CS 590: Algorithm

Week 2: C++ Review

In Suk Jang
Department of Computer Science
Stevens Institute of Technology

Concerns you may have…

- Do we really need to work with C++?
- Yes

- How about working with any languages you prefer to work with before you work with C++?
- For the first assignment, you may but from HW#2, **you must use C++.**

- When do we focus more on algorithm?
- Starting from week3, we will be more focusing on algorithms itself then the code or C++.

- Contents of report?
- You must write a report for each assignments when the testing algorithms are asked.
- Abstract
- Summary of results – tables and plots
- Analysis/evaluation of test
- Conclusion

Introduction to C++

- Introduction
- Evolution
- General Structure
- A Simple C++ Program

Introduction to C++

- The C++ programming language is a very powerful general – purpose programming language that *supports procedural programming as well as object* – oriented programming. It incorporates all the ingredients required for building software for large and complex problems.

- The C++ language is treated as *super set of C language* because the developer of C++ language have retained all features of C, enhanced some of the existing features, and incorporated new features to support for object – oriented programming.

- The importance of C++ can well be judged from the following statement:
- "Object – Oriented Technology is regarded as the ultimate paradigm for the modeling of information, be that information data or logic. The C++ has by now shown to fulfill this goal."

Evolution of C++ Language

- The C++ programming language was developed by **Bjarne Stroustrup** at *AT&T BELL LABORATORIES*, NEW JERSEY, USA, in the early 1980's.

- He found that as the problem size and complexity grows, it becomes extremely difficult to manage it using most of procedural languages, even with C language.

GENERAL STRUCTURE OF A C++ PROGRAM

Declarations: data types, function signatures, classes –

- Allows the compiler to check for type safety, correct syntax
- Usually kept in "header" (.h) files
- Included as needed by other files (to keep compiler happy)

```
class Simple {
public:
   Simple (int i);
   void print_i ();
private:
   int i_;
};
```

```
typedef unsigned int UINT32;
int usage (char * program_name);

struct Point2D {
  double x_;
  double y_;
};
```

Definitions: static variable initialization, function implementation

- The part that turns into an executable program
- Usually kept in "source" (.cpp) files

```
void Simple::print_i ()
{
 cout<< "i_is "<<i_<<endl;
}
```

Directives: tell compiler (or precompiler) to do something

GENERAL STRUCTURE OF A C++ PROGRAM

Section 1: Comments

- It contains the description about the program.

Section 2: Preprocessor Directives

- The frequently used **preprocessor** directives are include and define. These directives tell the preprocessor how to prepare the program for compilation. The include directive tells which header files are to be included in the program and the define directive is usually used to associate an identifier with a literal that is to be used at many places in the program.

Section 3: Global Declarations

- These declarations usually include the declaration of the data items which are to be shared between many functions in the program. It also include the decorations of functions.

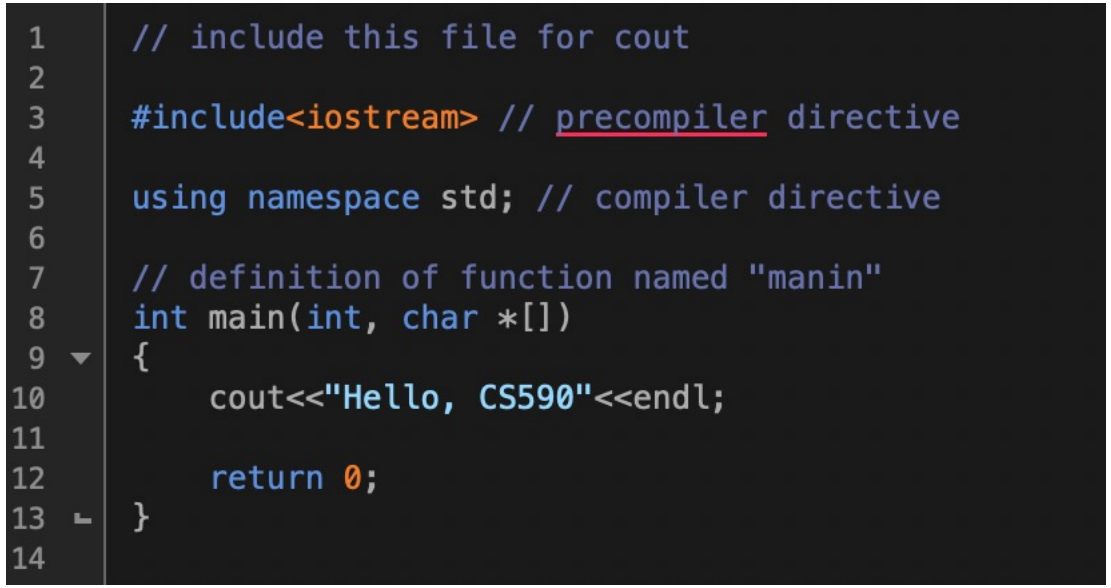
Section 4: Main function

- The execution of the program always begins with the execution of the main function. The main function can call any number of other functions, and those called function can further call other functions.

