CS 590: Algorithms

Week 1: Orientation & Advanced C++ Review

Fall 2023

CS 590: Algorithms

Week 1: Orientation

Fall 2023

1.1. Orientation

- About me
- Course Information
- Course Description
- Technology Requirements
- Grading Procedure and Policy

About Me

Prof. In Suk Jang

- Contact Information: ijang@stevens.edu
  - For a course related emails, please use Canvas
- Teaching Courses at Stevens: Algorithm, Machine Learning, Java Programming
- Research Interests: Black Holes, Machine Learning

Course Information

- Course Title: CS 590 Algorithms
- Course Period: Tuesday 6:30 PM – 9:00 PM
- Virtual Office Hours:
  - Friday 10 AM – 11 AM
  - Link: https://stevens.zoom.us/j/5516841287
- Reading Sources:
  - Textbook: Introduction to Algorithms, 3rd Edition, MIT press – CLRS
  - Lecture Notes and additional C++ documents are available in Canvas.

Course Description

- This course is on more complex data structures, algorithm design, and analysis using C++.
- Topics to discuss:
  - Asymptotic complexity analysis
  - Various sorting algorithms
  - Algorithm design techniques
  - Data structures
  - Graph algorithms

Course Description

After successful completion of this course, students will be able to:

- **Complexity** – Explain the meaning of big-O, Theta, and Omega notations. Calculate
- the asymptotic running time of standard algorithms, and use it to compare efficiency.
- **Master Theorem** – Use the Master Theorem to prove asymptotic assumptions
- **Sorting** - Compare and analyze basic and advanced sorting algorithms.
- **Trees** - Implement advanced search trees such as Binary Search and Red-Black Trees.
- **Algorithmic Design -** Apply standard algorithm design techniques such as the greedy technique, dynamic programming, hashing, and space/time trade-offs.
- **Graphs** - Implement standard algorithms using graphs and weighted graphs in C++ (e.g., DFS, BFS, MST, topological sort).
- **Shortest Paths** – Implement standard algorithms to solve the shortest path-finding problem. (Dijkstra, Bellman-Ford, Floyd-Warshall)

Technology Requirement

- Baseline technical skills necessary for online courses
  - Basic computer and web-browsing skills
  - Navigating Canvas
- Required Equipment
  - Computer: current Mac (OS X) or PC (Windows 7+) with high-speed internet connection
- Required Software
  - Microsoft Word
- Integrated Developing Environment (IDE)
  - Dev C++
  - Visual Studio
  - Eclipse
  - Online Compiler: https://www.onlinegdb.com/

Grading Procedure & Policy

Five Bi-Weekly Assignments	60%
Midterm (7th week)	15%
Final Exam (15th week)	25%

- **Grades will be based on:**

Any complaint regarding a grade must be presented **no later than 7 weekdays** following the publication of grades of respective assignments. Penalties for specific mistakes that are applied to exams, assignments and quizzes are equal for all students in the course. If you contact me to negotiate these penalties, I will not respond.

- **Late Policy**

Late assignments (even by 2 seconds) will be given **a -10% decrease penalty per day.**

Grading Procedure & Policy

- **Five Bi-Weekly Assignments (60%):**
  - The programming assignments will be done individually. No collaboration is allowed between students. **Any sign of collaboration will result in a 0 and be reported to the Honor Board.**
    - Programming assignments might be tested for similarity using MOSS or similar software.
    - Students caught collaborating for a second time will receive a failing grade (F) in the course.
  - No code from online resources is allowed to be used besides the code that I will share with you.
  - You can do the first assignment with another language if you are new to C++.
    - But will be penalized for not using C++ from Assignment 2. (15% for each assignment)
  - The most objective of assignments is to implement algorithms and analyze what we discuss in class.
  - All assignment results must be written in a report format and submitted with codes.
  - Each assignment will be released on Friday after the class.

