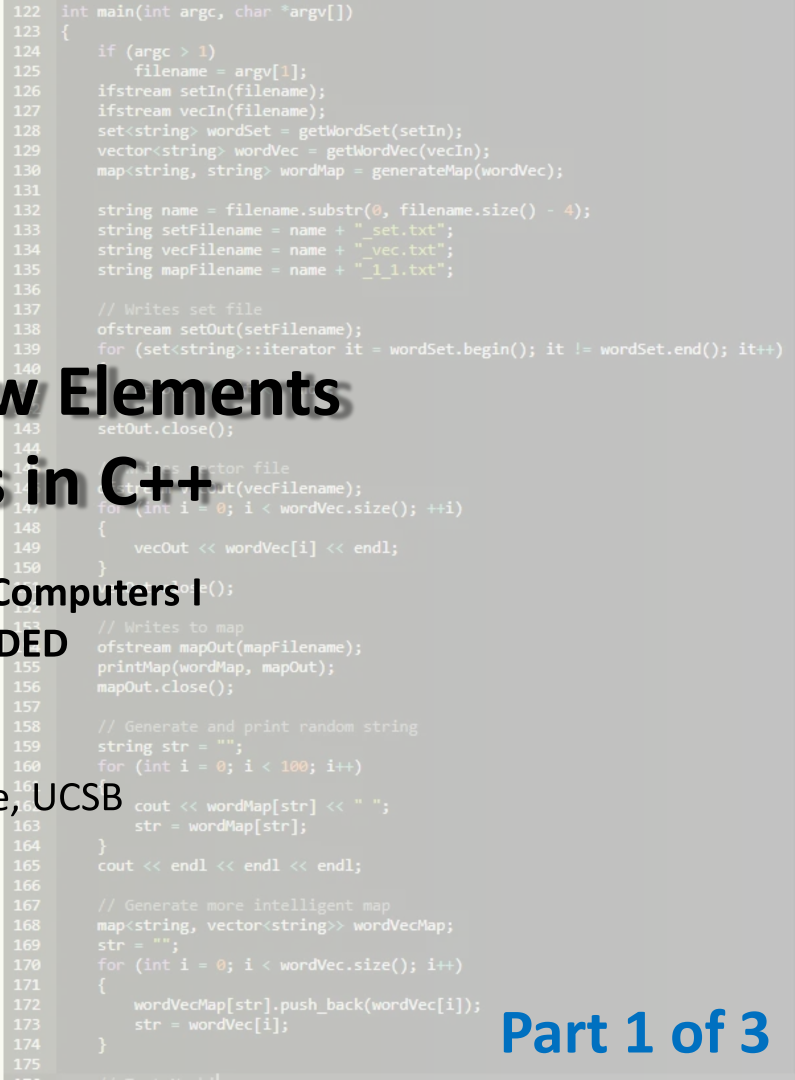
More on Program Flow Elements
Intro to Functions in C++

CS 16: Solving Problems with Computers I
Lecture #4 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB



```
int main(int argc, char *argv[])
{
    if (argc > 1)
        filename = argv[1];
    ifstream setIn(filename);
    ifstream vecIn(filename);
    set<string> wordSet = getWordSet(setIn);
    vector<string> wordVec = getWordVec(vecIn);
    map<string, string> wordMap = generateMap(wordVec);

    string name = filename.substr(0, filename.size() - 4);
    string setFilename = name + "_set.txt";
    string vecFilename = name + "_vec.txt";
    string mapFilename = name + "_1_1.txt";

    // Writes set file
    ofstream setOut(setFilename);
    for (set<string>::iterator it = wordSet.begin(); it != wordSet.end(); it++)

    setOut.close();

    // Writes vector file
    ofstream vecOut(vecFilename);
    for (int i = 0; i < wordVec.size(); ++i)
    {
        vecOut << wordVec[i] << endl;
    }
    vecOut.close();

    // Writes to map
    ofstream mapOut(mapFilename);
    printMap(wordMap, mapOut);
    mapOut.close();

    // Generate and print random string
    string str = "";
    for (int i = 0; i < 100; i++)
    {
        cout << wordMap[str] << " ";
        str = wordMap[str];
    }
    cout << endl << endl << endl;

    // Generate more intelligent map
    map<string, vector<string>> wordVecMap;
    str = "";
    for (int i = 0; i < wordVec.size(); i++)
    {
        wordVecMap[str].push_back(wordVec[i]);
        str = wordVec[i];
    }

```

Part 1 of 3

Lecture Outline

- Nested Loops
- Multiway Branching and the `switch` command
- Local vs. Global Variables

- Pre-Defined Functions
- User-Defined Functions
- Void Functions

Increments and Decrements by 1

In C++ you can increment-by-1 like this:

more common → `a++`

or like this:

```
++a
```

Similarly, you can decrement by:

```
a--  or  --a
```

Some Cool Uses of **x++**

Demo!