Section 5: Other functions as required

```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

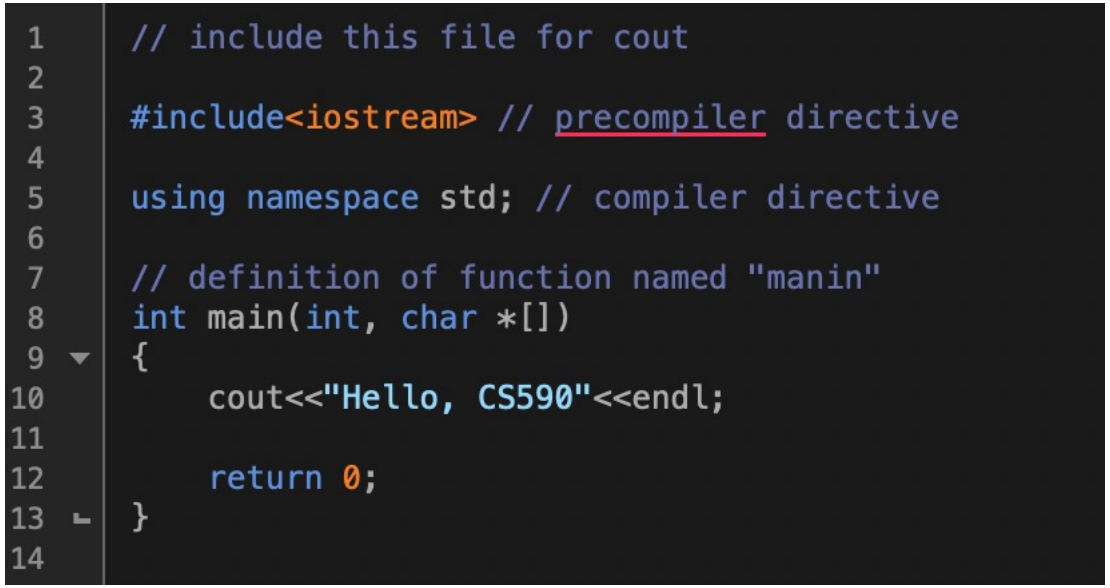
A Simple C++ Program



```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

What is #include<iostream> ?

- #include tells the pre-compiler to include a file
- Usually, we include header files
  - Contain *declarations* of structs, classes, functions
- Sometimes we include template *definitions*
  - Varies from compiler to compiler
  - Advanced topic, out of scope for this course, but feel free to look it up!
- <iostream> is the C++ label for a standard header file for input and output streams



```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

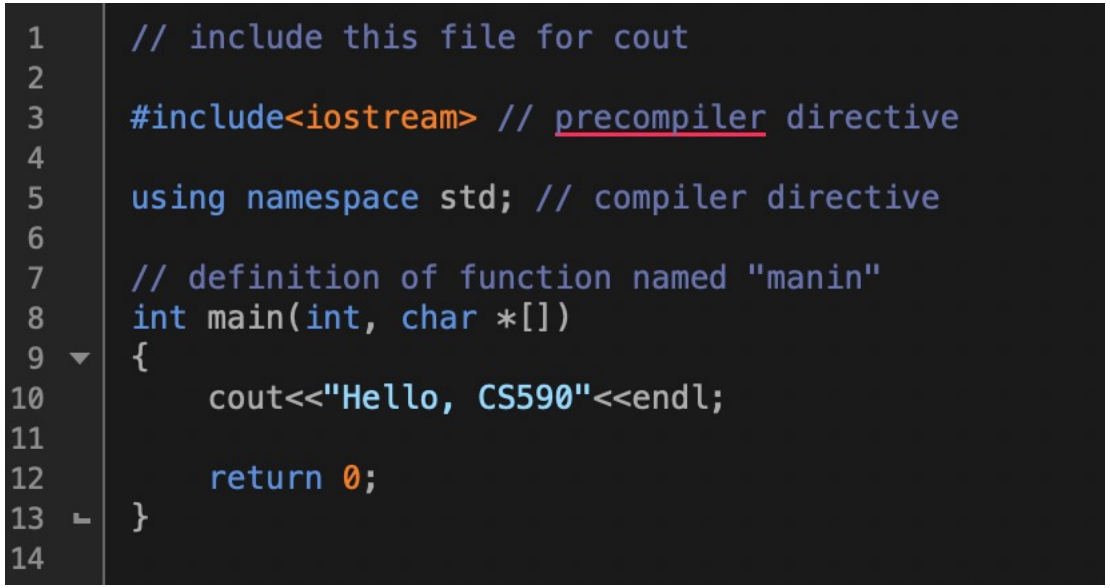
What is using namespace std; ?

- The **using** directive tells the compiler to include code from libraries that have separate *namespaces*
  - Similar idea to "packages" in other languages
- C++ provides a namespace for its standard library
  - Called the "standard name space" (written as std)
  - cout, cin, and cerr standard iostreams, and much more
- Namespaces reduce collisions between symbols
  - Rely on the **::**scoping operator to match symbols to them
  - If another library with namespace mylib defined **cout** we could say **std::cout vs. mylib::cout**
- Can also apply **using** more selectively:
  - *E.g.,*just **usingstd::cout**

```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

What is int main (int, char*[]) { ... } ?

- *Defines* the main function of any C++ program
- Who calls main?
  - The runtime environment, specifically a function often called something like **crt0** or **crtexe**
- What about the stuff in parentheses?
  - A list of types of the input arguments to function main
  - With the function name, makes up its *signature*
  - Since this version of main ignores any inputs, we leave off names of the input variables, and only give their types
- What about the stuff in braces?
  – It's the *body* of function main, its *definition*



```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

What's cout << "Hello, CS590" << endl; ?

- Uses the standard output iostream, named **cout**
  - For standard input, use **cin**
  - For standard error, use **cerr**
- << is an operator for *inserting* into the stream
  - A member **operator** of the ostream class
  - Returns a *reference* to stream on which its called
  - Can be applied repeatedly to references left-to-right
- **"hello, cs590"** is a C-style string
  - A14-position character *array* terminated by **'\0'**
- **endl** is an iostream manipulator
  - Ends the line, by inserting end-of-line character(s)
  - Also *flushes* the stream

```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

What about return 0;?

