15

Because the bases are closely related, you can easily convert to and from hexadecimal and binary almost just by looking at a number. Each hexadecimal digit represents 4 binary digits, so the number A in hex would be 1010 in binary, or 10 in decimal. The number F is decimal 15 or 1111, so FF would be 11111111, and so on... Try it yourself for the numbers below:

24. $FE_{16} =$
25. $FF_{16} =$
26. $DEAD_{16} =$
27. $ABCD_{16} =$

Now try to go the other way, converting into hexadecimal (note if a binary number does NOT break evenly into groups of 4 digits, just add zeros on the left side of the number until it does!):

28. $1010_2 =$
29. $0010_2 =$
30. $001_2 =$
31. $10101111_2 =$

Homework #3 - Handling Strings

Name:

1 String I/O

In Python, working with strings was easy! We never had to worry about how much space a string took up or what we would do if there were spaces in it. In C++ we do have to worry about these things. When we want to handle strings, we have to create a variable with the **string** type. Here's an example:

```
1
#include <iostream>
3 #include <string>

5 using namespace std;

7 int main() {
      string name;
9
      cout << "Enter your name: ";
11    cin >> name;
      cout << "Hello " << name << endl;
13
      return 0;
15 }
```

stringIO_1.cpp

This will work fine for single-word inputs like "steve" or "jeremy" but would fail on "jeremy pedersen" or "i am a hippo" because "cin" treats words with spaces separating them as separate strings. To deal with this, C++ provides us a function called "getline()". It works a lot more like Python's raw_input():

```
2 #include <iostream>
  #include <string>
4
  using namespace std;
6
  int main() {
8     string words;

10    cout << "Enter a sentence: ";
      getline(cin, words);
12    cout << "You said: " << words << endl;

14    return 0;
  }
```

stringIO_2.cpp

2 Functions

Just like Python, C++ has a way for us to make and use functions. Functions in C++ look like this (pay attention to the top part of the program):

```
1
#include <iostream>
3
using namespace std;
5
int add(int a, int b) {
7     return a + b;
}
9
int main() {
11    int a, b;

13    cout << "Enter 2 numbers please: ";
      cin >> a >> b;
15
      cout << "The total is: " << add(a,b) << endl;
17    return 0;
}
```

function.cpp

Instead of using "def" we start our functions by saying what **type** of data they will return. And instead of using tabs to say what belongs inside the function, we use the braces "{" and "}". Otherwise, using functions is very much like you are used to from Python.

3 Now You Try

Multi-word Strings

Write a short C++ program which can ask for a name and print it back out. It should work for names that are **more than one word long** (i.e. names that have spaces in them such as "Kevin Gu" or "Jeremy Pedersen"). It should look like this when it runs:

```
Enter a name: Jeremy Pedersen
Hello, Jeremy Pedersen
```

Inputting numbers

Write a program which can change temperatures from C to F. For instance, if you type in 100, the program should give you the answer 212 (because 100 C = 212 F). You can use the formula $F = \frac{9}{5} \cdot C + 32$ to do the conversion from C to F. You should put the code to change temperature **inside a function** like this one:

```
float tempConv(float c) {
2   f = // Your code here
    return f;
4 }
```

Homework #3 - Functions and Pointers

Name:

1 Pointers

Pointers are special variables that hold addresses. So a pointer doesn't tell you what your data is, it tells you **where to find it**. In Python, we did not use pointers because we didn't need to know where our data was: we always let the Python interpreter manage that for us.

Sometimes in C++ we need to manage our own data, or even ask the computer for extra space (for instance, if we need to make an array bigger). When we do this, the computer will give us a **pointer** telling us where in computer memory we can put our data.

Pointers are also great if we need to have a function that can change its own **operands** (the data we put into the function's parentheses "()"). For instance we could write something like this:

```
1
#include <iostream>
3 #include <vector>

5 using namespace std;

7 void minmax(int* a) {
      if (*a > 1) {
9         *a = 1;
      }
11    if (*a < 1) {
          *a = 0;
13    }
}
15
int main() {
17
      int x;
19
      cout << "Enter a number: ";
21    cin >> x;

23    minmax(&x);

25    cout << "Result is: " << x << endl;

27    return 0;
}
```

pointerFunc.cpp

Notice that the function does NOT have to **return** any numbers (i.e. give back an answer). Instead, it directly changes the data we gave to it.

