CS590/CPE590

Algorithms
Introduction

Kazi Lutful Kabir

Spring 2023

Introduction

Course Description

- This is a course on more complex data structures, and algorithm design and analysis, using one or more modern imperative language(s), as chosen by the instructor. Topics include advanced and/or balanced search trees; further asymptotic complexity analysis; standard algorithm design techniques; graph algorithms; complex sort algorithms; and other "classic" algorithms that serve as examples of design techniques.

Introduction

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 basic and advanced search trees such as Binary Search Trees, and Red-Black Trees.
- **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)
- **Algorithmic Design -** Apply standard algorithm design techniques such as the greedy technique, dynamic programming, hashing, and space/time trade-offs.

Technological Requirements

Baseline Technical Skills
-- Basic computer and web-browsing skills
-- Navigating Canvas

Required Equipment
Computer: current Mac (OS X) or PC (Windows 10+) with internet-access

Required Software
Microsoft Word/Latex

Integrated Developing Environment (IDE)
Dev C++
Visual Studio
Eclipse
Virtual Box
https://www.onlinegdb.com/online_c++_compiler

Introduction to Algorithms

- Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output. That means, A sequence of well-defined computational steps that takes a value/set of values (input) and produces a value/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)
- Instance of a problem consists of the input needed to compute a solution to the problem.
- Correct algorithm solves the given computational problem.

- Example:

  Sorting (We will investigate Sorting Algorithms in detail in Lecture-3):
- Input: ⟨31, 41, 59, 26, 41, 58⟩
- Output: ⟨26, 31, 41, 41, 58, 59⟩

Algorithms

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for any legitimate input in a finite amount of time.

Interesting Points

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works must be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

Problem 1

What subtle concept(s) are we trying to emphasize?

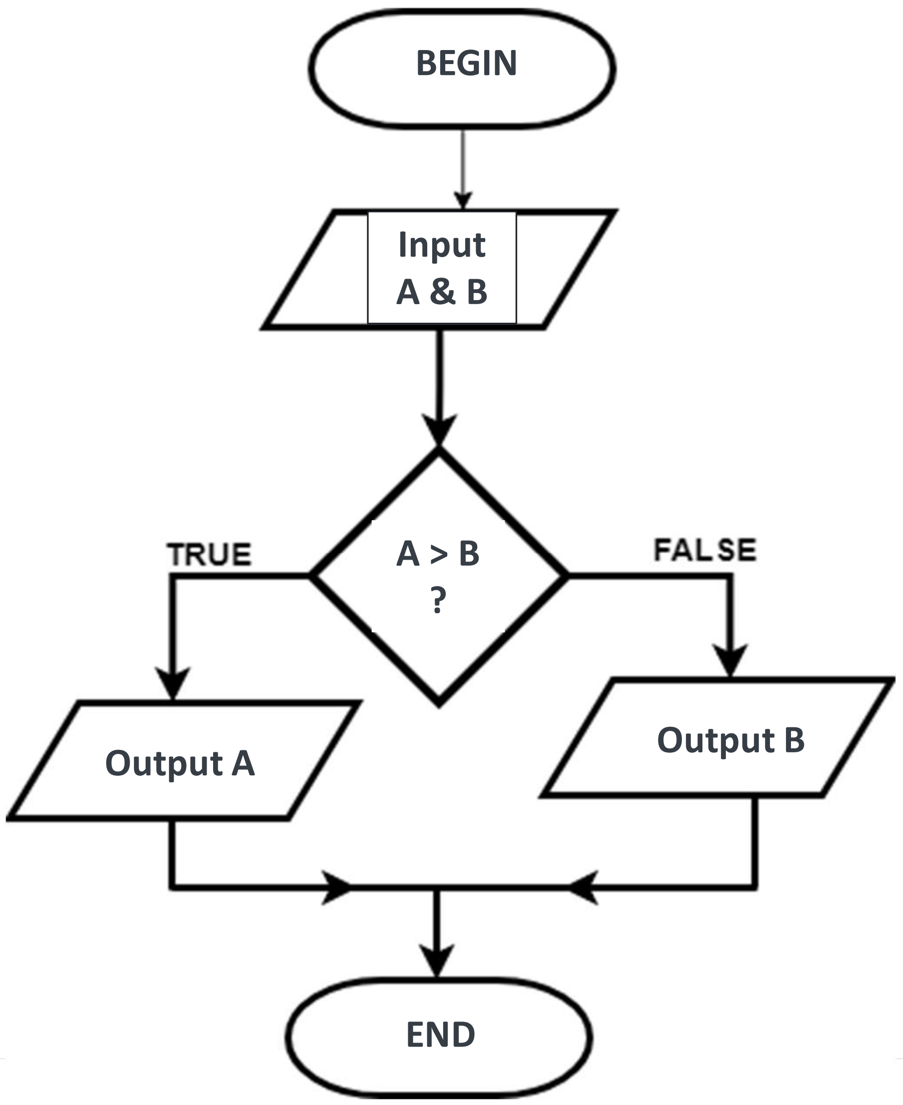
- **Glove Selection**
- There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to get at least one matching pair?
  - The best case?
  - The worst case?

