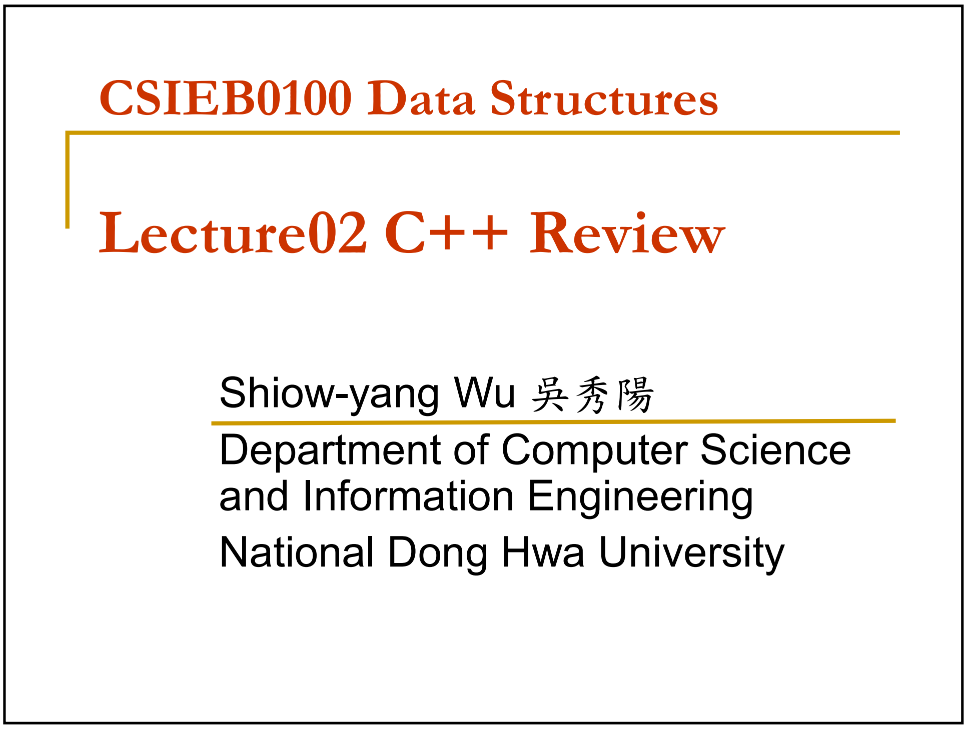


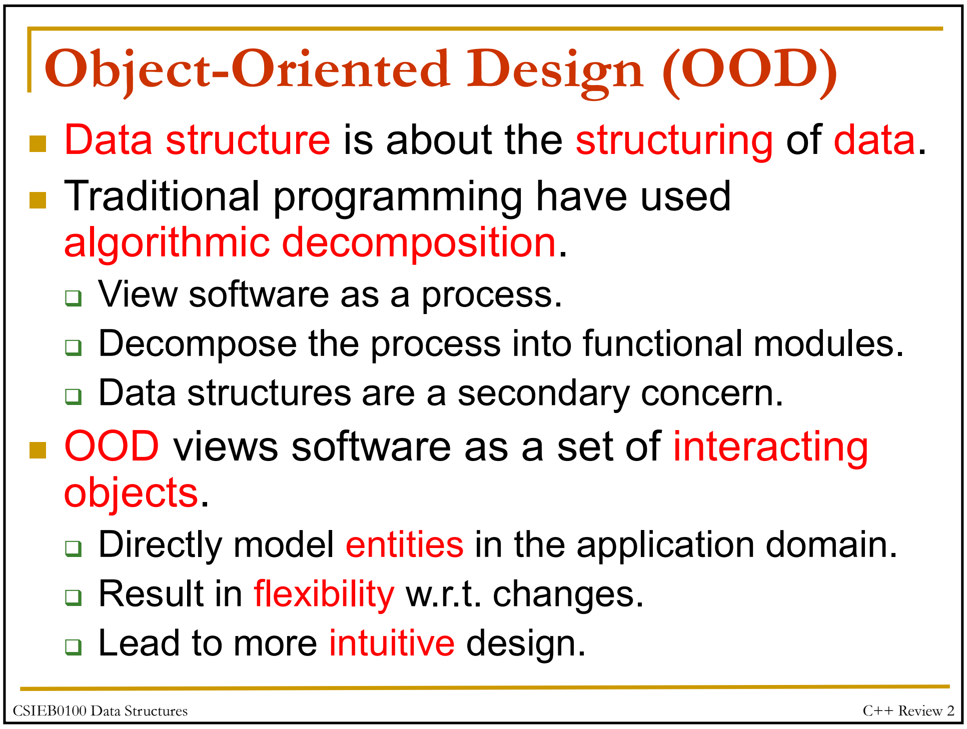
CSIEB0100 Data Structures

Lecture02 C++ Review

Shiow-yang Wu 吳秀陽

Department of Computer Science and Information Engineering

National Dong Hwa University

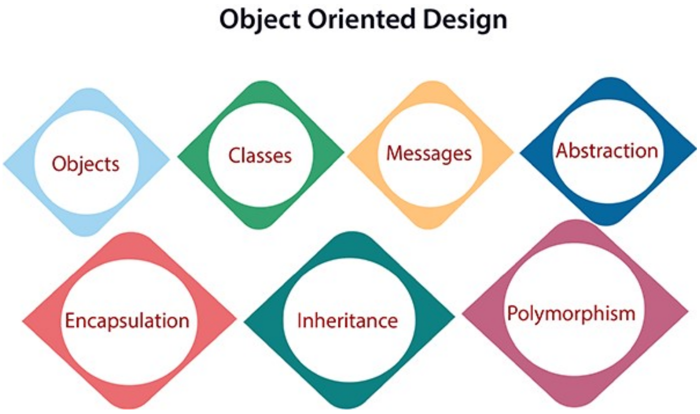


Object-Oriented Design (OOD)

- Data structure is about the structuring of data.
- Traditional programming have used algorithmic decomposition.
  - View software as a process.
  - Decompose the process into functional modules.
  - Data structures are a secondary concern.
- OOD views software as a set of interacting objects.
  - Directly model entities in the application domain.
  - Result in flexibility w.r.t. changes.
  - Lead to more intuitive design.

Key Concepts in OOD

- You need to have solid understanding of the following terms



(https://www.javatpoint.com/software-engineering-object-oriented-design)

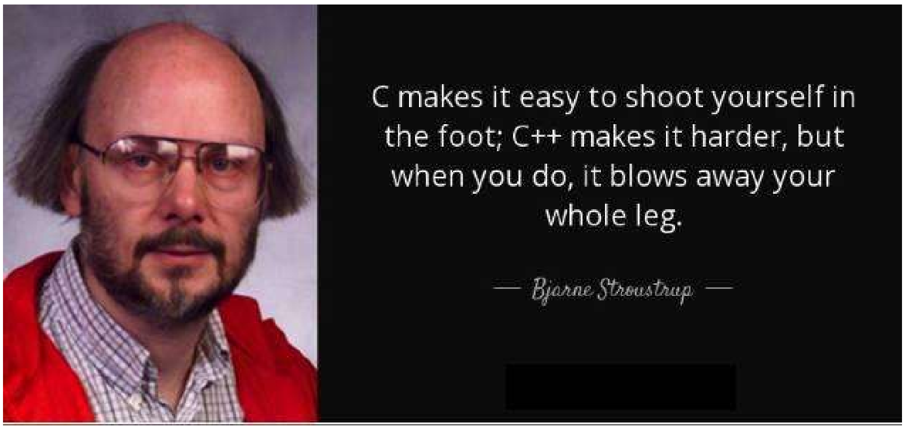
Object-Oriented Concepts

- Object: an entity with behavior (i.e. performs computation) and a local state. (Encapsulation)
- Object-oriented programming(OOP):
  - Representing entities with objects.
  - Each object is an instance of a class.
  - Classes are related to each other by inheritance.
  - Problems/solutions are modeled/provided by a set of objects interacting with each other by passing messages.
- A programming language that supports OOP is called an object-oriented language.
- All other features of OO are related directly or indirectly with the basic characteristics above.

Evolution of Languages

- 1st gen language: Fortran, for evaluating math expression
- 2nd gen language: Pascal, C, for algorithmic and structured programming
- 3rd gen language: Modula, for ADTs
- 4th gen language: C++, Objective C, Smalltalk, for OOP.
- C++ was designed by Bjarne Stroustrup of AT&T Bell Lab. (next slide)
- C/C++ form the basics of many modern langs.

Bjarne Stroustrup



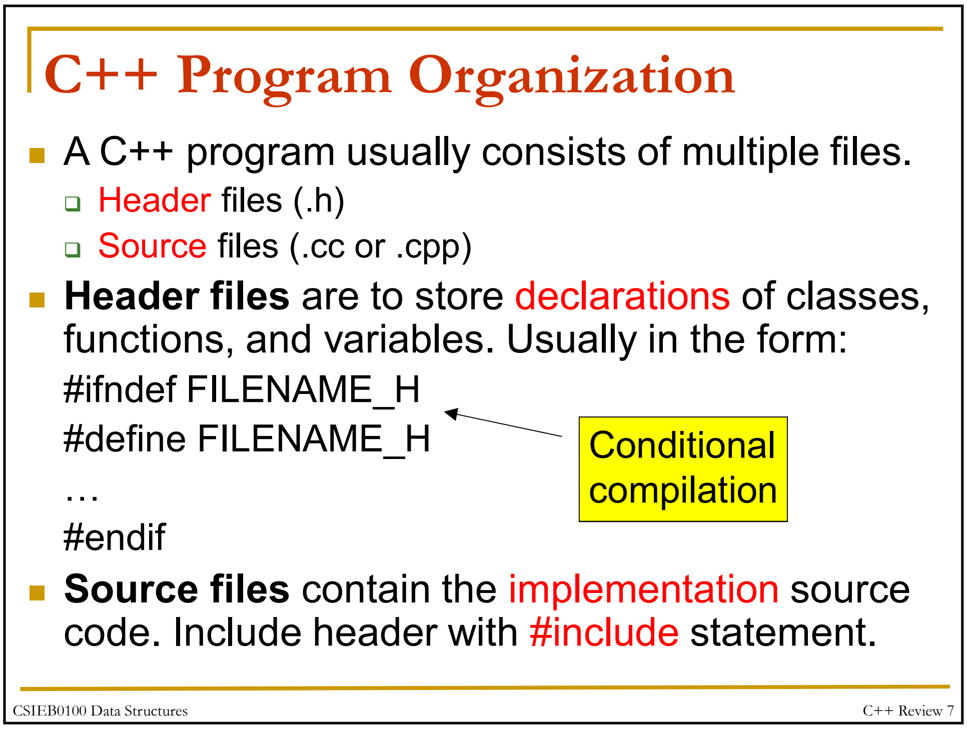
C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.

— Bjarne Stroustrup —

[redacted]

- How to pronounce his name?

(https://youtu.be/9QKHg8wj4MA)



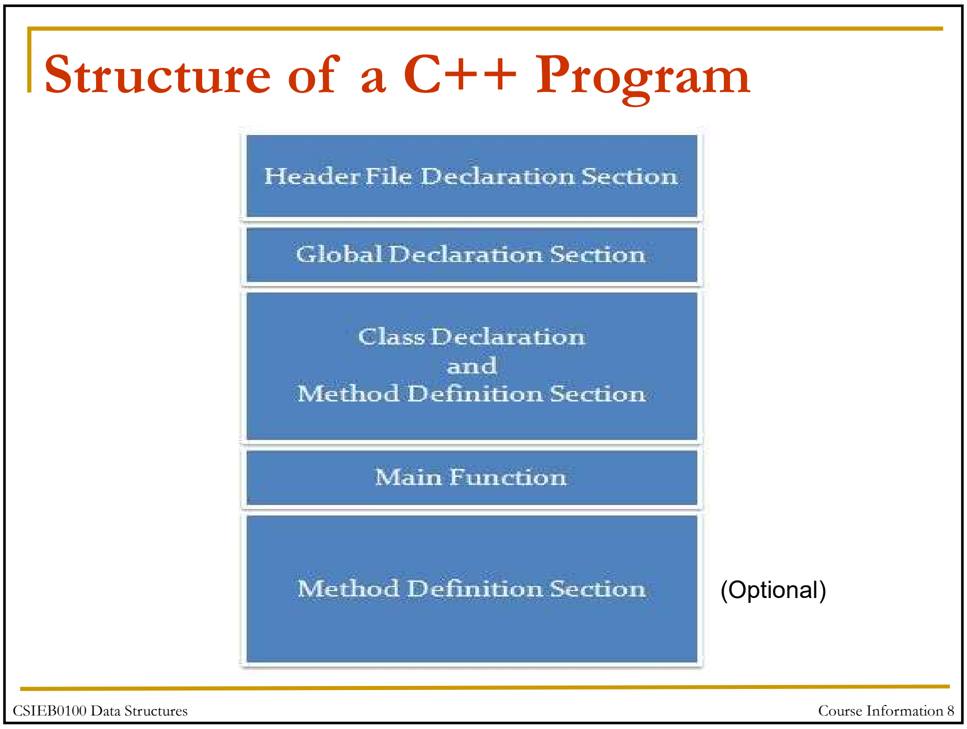
C++ Program Organization

- A C++ program usually consists of multiple files.
  - Header files (.h)
  - Source files (.cc or .cpp)
- **Header files** are to store declarations of classes, functions, and variables. Usually in the form:

```
#ifndef FILENAME_H
#define FILENAME_H
…
#endif
```

Conditional compilation

- **Source files** contain the implementation source code. Include header with #include statement.



Structure of a C++ Program

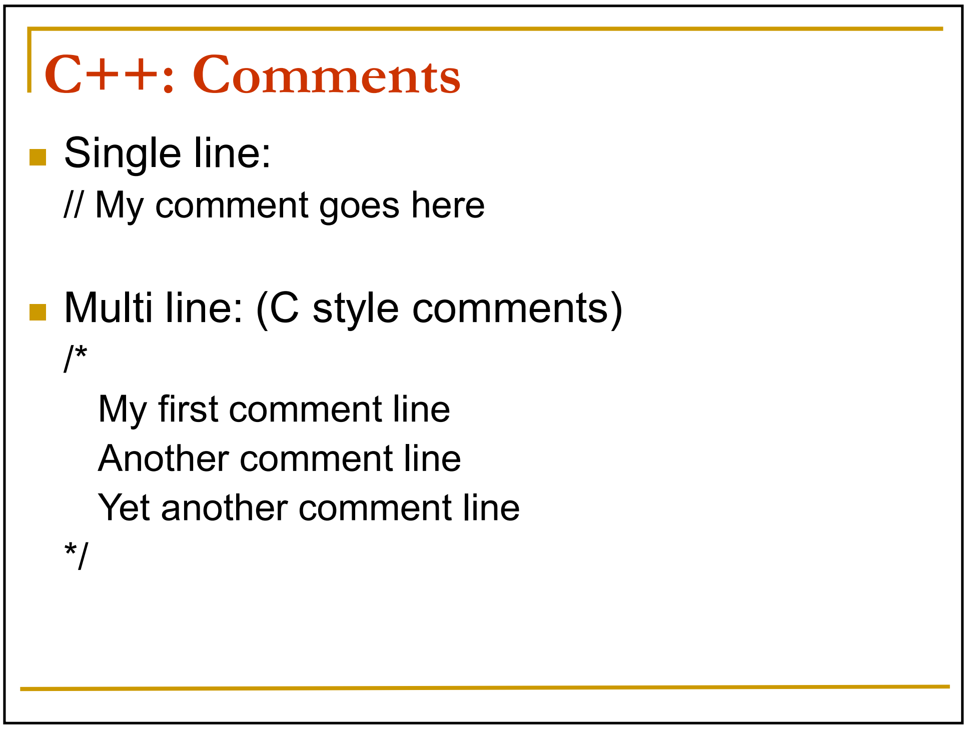
Header File Declaration Section

Global Declaration Section

Class Declaration
and
Method Definition Section

Main Function

Method Definition Section (Optional)