2 Now You Try

2.1 Temperature Conversion, Again

Write another function to convert temperatures from C to F. The difference is that this function should NOT return any numbers. So it will look like this:

```
void changeTemp(float* temp) {
  *temp = // Your code here
}
```

Note the "*" which tells the computer that the variable "temp" is actually a pointer. Also pay close attention to the example code on the page above this one. They use "*" whenever they change the data pointed to by the pointer (as in *a = 0). Doing "*a" tells the computer "go get the data at the address saved in "a", and change it". This is how the function in the example code can change "a" **directly**. Compare that to our last program, where our function had to do something like this:

```
F = tempConv(C);
```

Notice that C is NOT changed by the tempConv function. Instead, tempConv **returns** a value which is then saved into F.

2.2 Range Limiter

There are lots of science and engineering problems where you have to make sure that a number is between two other numbers. For instance, you might have to do some math with a number x where x is only allowed to be between 0 and 100 ($0 \leq X \leq 100$).

Write a program which asks the user to enter a number. The program then checks if the number is between -10 and 10. If the number is less than -10, the program changes it to -10. If the number is more than 10, the program changes it to 10. If it is between -10 and 10, the program does not change it. Your program should look like this when it runs:

```
Enter a number: -234234
Changed to: -10
Enter a number: 79789
Changed to: 10
Enter a number: 5
Changed to: 5
```

Your program should use a function like this to change the numbers the user enters:

```
void limit(int* x) {
  // Your code here
}
```

Homework #4 - Arrays and Vectors

Name:

1 Intro

In C++, there's no obvious replacement for the Python list. Probably the simplest thing C++ has that is like a list is the **array**. That said, there are two big differences between Python lists and C++ arrays:

- You must know the size of a C++ array before you use it
- Everything in the array must be the same type (no mixing ints and strings)

Here is some simple code to create an array of 10 integers in C++:

```
1
#include <iostream>
3
using namespace std;
5
int main(){
7
    int myArray[10]; // Make an array that can hold 10 numbers
9   int i;

11  // Fill array
    cout << "Enter 10 numbers: ";
13  for (i = 0; i < 10; i++) {
      cin >> myArray[i];
15  }

17  // Print out array
    for (i = 0; i < 10; i++) {
19    cout << myArray[i] << " ";
    }
21  cout << endl;

23  return 0;
}
```

fixedArray.cpp

We can get around the need to know in advance how big the array is by using C++'s **new** and **delete** operators. These let us ask the computer for more memory while the computer is running. See the example on the next page.

```cpp

#include <iostream>

using namespace std;

int main(){
    int arraySize, i;

    cout << "How big do you want your array?: ";
    cin >> arraySize; // Ask for array size

    // Allocate a new array
    int* array = new (nothrow) int[arraySize];

    // Read in elements
    for (i = 0; i < arraySize; i++){
        cin >> array[i];
    }

    // Print array out backwards
    for (i = 0; i < arraySize; i++){
        cout << array[i] << " ";
    }
    cout << endl;

    // Free memory and return
    delete[] array;
    return 0;
}
```

dynamicArray.cpp

Here, we ask the user how large the array needs to be, and then **allocate** space in memory to hold it.

2 Vectors

Using "new" and "delete" creates a lot of extra work for us! In the example above we had to create an integer pointer (int *) and then use new to ask the computer for memory. What happens if later on we need the array to grow again? Or we want to make a copy of it into a new place in computer memory? These situations will make our code ugly and difficult to read. Luckily someone has already thought of this and has written the "vector" library! This library takes care of asking for memory for us, and makes our lives much simpler. A simple program to read N numbers into a vector and print them back out would look like this (see next page):

```cpp
// This code demonstrates basic vector usage in C++

#include <iostream>
#include <vector>

using namespace std;

int main(){
    int arraySize, i, tmp;
    vector<int> array;

    cout << "How big is your array?: ";
    cin >> arraySize;

    for (i = 0; i < arraySize; i++){
        cin >> tmp;
        array.push_back(tmp); // Vectors grow dynamically
    }

    for (i = 0; i < array.size(); i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    return 0;

}
```

vectorUsage.cpp

See how much easier that is? No worrying about the final size of the vector, and no worrying about asking for more memory: the vector handles it all!