Grading Procedure & Policy

Midterm (15%):

- Time: 10/18 Tuesday 6:30 – 8 PM
- Format: In-person
- Topics: Computational complexities and sorting algorithms

Final Exams (25%):

- Time: 12/13 Tuesday 6:30 – 9 PM
- Format: TBA
- Topics: Cumulative

1.2. Introduction to Algorithm

- Algorithm
- Data Structure
- Efficiency

Algorithms

What are Algorithms?

- Any well-defined computational *procedure* takes some value, or set of values, as input and produces some value, or set of values, as output.
- *Sequences of computational steps* that transform the input into the output
- Tools for *solving* a well-specified computational problem (input/output relationship)
- An instance of a problem consists of the input needed to compute a solution to the problem.
- The correct algorithm solves the given computational problem.

Data Structure

- A way to store and organize data to facilitate access and modification.
- No single data structure is optimal for all purposes.
- Usually optimized for a specific problem set.
- Important to know the strength and limitations of several of them.

Efficiency

Computing time and memory are bounded resources.

Efficiency:

- Different algorithms that solve the same problem often differ in their efficiency.
- More significant than differences due to hardware (CPU, memory, disks, ...) and software (OS, programming language, compiler, ...).

Algorithms and other technologies

Consider algorithms like computer hardware as a technology

In this course, we care most about ***asymptotic performance.***
For example, how does the algorithm behave when the problem gets very large?

- Running time
- Memory/storage requirements
- Bandwidth/power requirements/logic gates/etc.

CS 590: Algorithms

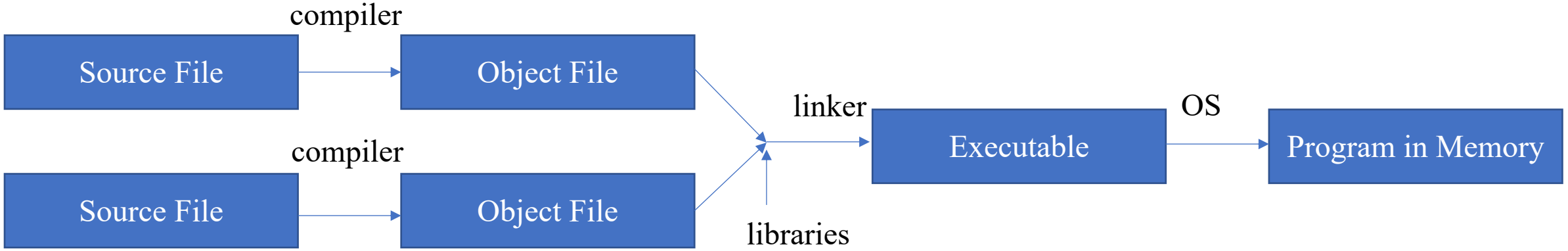
Week 1: Advanced C++ Review

Fall 2023

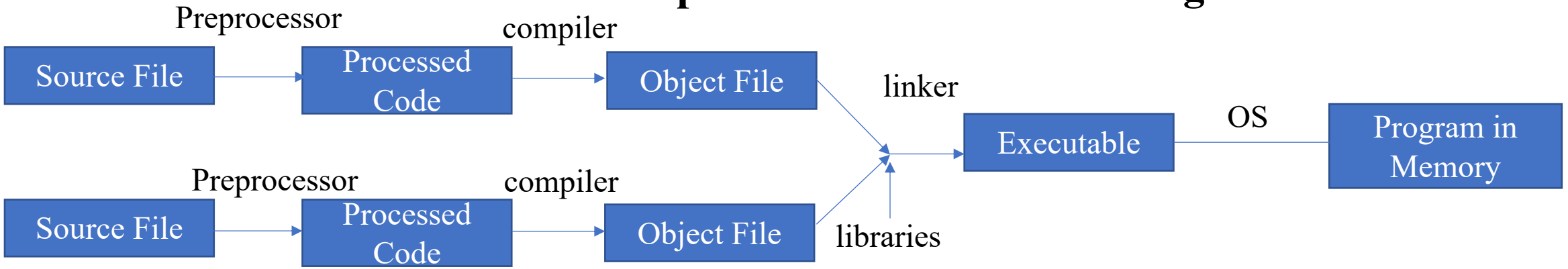
2.1. C++ Review:

- Token
- Basic Language Features
- Variables
- Arrays
- Strings
- Pointers
- Functions
- Class

Source files to processor diagram



Source files to processor modification diagram



Tokens

- **Tokens** are the minimal chunk of programs with meaning to the compiler – the smallest meaningful symbols in the language.
- Six kinds of tokens are…

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 new expressions.
  - 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 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 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 used on most 32-bit computers.

Variable

- 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.

```
# 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;
}
```

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* in 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 is a 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 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;
}
```

- Is **fragment2** correctly initialized?
- Is **fragment3** declared or initialized?
- Is **finalString** initialized?

  - strcpy and strcat functions copy and concatenate strings, respectively.
  - Can you guess what **finalString** will display?

Pointers

- When you refer to the variable by name in your code, the computer must take two steps:
  - Look up the address that the variable name corresponds to
  - 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:
  - `&x` evaluates to the address of `x` in memory.
  - `*(&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 initialize to the x address.
- Pointers can have any type of value:

```
data_type *pointer_name;
```

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

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

- We can use the dereferenced pointer as a single value:

```
*ptr = 5;
```

Pointers

- 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 to declare a pointer, **void** specifies that the pointer is "universal".
- What will the output be?

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

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

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 pointer is valid?

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

Pointers

References

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

Consider a code example…

- 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 the 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 you want to refer 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 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. take a variable's address (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 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 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 the elements below?

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

Pointers