Problem 2

- Find the square root of a number.
- **Input range**: non-negative numbers!
- Why?

Problem 3

- **Find the max between two numbers**
- Unambiguous steps. How?

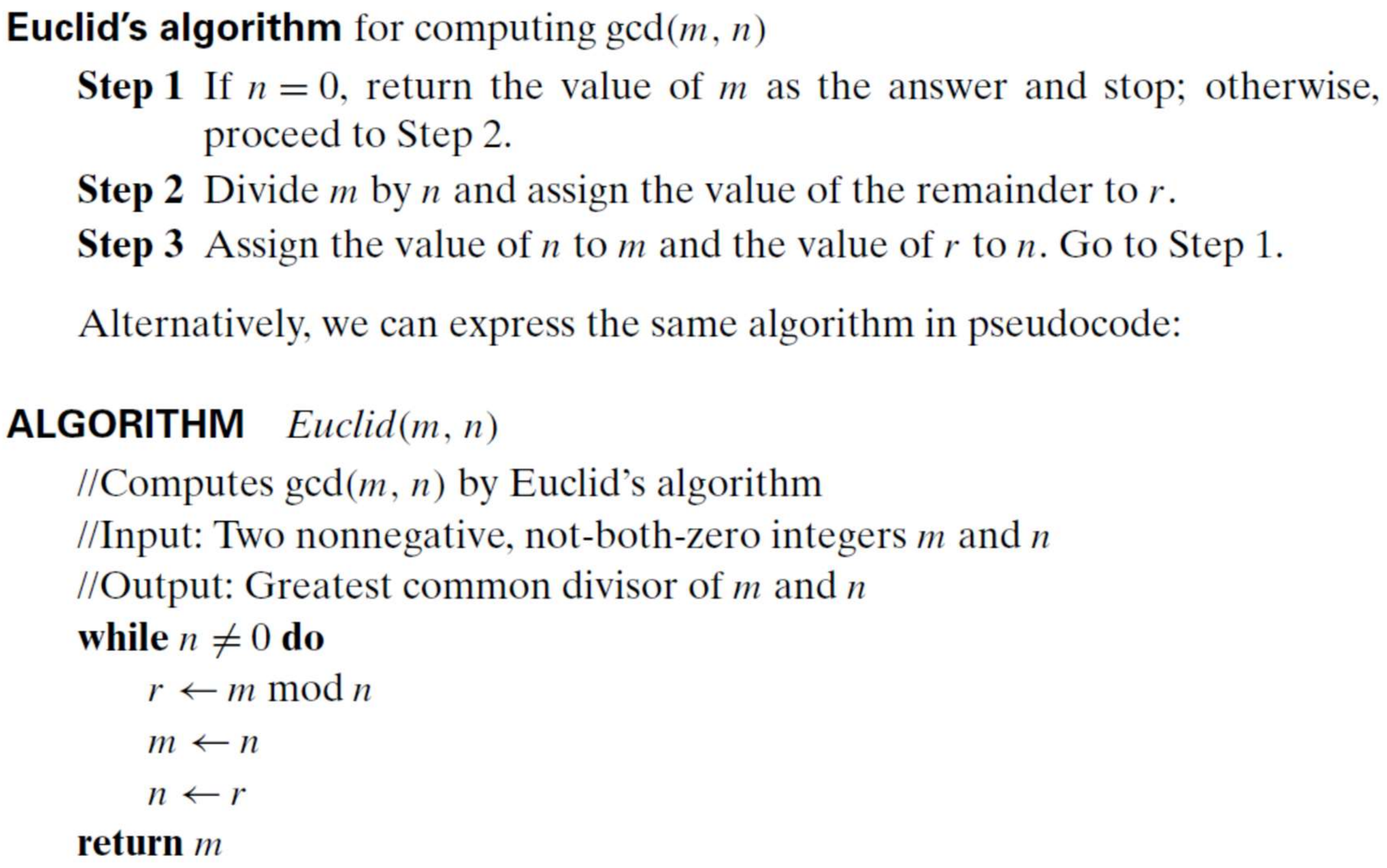


Simple Algorithm: GCD

Greatest Common Divisor

- The greatest common divisor of two integers m and n, denoted gcd(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.
- How did you solve this type of problem in your math courses?
- Work out the gcd(60, 24) now. How many steps did you take?

Simple Algorithm: GCD (Euclid's Algorithm)