3 Now you try

3.1 Arrays Challenge

Use your new array knowledge to solve the arrays problem on HackerRank

3.2 Vector Challenge

Write a program which lets the user enter as many numbers as they want. When the user enters 9999, the program should stop and print out all the numbers that the user has entered so far. Like this:

```
Enter a number: 5
Enter a number: 25
Enter a number: 36
Enter a number: -4
Enter a number: 22
Enter a number: 9999
Your vector contains:
5 25 36 -4 22
```

Homework #5 - Sorting

Name:

1 Intro

You've now learned enough C++ that we can start talking about the idea of an **algorithm**. An algorithms is the way you do something. For instance, an algorithm for washing dishes might looks like:

1. Turn on sink
2. Put soap on sponge
3. Pick up dish (bowl, plate, fork, spoon, knife, or cup)
4. Wash with sponge
5. Dry
6. Go back to step 3
7. Stop when there are no more dishes

Actually, an algorithm is not too different from a program. It's a list of instructions that tell you **how** to do something. The difference is that usually an algorithm only talks about how to do one very specific job, usually something mathematical like searching or sorting. Programs often have a lot of **other** work to do like opening and closing files, waiting for users to click on things, etc... and algorithms do not deal with that.

2 Sorting

We talked a little bit about sorting today in class. There are many different ways to sort, and **how** you sort can affect how quickly you can sort. For homework, we'll try to write two simple sorting algorithms, Bubble Sort and Insertion Sort.

For most problems, Bubble Sort is very slow, but there are some times when it is the right choice. For instance, if most of your lists look like this:

```
1 2 3 4 5 6 8 7 9 10 11 12
```

Notice that only 8 and 7 are not in the right order in this list. For lists that are **almost** in the right order already with some things only one or two places away from the correct position, then Bubble Sort is a good choice. We will talk more about this later in the class. Not only does the **algorithm** you use determine how fast you can solve a problem, but sometimes the problem can help you decide which algorithm is the right one to use. Not every kind of data is the same, so there is no **best** algorithm for all situations.

Remember, in algorithms there is **no substitute for understanding the problem you are trying to solve**.

2.1 Bubble Sort

Bubble Sort is perhaps the simplest sorting algorithm. Bubble sort for a list of numbers works like this:

1. Look at the 1st and 2nd numbers in the list
2. If the numbers are not in the right order, flip them
3. Look at the 2nd and 3rd numbers in the list
4. If the numbers are not in the right order, flip them
5. Continue to the end of the list, flipping any out of order number pairs
6. Start again from the beginning of the list
7. If you go through the whole list without flipping any numbers, stop

You can use the code in emptySort.cpp to get started. You just need to write the sorting code in the middle. The code to create and print the array(s) is already written for you in that file.

2.2 Insertion Sort

Below is the pseudo-code for insertion sort. Your job is to turn this into a real C++ program which you can show me in the next class.

```
// Insertion sort pseudo-code
for j = 1 to A.length-1
    key = A[j]
    i = j-1;
    while i >= 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key;
```

insertionSortPseudo.cpp

Think about how you could convert this into a real C++ program. How does it work? What happens when this code finds two numbers that are already in order? What about two numbers that are **not** in order? You can start with the C++ code on the next page (emptySort.cpp)

```cpp
#include <iostream>

using namespace std;

int main(){

    int simpleArray[] = {6, 5, 4, 3, 2, 1};

    // Print original array
    cout << "Original array: " << endl;
    for (int i = 0; i < 6; i++){
        cout << simpleArray[i] << " ";
    }
    cout << endl;

    // YOUR CODE HERE

    // Print sorted array
    cout << "Sorted array: " << endl;
    for (int i = 0; i < 6; i++){
        cout << simpleArray[i] << " ";
    }
    cout << endl;

    return 0;
}
```

emptySort.cpp

Your new code should go in the middle, between the two for loops.