```
1  // my first pointer
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int firstvalue, secondvalue;
8    int * mypointer;
9
10   mypointer = &firstvalue;
11   *mypointer = 10;
12   mypointer = &secondvalue;
13   *mypointer = 20;
14   cout << "firstvalue is " << firstvalue << '\n';
15   cout << "secondvalue is " << secondvalue << '\n';
16   return 0;
17 }
```

- mypointer = &firstvalue;
- *mypointer = 10;
- mypointer = &secondvalue;
- *mypointer = 20;

Pointers

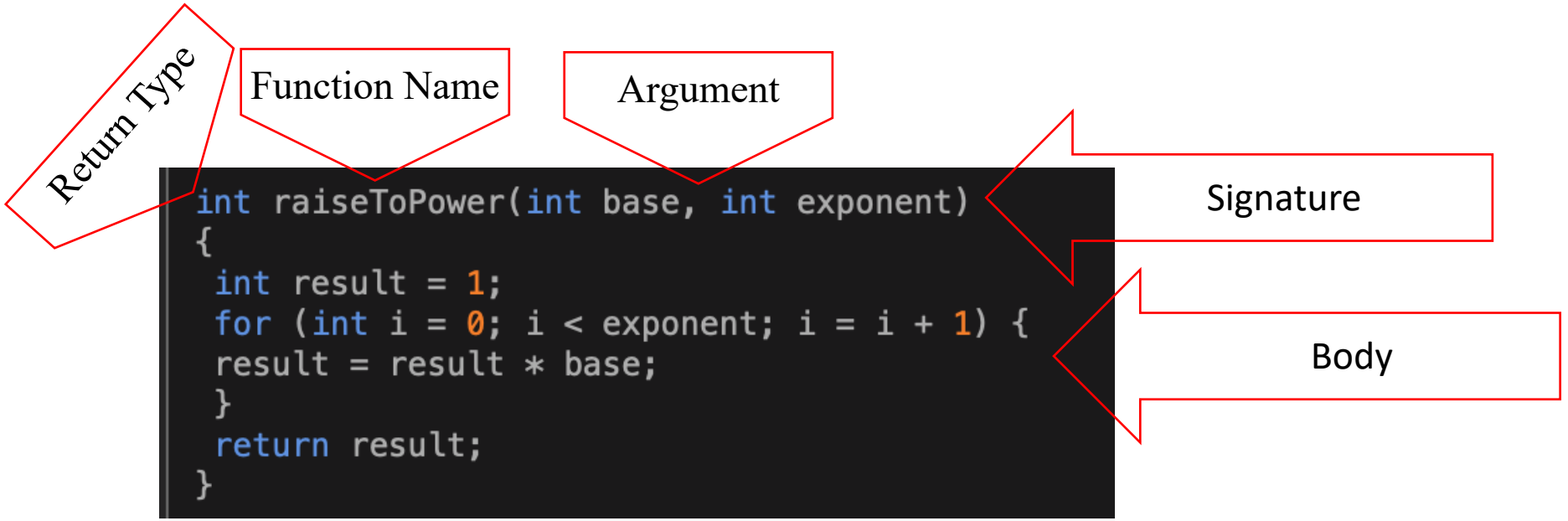
```
1  // more pointers
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int firstvalue = 5, secondvalue = 15;
8    int * p1, * p2;
9
10   p1 = &firstvalue;  // p1 = address of firstvalue
11   p2 = &secondvalue; // p2 = address of secondvalue
12   *p1 = 10;          // value pointed to by p1 = 10