Euclid's algorithm for computing gcd(m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

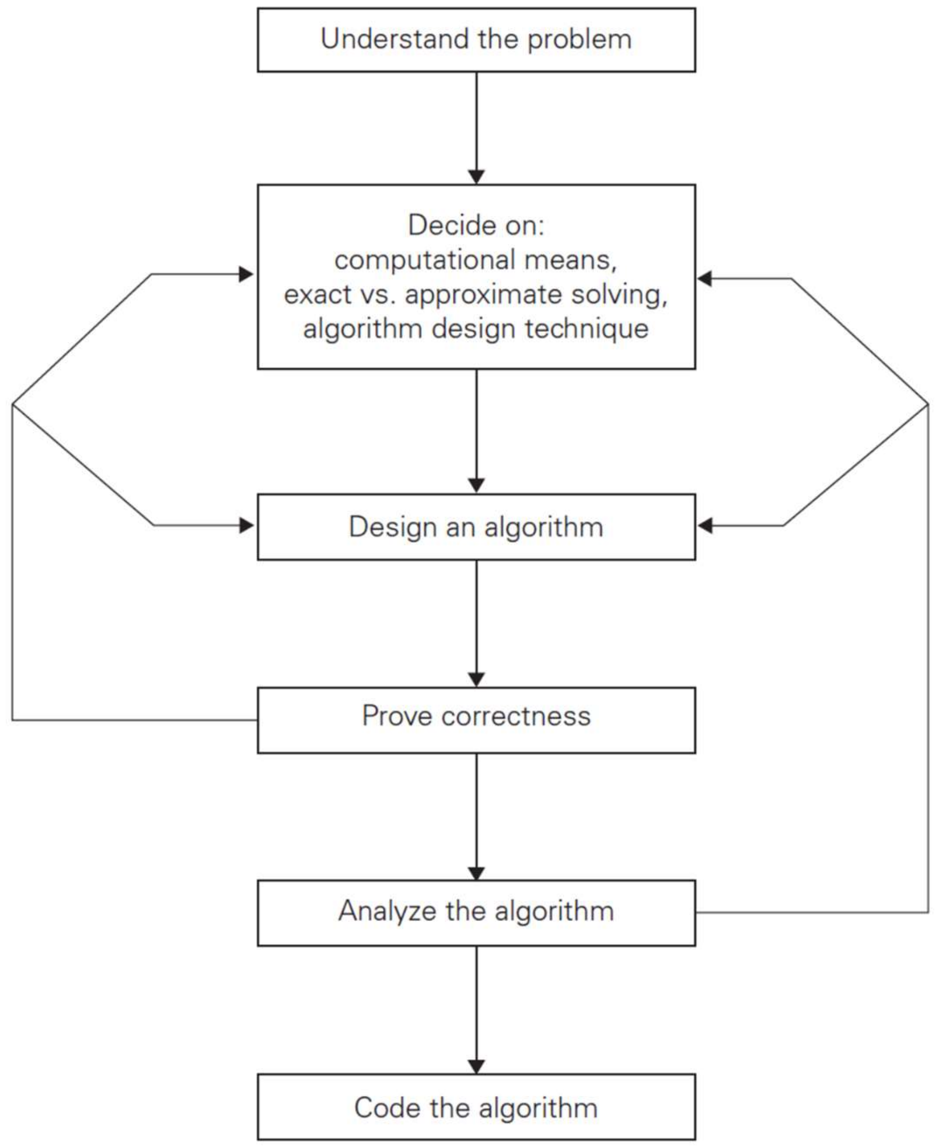
Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

```
ALGORITHM  Euclid(m, n)
    //Computes gcd(m, n) by Euclid's algorithm
    //Input: Two nonnegative, not-both-zero integers m and n
    //Output: Greatest common divisor of m and n
    while $n \neq 0$ do
        $r \leftarrow m \bmod n$
        $m \leftarrow n$
        $n \leftarrow r$
    return m
```

Algorithm Design and Analysis Process

- A Good Algorithm is the result of repeated effort and rework.