2.3 Ok...what about big to small?

Write a C++ program that sorts in reverse. Big numbers first, small numbers last. You can start with the code you wrote for the last question.

2.4 A final challenge

How would you sort words from A-Z? Think about an algorithm to do this. You don't need to write real code, but please write the **pseudocode** for your algorithm here (or on the back of the page if you run out of room):

1.
2.
3.
4.
5.
6.
7.

Homework #6 - Preprocessor Directives

Name:

1 Intro

C++ programs are **compiled**. This means that they are run through another computer program (called a compiler) which turns them into **machine code**, which is just a series of instructions for the computer, encoded as patterns of "1" and "0". During that process, our C++ files get read by the compiler, sometimes more than once. The first step in creating our program is done by a part of the compiler called the **preprocessor**. The job of the preprocessor is to make changes to our code before it is turned into machine code. You have gotten used to starting your programs with:

```
1 #include <iostream>
```

But why do we do this? In C++, lines that start with "#" are special instructions for the preprocessor. The "#include <iostream>" tells the preprocessor to go find a C++ file called "iostream" and copy it into our file. We do this so that we can use "cin" and "cout". In fact, any time we want to include functions or tools from another C++ file we will use "#include". It is also possible to tell the preprocessor to replace certain words in our code with numbers, like this:

```
1 #include <iostream>
  #define INF 1000
3
  using namespace std;
5
  int main() {
7   int i = INF;
    cout << i << endl;
9   return 0;
  }
```

When we build and run this program, the INF will be replaced by the number 1000. We can also use this behavior to create something called a "macro". This lets us replace parts of our code with something that looks a bit like a function. For instance, like this (see next page):

```
#include <iostream>
#define min(a, b) a = a < b ? a : b

using namespace std;

int main() {
  int a, b;
  cin >> a;
  cin >> b;
  min(a,b);
  cout << a;
  return 0;
}
```

Here, the min(a,b) code will be **replaced** by the code from the #define, and will change "a" to be equal to the smaller number (either a or b).

A word of warning: The preprocessor is a powerful tool but it is very easy to use it wrong. Remember, the preprocessor is **not** smart. It simply looks for text that matches #define and then replaces it with whatever was in the #define. This can result in very unclear and strange behavior in your programs, and it can make them hard to debug, because the compiler will warn you about errors in your code which happened **after** the preprocessor runs...so the line numbers of the error messages may not match the line numbers in your file editor. Similarly it is easy to create a #define like this one which is **very** confusing for other programmers:

```
#include <iostream>
#define return return;}

using namespace std;

int main() {
  cout << "Hello world!" << endl;
  return
```

And this code **will** compile and run! Because "return" gets replaced with "return;}" which fills in the missing semicolon (;) and brace (}) for us. But it makes the program very confusing to read. So a word of warning: **if you must use #define, don't use it often, and only use it for very simple things**.

2 Now you try...

2.1 Making range(0,10) work in C++

Add one or more #define statements to this code so that it will run correctly. Do not change any of the code itself, just add #define statements.

```
#include <iostream>
using namespace std;
int main() {
  for range(0,10) {
    cout << i << endl;
  }
  return 0;
}
```

2.2 HackerRank Challenge

Solve the HackerRank C++ preprocessor challenge. You may find the hints below useful.

2.2.1 Hint: Deleting code with #define

You may need to add the following #define to ignore the "FUNCTION(a,b)" lines and instead write separate #define lines for the minimum() and maximum() functions:

```
#define FUNCTION(name,op) /* */
```

2.2.2 Hint: Doing if/else comparisons

You can do a really quick if/else statement in C++ like this:

```
(a == b) ? a:b
```

If a and b are equal, this code returns the number a. If a and b are not equal, this code returns the number b. Think about how you could use this with "<" and ">" to make **minimum** and **maximum** #define statements.

2.2.3 Hint: Adding quotes

It might also be useful for you to know that a #define like this...

```
#define makeString(thing) #thing
```

Will put quotes "" around whatever words you put into its parentheses (). You can use this to write a **toStr()** #define.