- In a while loop, you *usually always* need to increment a counter var.

Example:

```
int max = 0;
while (max < 4)
{
   cout << “hi” << endl;
   max++;
}
```

What will this print out?

```
hi
hi
hi
hi
```

Some Cool Uses of **x++**

- You can make a slight change and save a line of code!

Example:

```
int max = 0;
while (max++ < 4)
{
   cout << “hi” << endl;
}
```

When to use **x++** vs **++x**

- **x++** will assess **x** *then* increment it
- **++x** will increment **x** first, *then* assess it

- 95% of the time, you will use the first one

- In *while* statements, it **makes** a difference
- In *for* statements, it **won't make** a difference

Examples

```
for (int c = 0; c < 4; c++)
   cout << “hi” << endl;
```

Prints “hi” 4 times

```
for (int c = 0; c < 4; ++c)
   cout << “hi” << endl;
```

```
int max = 0;
while (max++ < 4)
{
   cout << “hi” << endl;
}
```

Prints “hi” 4 times

```
int max = 0;
while (++max < 4)
{
   cout << “hi” << endl;
}
```

Prints “hi” 3 times

What Happens If…

```
x = 1;
while (x > 0)
{
   cout << x << endl;
}
```

Answer:

The while loop is never finished!!

The program will be “stuck” or “hang”…

This is known as an “infinite loop”

Infinite Loops

- Loops that never stop – **must** be avoided!
  - Your program will either “hang” or just keep spewing outputs for ever

- The loop body should contain a line that will eventually cause the Boolean expression to become false (to make the loop to end)

- **Example**: Goal: Print all positive odd numbers less than 6

```
x = 1;
while (x != 6)
{
  cout << x << endl;
  x = x + 2;
}
```

What is the problem with this code and why?

```
Infinite Loop! x will never be 6!
```

What simple fix can undo this bad design?

```
while ( x < 6 )
```

Using **for-loops** For Sums

- To create an **accumulated sum**, in a for-loop:

```
int sum = 0;
for(int count = 0; count < 10; count++)
   {
        cin >> next;
        sum = sum + next;  // can also use sum += next;
   }
```

- Note that “sum” must be initialized prior to the loop body!
  - **Why?**

Using **for-loops** For Products

- Forming an **accumulated product** is very similar to the sum example

```
int product = 1;
for(int count = 0; count < 10; count++)
{
        cin >> next;
        product = product * next;
                                // can also use product *= next;
 }
```

- Note that "product" must be initialized prior to the loop body
  - Product is initialized to **1**, not 0!

Ending a *While* Loop

- A for-loop is generally the choice when there is **a predetermined** number of iterations
- When you DON'T have a predetermined number of iterations, you will want to use **while loops**

The are 3 common methods to END a while loop:

1. *List ended with a sentinel value:* Using a particular value or calculation to signal the end
2. *Ask before iterating:* Ask if the user wants to continue before each iteration
3. *Running out of input:* Using the *eof* function to indicate the end of a file (more on this when we discuss file I/Os)

1. List Ended With a Sentinel Value

Demo!

```
cout  << "Enter a positive integer and I will give you its double!\n"
      << "Place a negative integer to quit.\n";
cin >> number;
while (number >= 0)
{
   cout << “The double of that is: ” << 2*number << endl;
   cin >> number;
}
```

Note that the sentinel value (number) is **read**, **but not processed** at the end

2. Ask Before Iterating

Demo!

```
char ans;
cout << "Are you satisfied yet? (Y/N) ";
cin >> ans;

while (( ans == 'N')  || (ans == 'n'))
{
    cout << "How about now? Are you satisfied yet? (Y/N) ";
    cin >> ans;
}
```