Data Structure

- A way to store and organize data in order to facilitate access and modification.
- No single data structure optimal for all purposes.
- Usually optimized for a specific problem setting.
- 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, ...).

Other Factors

In this course, we care most about **asymptotic performance**

– How does the algorithm behave as the problem size gets very large?

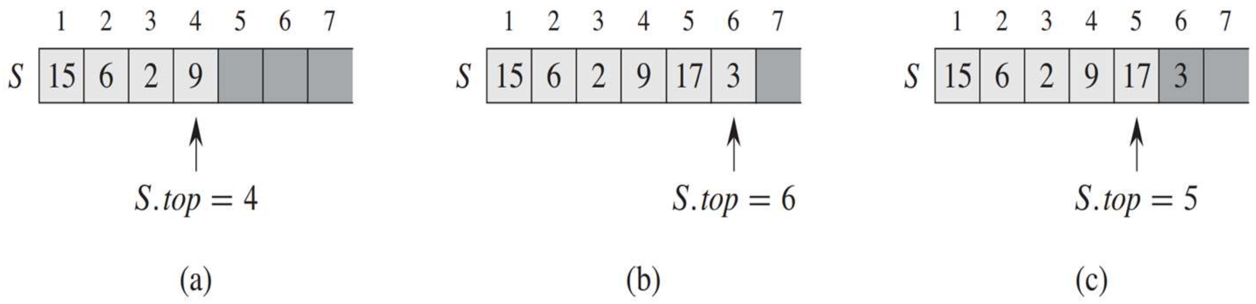
- Running time
- Memory/storage requirements
- Bandwidth/power requirements/logic gates and so on.

Data Structure : Review

- Stack
- Queue
- Linked List

Stack

- **LIFO:** last-in-first-out



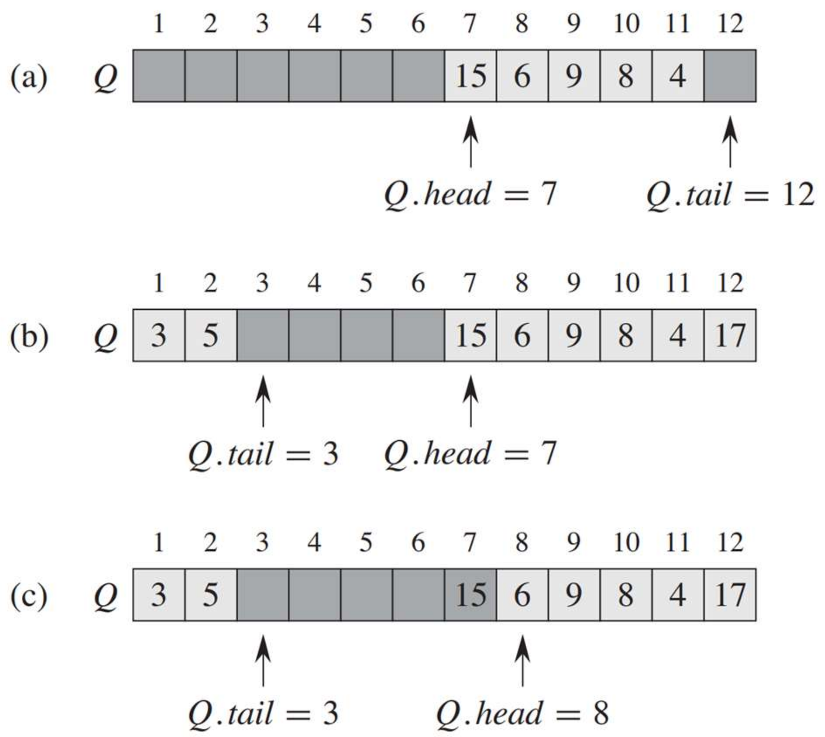
```
STACK-EMPTY(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top − 1
4      return S[S.top + 1]
```