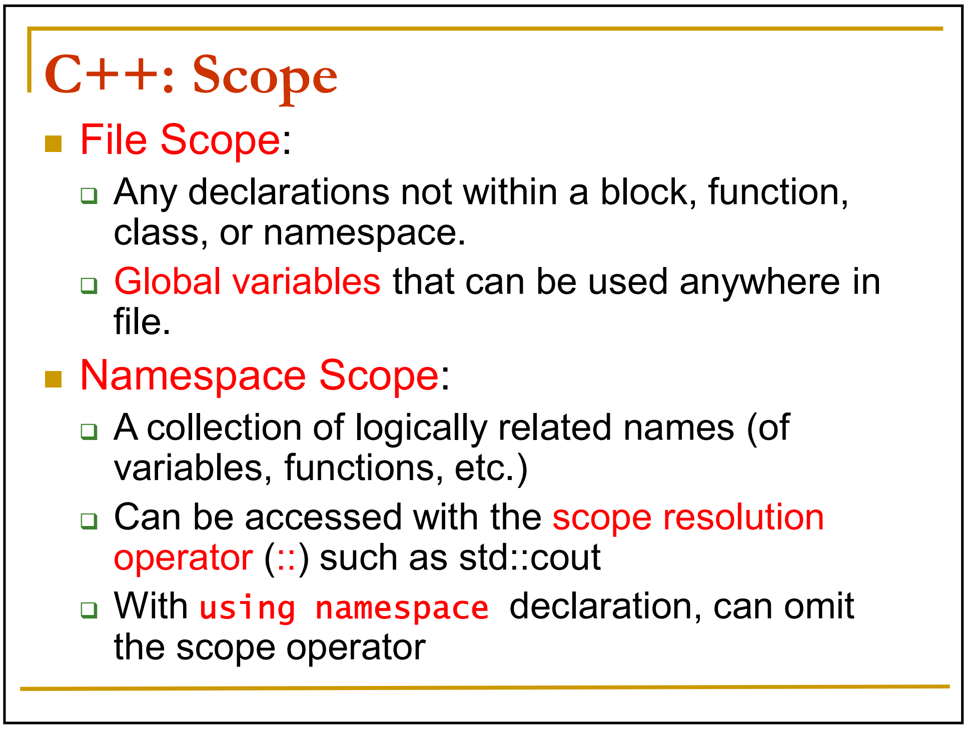
C++: Comments

- Single line:

```
// My comment goes here
```

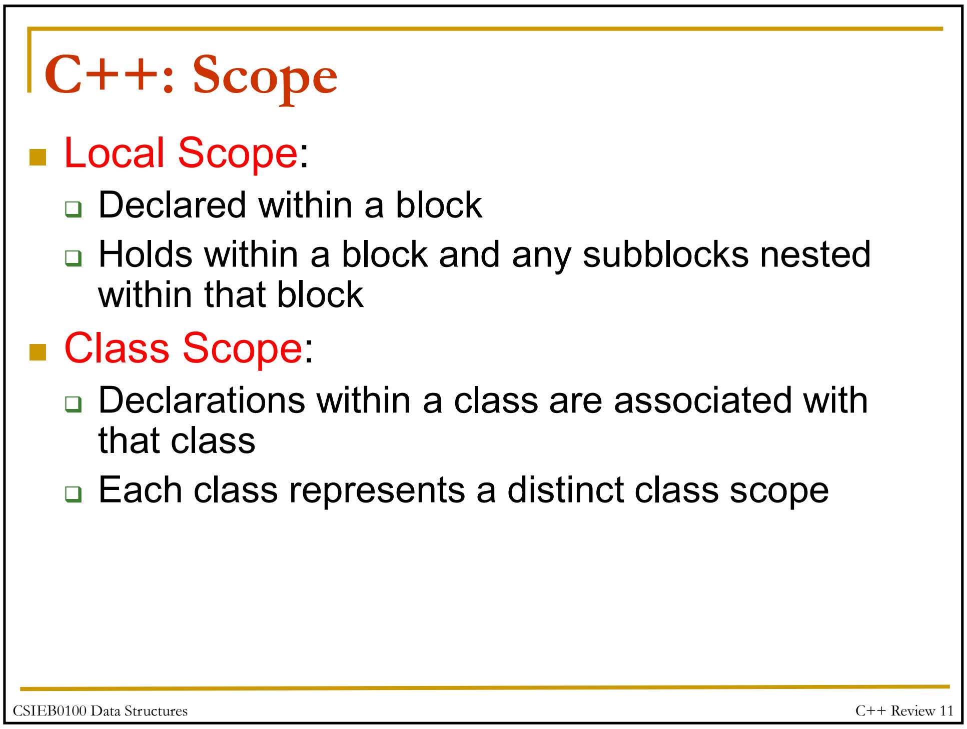
- Multi line: (C style comments)

```
/*
   My first comment line
   Another comment line
   Yet another comment line
*/
```



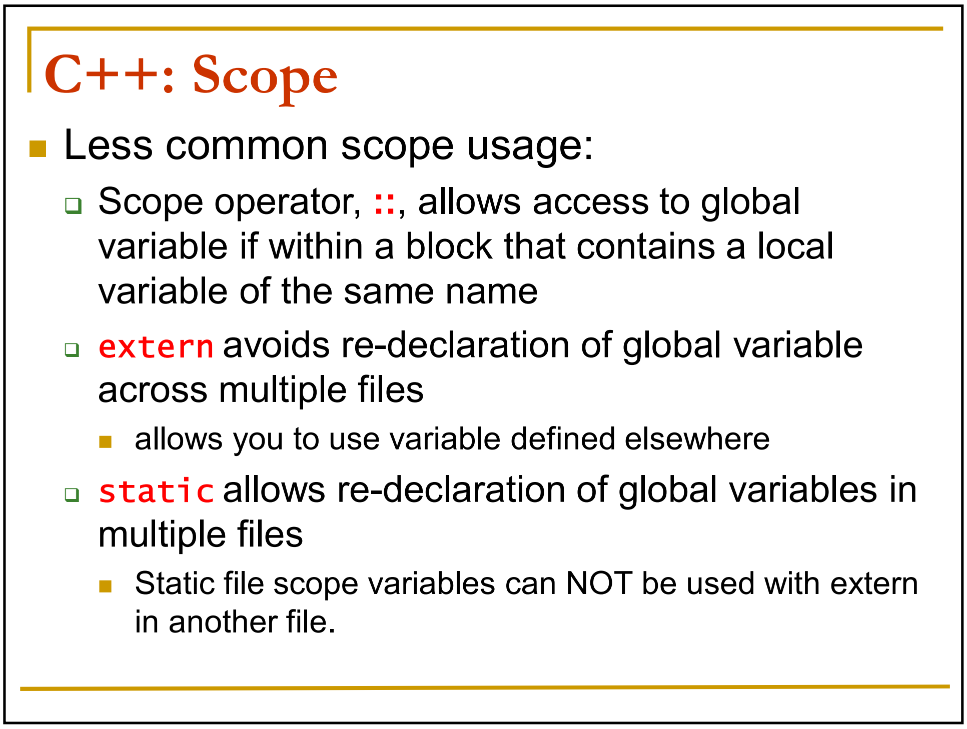
C++: Scope

- File Scope:
  - Any declarations not within a block, function, class, or namespace.
  - Global variables that can be used anywhere in file.
- Namespace Scope:
  - A collection of logically related names (of variables, functions, etc.)
  - Can be accessed with the scope resolution operator (::) such as std::cout
  - With `using namespace` declaration, can omit the scope operator



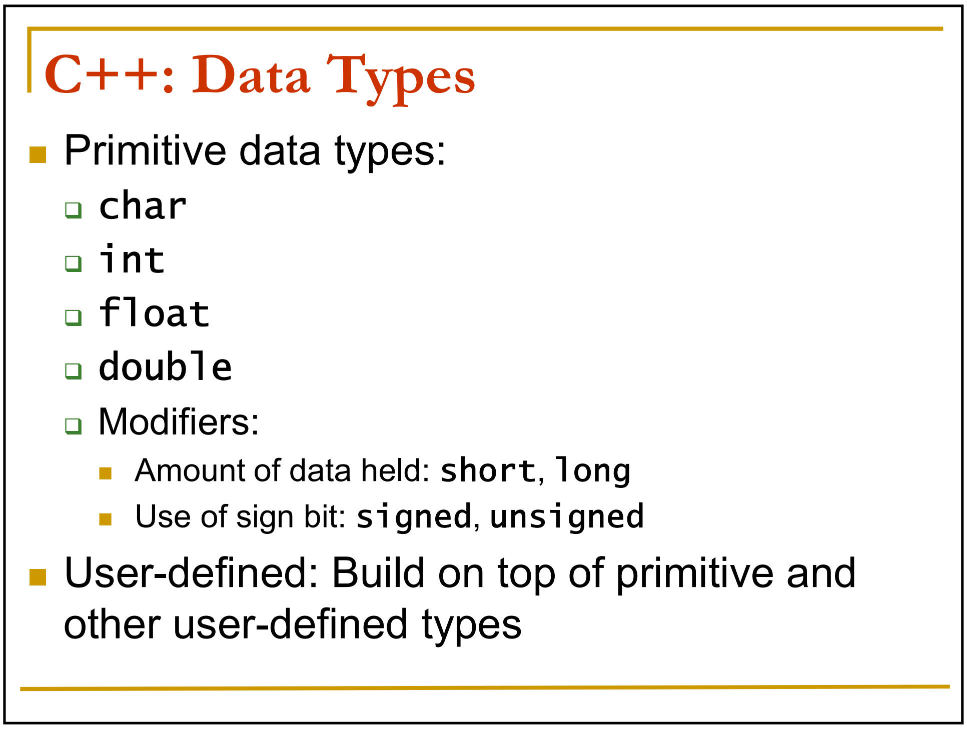
C++: Scope

- Local Scope:
  - Declared within a block
  - Holds within a block and any subblocks nested within that block
- Class Scope:
  - Declarations within a class are associated with that class
  - Each class represents a distinct class scope



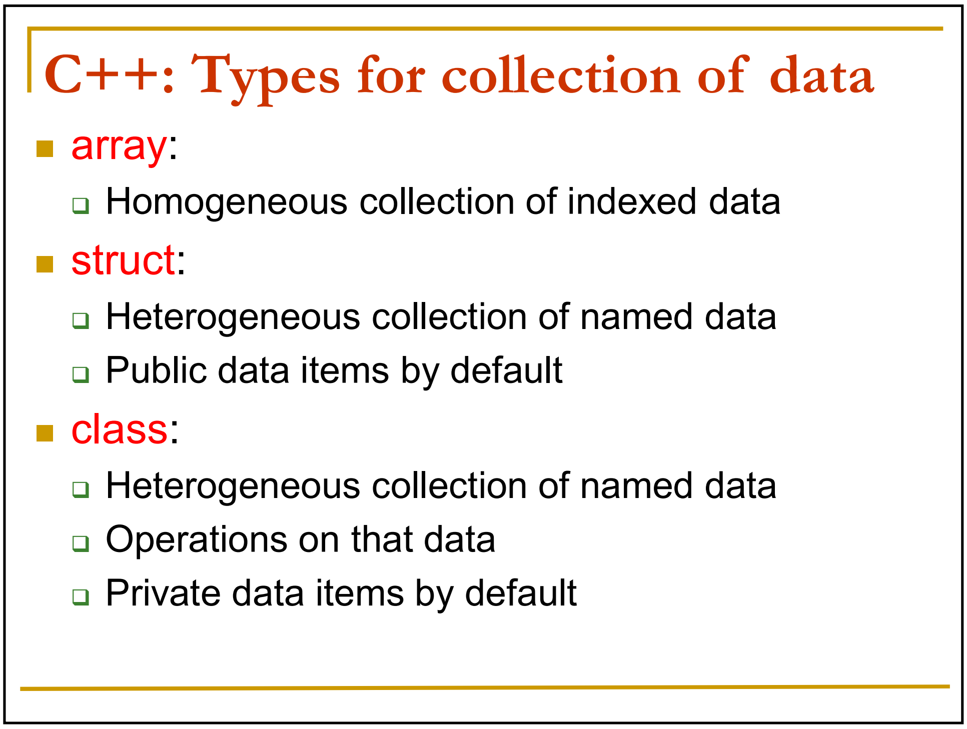
C++: Scope

- Less common scope usage:
  - Scope operator, `::`, allows access to global variable if within a block that contains a local variable of the same name
  - `extern` avoids re-declaration of global variable across multiple files
    - allows you to use variable defined elsewhere
  - `static` allows re-declaration of global variables in multiple files
    - Static file scope variables can NOT be used with extern in another file.



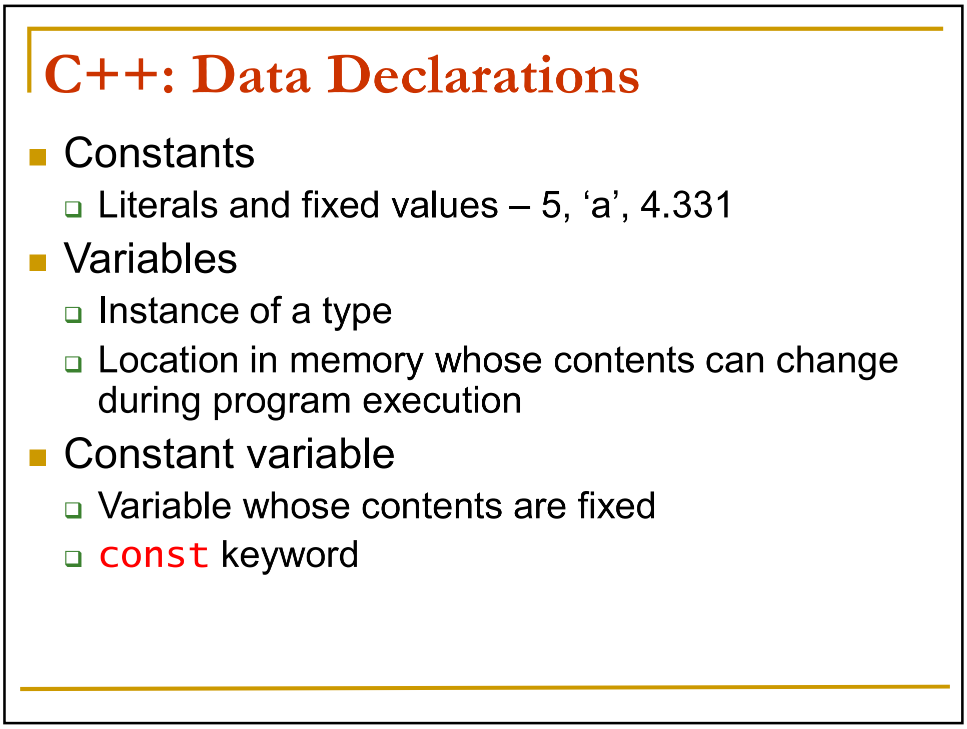
C++: Data Types

- Primitive data types:
  - `char`
  - `int`
  - `float`
  - `double`
  - Modifiers:
    - Amount of data held: `short`, `long`
    - Use of sign bit: `signed`, `unsigned`
- User-defined: Build on top of primitive and other user-defined types



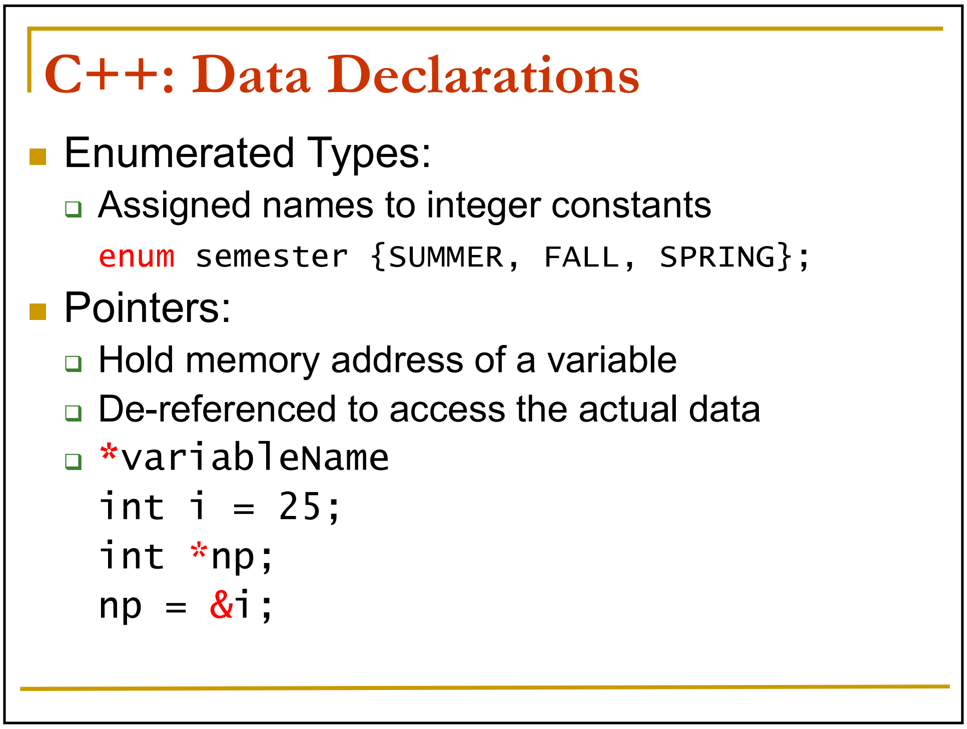
C++: Types for collection of data

- array:
  - Homogeneous collection of indexed data
- struct:
  - Heterogeneous collection of named data
  - Public data items by default
- class:
  - Heterogeneous collection of named data
  - Operations on that data
  - Private data items by default



C++: Data Declarations

- Constants
  - Literals and fixed values – 5, ‘a’, 4.331
- Variables
  - Instance of a type
  - Location in memory whose contents can change during program execution
- Constant variable
  - Variable whose contents are fixed
  - `const` keyword



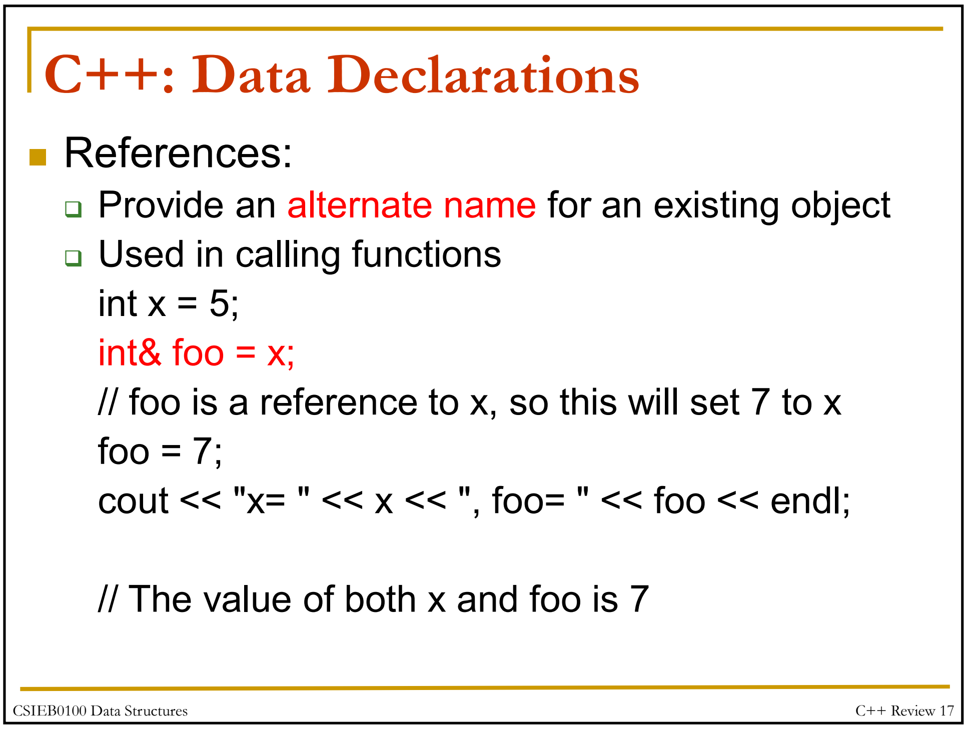
C++: Data Declarations

- Enumerated Types:
  - Assigned names to integer constants

```
enum semester {SUMMER, FALL, SPRING};
```