Nested Loops

- The body of a loop may contain any kind of statement, ***including another loop***

- When loops are nested, **all iterations of the inner loop** are **executed for each iteration of the outer loop**

- *ProTip:* Give serious consideration to making the inner loop a function call to make it easier to read your program
  - More on functions later…

Example of a Nested Loop

- You want to collect the total grades of 100 students in a class
- Each student has multiple scores
  - Example: multiple homeworks, multiple quizzes, etc…
- You go through each student – one at a time – and get their scores
  - You calculate a sub-total grade for each student
- Then after collecting every student score, you calculate a grand total grade of the whole class and a class average (grand total / no. of students)

Example of a Nested Loop

```
int students(100);
double grade(0), subtotal(0), grand_total(0);

for (int count = 0; count < students; count++)
{
    cout << “Starting with student number: ” << count << endl;
    cout << “Enter grades. To move to the next student, enter a negative number.\n”
    cin >> grade;
    while (grade >= 0)
    {
        subtotal = subtotal + grade;
        cin >> grade;
    } // end while loop
    cout << “Total grade count for student ” << count << “is ” << subtotal << endl;
    grand_total = grand_total + subtotal;
    subtotal = 0;
} // end for loop

cout << “Average grades for all students= ” << grand_total / students << endl;
```

Multiway Branching

- Nesting (embedding) one if/else statement in another.

```
if (count < 10)
{
    if ( x < y )
    {
        cout << x << " is less than " << y;
    }
    else
    {
        cout << y << " is less than " << x;
    }
}
```

Note the tab indentation at each level of nesting.

Defaults in Nested IF/ELSE Statements

- When the conditions tested in an if-else-statement are mutually exclusive, the final if-else can sometimes be omitted

EXAMPLE:

```
if (guess > number)
    cout << “Too high.”;
else if (guess < number)
    cout << “Too low.”;
else if (guess == number)
    cout << “Correct!”;
```

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.”;
else cout << "Correct!";
```

i.e. All other possibilities

A Better Way… Using **switch**

An alternative for constructing multi-way branches

```
switch (variable)
{
    case variable_value1:
        statements;
        break;

    case variable_value2:
        statements;
        break;

    …   …   …

    default:
        statements;
    }
```

Controlling statement

“break” statement is important – you cannot forget it!

Demo!

The Controlling Statement

- A `switch` statement's controlling statement must return one of these basic types:
  - – A **bool** value
  - – An **int** type
  - – A **char** type

- `switch` will <u>not</u> work with **strings** in the controlling statement.

Can I Use the **break** Statement in a *Loop*?

- We saw the use of break as important in the switch-case…
- Technically, the **`break`** statement can be used to exit a loop (i.e. force it to) before normal termination
- **But it’s not good design practice!**
  - Its use is considered “sloppy” in loops
  - In this class, **do NOT use it outside of `switch`**

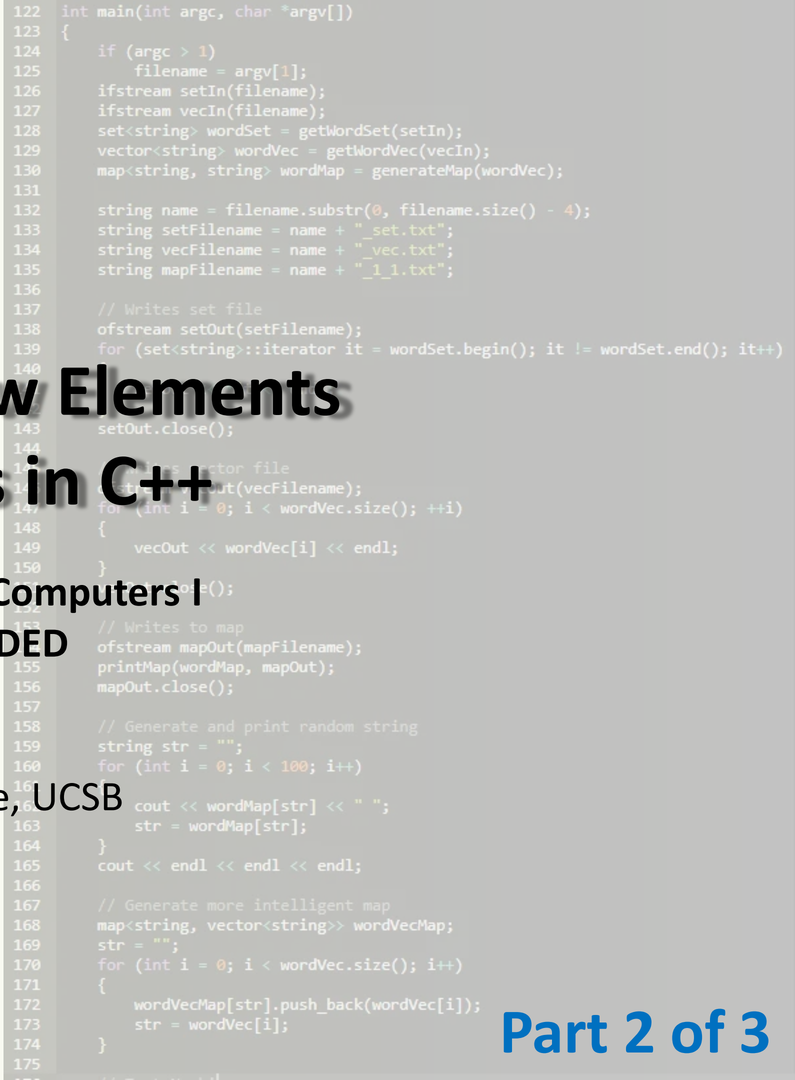
END OF PART 1

More on Program Flow Elements
Intro to Functions in C++

CS 16: Solving Problems with Computers I
Lecture #4 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 2 of 3