Queue

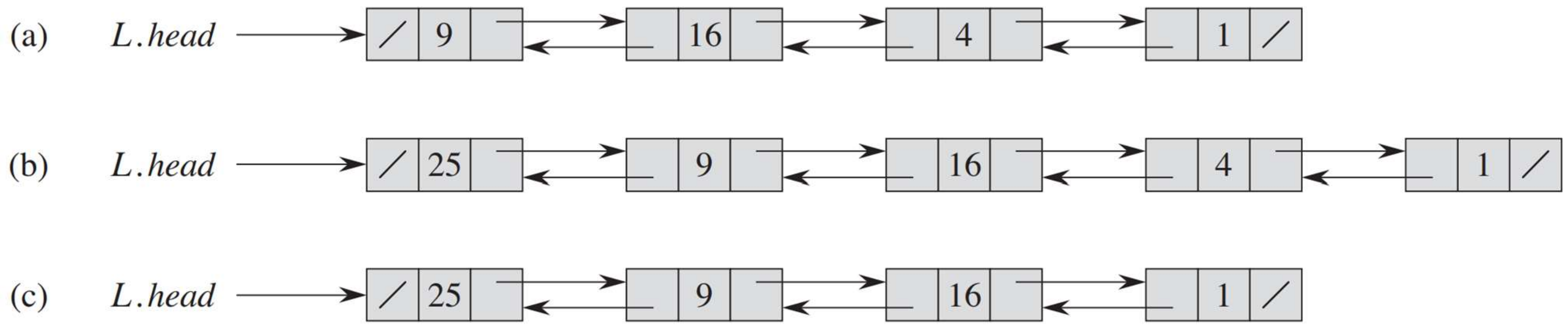
- **FIFO:** first-in-first-out



```
ENQUEUE($Q, x$)
1  $Q[Q.tail] = x$
2  if $Q.tail == Q.length$
3      $Q.tail = 1$
4  else $Q.tail = Q.tail + 1$
```

```
DEQUEUE($Q$)
1  $x = Q[Q.head]$
2  if $Q.head == Q.length$
3      $Q.head = 1$
4  else $Q.head = Q.head + 1$
5  return $x$
```

Linked-list



```
LIST-SEARCH$(L, k)$
1  $x = L.head$
2  while $x \neq$ NIL and $x.key \neq k$
3      $x = x.next$
4  return $x$
```

```
LIST-INSERT$(L, x)$
1  $x.next = L.head$
2  if $L.head \neq$ NIL
3      $L.head.prev = x$
4  $L.head = x$
5  $x.prev =$ NIL
```

```
LIST-DELETE$(L, x)$
1  if $x.prev \neq$ NIL
2      $x.prev.next = x.next$
3  else $L.head = x.next$
4  if $x.next \neq$ NIL
5      $x.next.prev = x.prev$
```

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

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.