- The main function should return an integer
  - By convention it should return 0 for success
  - And a non-zero value to indicate failure
- The program should not exit any other way
  - Letting an exception propagate uncaught
  - Dividing by zero
  - Dereferencing a null pointer
  - Accessing memory not owned by the program
    - Indexing an array “out of range” can do this
    - Dereferencing a “stray” pointer can do this

Outline

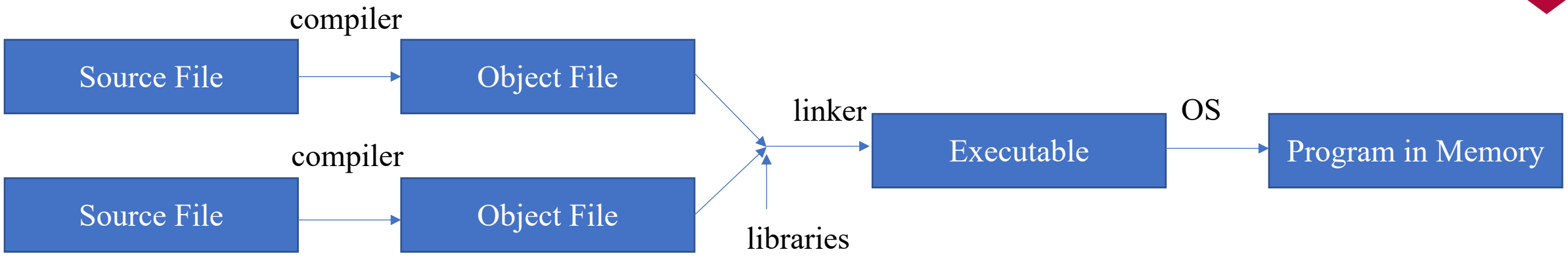
C++ Review:

- Token
- Variables
- Arrays
- Strings
- Pointers
- Functions
- Class

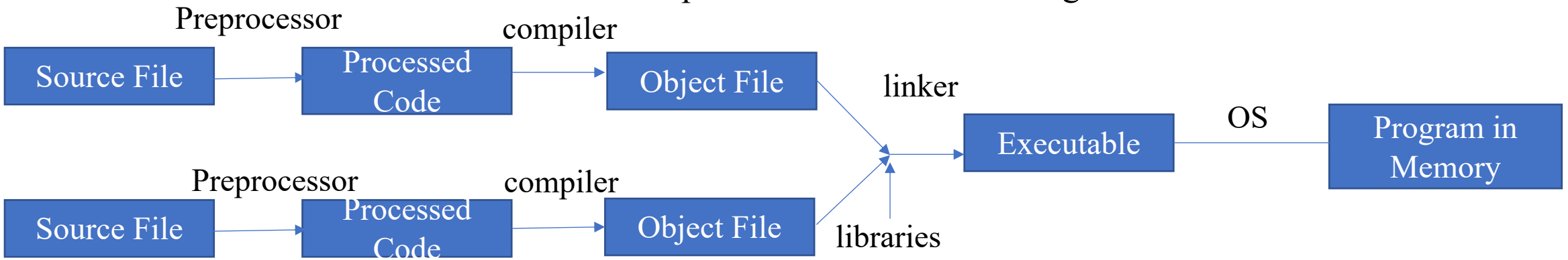
Mathematics:

- Exponents
- Logarithms

Source files to processor diagram



Source files to processor modification diagram



Tokens

```
1   // include this file for cout
2
3   #include<iostream> // precompiler directive
4
5   using namespace std; // compiler directive
6
7   // definition of function named "manin"
8   int main(int, char *[])
9   {
10      cout<<"Hello, CS590"<<endl;
11
12      return 0;
13  }
14
```

- **Tokens** are the minimal chunk of program that have meaning to the compiler – the smallest meaningful symbols in the language.
- This code displays all 6 kinds of tokens

Token Types	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	int
Identifiers	Name of things that are not built into the language	cout, std
Literals	Basic constant values whose value is specified directly in the source code	“Hello, CS590”
Operators	Mathematical or logical operations	<<
Punctuation/Separators	Punctuation defining the structure of a program	{ }, ;
Whitespace	Spaces of various sorts; ignored by the compiler	// include this file for cout

Basic Language Features

Values and Statements

- A **statement** is a unit of code that does something – a basic building block of a program.
- An **expression** is a statement that has a value – for instance, a number, a string, the sum of two numbers, etc.
- **Not** every statement is an expression. E.g., #include statement.

Operators

- Operators act on expressions to form a new expression.
  - Mathematical: +, -, *, /, %
  - Logical: and, or, etc.
  - Bitwise: manipulates the binary representation of numbers, e.g., |, ^, <<, etc.