13   *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
14   p1 = p2;           // p1 = p2 (value of pointer is copied)
15   *p1 = 20;          // value pointed to by p1 = 20
16
17   cout << "firstvalue is " << firstvalue << '\n';
18   cout << "secondvalue is " << secondvalue << '\n';
19   return 0;
20 }
```

Functions

Advantages

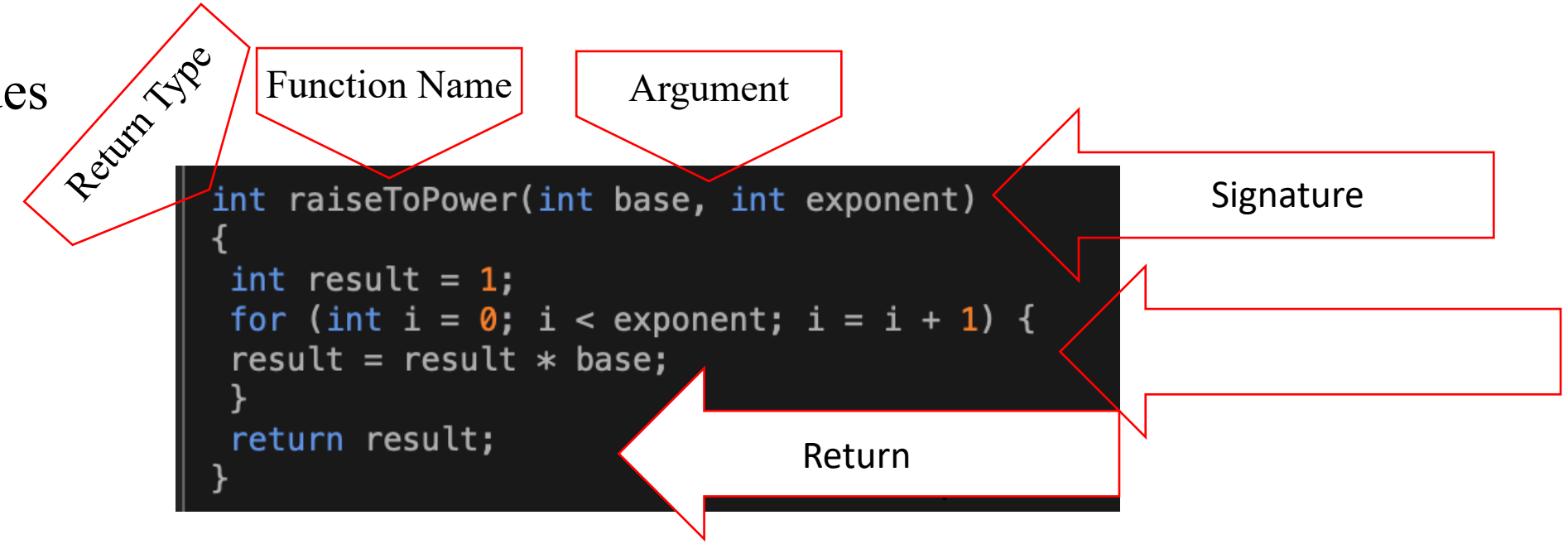
- Readability: sqrt(5) is clearer than copy-pasting in an algorithm to compute the square root, e.g., 5^(0.5).
- Maintainability: To change the algorithm, change the function (vs. changing it everywhere you ever used it).
- Code reuse: Let other people use algorithms you've implemented.

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.
- The 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;
}
```