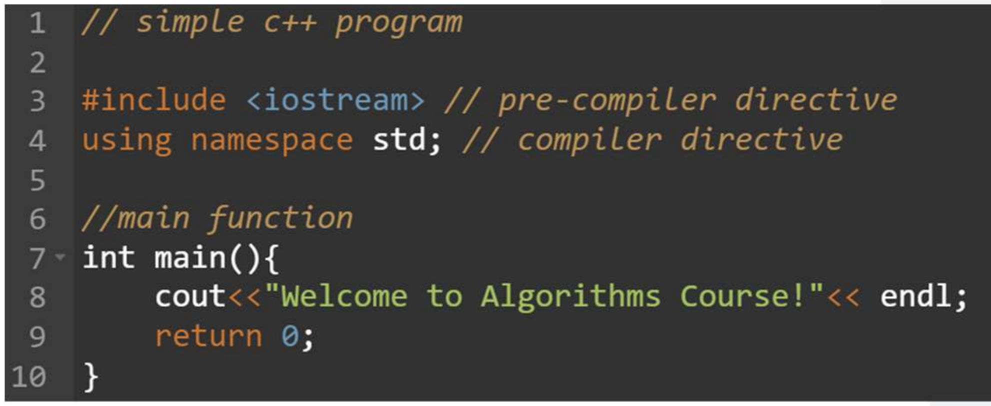
Section 3: Global Declarations

- 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

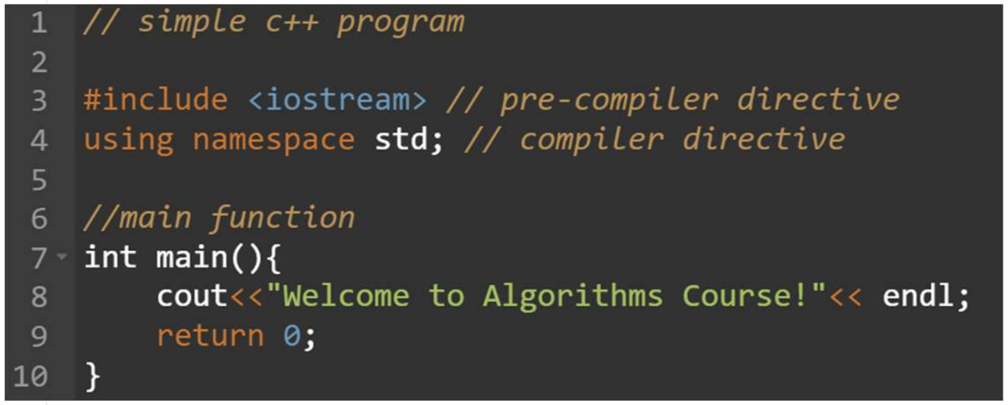


```
// simple c++ program

#include <iostream> // pre-compiler directive
using namespace std; // compiler directive

//main function
int main(){
    cout<<"Welcome to Algorithms Course!"<< endl;
    return 0;
}
```

A Simple C++ Program



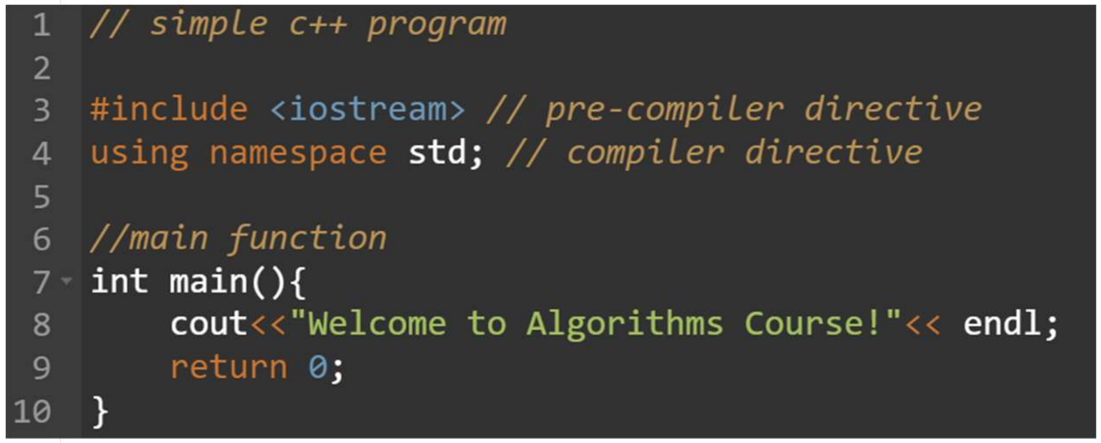
```
// simple c++ program

#include <iostream> // pre-compiler directive
using namespace std; // compiler directive

//main function
int main(){
    cout<<"Welcome to Algorithms Course!"<< endl;
    return 0;
}
```

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



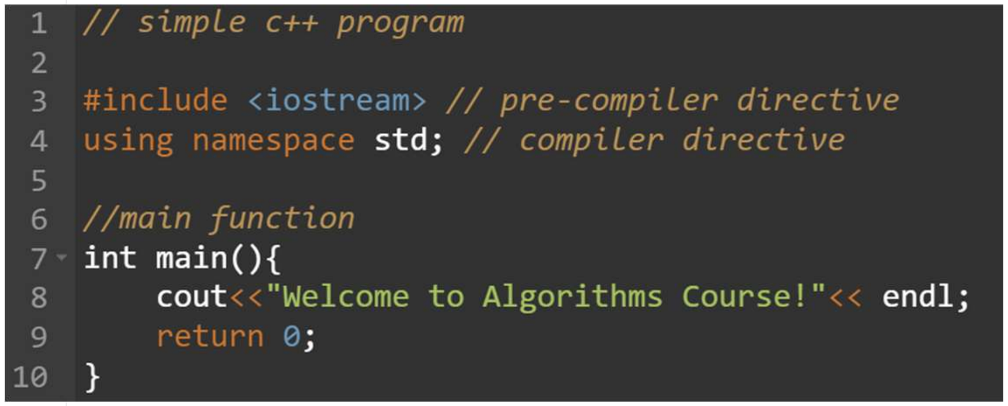
```cpp
// simple c++ program

#include <iostream> // pre-compiler directive
using namespace std; // compiler directive

//main function
int main(){
    cout<<"Welcome to Algorithms Course!"<< endl;
    return 0;
}
```