Basic Language Features

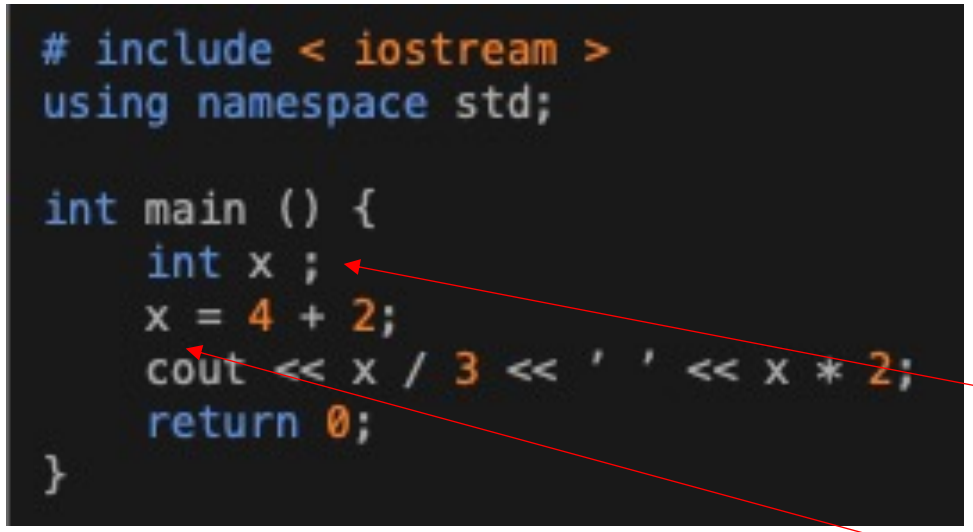
Data Types

- Every expression has a type – integer, floating-point, string
- Data of different types take a different amounts of memory to store.
- An operation can be performed on compatible types and normally produces a value of the same type as its operands.

Type Names	Description	Size (byte)	Range
char	Single text character or small integer. Indicated with single quotes ('a', '3').	1	signed: -128 to 127 unsigned: 0 to 255
int	Larger integer	4	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean (true/false). Indicated with the keywords true and false.	1	Just true (1) or false (0).
double	"Doubly" precise floating point number.	8	+/- 1.7e +/- 308 (15 digits)

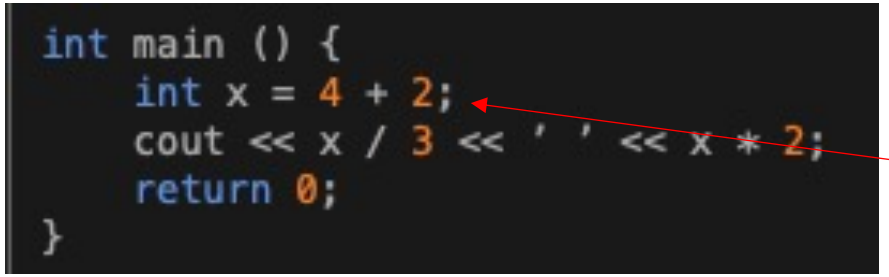
- A *signed* integer is one that can represent a negative number; an *unsigned* integer will never be interpreted as negative.
- There are actually 3 integer types: short, int, and long, in non-decreasing order of size.
  - memory usage or huge numbers.
- The sizes/ranges for each type are not fully standardized; those shown above are the ones used on most 32-bit computers.

Variable



```
# include < iostream >
using namespace std;

int main () {
    int x ;
    x = 4 + 2;
    cout << x / 3 << ' ' << x * 2;
    return 0;
}
```



```
int main () {
    int x = 4 + 2;
    cout << x / 3 << ' ' << x * 2;
    return 0;
}
```

- Use ***variables*** to give a value a name so we can refer to it later.
- The name of a variable is an *identifier token*. Identifiers may contain numbers, letters, and underscores (_), and *may not start with a number*.
- The **declaration** of the variable x – must tell the compiler what type x will be so that it knows how much memory to reserve for it and what kinds of operations may be performed on it.
- The **initialization** of x – specify an initial value for it. This introduces a new operator: =, the assignment operator.
- A single statement does both declaration and initialization.

Arrays

- An array is a fixed number of elements of the same type stored sequentially in memory.
  type arrayName[dimension];
- The elements of an array can be accessed by using an index into the array.
- Arrays in C++ are zero-indexed, so the first element has an index of 0.
- Like normal variables, the elements of an array must be initialized before they can be used.
- The array be multidimensional array.
  type arrayName[dimension1][dimension2];
- Dimensions must always be provided when initializing multidimensional arrays.
- Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory.

```
int arr[4];

arr[0] = 6;
arr[1] = 0;
arr[2] = 9;
arr[3] = 6;

int arr[4] = {6, 0, 9, 6};

int arr[] = {6, 0, 9, 6};
```

```
int twoDimArray[2][4] = { 6, 0, 9, 6, 2, 0, 1, 1 };
int twoDimArray[2][4] = { { 6, 0, 9, 6 } , { 2, 0, 1, 1 } };
```

Strings

- String literals such as “Hello, world!” are actually represented by C++ as a sequence of characters in memory. In other words, a string is simply a character array and can be manipulated as such.

```
char helloworld[] = { 'H', 'e', 'l', 'l', 'o', ',', ' ',
                      'w', 'o', 'r', 'l', 'd', '!', '\0' };
```

- The character array helloworld ends with a special character, ‘\0’, known as the null character.
- Character arrays can also be initialized using string literals.

```
char helloworld[] = "Hello, world!"
```

- The individual characters in a string can be manipulated either directly by the programmer or by using special functions provided by the C/C++ libraries. These can be included in a program through the use of the **#include** directive:
  - cctype (ctype.h): character handling
  - cstdio (stdio.h): input/output operations
  - cstdlib (stdlib.h): general utilities
  - cstring (string.h): string manipulation

Strings

```
#include <iostream>
#include <cctype>
using namespace std;

int main() {
    char messyString[] = "t6H0I9s6.iS.999a9.STRING";
    char current = messyString[0];
    for(int i = 0; current != '\0'; current = messyString[++i]) {
        if(isalpha(current))
            cout << (char)(isupper(current) ? tolower(current) : current);
        else if(ispunct(current))
            cout << ' ';
    }
    cout << endl;
    return 0;
}
```

- The *is-* functions check whether a given character is an **alphabetic character**, an **uppercase letter**, or a **punctuation character**, respectively.
- These functions return a *Boolean* value of either true or false.
- The *tolower* function converts a given character to lowercase.