- Pointers:
  - Hold memory address of a variable
  - De-referenced to access the actual data
  - `*variableName`

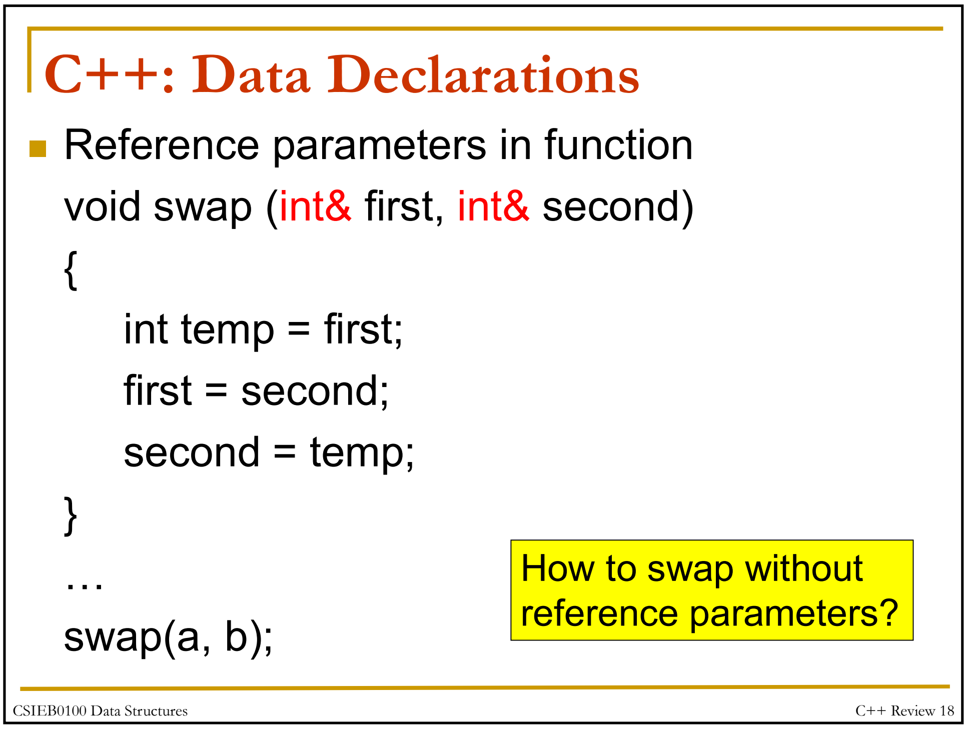
```
int i = 25;
int *np;
np = &i;
```



C++: Data Declarations

- References:
  - Provide an alternate name for an existing object
  - Used in calling functions

```
int x = 5;
int& foo = x;
// foo is a reference to x, so this will set 7 to x
foo = 7;
cout << "x= " << x << ", foo= " << foo << endl;

// The value of both x and foo is 7
```



C++: Data Declarations

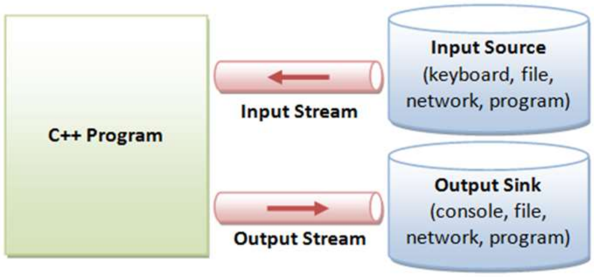
- Reference parameters in function

```
void swap (int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}
…
swap(a, b);
```

How to swap without reference parameters?

C++ Stream I/O Concept

- C++ I/O are based on streams (sequence of bytes flowing in and out of the programs).


C++ Program
Input Stream
Output Stream
Input Source
(keyboard, file, network, program)
Output Sink
(console, file, network, program)

Internal Data Formats:

- Text: char, wchar_t
- int, float, double, etc.

External Data Formats:

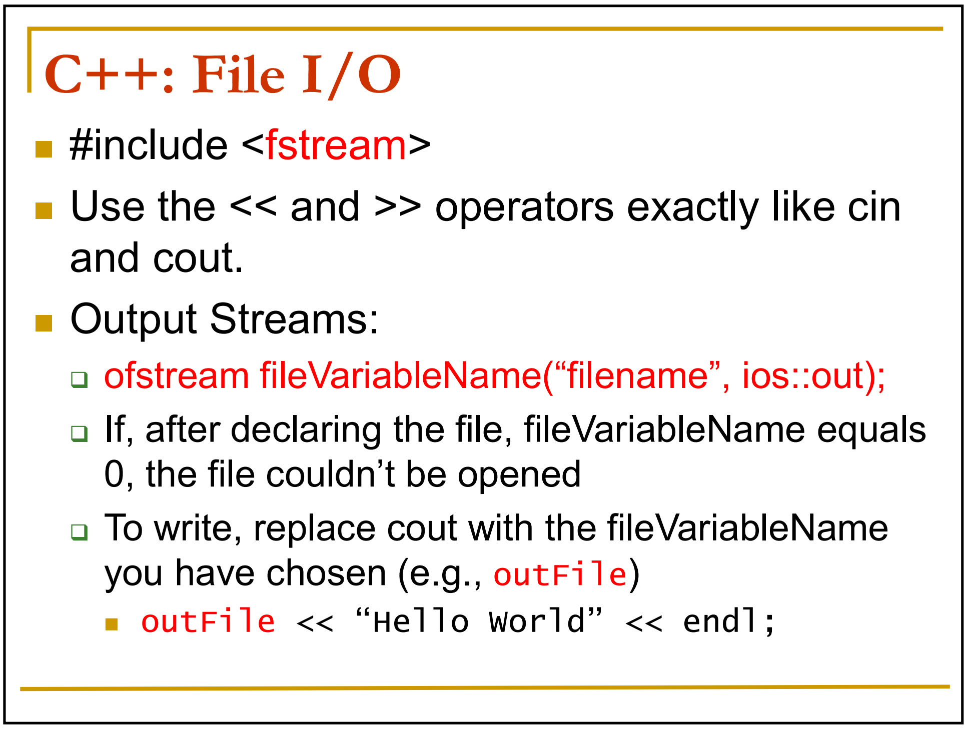
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

C++: Standard I/O

- #include <iostream>

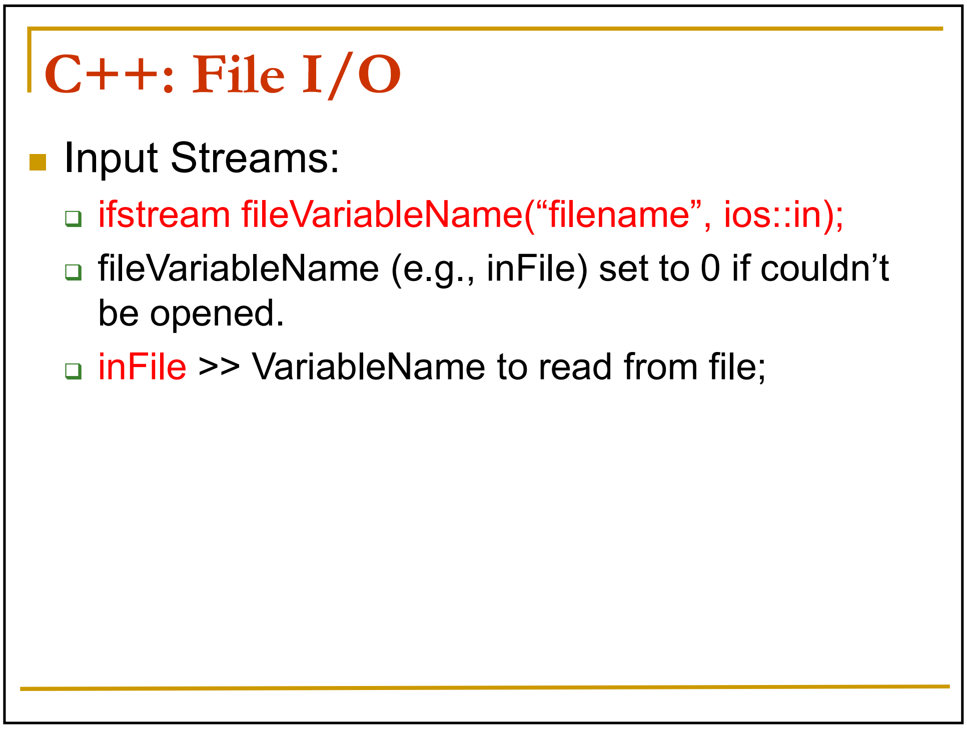
- Writing to screen:
  - cout << variable << “string literal” << endl;

- Reading from keyboard:
  - cin >> variable;



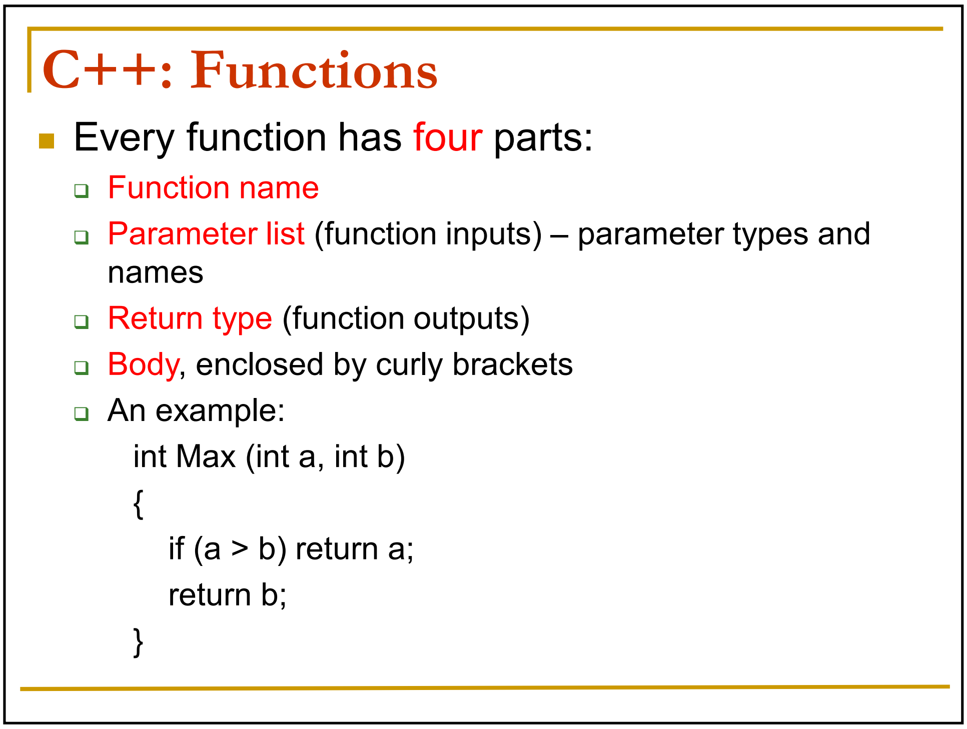
C++: File I/O

- #include <fstream>
- Use the << and >> operators exactly like cin and cout.
- Output Streams:
  - ofstream fileVariableName(“filename”, ios::out);
  - If, after declaring the file, fileVariableName equals 0, the file couldn’t be opened
  - To write, replace cout with the fileVariableName you have chosen (e.g., `outFile`)
    - `outFile << “Hello World” << endl;`



C++: File I/O

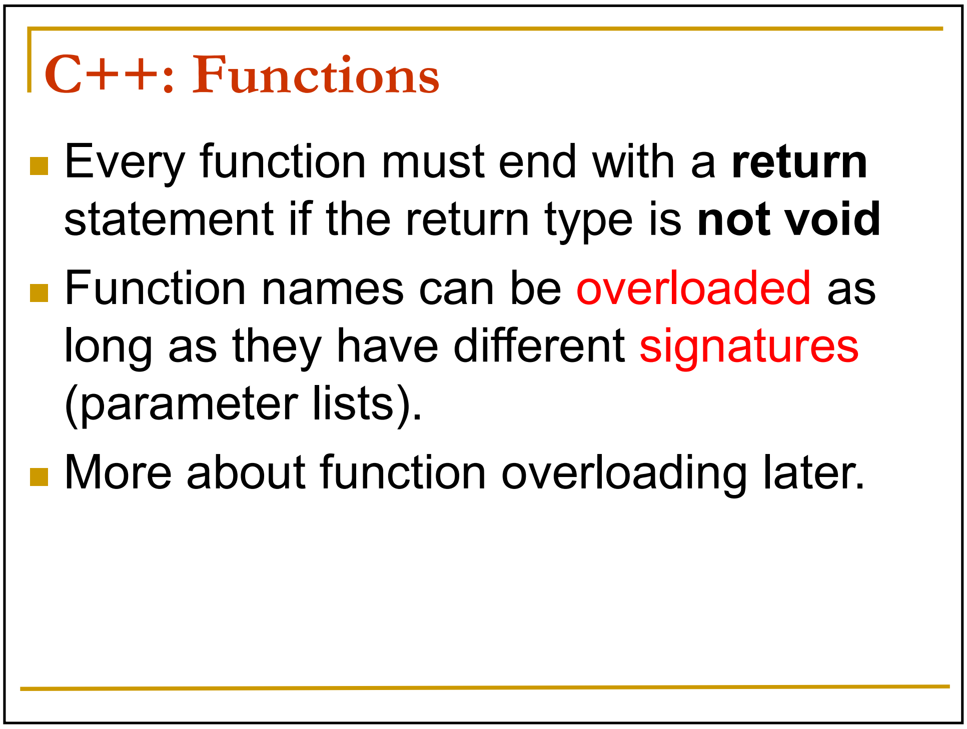
- Input Streams:
  - ifstream fileVariableName(“filename”, ios::in);
  - fileVariableName (e.g., inFile) set to 0 if couldn’t be opened.
  - inFile >> VariableName to read from file;



C++: Functions

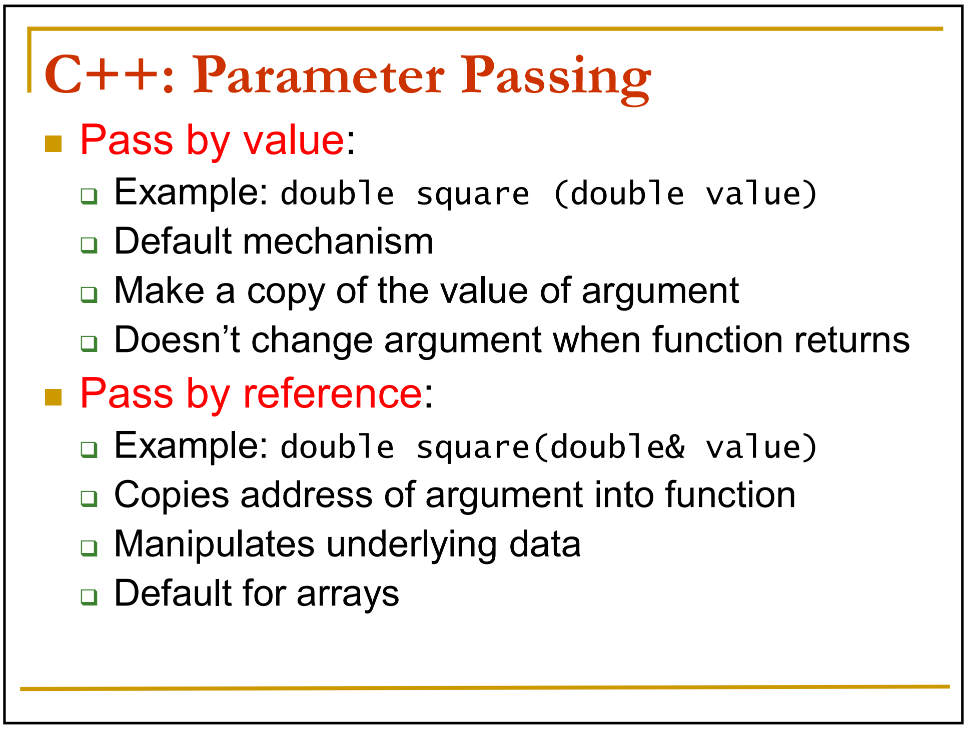
- Every function has four parts:
  - Function name
  - Parameter list (function inputs) – parameter types and names
  - Return type (function outputs)
  - Body, enclosed by curly brackets
  - An example:

```
int Max (int a, int b)
{
    if (a > b) return a;
    return b;
}
```



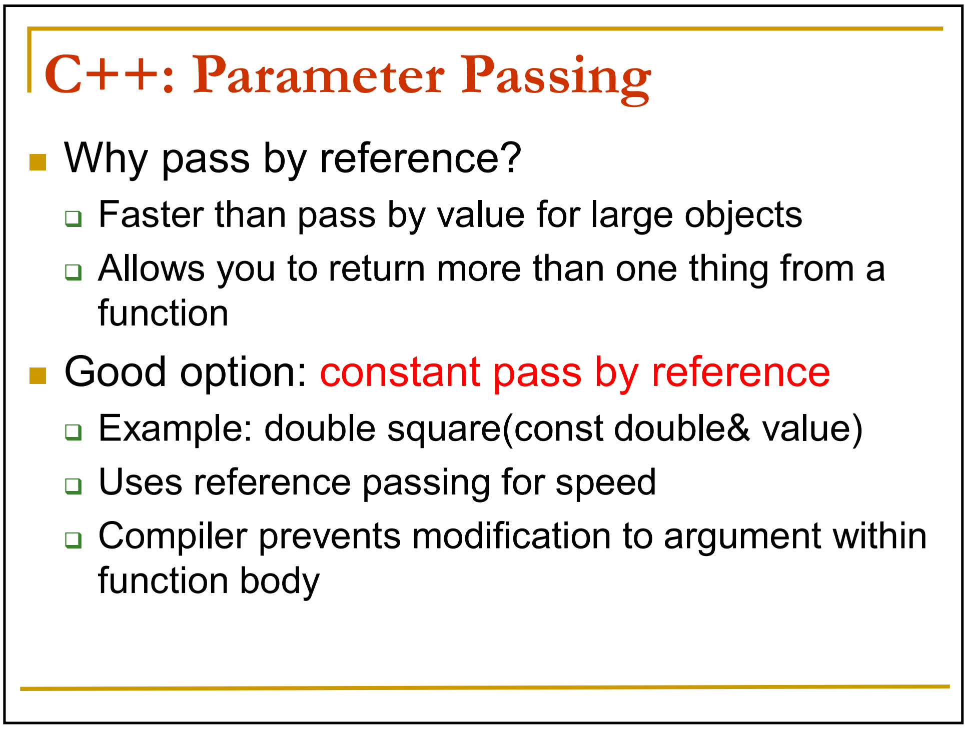
C++: Functions

- Every function must end with a **return** statement if the return type is **not void**
- Function names can be overloaded as long as they have different signatures (parameter lists).
- More about function overloading later.