Homework #7 - Structs and Classes

Name:

1 Structs

One of the **big ideas** in C++ is something called OOP or **object oriented programming**. There are lots of real world programming problems that are best solved by putting a lot of data together into a new **type** that you make yourself. Grouping related data together can make your programs easier to design and easier to read and modify. In C++, you can group data together with a **struct**. A struct looks like this:

```
struct Student {
  string name;
  int age;
  float score;
};
```

structs.cpp

This struct would make it very easy to keep track of a list of students in an array, because now a single array can contain all this data together in one place (no need for a "student names" array and a "student ages" array, etc...). Creating an array to hold 3 students is now easy:

```
    // Make a struct that can hold 3 students
    Student students[3];
```

structs.cpp

Let's say you wanted to change the name of the first student in the array, students[0]. You'd just write:

```
students[0].name = "Frank";
```

You could of course write a loop to fill in all the elements of the array, as well:

```
    for (int i = 0; i < 3; i++) {
        cout << "Enter name, age, and score..." << endl;
        cout << "Name: ";
        cin >> students[i].name;
        cout << "Age: ";
        cin >> students[i].age;
        cout << "Score: ";
        cin >> students[i].score;
        cout << endl;
    }
```

structs.cpp

2 Classes

Letting us group data into new types like this is a great feature, but C++ goes farther than that. It also allows us to put **functions** together with our data. Now, our data **and** the functions we need to work with it can all be in one place. We use classes to do this. Here's an example of the Student struct rewritten as a class:

```
class Student {
    // Data
    private:
      string name;
      int age;
      float score;

    // Methods (functions inside the class)
    public:
      void setData(string selfName, int selfAge, float selfScore) {
          name = selfName;
          age = selfAge;
          score = selfScore;
      }

      void printData() {
          cout << "Name: " << name << endl;
          cout << "Age: " << age << endl;
          cout << "Score: " << score << endl;
      }
};
```

classes.cpp

Notice that the work of printing out everything inside the class can now be hidden inside the class. If I want to print out all the details for students[1] I would just say:

```
students[1].printData();
```

To do this with a struct, I'd have to write:

```
cout << "Name: " << students[1].name << endl;
cout << "Age: " << students[1].age << endl;
cout << "Score: " << students[1].score << endl;
```

Having functions saved inside with the data makes classes much more powerful than structs. There are lots of situations - in fact - where classes are very helpful because they can hide the details of their insides behind "set()" and "get()" functions, so programmers don't need to know what's inside a class in order to use it. These "get()" and "set()" functions can also make sure that anybody using the class is changing the class's data in the correct way.

3 Now you try...

3.1 Using Structs

Write a struct for use in a Zoo. You should make a struct that can store:

- An animal's name (ex: Billy)
- What it eats (ex: Bamboo)
- What type it is (ex: Panda)
- It's age (ex: 12)
- How many legs it has (ex: 4)

Once you have written this struct, you should make an array big enough to hold 3 animals, and make a loop to let the user enter the information for these 3 animals (name, age, legs, etc...). You can look at **structs.cpp** for an example.

3.2 Using Classes

Change your Zoo struct into a Zoo class. You should add functions to **set** the animal's name/-food/type/age/legs and another type to **get** these things and print them out. Start with the code in **classes.cpp**.

3.3 Final Challenge

Make a class called "Array" that can hold an array. The class should have functions to change what's stored in the array and print out the array. Something like this would be a good start:

```
class Array {
  // Data
  private:
    vector<int> array;

  // Methods (functions inside the class)
  public:
    void printArray() {
      // YOUR CODE HERE
    }

    void changeItem(int index, int newValue) {
      // YOUR CODE HERE
    }

};
```

Homework #8 - Operator Overloading

Name:

1 Changing the Meaning of Operators

We talked last class about the general idea of a **class**, which is a collection of data and functions stored together in the computer's memory, inside a single variable. We talked about how - in general - we write "set" and "get" functions to access data inside our classes. Well, what do we do if we want to use built-in operations like "+", "-", "*", or "/" with our classes? What if we want to be able to print our classes out nicely using "<<"? This can be done with something called **operator overloading**. Basically, we can teach the computer a **new meaning** for operations like "+" or "<<", and this can make our classes even easier to work with. Here is an example of a Point class for storing 2D points (x, y):

```
class Point {
    // Data
    private:
      int x,y;

    // Methods (functions inside the class)
    public:
        // Empty constructor
        Point();

        // Constructor
        Point(int xp, int yp);

        // "+" operator overload
        Point operator+(const Point& otherPoint);

        friend ostream& operator<<(ostream& os, const Point& pt);

        void setPoint(int xp, int yp) {
            x = xp;
            y = yp;
        }

        int getX() {
            return x;
        }

        int getY() {
            return y;
        }
};
```

operatorOverloading.cpp

It looks a lot like the classes we used before. It still has "set" and "get" functions, but there's some other new stuff as well. Let's talk about that.

1.1 Constructors

You'll notice that there are two functions in the class that look like this:

```
// Empty constructor
Point();

// Constructor
Point(int xp, int yp);
```

These are called **constructors**. Their job is to set up everything inside the class each time we make a new copy of it (a new copy of a class is called an **object**). When we first make an **object** we sometimes do it like this:

```
Point p;
```

And sometimes we want to do it in a way that sets up all the data inside the object, like this:

```
Point p = Point(2,5);
```

This sets up a new Point with $x = 2$ and $y = 5$. This is why there are **two** constructors: the "empty" one handles cases where we don't need to set up the data inside our object, or when we aren't making a new object yet, and the non-empty one handles cases where we are making a new object and need to set it up.

1.2 Overloading Functions

So why are we allowed to have two different functions with the same name? We have two **constructors**, after all:

```
// Empty constructor
Point();

// Constructor
Point(int xp, int yp);
```

They are both called "Point", so how does the computer know which one we want to use? In C++, you are allowed to have more than one function with the same name, as long as the functions have different numbers of **arguments** (the things inside the parentheses "()"). Here, we have one constructor that takes two arguments, and one that takes none. This idea is called **overloading**. Basically, when the computer is making our program into machine code, it can tell which function is the right one to use by checking to see how many **arguments** and what **type** of arguments we have put inside the "()".

1.3 Prototypes vs Definitions

Well hold on, this just keeps getting stranger! These two constructors do not contain any code! There's no "{}". They simply end with ";" like this:

```
// Constructor
Point(int xp, int yp);
```

So how does this function work? It doesn't actually do anything! In C++, you can actually split functions into two parts. One part is called a **prototype** and one part is called a **definition**. The **prototype** just tells us what kind of **arguments** go into the function. Later on in our code, we have to also write a **definition** which says says how the function will actually work. The strange thing about **constructors** is that their **definitions** go outside the class. So later on in our code, we have:

```
// Empty constructor
Point::Point() {

}

// Tell the computer how to make a new point
Point::Point(int xp, int yp) {
    x = xp;
    y = yp;
}
```

operatorOverloading.cpp

These actually tell the computer what the two **constructors** should do. The "Point::" part in front just tells the computer that these two functions belong to the class "Point".

1.4 Overloading Operators

We need the idea of a **prototype** for operator overloading. If you look back at the first page, you'll see that there's a **prototype** for the "+" operator inside our Point class...but there's no code saying how it works. That's farther down in our program code and looks like this:

```
// Tell the computer what "+" means for the Point class
Point Point::operator+(const Point& otherPoint) {
    Point newPoint(x+otherPoint.x, y+otherPoint.y);
    return newPoint;
}
```

operatorOverloading.cpp

The "+" operator has to be **defined** outside of the Point class so that we can use it without also saying the name of the class (like this code to make a new point):

```
Point x = Point.Point()
```

Putting the **definition** outside the class keeps us from having to say "Point.Point()" every time we want to use the constructor, or "Point.+(otherPoint)" every time we want to add points together.

2 Your Turn

2.1 Overloading Math

Change your Array class so that it overloads "+", "-", "*", and "/" for arrays (so that you can multiply all the numbers in two arrays together just by doing "A * B", for instance). You only need to make this work for arrays that are the same length.