- The for loop takes each successive character from messyString until it reaches the *null character.*

- What is the resulting output?

Strings

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char fragment1[] "I'm a s";
    char fragment2[] = "tring"
    char fragment3[20];
    char finalString[20] = "";

    strcpy(fragment3, fragment1);
    strcat(finalString, fragment3);
    strcat(finalString, fragment2);

    cout<<finalString;
    return 0;
}
```

- What do we need to fix here?
- Is **fragment3** declared or initialized?
- Is **finalString** initialized?

Knowing that *strcpy* and *strcat* functions copy and concatenate strings, respectively, can you guess what **finalString** will display?

Pointers

Motivations

When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the & and * operators:

1. &x evaluates to the address of x in memory.
2. *(&x) takes the address of x and *dereferences* it – it retrieves the value at that location in memory. *(&x) thus evaluates to the same thing as x.

Memory addresses, or **pointers**, allow us to manipulate data much more flexibly.

- Manipulating the memory addresses of data can be more efficient than manipulating the data itself.
  - More flexible pass-by-reference
  - Manipulate complex data structures efficiently, even if their data is scattered in different memory locations
  - Use polymorphism – calling functions on data without knowing exactly what kind of data it is

A pointer that stores the address of some variable x is said “to point to x”.

Pointers

Pointer Syntax/Usage

The pointer named ptr that points to an integer variable name x is declared as

```
int *ptr = &x;
```

Declares the pointer to an integer value, which we are initializing to the address of x.

Pointers can have any type value:

```
data_type *pointer_name;
```

We can dereference the pointer with the * operator to access its value:

```
cout << *ptr;
```

We can use deferenced points as l-values:

```
*ptr = 5;
```

Pointers

```
void squareByPtr ( int * numPtr ) {
    * numPtr = * numPtr * * numPtr ;
}

int main () {
    int x = 5;
    squareByPtr (&x);
    cout << x;
}
```

- When used as a function return type, the **void** keyword specifies that the function does not return a value.
- When used for a function's parameter list, **void** specifies that the function takes no parameters.
- When used in the declaration of a pointer, **void** specifies that the pointer is "universal".

- Can you guess what the output will be?

Pointers

Constant Pointers

Declares a changeable pointer to a constant integer.

Declares a constant pointer to changeable integer data.

How about this?

```
const int *ptr;

int *const ptr;

const int *const prt:
```

Null Pointers

Some pointers do not point to valid data.
Any point set to 0 is called a ***null pointer***.
Do you think this this pointer is valid?

```
int *myFunc() {
    int phantom = 4;
    return &phantom;
}
```

Pointers

References