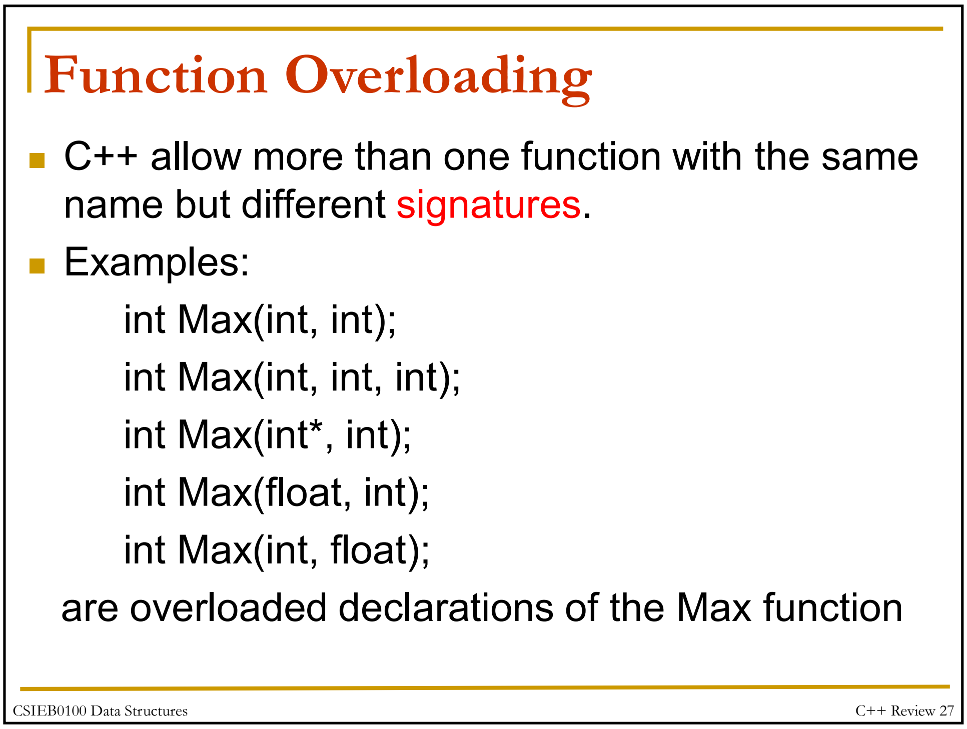
C++: Parameter Passing

- Pass by value:
  - Example: `double square (double value)`
  - Default mechanism
  - Make a copy of the value of argument
  - Doesn't change argument when function returns
- Pass by reference:
  - Example: `double square(double& value)`
  - Copies address of argument into function
  - Manipulates underlying data
  - Default for arrays



C++: Parameter Passing

- Why pass by reference?
  - Faster than pass by value for large objects
  - Allows you to return more than one thing from a function
- Good option: constant pass by reference
  - Example: double square(const double& value)
  - Uses reference passing for speed
  - Compiler prevents modification to argument within function body

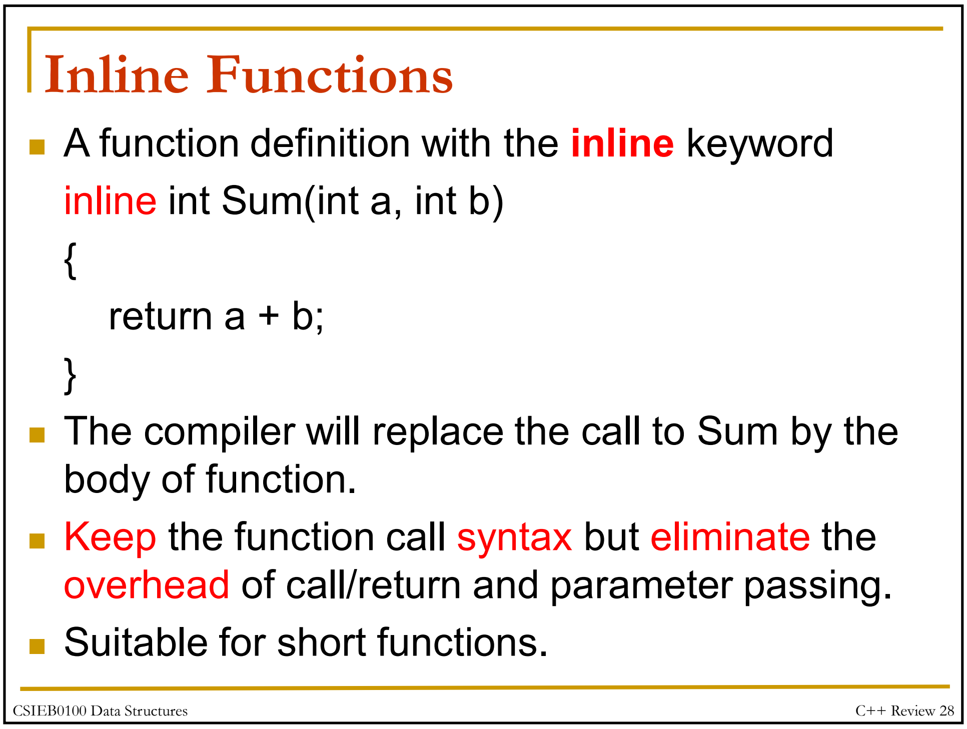


Function Overloading

- C++ allow more than one function with the same name but different signatures.
- Examples:

```
    int Max(int, int);
    int Max(int, int, int);
    int Max(int*, int);
    int Max(float, int);
    int Max(int, float);
```

are overloaded declarations of the Max function

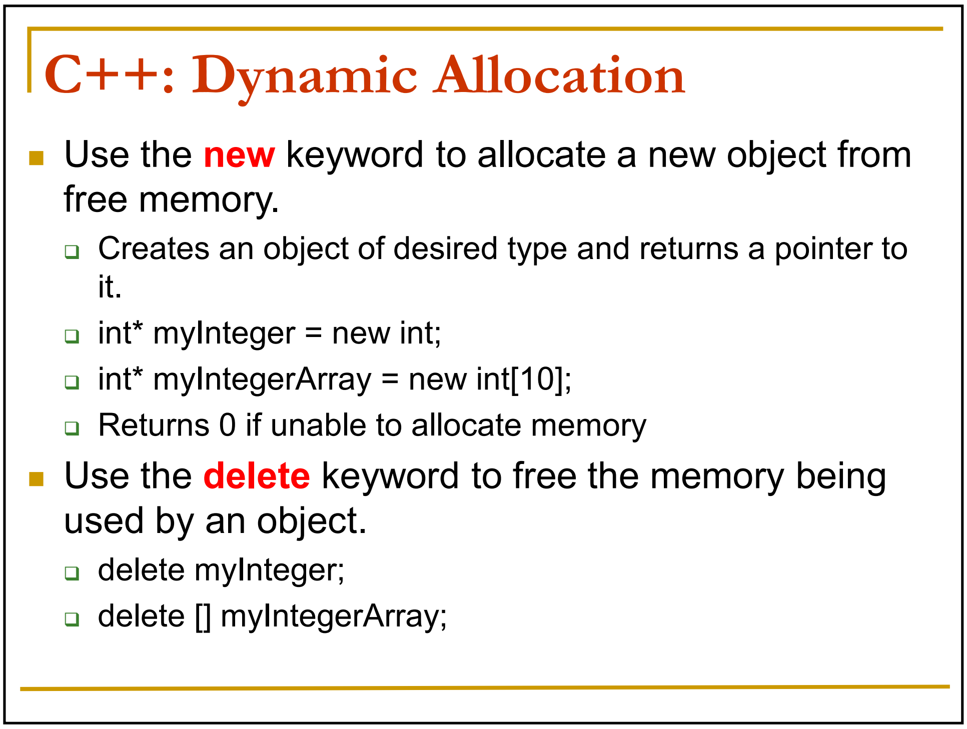


Inline Functions

- A function definition with the **inline** keyword

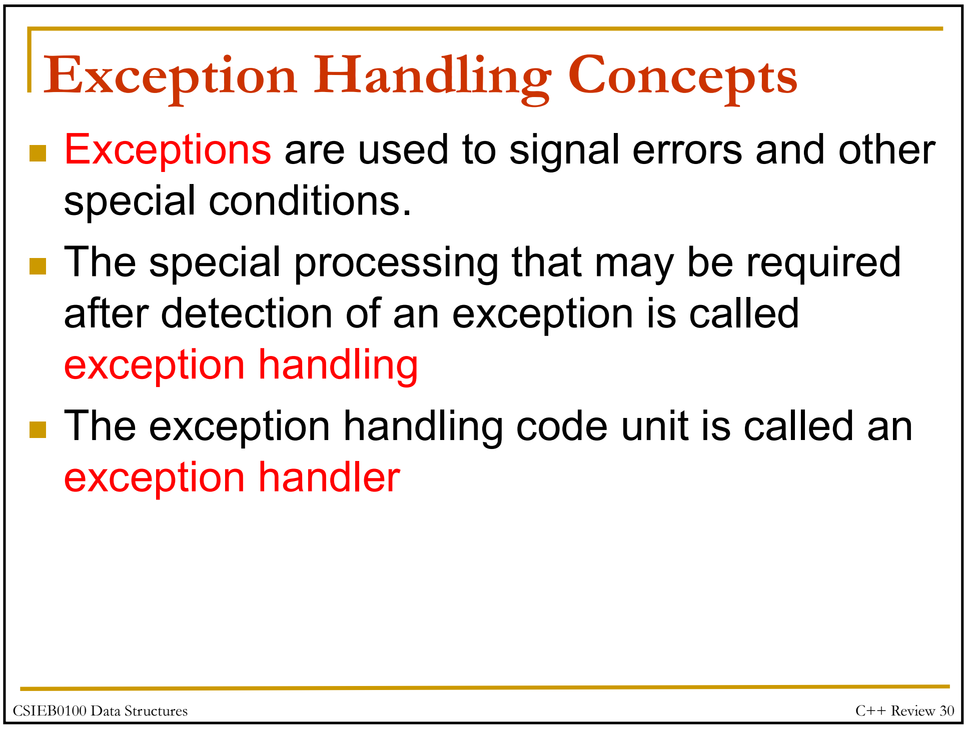
```
inline int Sum(int a, int b)
{
   return a + b;
}
```

- The compiler will replace the call to Sum by the body of function.
- Keep the function call syntax but eliminate the overhead of call/return and parameter passing.
- Suitable for short functions.



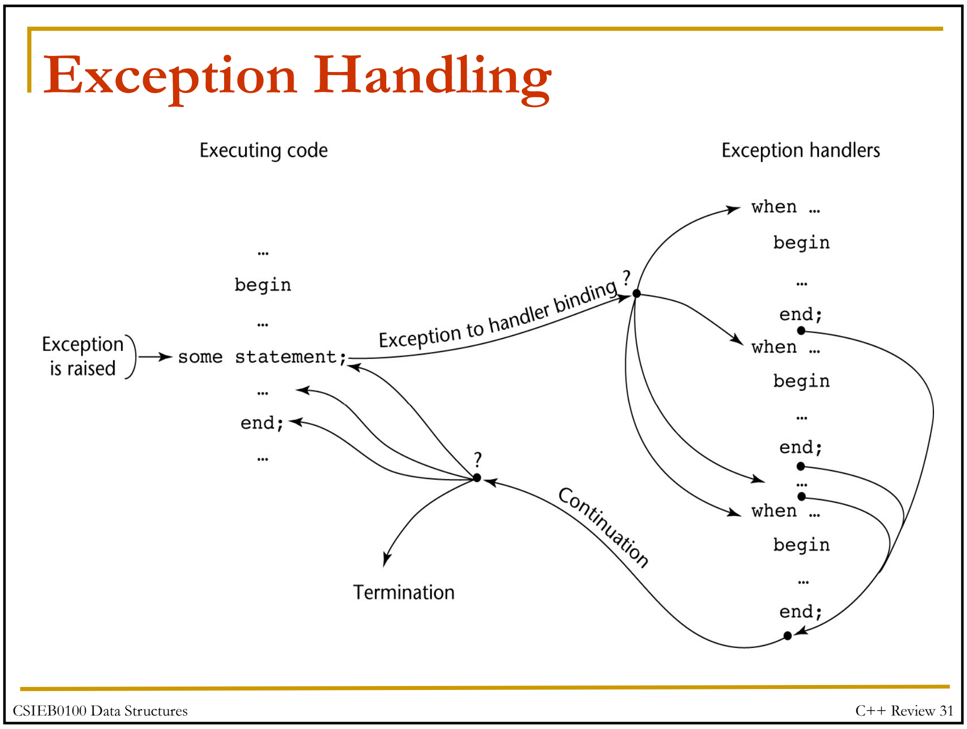
C++: Dynamic Allocation

- Use the **new** keyword to allocate a new object from free memory.
  - Creates an object of desired type and returns a pointer to it.
  - int* myInteger = new int;
  - int* myIntegerArray = new int[10];
  - Returns 0 if unable to allocate memory
- Use the **delete** keyword to free the memory being used by an object.
  - delete myInteger;
  - delete [] myIntegerArray;

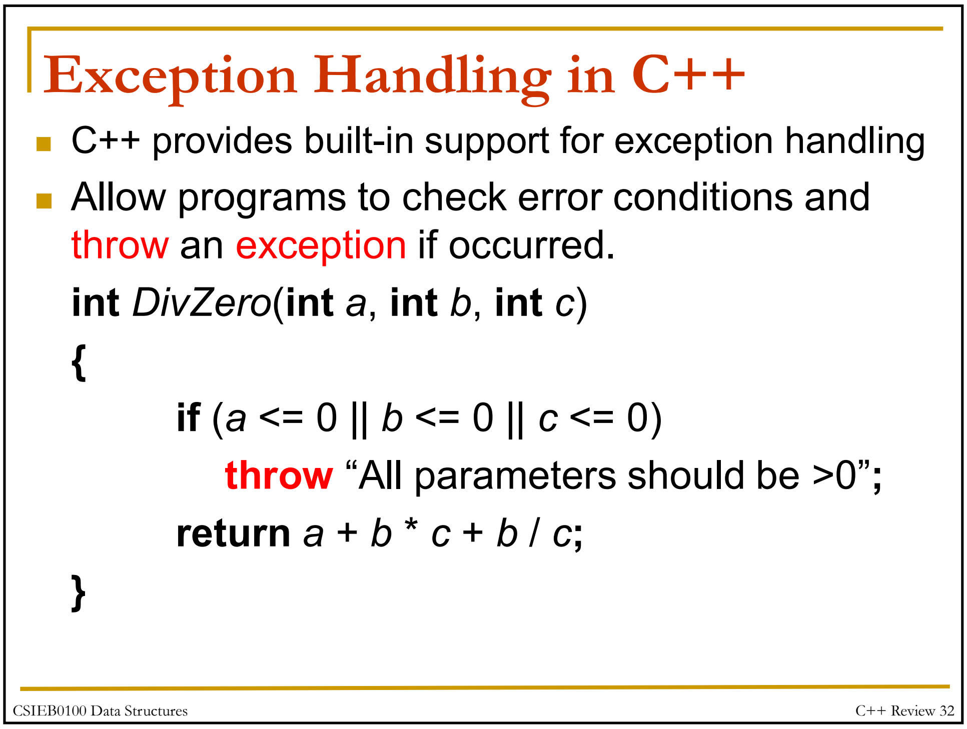


Exception Handling Concepts

- Exceptions are used to signal errors and other special conditions.
- The special processing that may be required after detection of an exception is called exception handling
- The exception handling code unit is called an exception handler



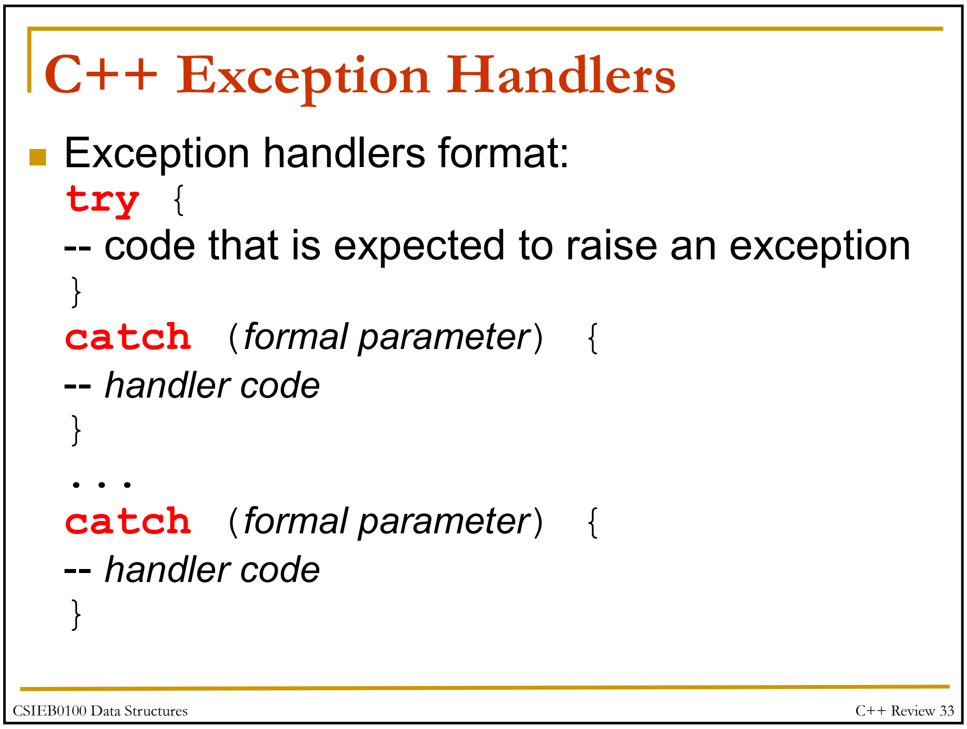
Exception Handling



Exception Handling in C++

- C++ provides built-in support for exception handling
- Allow programs to check error conditions and throw an exception if occurred.