- printOnNewLine 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)
    }
}
```

- Recursive Algorithms:
- 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.
  - It 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 that 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 the field is members.

Fields can have different types.

Access Modifier:

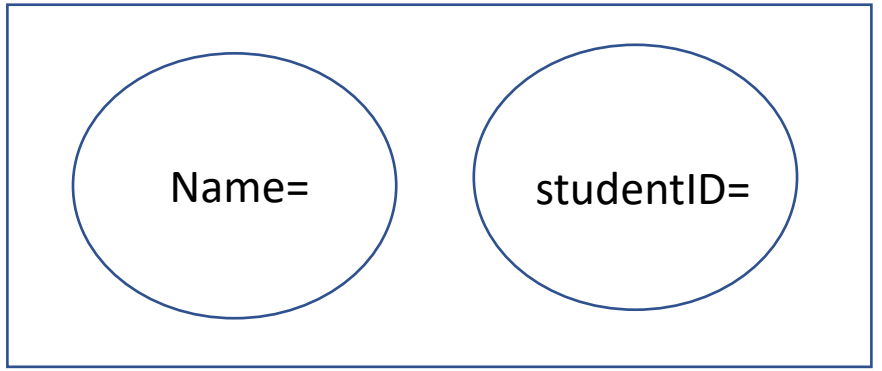
- Private: can be accessed within the class (default)
- Public: can be accessed from anywhere
- **Structs** – same as classes, but the default access modifier is public.

Class

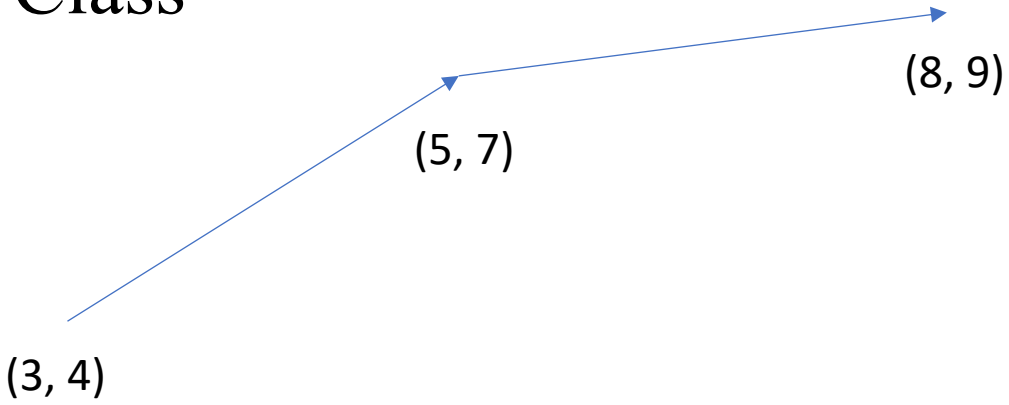
```
class CS590Class {
public:
    char *name;
    int studentID;
};
```

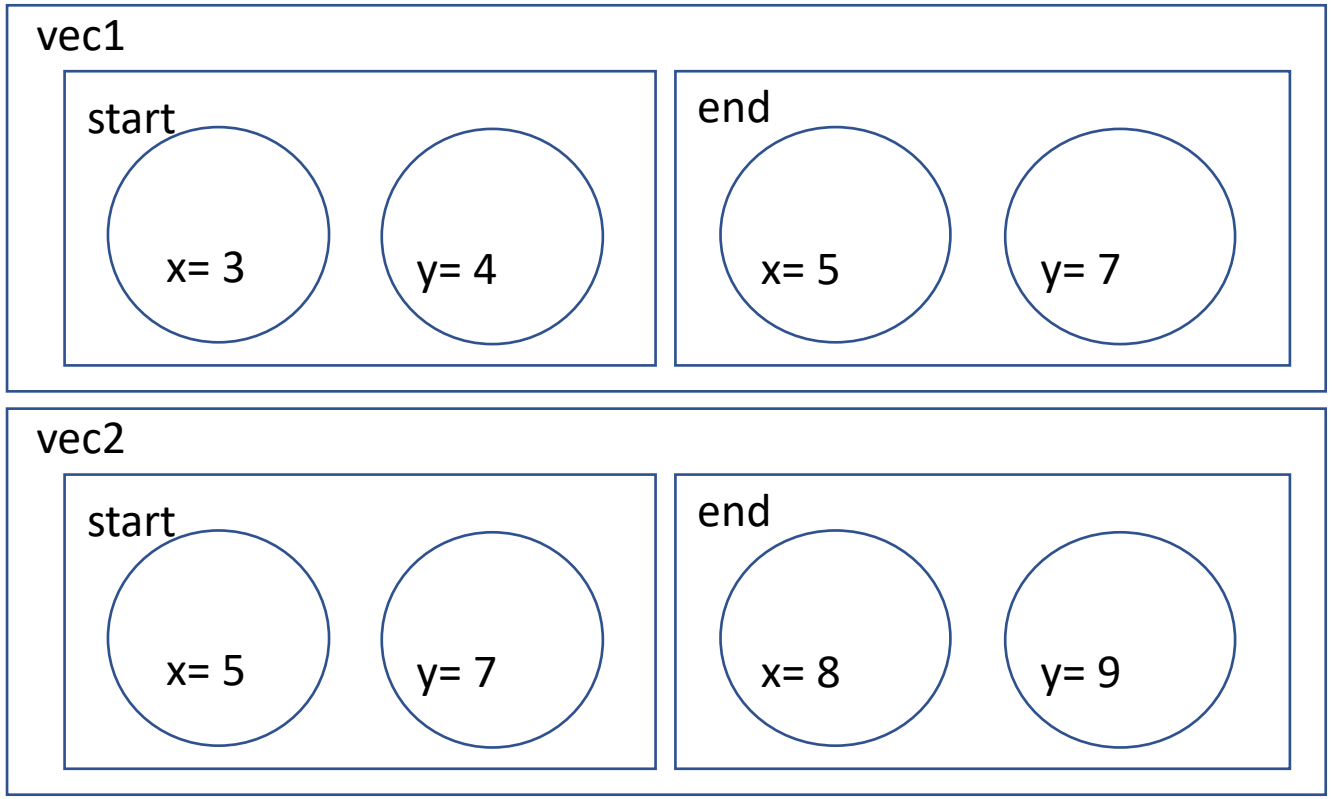
```
int main() {
    CS590Class student1;
    CS590Class student2;
    student1.name = "Jang"
}
```

- 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);
}
```

Conclusion

C++ Review

- Although the things we discussed are not all we need to know, this is a good starting point if you do not have C++ experience.

Next Week

- We will discuss sort algorithms and growth functions.
- We will discuss the first homework.