```
int y;
int &x = y;
```

What would happen if x changes?
What would happen if y changes?
Why?

References are pointers that are dereferenced every time they are used.
Then what are differences between using pointers and references?

- References are sort of pre-dereferenced – you do not dereference them explicitly.
- You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, references must always be initialized when they are declared.
- When writing the value that you want to make a reference to, you do not put an & before it to take its address, whereas you do need to do this for pointers.

Pointers

The * operator is used in two different ways:

1. When declaring a pointer, * is placed before the variable name to indicate that the variable **being declared is a pointer** – say, a pointer to an int or char, not an int or char value.
2. When using a pointer that has been set to point to some value, * is placed before the pointer name to dereference it – to access or set the value it points to.

A similar distinction exists for &, which can be used either

1. to indicate a reference data type (as in int &x;), or
2. to take the address of a variable (as in int *ptr = &x;).

Pointers

Pointers and Arrays

The name of an array is actually a pointer to the first element in the array.

- myArray[3] tells the compiler to return the element that is 3 away from the starting element of myArray.
- passing an array is really passing a pointer.
- This is why array indices start at 0.
- Array-subscript notation (myArray[3]) can be alternatively expressed as *(myArray + 3).

char *String

- a string is actually an array of characters.
- You are setting a pointer to point to the first character in the array that holds the string.
- Can you modify one of elements below?

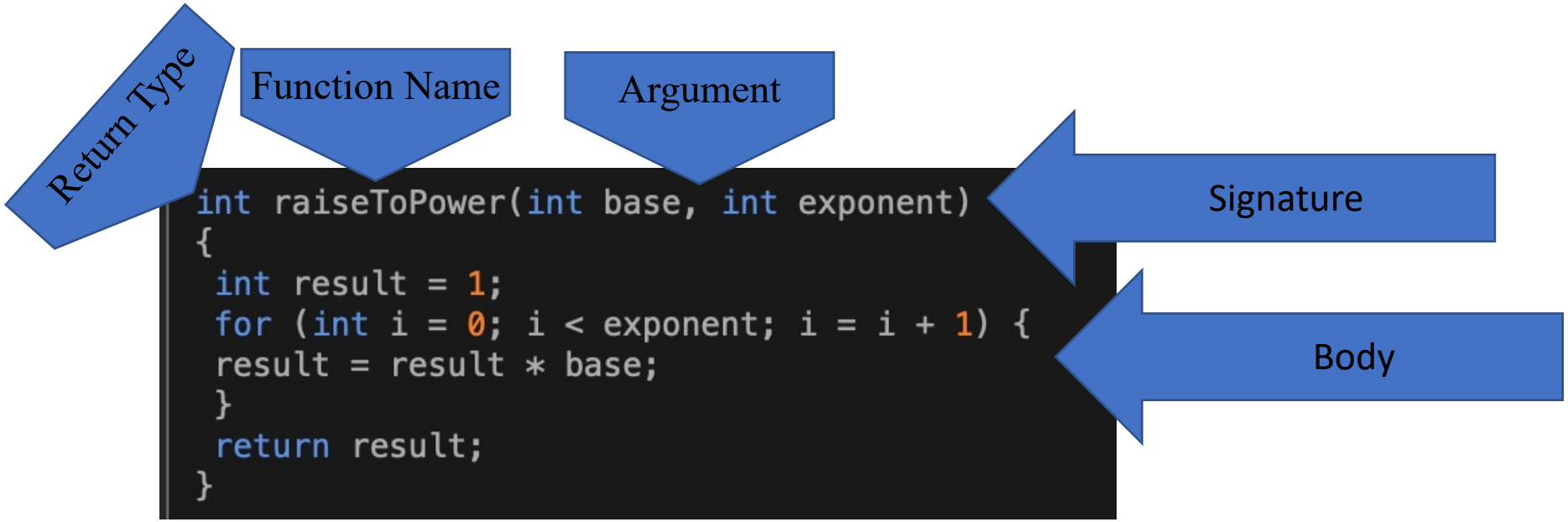
```
char stringName1[] = {'6','.','2'};
char *stringName2 = "6.2"
```

Functions

Advantages

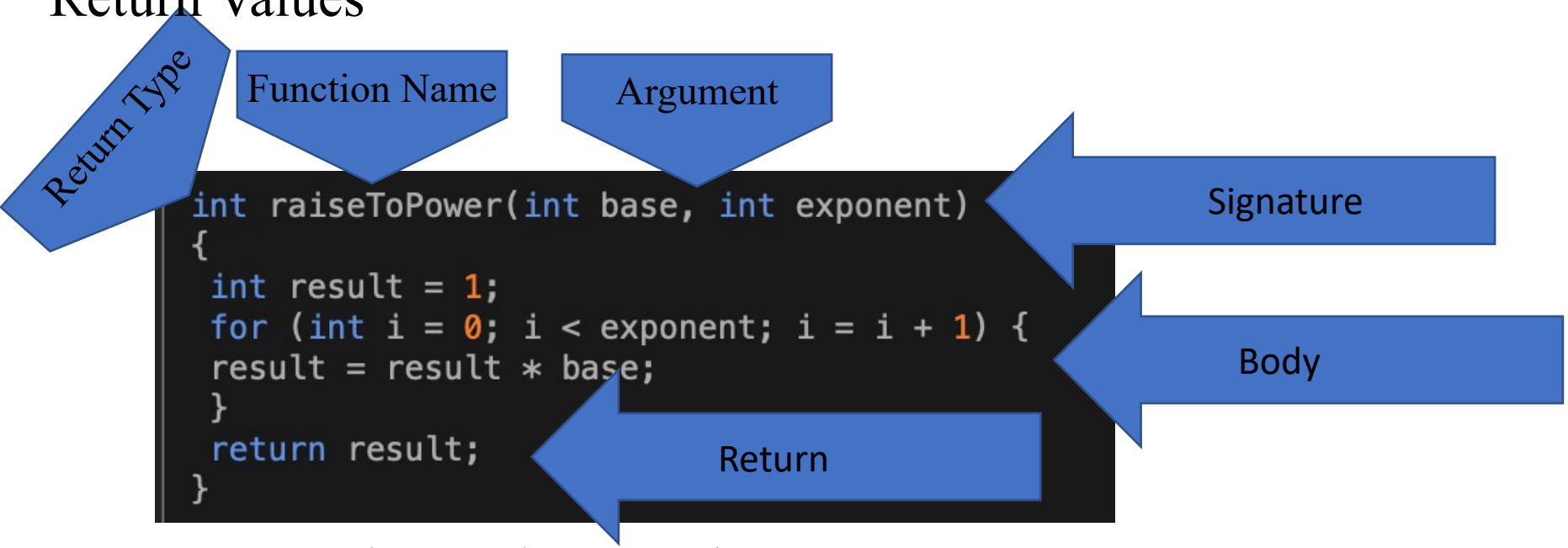
- Readability: sqrt(5) is clearer than copy-pasting in an algorithm to compute the square root, e.g., 5^(0.5).
- Maintainability: To change the algorithm, just change the function (vs changing it everywhere you ever used it).
- Code reuse: Lets other people use algorithms you've implemented.

Function Declaration Syntax



Functions

Return Values



- Up to one value may be returned; **it must be the same type as the return type**.
- If no values are returned, give the function a void return type
  - Note that you cannot declare a variable of type void
- Return statements don't necessarily need to be at the end.
- Function returns as soon as a return statement is executed.

```
void printNumberIfEven(int num) {
    if (num % 2 == 1) {
    cout << "odd number" << endl;
    return;
    }
    cout << "even number; number is " << num << endl;
}
```

Functions

Argument Type Matters

```
void printOnNewLine(int x) {
    cout << x << endl;
}

void printOnNewLine(char *x) {
    cout << x << endl;
}
```

- Many functions with the same name, **but different arguments**.
- The function called is the one whose arguments match the invocation.

Functions

```
int foo() {
    return bar()*2
}

int bar() {
    return 3;
}
```

```
int square(int z);
int cube(int x){
    return x*square(x);
}
int square(int x):{
    return x*x
}
```

- Function declarations need to occur **before invocations**.
  - Solution 1: reorder function declarations
  - Solution 2: use a function prototype; informs the compiler you'll implement it later
- Function prototypes should match the signature of the method, though argument names don't matter
- Function prototypes are generally put into separate header files
  - Separates specification of the function from its implementation

Functions

Recursion

```
int fibonacci(int n){
    if (n==0 || n == 1){
    return 1;
    } else {
        return fibonacci(n-2) + fibonacci(n-1)
    }
}
```

- Functions can call themselves.
- Fib(n) = fib(n-1)+fin(n-2) can be easily expressed via a recursive implementation.