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



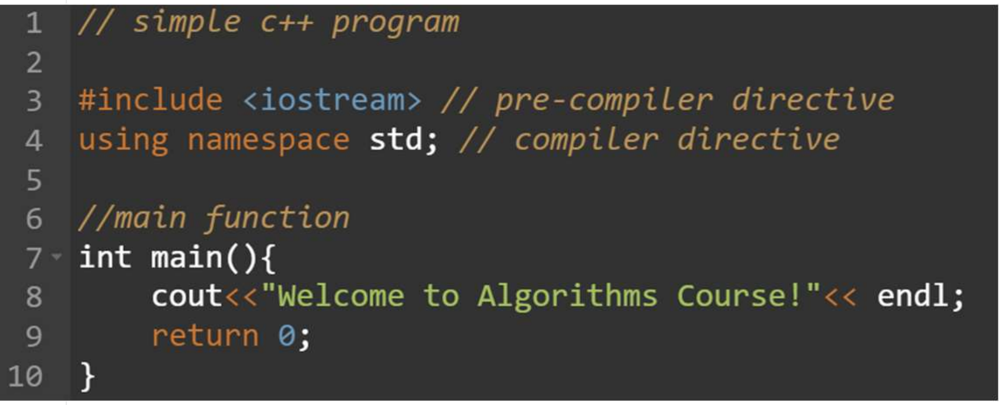
```cpp
// simple c++ program

#include <iostream> // pre-compiler directive
using namespace std; // compiler directive

//main function
int main(){
    cout<<"Welcome to Algorithms Course!"<< endl;
    return 0;
}
```

What is int main () { ... } ?

- *Defines* the main function of any C++ program
- Who calls main?
  - The runtime environment
- Can we have 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



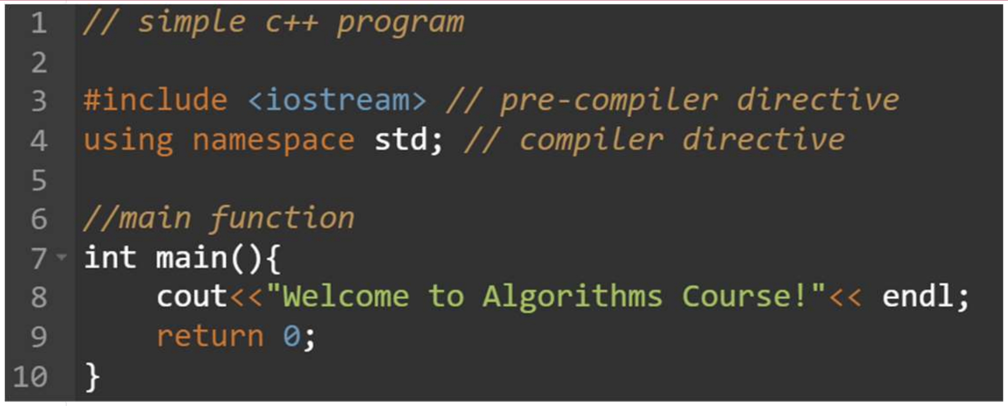
```
// simple c++ program

#include <iostream> // pre-compiler directive
using namespace std; // compiler directive

//main function
int main(){
    cout<<"Welcome to Algorithms Course!"<< endl;
    return 0;
}
```

What's cout << "Welcome to Algorithms Course!" << 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 it is called
  - Can be applied repeatedly to references left-to-right
- Output a string "Welcome to Algorithms Course!"
- **endl** means newline



```
// simple c++ program

#include <iostream> // pre-compiler directive
using namespace std; // compiler directive

//main function
int main(){
    cout<<"Welcome to Algorithms Course!"<< endl;
    return 0;
}
```

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 null pointer can do this

Contents of this presentation are partially adapted from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-1)
and are also based on
Book Chapter-1 & 10, Introduction to Algorithms by *Cormen, Leiserson, Rivest, & Stein*

THANK **YOU**

Stevens Institute of Technology