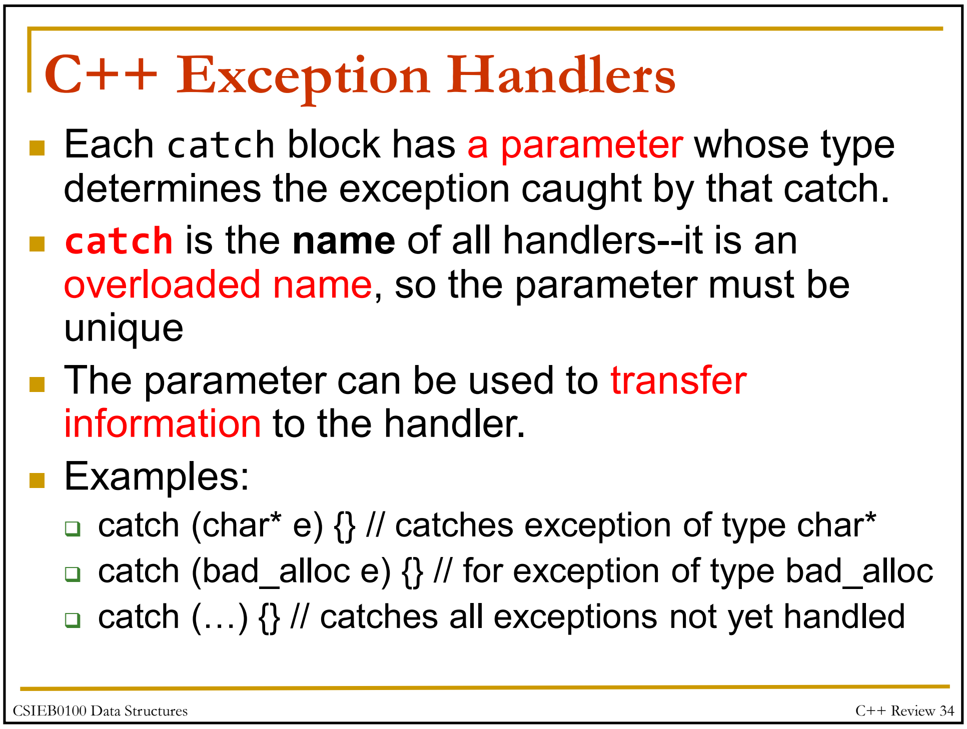
```
int DivZero(int a, int b, int c)
{
        if (a <= 0 || b <= 0 || c <= 0)
            throw "All parameters should be >0";
        return a + b * c + b / c;
}
```



C++ Exception Handlers

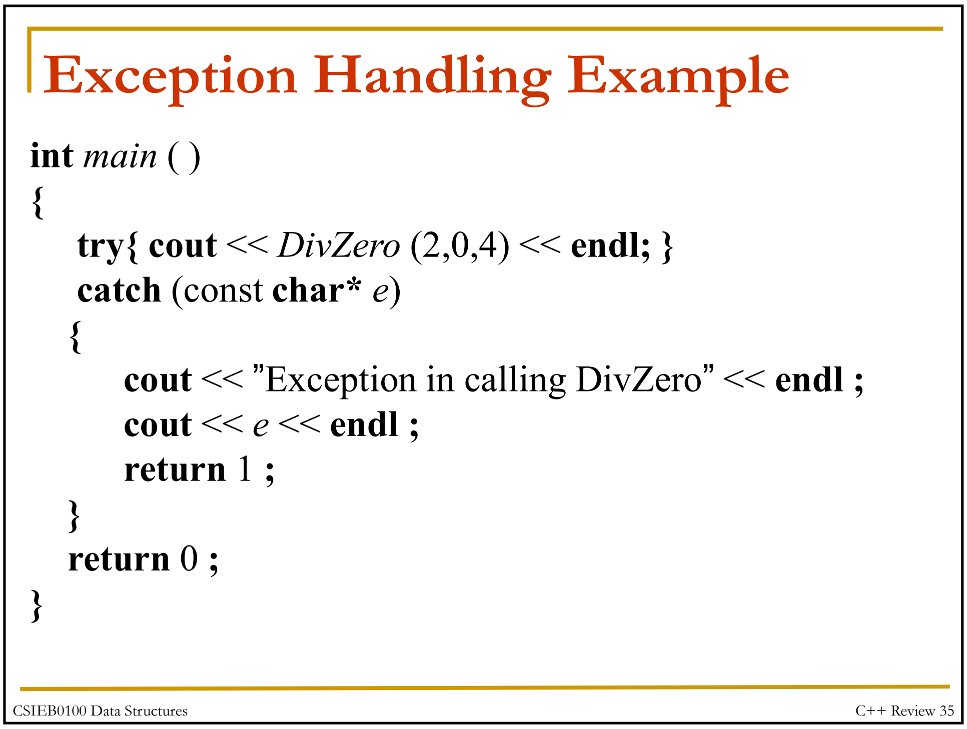
- Exception handlers format:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
. . .
catch (formal parameter) {
-- handler code
}
```



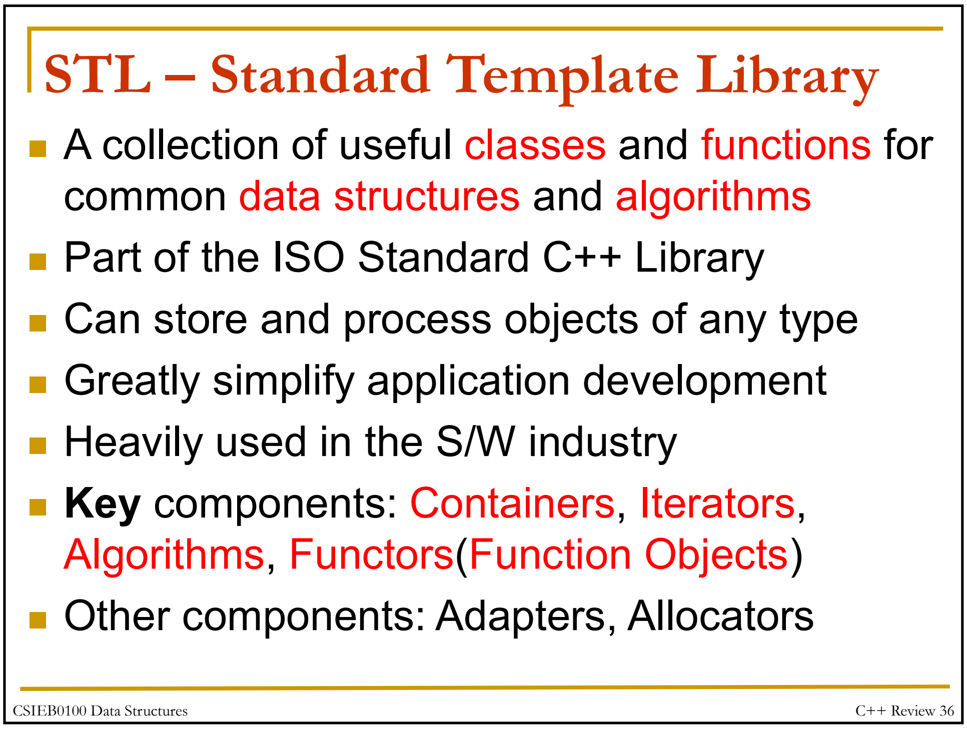
C++ Exception Handlers

- Each `catch` block has a parameter whose type determines the exception caught by that catch.
- `catch` is the **name** of all handlers--it is an overloaded name, so the parameter must be unique
- The parameter can be used to transfer information to the handler.
- Examples:
  - catch (char* e) {} // catches exception of type char*
  - catch (bad_alloc e) {} // for exception of type bad_alloc
  - catch (…) {} // catches all exceptions not yet handled



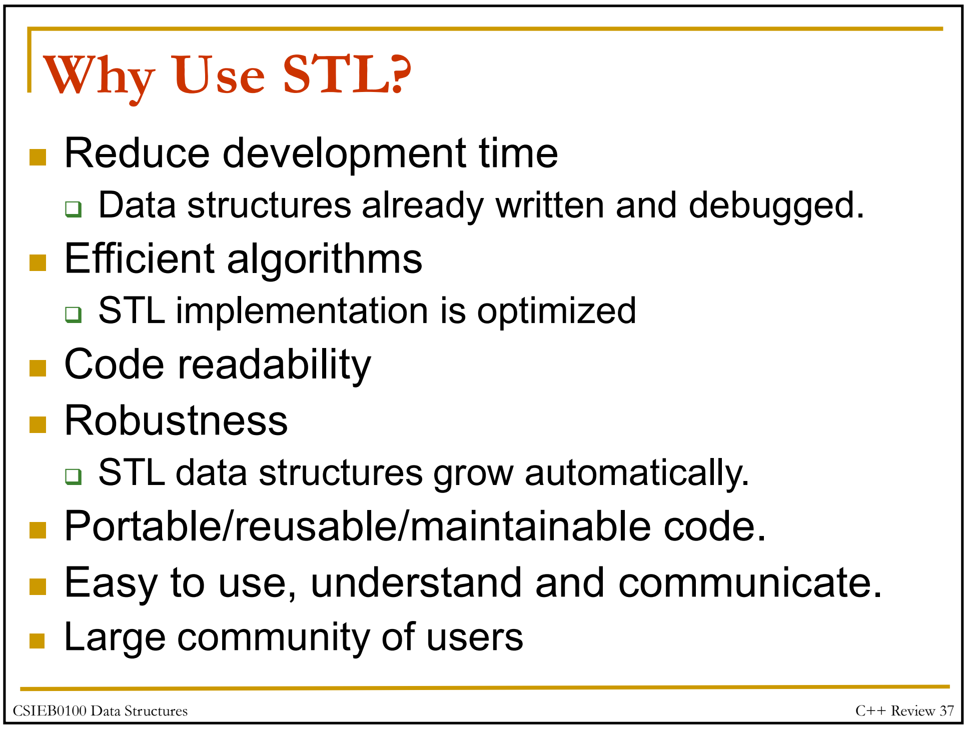
Exception Handling Example