```
int numCalls = 0;
void foo() {
    ++numCalls;
}
int main(){
    foo(); foo(); foo();
    cout<< numCalls << endl;
}
```

How many times is function foo() called?

- Use a global variable to determine this.
  - Can be accessed from any function

Functions

Scope

- the extent up to which something can be worked with.
  - Where a variable was declared, determines where it can be accessed from
  - numCalls has global scope – can be accessed from any function
  - result has function scope – each function can have its own separate variable named result

```
int raiseToPower(int base, int exponent){
 numCalls = numCalls + 1;
 int result = 1;
 for (int i = 0; i < exponent; i = i + 1) {
    result = result * base;
 }
 return result;
}

int max(int num1, int nume2){
    numCalls = numCalls +1;
    int result;
    if (num1 > num2) {
        result = num1;
    }
    else {
        result = num2;
    }
    return result;
}
```

Functions

Scope

```
double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
    }
 return estimate;
}
```

Cannot access variables that are out of scope

Functions

Scope

```
double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
        if (i == 29)
            return estimate;
    }
    return -1;
}
```

Solution 1: move the code

```
double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    double estimate;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
    }
    return estiamte;
}
```

Solution 2: declare the variable in a higher scope

Functions

Pass by value vs. by reference

```
void increment(int a) {
 a = a + 1;
 cout << "a in increment " << a << endl;
}

int main() {
 int q = 3;
 increment(q);
 cout << "q in main " << q << endl;
}
```

```
void increment(int &a) {
 a = a + 1;
 cout << "a in increment " << a << endl;
}

int main() {
 int q = 3;
 increment(q);
 cout << "q in main " << q << endl;
}
```

Functions

Returning multiple values

- Passing output variables by reference overcomes this limitation.

```
int divide(int numerator, int denominator, int &remainder) {
 remainder = numerator % denominator;
 return numerator / denominator;
}
int main() {
 int num = 14;
 int den = 4;
 int rem;
 int result = divide(num, den, rem);
 cout << result << "*" << den << "+" << rem << "=" << num << endl;
}
```

- Observe how some functions are closely associated with a particular class
- Methods: functions which are part of a class
  - Implicitly pass the current instance

Class

A user-defined datatype which groups together related pieces of information.