2.2 Overloading output operators

Look at the example code for the Point class and notice that "<<" is overloaded too. Try to overload this in your Array class as well, so that you can do things like this:

```
1 cout << A << endl;
```

Homework #9 - Recursion

Name:

1 Recursion

You've probably seen code that looks like this before:

```
for (int i = 10; i > 0; i--) {
  cout << i << endl;
}
```

It's just a regular "for" loop that counts down from 10 to 1. Is there any other way to count from 10 to 1? Without a loop? And without just writing "cout" ten times? Turns out there is:

```
void countDown(int start) {
    if (start > 1) {
        cout << start << endl;
        countDown(start - 1);
    }
    else {
        cout << 1 << endl;
    }
}

int main() {

    countDown(10);

    return 0;
}
```

simpleRecursion.cpp

If you take a close look at the function at the top, you'll notice that **it calls itself**! Functions are completely allowed to do this. You can write a C++ function that uses itself. This can be a difficult concept to understand at first, but it's very powerful and useful once you understand it.

1.1 The Termination Condition

Usually, recursive functions have an if/else inside that checks the value of an **argument** passed to the function. If the argument meets a certain condition, the recursion can continue (the function calls itself again), but with the argument changed (in this case, reduced by 1). If the argument does **not** meet the condition, the function does a "return". When this happens, we say the recursion has **terminated**. This means the function stops calling itself, and the computer goes "back up the call stack". That means that all the functions that were called finish their work and go back into our program's main() function.

1.2 Factorial

Though it's hard to believe when you first see it, some algorithms are easiest to write if you write them recursively. A great example is **factorial**. You're probably familiar with factorial from math class. You write "factorial five" like this:

$$5!$$

And it is solved like this:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Imagine a recursive function to solve this. You could have a function that calls itself on 5, then 4, then 3....down to 1 then stops and **terminates the recursion**. Something like this:

```
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n-1);
    }
    else {
        return 1;
    }
}
```

factorial.cpp

When the value "n" gets to 1, the function stops calling itself and just returns "1". All the functions above it would have done "n * factorial(n-1)", so the computer ended up doing something like this:

```
factorial(5) * factorial(4) * factorial(3) * factorial(2) * factorial(1)
```

Doing factorial(1) just returns a 1 into the function factorial(2), which does the math "2 * 1" and returns a "2" to factorial(3), which does "3 * 2" which returns the value "6" to factorial(4) which does "4 * 6" and returns "24" to factorial(5) which does "5 * 24" and then returns the final answer "120". So the recursion **collapses**, kind of like a folding telescope. To illustrate this for you, imagine a program that did "cout << num;" and then called itself on "num + 1", stopping when "num == 4". If we started it at "0" we'd get something like this when it ran:

```
1  recursiveFunction(0)
2  printf(0)
3          recursiveFunction(0+1)
4          printf(1)
5                  recursiveFunction(1+1)
6                  printf(2)
7                          recursiveFunction(2+1)
8                          printf(3)
9                                  recursiveFunction(3+1)
10                                 printf(4)
```

Figure 1: A Simple Recursion

2 Now You Try

There is a famous **series** (a set of numbers counting up or down) in mathematics called the **Fibonacci series**. It goes like this:

```
1 1 2 3 5 8 13 21 34 55 89 144...
```

Can you see a pattern here? $0+1=1$, the next number in the series. Then $1+1=2$, the next number. $1+2=3$, $2+3=5$, $3+5=8$, and so on. Try to write a **recursive** C++ program that print out this series. You should have a function in your code that looks like this:

```
void fibonacci(int a, int b, int n){
  // YOUR CODE HERE
}
```

If you wanted the first 12 numbers, you'd set "a" and "b" to be "0" and "1" and you'd set "n" to "12". Like this:

```
int main() {

    cout << "The first 12 Fibonacci numbers: " << endl;
    fibonacci(0, 1, 12);

    return 0;
}
```

Which would print out:

```
1 1 2 3 5 8 13 21 34 55 89 144
```

Think about how the 3 arguments to fibonacci should **change** each time the function is called. Think about what number the fibonacci function should print out each time it is called. Think about what should happen when "n == 0". Have a look at the files **factorial.cpp** and **simpleRecursion.cpp** for inspiration.