```
int main ( )
{
    try{ cout << DivZero (2,0,4) << endl; }
    catch (const char* e)
   {
        cout << "Exception in calling DivZero" << endl ;
        cout << e << endl ;
        return 1 ;
   }
   return 0 ;
}
```



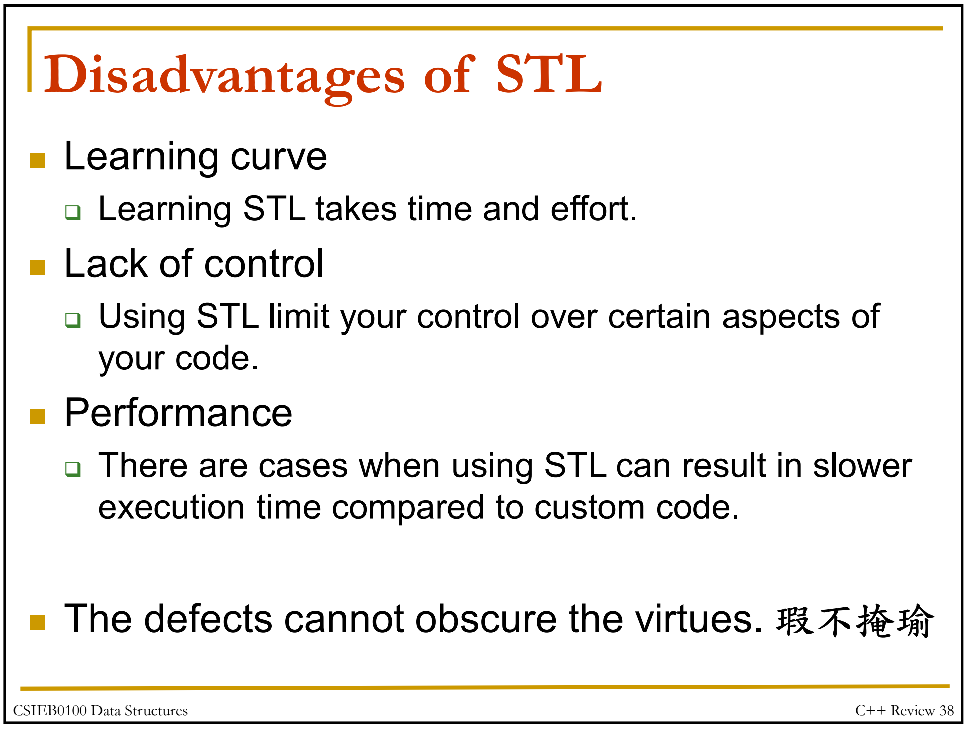
STL – Standard Template Library

- A collection of useful classes and functions for common data structures and algorithms
- Part of the ISO Standard C++ Library
- Can store and process objects of any type
- Greatly simplify application development
- Heavily used in the S/W industry
- **Key** components: Containers, Iterators, Algorithms, Functors(Function Objects)
- Other components: Adapters, Allocators



Why Use STL?

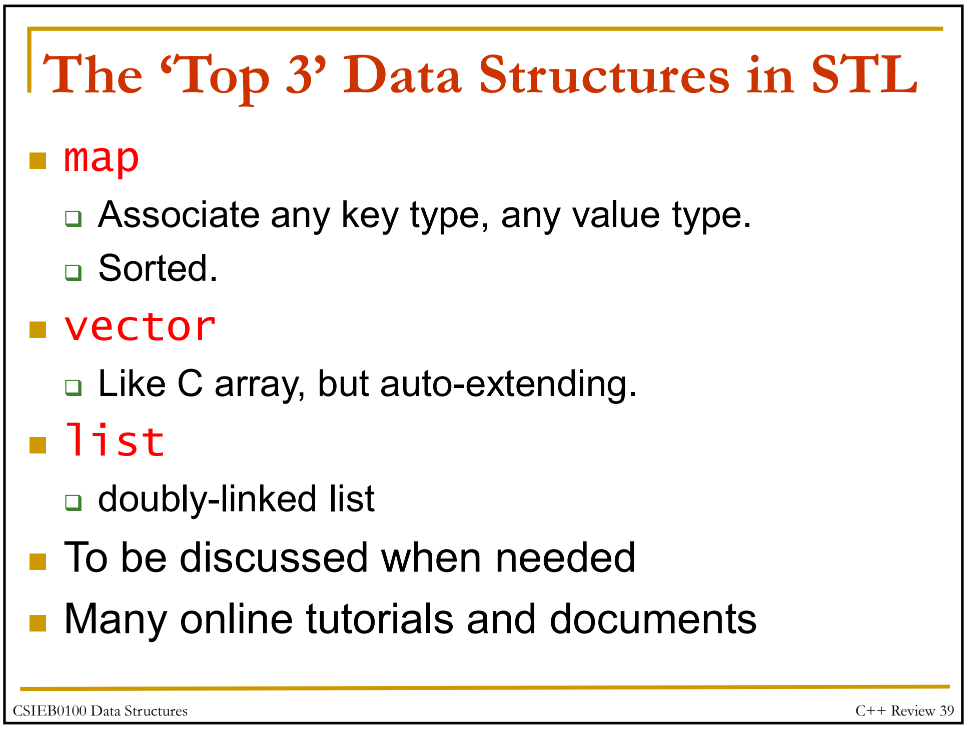
- Reduce development time
  - Data structures already written and debugged.
- Efficient algorithms
  - STL implementation is optimized
- Code readability
- Robustness
  - STL data structures grow automatically.
- Portable/reusable/maintainable code.
- Easy to use, understand and communicate.
- Large community of users



Disadvantages of STL

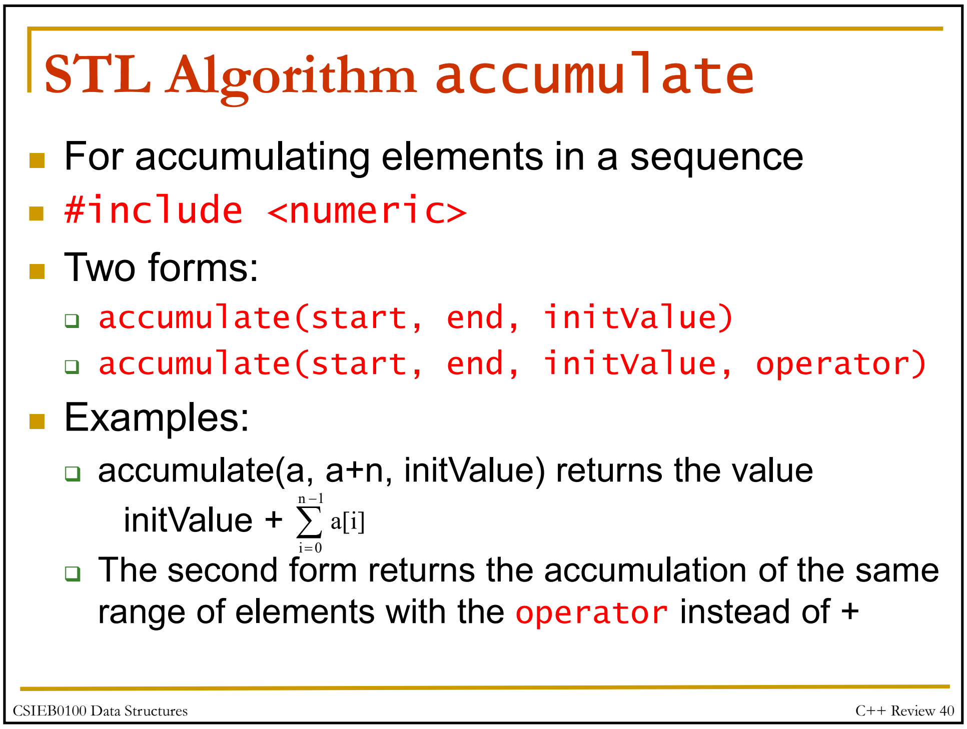
- Learning curve
  - Learning STL takes time and effort.
- Lack of control
  - Using STL limit your control over certain aspects of your code.
- Performance
  - There are cases when using STL can result in slower execution time compared to custom code.

- The defects cannot obscure the virtues. 瑕不掩瑜



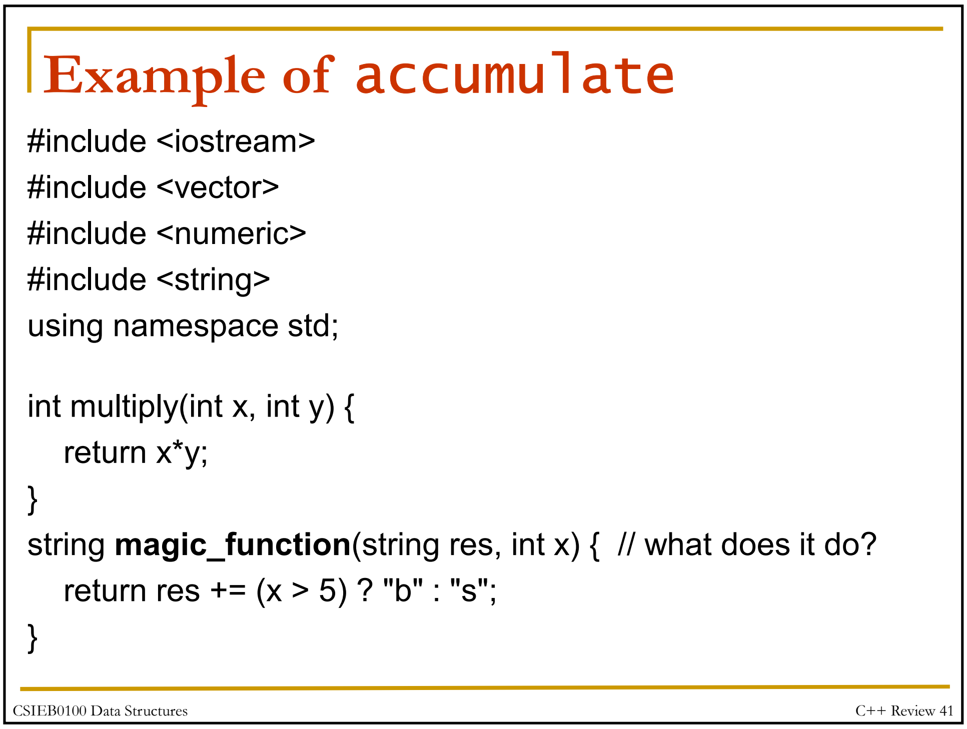
The ‘Top 3’ Data Structures in STL

- `map`
  - Associate any key type, any value type.
  - Sorted.
- `vector`
  - Like C array, but auto-extending.
- `list`
  - doubly-linked list
- To be discussed when needed
- Many online tutorials and documents



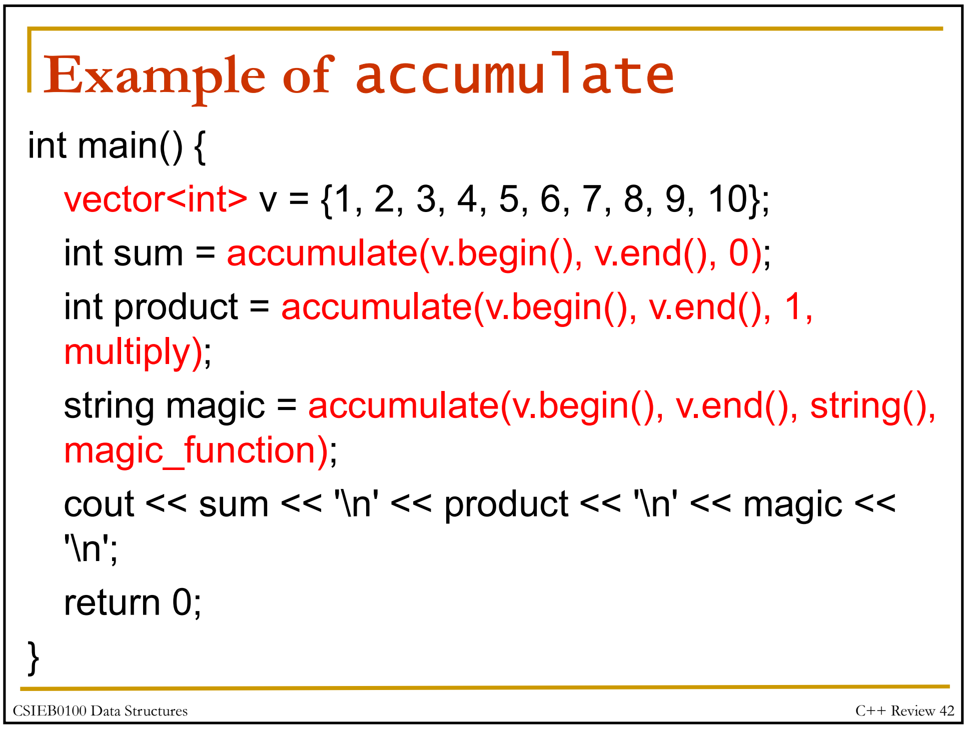
STL Algorithm accumulate

- For accumulating elements in a sequence
- `#include <numeric>`
- Two forms:
  - `accumulate(start, end, initValue)`
  - `accumulate(start, end, initValue, operator)`
- Examples:
  - accumulate(a, a+n, initValue) returns the value
    initValue + $\sum_{i=0}^{n-1} a[i]$
  - The second form returns the accumulation of the same range of elements with the `operator` instead of +



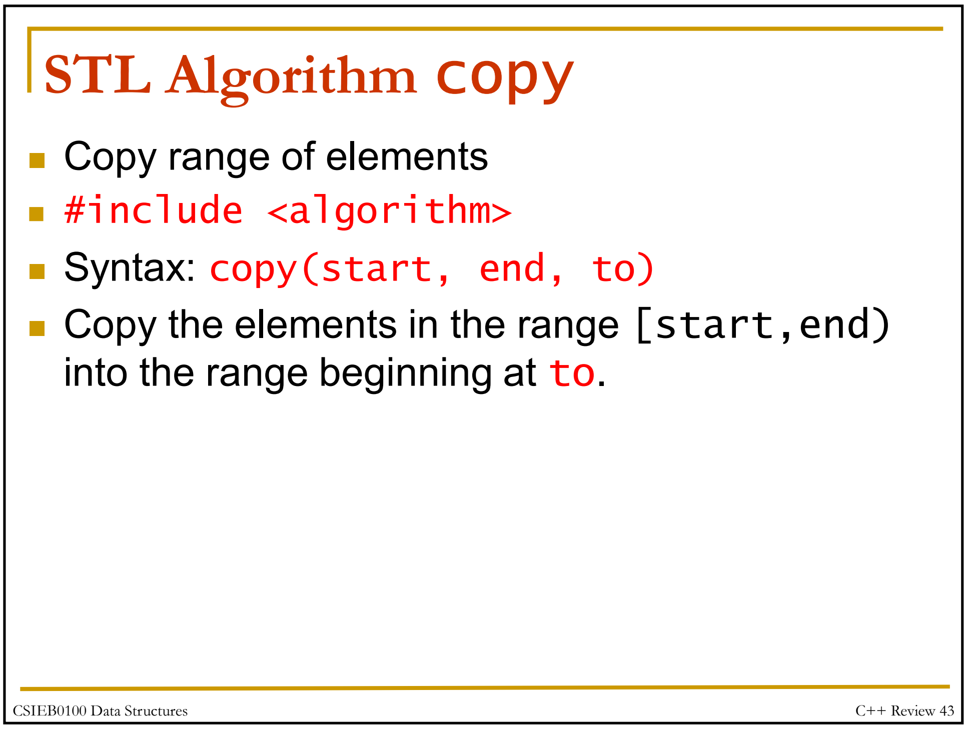
Example of accumulate

```
#include <iostream>
#include <vector>
#include <numeric>
#include <string>
using namespace std;

int multiply(int x, int y) {
    return x*y;
}
string magic_function(string res, int x) {  // what does it do?
    return res += (x > 5) ? "b" : "s";
}
```



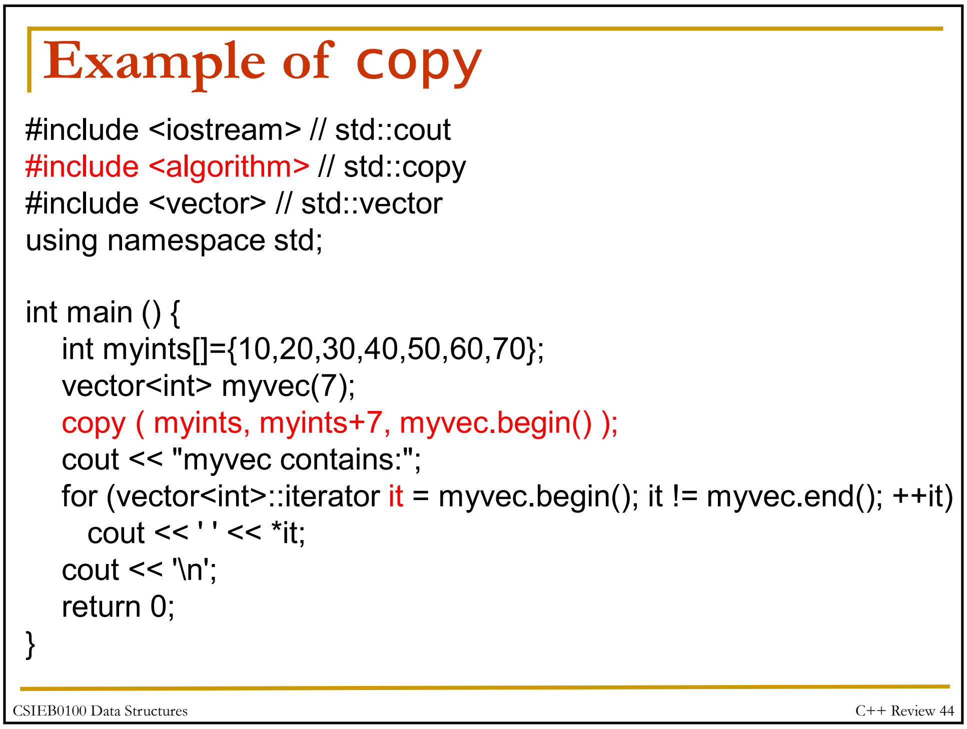
Example of accumulate

```
int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = accumulate(v.begin(), v.end(), 0);
    int product = accumulate(v.begin(), v.end(), 1,
    multiply);
    string magic = accumulate(v.begin(), v.end(), string(),
    magic_function);
    cout << sum << '\n' << product << '\n' << magic <<
    '\n';
    return 0;
}
```



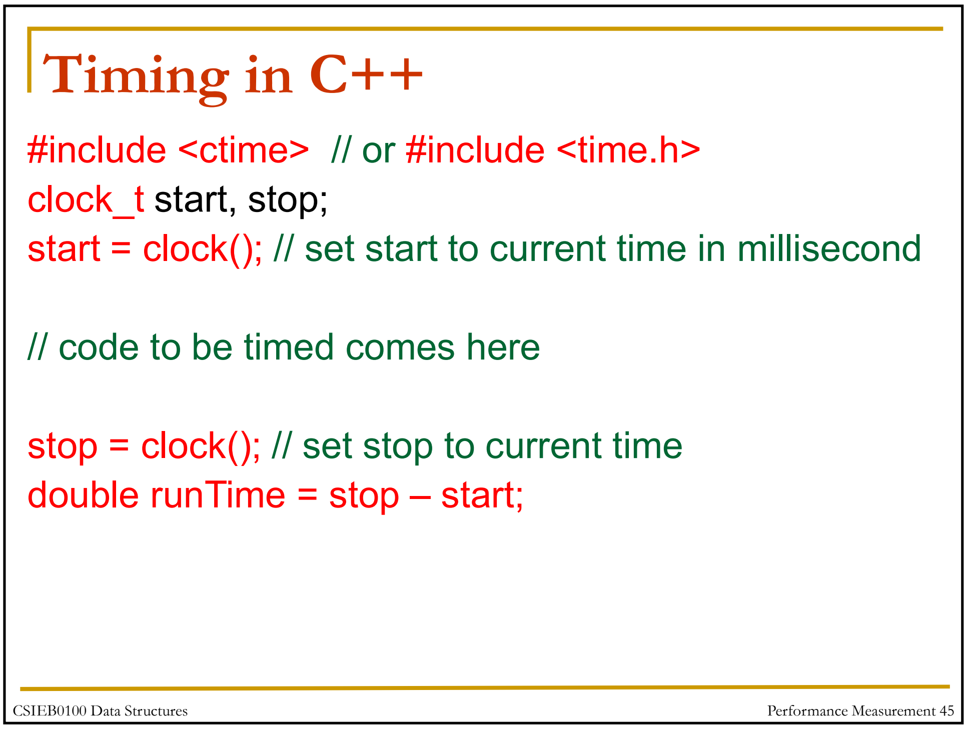
STL Algorithm copy

- Copy range of elements
- #include <algorithm>
- Syntax: copy(start, end, to)
- Copy the elements in the range [start,end) into the range beginning at to.



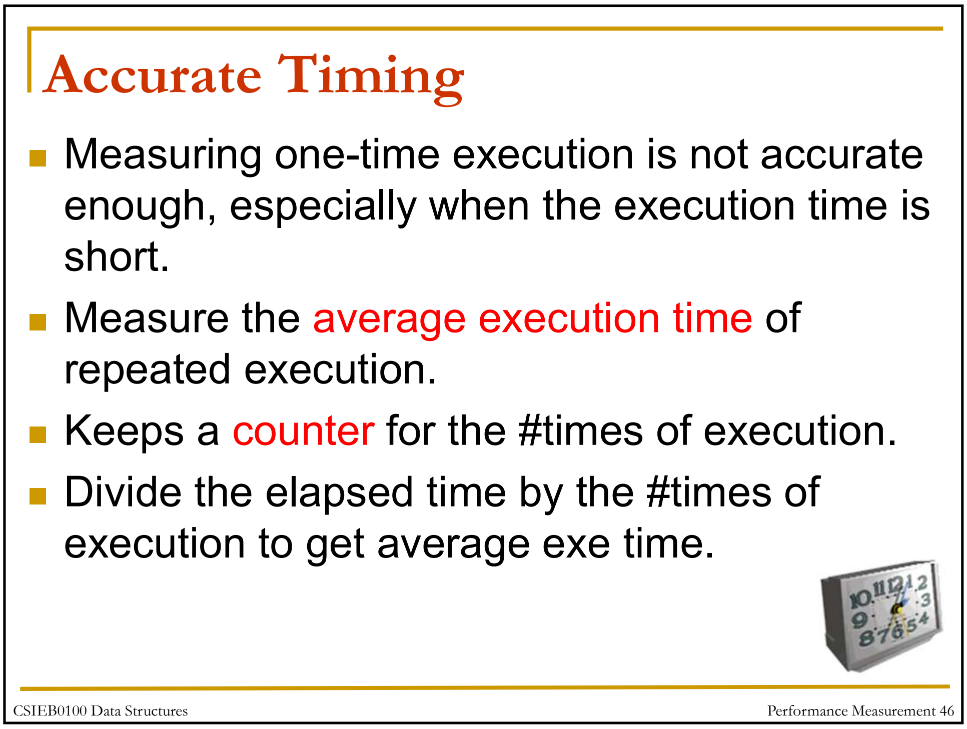
Example of copy