```
int main(int argc, char *argv[])
{
    if (argc > 1)
        filename = argv[1];
    ifstream setIn(filename);
    ifstream vecIn(filename);
    set<string> wordSet = getWordSet(setIn);
    vector<string> wordVec = getWordVec(vecIn);
    map<string, string> wordMap = generateMap(wordVec);

    string name = filename.substr(0, filename.size() - 4);
    string setFilename = name + "_set.txt";
    string vecFilename = name + "_vec.txt";
    string mapFilename = name + "_1_1.txt";

    // Writes set file
    ofstream setOut(setFilename);
    for (set<string>::iterator it = wordSet.begin(); it != wordSet.end(); it++)

    setOut.close();

       ctor file
    ut(vecFilename);
    for (int i = 0; i < wordVec.size(); ++i)
    {
        vecOut << wordVec[i] << endl;
    }
    ose();

    // Writes to map
    ofstream mapOut(mapFilename);
    printMap(wordMap, mapOut);
    mapOut.close();

    // Generate and print random string
    string str = "";
    for (int i = 0; i < 100; i++)

        cout << wordMap[str] << " ";
        str = wordMap[str];
    }
    cout << endl << endl << endl;

    // Generate more intelligent map
    map<string, vector<string>> wordVecMap;
    str = "";
    for (int i = 0; i < wordVec.size(); i++)
    {
        wordVecMap[str].push_back(wordVec[i]);
        str = wordVec[i];
    }
```

Note About Blocks

- ***Recall***: A block is a section of code enclosed by **{…}** braces
- Variables declared within a block, are **local to the block**
  - An exclusivity feature
  - These variable are said to have the block as their ***scope***.
  - They can used inside this block *and nowhere else!*
- Variable names declared inside the block **cannot** be re-used outside the block

Local vs. Global Variables

- **Local variables** only work in a specified **block of statements**
  - If you try and use them outside this block, they won't work

- **Global variables** work in the **entire program**

- There are standards to each of their use
  - Local variables are **much preferred** as global variables can cause conflicts in the program
  - Sometimes we want to define **constants** and use them as globals

Local vs. Global Variables – Example

Demo!

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

int main( )
{
   int age(0);
   for (int c = 0; c < 10; c++)
   {
      cout << age*c << endl;
      age += (2*c + 4);
   }
   return 0;
}
```

Local to main( )

Local to the for-loop

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

int age(0);
int main( )
{
   for (int c = 0; c < 10; c++)
   {
      cout << age*c << endl;
      age += (2*c + 4);
   }
   return 0;
}
```

Globally declared

Global Constants – Example

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

const double PI=3.14159;
int main( )
{
    double angle=0;
    while (angle <= 2*PI)
    {
        cout << "sin(" << angle << ") = ";
        cout << sin(angle);
        angle += PI/4;
    }
    return 0;
}
```

Globally declared

FUNCTIONS in C++

Predefined Functions in C++

- C++ comes with “built-in” libraries of predefined functions
- Example: sqrt function (found in the library ***cmath***)
  - Computes and returns the square root of a number

```
the_root = sqrt(9.0);
```

  - The number 9 is called *the argument*
- Can variable **the_root** be either int or double?