```
class Vector{
public;
    double xStart;
    double xEnd;
    double yStart;
    double yEnd;
};
```

This indicates that the new datatype we're defining is called Vector

Fields indicate what related pieces of information our datatype consists of – Another word for field is members.

Fields can have different types.

Access Modifier:

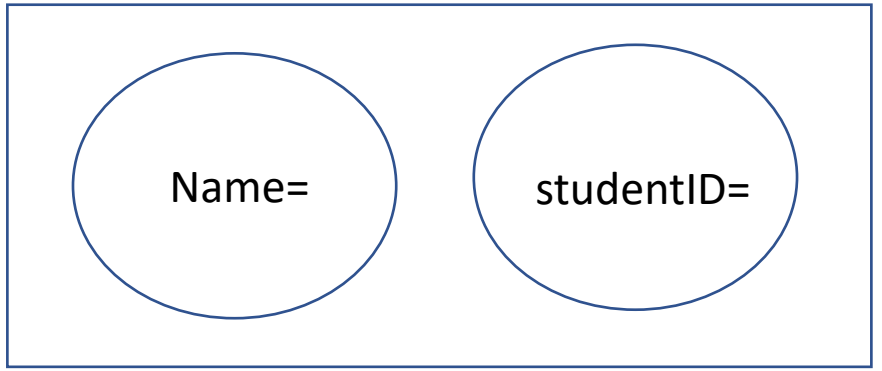
- Private: can be accessed within the class (default)
- Public: can be accessed from anywhere
- **Structs** – same as classes but default access modifier is public.

Class

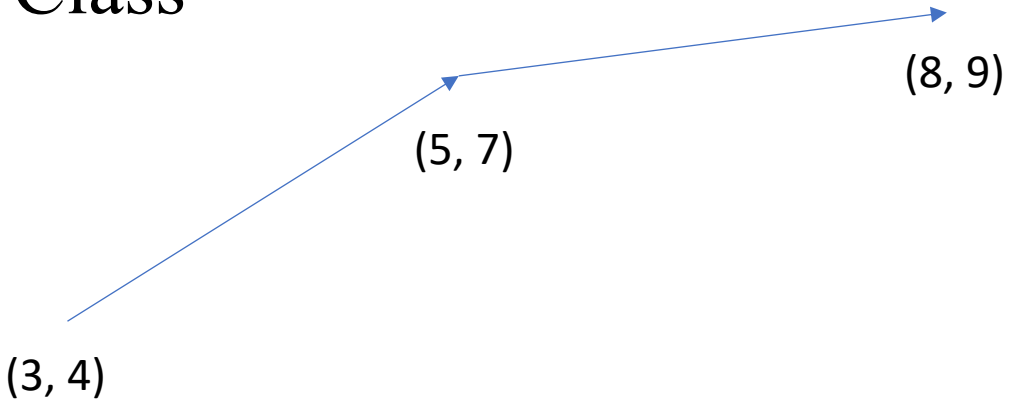
```
class CS590Class {
public:
    char *name;
    int studentID;
};
```

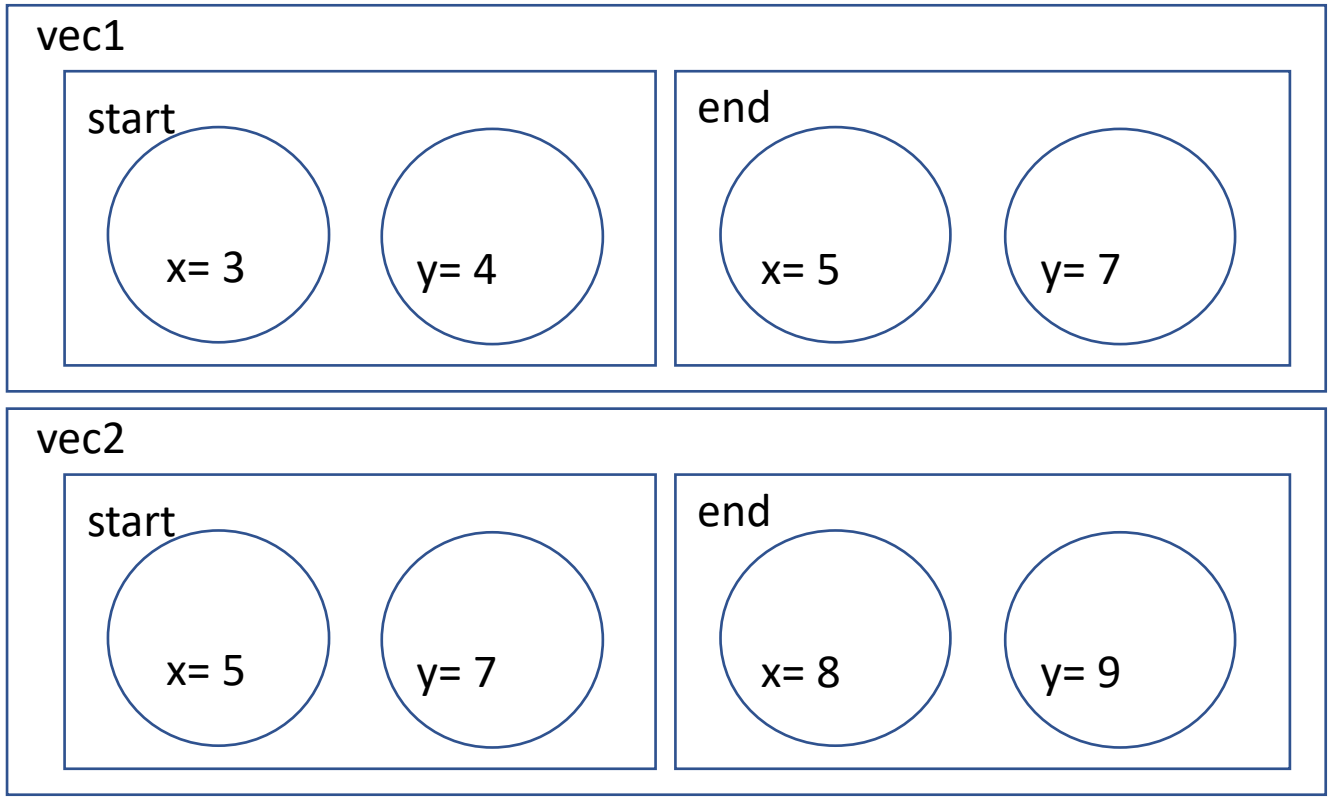
```
int main() {
    CS590Class student1;
    CS590Class student2;
    student1.name = "Jang"
}
```

- An instance is an occurrence of a class. Different instances can have their own set of values in their fields.
- If you wanted to represent 2 different students (who can have different names and IDs), you would use 2 instances of CS590 Students.
- Defines 2 instances of CS590: one called student1, the other called student2.
- To access fields of instances, use variable.fieldName



Class





Practice:

- A point consists of an x and y coordinate
- A vector consists of 2 points: a start and a finish
- Assigning instances for fields

Class

Passing classes to functions

- Passing by value passes a copy of the class instance to the function; changes aren't preserved

```
class Point { public: double x, y; };
void offsetPoint(Point p, double x, double y) {
 p.x += x;
 p.y += y;
}
int main() {
 Point p;
 p.x = 3.0;
 p.y = 4.0;
 offsetPoint(p, 1.0, 2.0);
 cout << "(" << p.x << "," << p.y << ")";
}
```

Class

Constructors

- Method that is called when an instance is created
- Can accept parameters
- Can have multiple constructors

```
class Point {
public:
    double x, y;
    Point() {
        x = 0.0; y = 0.0; cout << "default constructor" << endl;
    }
    Point(double nx, double ny) {
        x = nx; y = ny; cout << "2-parameter constructor" << endl;
    }
};

int main() {
    Point p;
    Point q(2.0, 3.0);
}
```

Mathematic Review – Exponents & Logarithms

Exponents

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

Logarithms

$$X^A = B \text{ i.f.f. } \log_X B = A.$$

$$\log_A B = \frac{\log_C B}{\log_C A}; A, B, C > 0 \ \& \ A \neq 1$$

$$\log AB = \log A + \log B$$

$$\log \frac{A}{B} = \log A - \log B$$

$$\log(A^B) = B \log A$$

Conclusion

C++ Review

- Although things we discussed are not all we need to know, yet this is a good starting point if you did not have C++ experiences before.

Math Review

- We do not use a heavy math in this class but knowing general properties of exponents and logarithms will be helpful.

Next Week

- Sorting will be discussed.
- Will discuss the quiz & homework.