```
#include <iostream> // std::cout
#include <algorithm> // std::copy
#include <vector> // std::vector
using namespace std;

int main () {
    int myints[]={10,20,30,40,50,60,70};
    vector<int> myvec(7);
    copy ( myints, myints+7, myvec.begin() );
    cout << "myvec contains:";
    for (vector<int>::iterator it = myvec.begin(); it != myvec.end(); ++it)
       cout << ' ' << *it;
    cout << '\n';
    return 0;
}
```



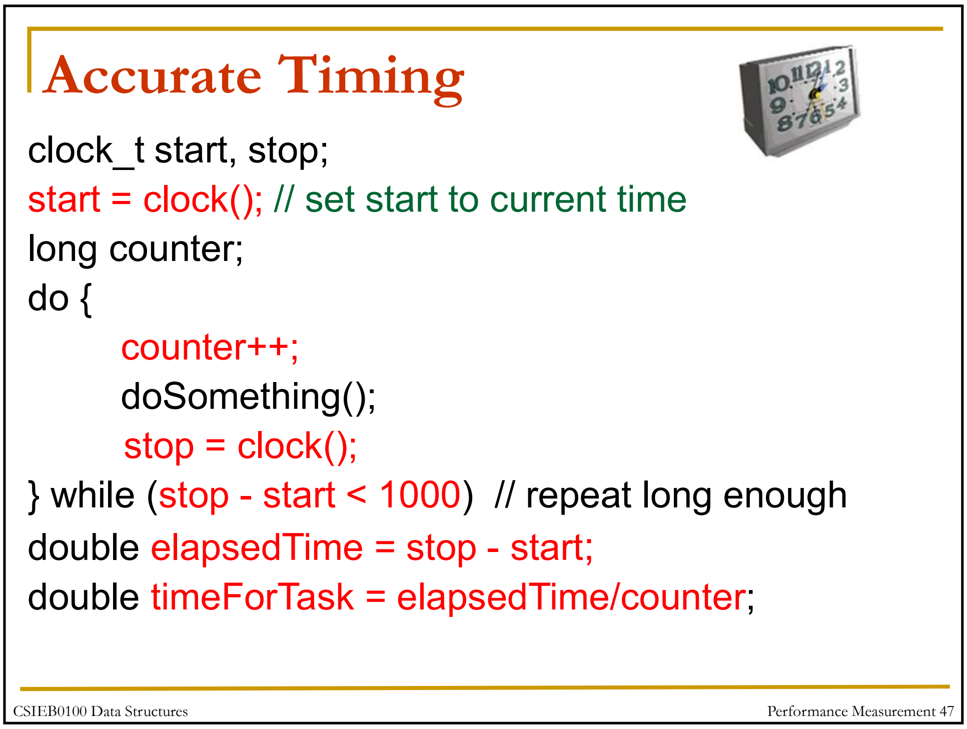
Timing in C++

```
#include <ctime>  // or #include <time.h>
clock_t start, stop;
start = clock(); // set start to current time in millisecond

// code to be timed comes here

stop = clock(); // set stop to current time
double runTime = stop – start;
```



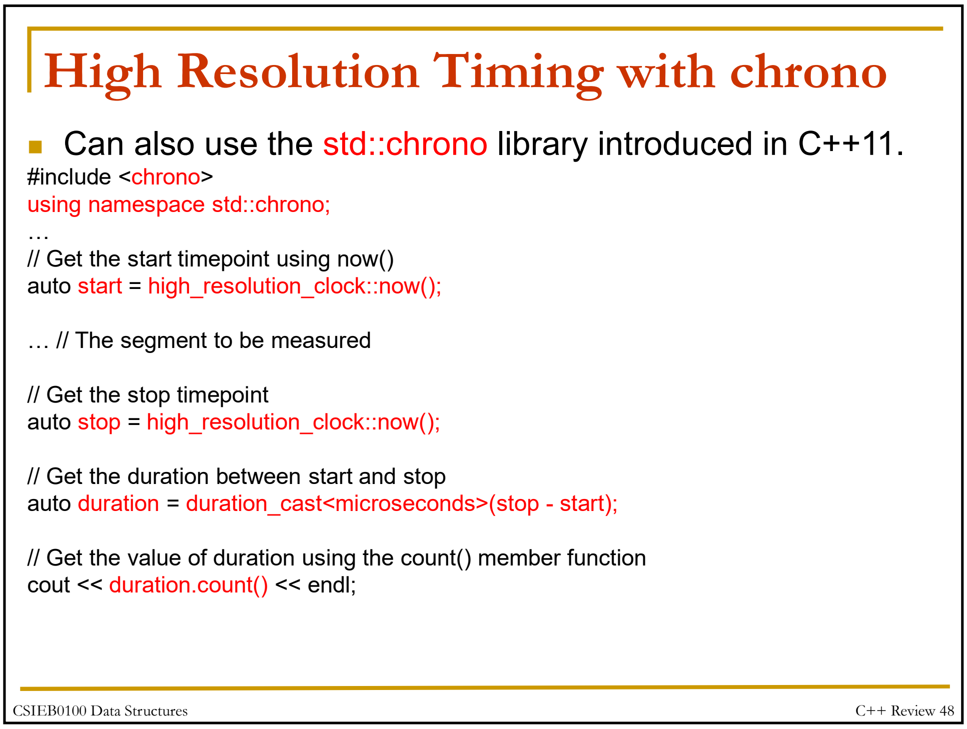
Accurate Timing

- Measuring one-time execution is not accurate enough, especially when the execution time is short.
- Measure the average execution time of repeated execution.
- Keeps a counter for the #times of execution.
- Divide the elapsed time by the #times of execution to get average exe time.



Accurate Timing

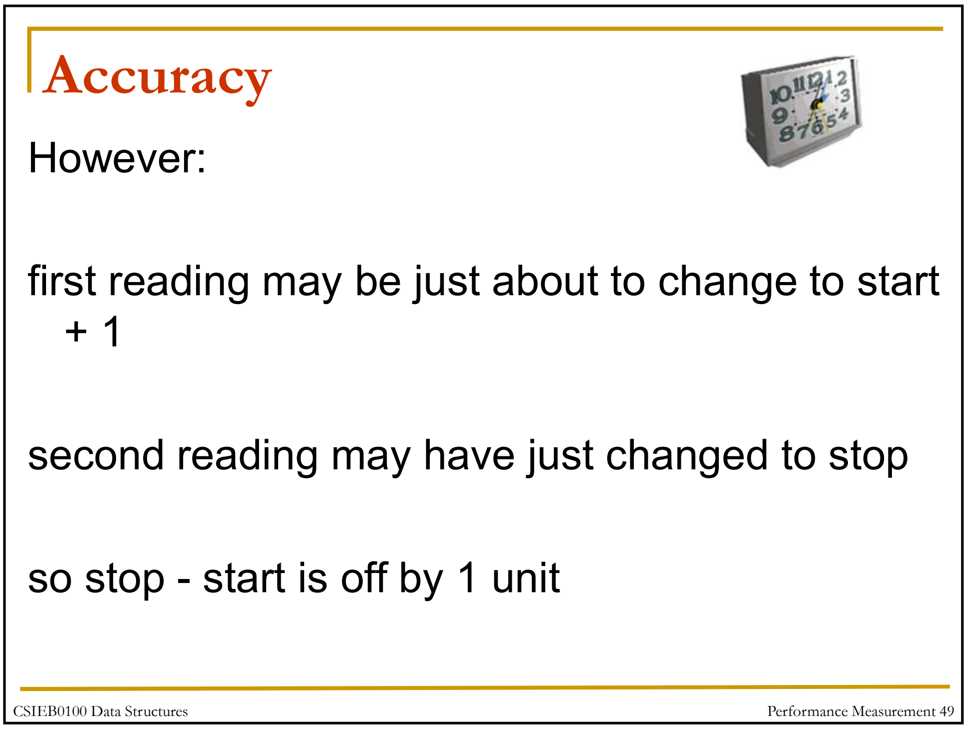
```
clock_t start, stop;
start = clock(); // set start to current time
long counter;
do {
        counter++;
        doSomething();
        stop = clock();
} while (stop - start < 1000)  // repeat long enough
double elapsedTime = stop - start;
double timeForTask = elapsedTime/counter;
```



High Resolution Timing with chrono

- Can also use the std::chrono library introduced in C++11.

```
#include <chrono>
using namespace std::chrono;
…
// Get the start timepoint using now()
auto start = high_resolution_clock::now();

… // The segment to be measured

// Get the stop timepoint
auto stop = high_resolution_clock::now();

// Get the duration between start and stop
auto duration = duration_cast<microseconds>(stop - start);

// Get the value of duration using the count() member function
cout << duration.count() << endl;
```



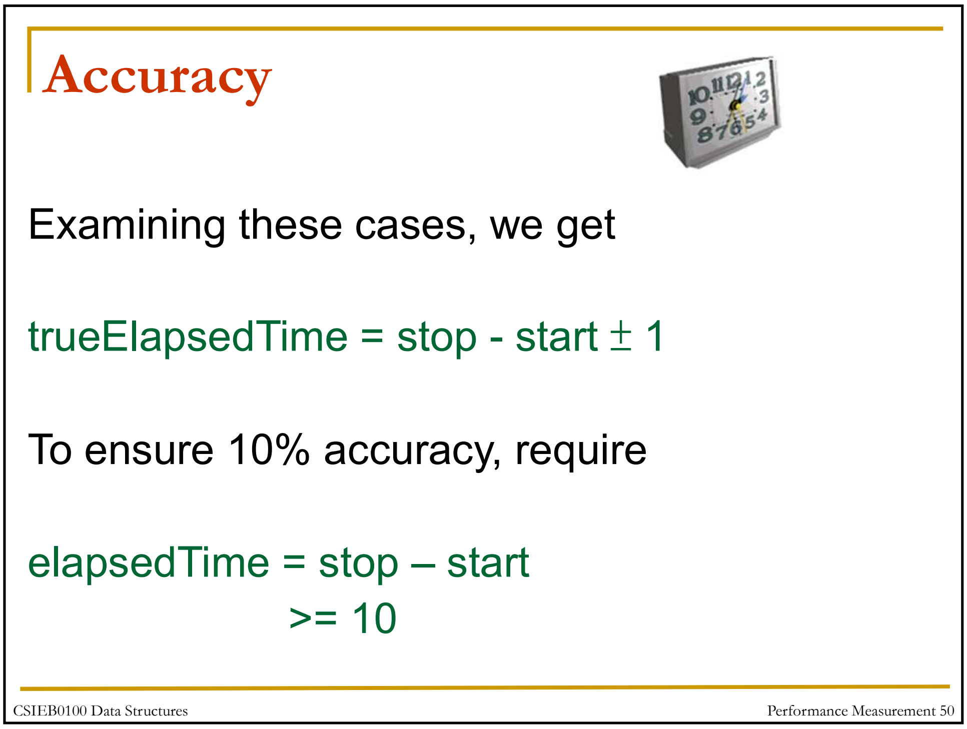
Accuracy

However:

first reading may be just about to change to start + 1

second reading may have just changed to stop

so stop - start is off by 1 unit



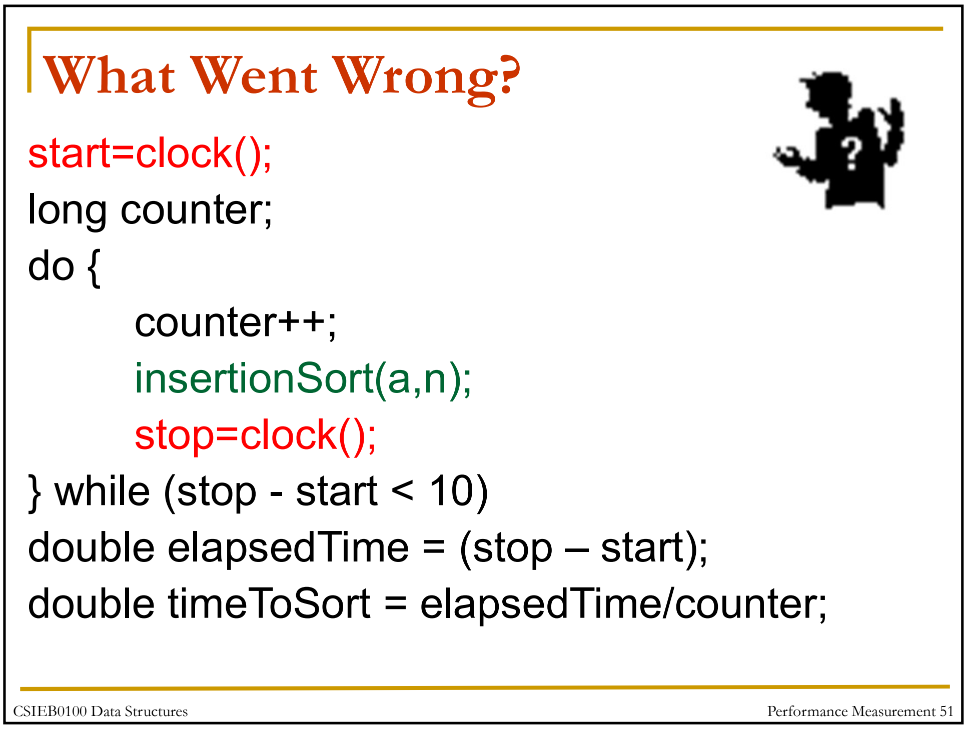
Accuracy

Examining these cases, we get

trueElapsedTime = stop - start ± 1

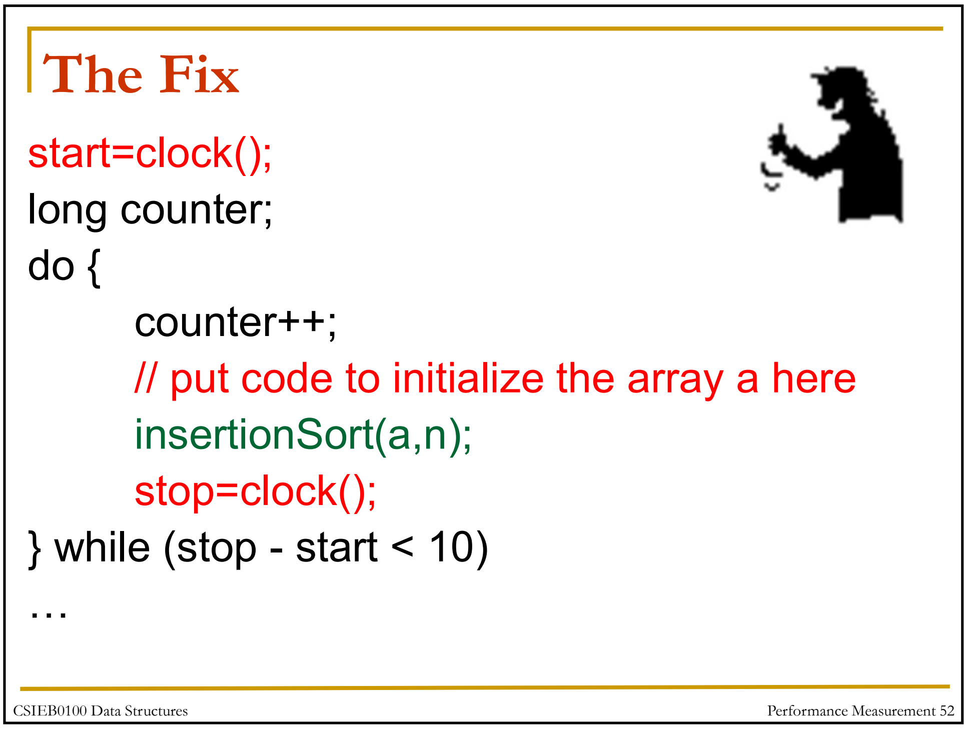
To ensure 10% accuracy, require

elapsedTime = stop – start
>= 10



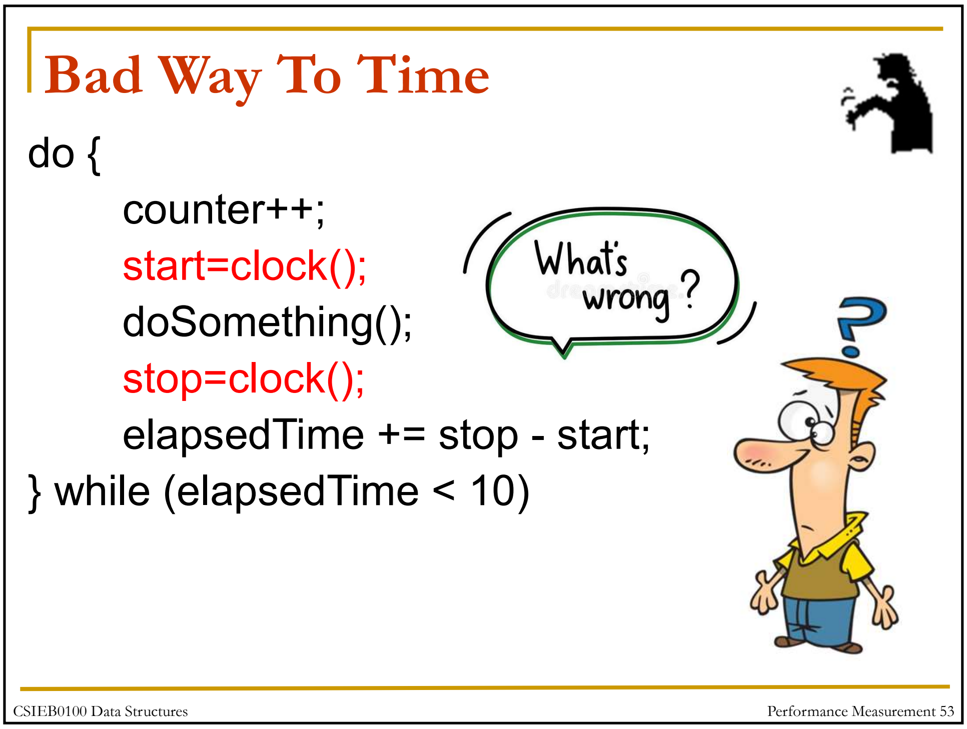
What Went Wrong?