Notes on the **cmath** Library

- Standard math library in C++
- Contains several useful math functions, like

`cos( ), sin( ), exp( ), log( ), pow( ), sqrt( )`

- To use it, you must import it at the start of your program

#include <cmath>

  – You can find more information on this library at: http://www.cplusplus.com/reference/cmath/

Other Predefined **cmath** Functions

- pow(x, y) --- `double value = pow(2, -8);`
  - Returns 2^{-8} , a double value (value = 0.00390625)
  - Arguments are of type double

- sin(x), cos(x), tan(x), etc… --- `double value = sin(1.5708);`
  - Returns sin($\pi/2$) (value = 1) – note it's in radians
  - Argument is of type double

Other Predefined **cmath** Functions

- abs(x) --- `int value = abs(-8);`
  - Returns absolute value of argument x
  - Return value is of type **int**
  - Argument is of type int

- fabs(x) --- `double value = fabs(-8.0);`
  - Also returns absolute value of argument x
  - Return value is of type **double**
  - Argument is of type double

Random Number Generation: Step 1

- Not true-random, but pseudo-random numbers.

  Must

  ```
  #include <cstdlib>
  #include <ctime>
  ```

- First, ***seed*** the random number generator (only need to do this once)

  ```
  srand(time(0));  //place inside main( )
  ```

  – **time( )** is a pre-defined function in the **ctime** library: gives current system time
  (it gives the current system time)

  – It's used here because it generates a *distinctive enough seed*, so that **rand( )** generates a "good enough" random number.

Random Number Generation: Step 2

- Next, use the **rand( )** function, which returns a random integer that is greater than or equal to 0 and less than RAND_MAX
(a library-dependent value, but is at least 32767)

```
int r = rand();
```

- But what if you want to generate random numbers in other ranges? Example, between 1 and 6?

Random Numbers

Demo!

- Use **%** and **+** to scale to the number range you want

- For example to get a random number bounded from 1 to 6 to simulate rolling a six-sided die:

```
int die = (rand( ) % 6) + 1;
```