```
start=clock();
long counter;
do {
        counter++;
        insertionSort(a,n);
        stop=clock();
} while (stop - start < 10)
double elapsedTime = (stop – start);
double timeToSort = elapsedTime/counter;
```



The Fix

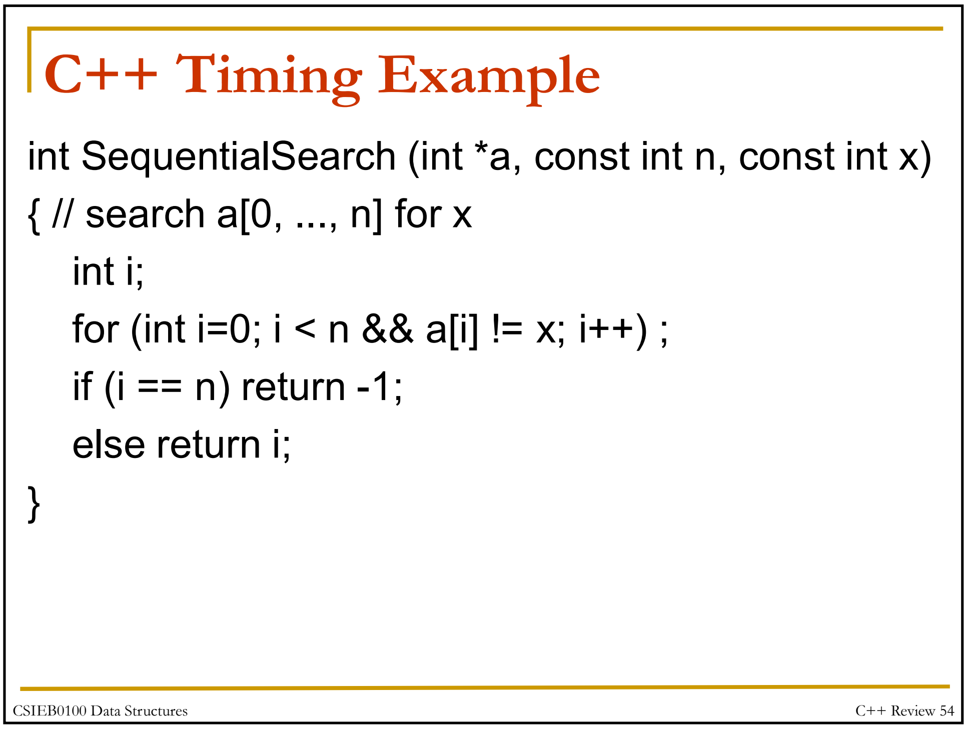
```
start=clock();
long counter;
do {
        counter++;
        // put code to initialize the array a here
        insertionSort(a,n);
        stop=clock();
} while (stop - start < 10)
…
```



Bad Way To Time

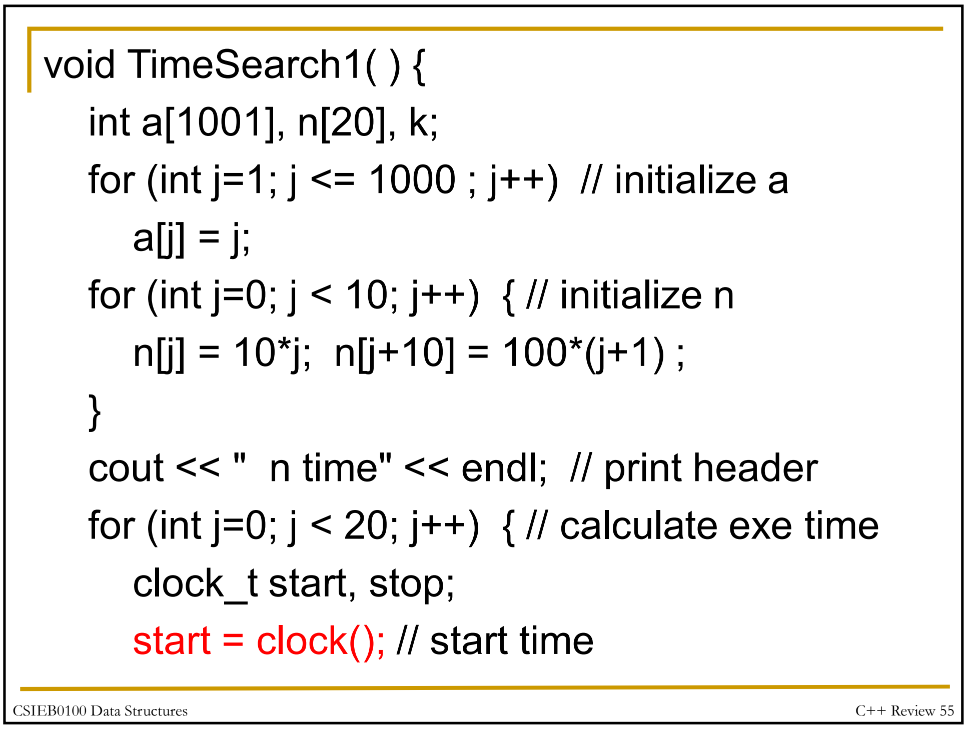
```
do {
        counter++;
        start=clock();
        doSomething();
        stop=clock();
        elapsedTime += stop - start;
} while (elapsedTime < 10)
```

What's wrong?

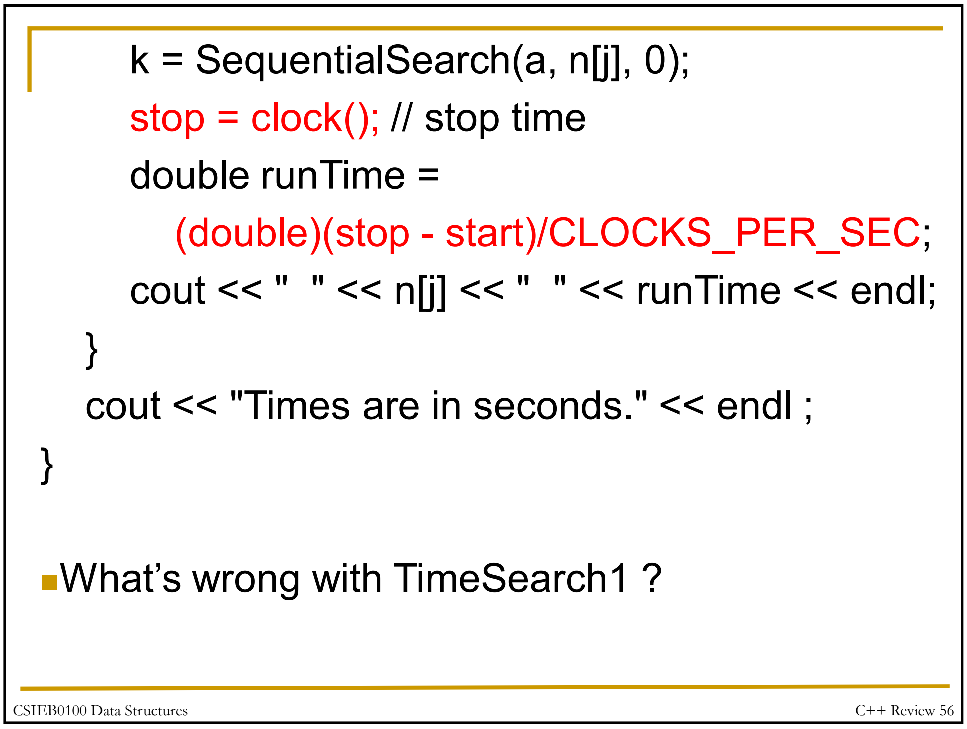


C++ Timing Example

```
int SequentialSearch (int *a, const int n, const int x)
{ // search a[0, ..., n] for x
    int i;
    for (int i=0; i < n && a[i] != x; i++) ;
    if (i == n) return -1;
    else return i;
}
```

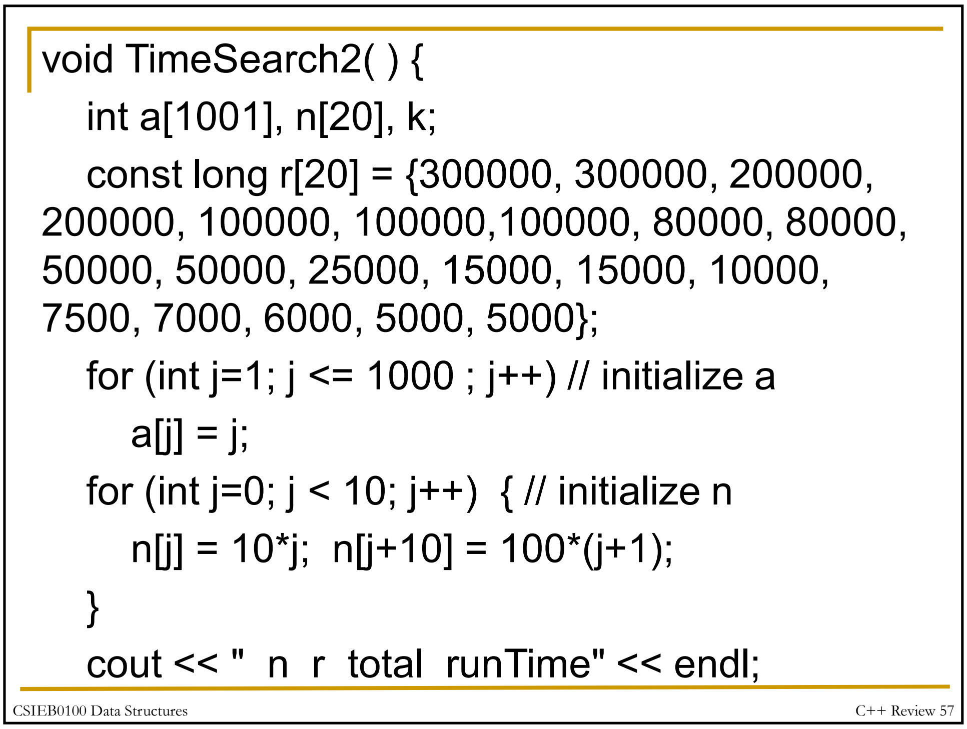


```
void TimeSearch1( ) {
    int a[1001], n[20], k;
    for (int j=1; j <= 1000 ; j++)  // initialize a
        a[j] = j;
    for (int j=0; j < 10; j++)  { // initialize n
        n[j] = 10*j;  n[j+10] = 100*(j+1) ;
    }
    cout << "  n time" << endl;  // print header
    for (int j=0; j < 20; j++)  { // calculate exe time
        clock_t start, stop;
        start = clock(); // start time
```

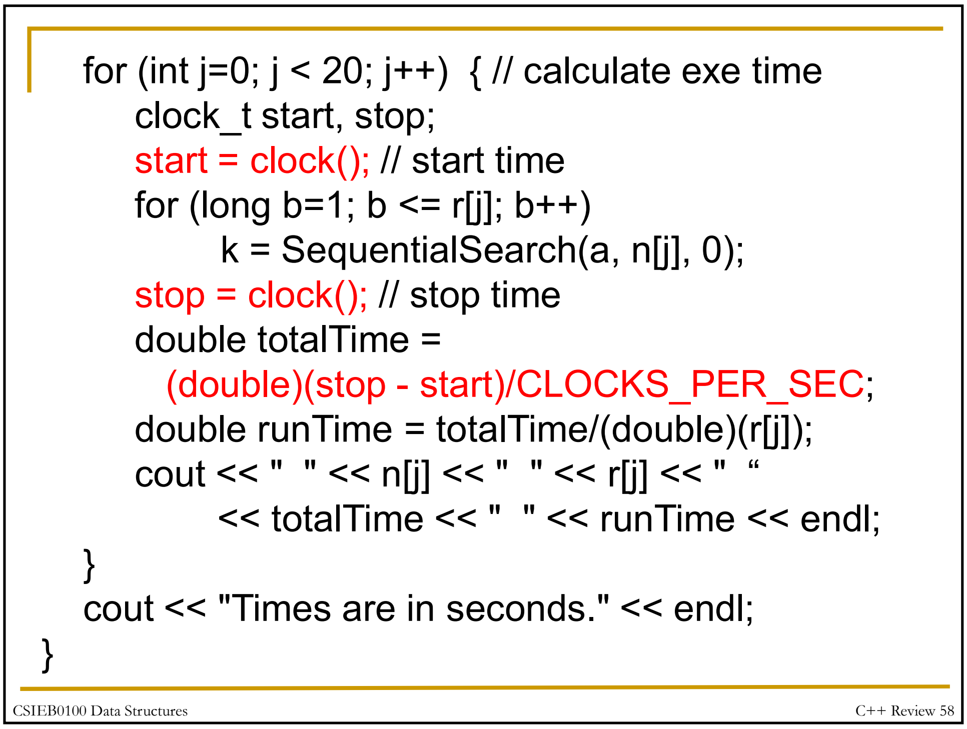


```
        k = SequentialSearch(a, n[j], 0);
        stop = clock(); // stop time
        double runTime =
            (double)(stop - start)/CLOCKS_PER_SEC;
        cout << "  " << n[j] << "  " << runTime << endl;
    }
    cout << "Times are in seconds." << endl ;
}
```

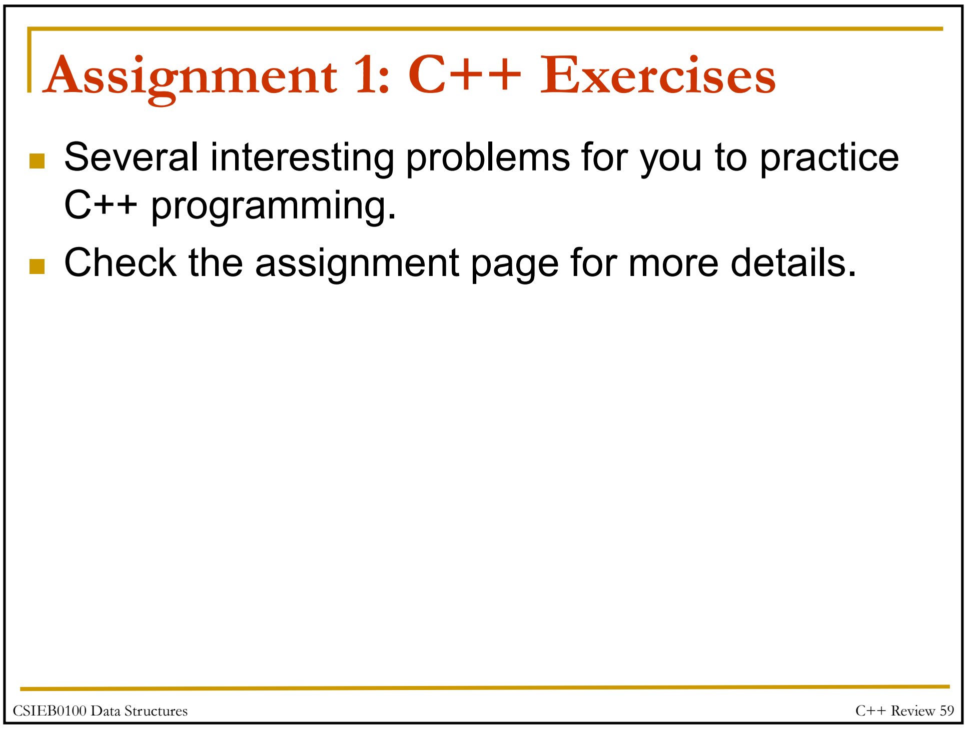
- What’s wrong with TimeSearch1 ?



```
void TimeSearch2( ) {
    int a[1001], n[20], k;
    const long r[20] = {300000, 300000, 200000,
200000, 100000, 100000,100000, 80000, 80000,
50000, 50000, 25000, 15000, 15000, 10000,
7500, 7000, 6000, 5000, 5000};
    for (int j=1; j <= 1000 ; j++) // initialize a
        a[j] = j;
    for (int j=0; j < 10; j++)  { // initialize n
        n[j] = 10*j;  n[j+10] = 100*(j+1);
    }
    cout << "  n  r  total  runTime" << endl;
```



```
    for (int j=0; j < 20; j++)  { // calculate exe time
        clock_t start, stop;
        start = clock(); // start time
        for (long b=1; b <= r[j]; b++)
              k = SequentialSearch(a, n[j], 0);
        stop = clock(); // stop time
        double totalTime =
          (double)(stop - start)/CLOCKS_PER_SEC;
        double runTime = totalTime/(double)(r[j]);
        cout << "  " << n[j] << "  " << r[j] << "  “
              << totalTime << "  " << runTime << endl;
    }
    cout << "Times are in seconds." << endl;
}
```



Assignment 1: C++ Exercises

- Several interesting problems for you to practice C++ programming.
- Check the assignment page for more details.