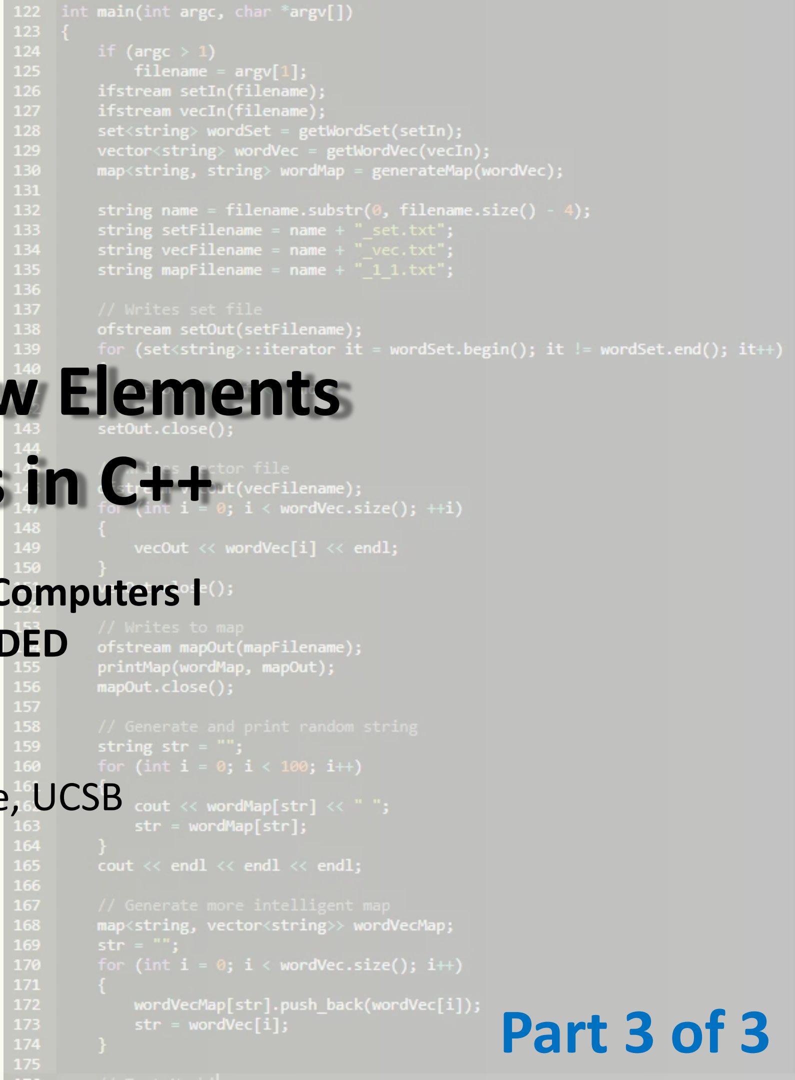
END OF PART 2

More on Program Flow Elements
Intro to Functions in C++

CS 16: Solving Problems with Computers I
Lecture #4 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 3 of 3



```
int main(int argc, char *argv[])
{
    if (argc > 1)
        filename = argv[1];
    ifstream setIn(filename);
    ifstream vecIn(filename);
    set<string> wordSet = getWordSet(setIn);
    vector<string> wordVec = getWordVec(vecIn);
    map<string, string> wordMap = generateMap(wordVec);

    string name = filename.substr(0, filename.size() - 4);
    string setFilename = name + "_set.txt";
    string vecFilename = name + "_vec.txt";
    string mapFilename = name + "_1_1.txt";

    // Writes set file
    ofstream setOut(setFilename);
    for (set<string>::iterator it = wordSet.begin(); it != wordSet.end(); it++)

    setOut.close();

                ctor file
                ut(vecFilename);
             i = 0; i < wordVec.size(); ++i)
    {
        vecOut << wordVec[i] << endl;
    }
           ose();

    // Writes to map
    ofstream mapOut(mapFilename);
    printMap(wordMap, mapOut);
    mapOut.close();

    // Generate and print random string
    string str = "";
    for (int i = 0; i < 100; i++)

        cout << wordMap[str] << " ";
        str = wordMap[str];
    }
    cout << endl << endl << endl;

    // Generate more intelligent map
    map<string, vector<string>> wordVecMap;
    str = "";
    for (int i = 0; i < wordVec.size(); i++)
    {
        wordVecMap[str].push_back(wordVec[i]);
        str = wordVec[i];
    }
```

Programmer-Defined Functions

- In C++, you can create your own functions
  - You can have them “do things” based on ***input arguments***
  - These functions can also ***return*** a value or ***NOT***

- You have to declare functions as “types”
  - That is, what “type” of data they return (if any)
  - Example (here, **x** and **y** are the *input arguments*):

`double functionX(int x, int y)`	returns a double
`string functionX(int x, int y)`	returns a string
`void functionX(int x, int y)`	returns nothing

Programmer-Defined Functions

- There are 2 necessary components for using functions in C++

- **Function declaration** (a.k.a function prototype)
  - Just like declaring variables
  - <u>Must</u> be placed *outside* the **main( )**, *usually* just before it
  - <u>Must</u> be placed *before* the function is ***defined*** & ***called***

- **Function definition**
  - This is where you define the function itself (all the details go here)
  - <u>Must</u> be place *outside* the **main( )**
  - Can be before **main( )** or after it, *often* placed after it

Function Declaration

- Shows how the function is ***called*** from **main( )** or from other functions

- **Must** appear in the code *before* the function can be called

- Syntax:

```
Type_returned Function_Name(Parameter_List);
//Comment describing what function does
```

; *Needed for declaration statement*

E.g:

```
double interestOwed(double principle, double rate);
//Calculates the interest owed on a loan
```

Function Definition

- Describes ***how*** the function does its task
- Can appear before or after the function is called

- Syntax:

```
Type_returned  Function_Name(Parameter_List)
   {
         //code to make the function work
   }
```

Example of a Simple Function in C++

Demo!

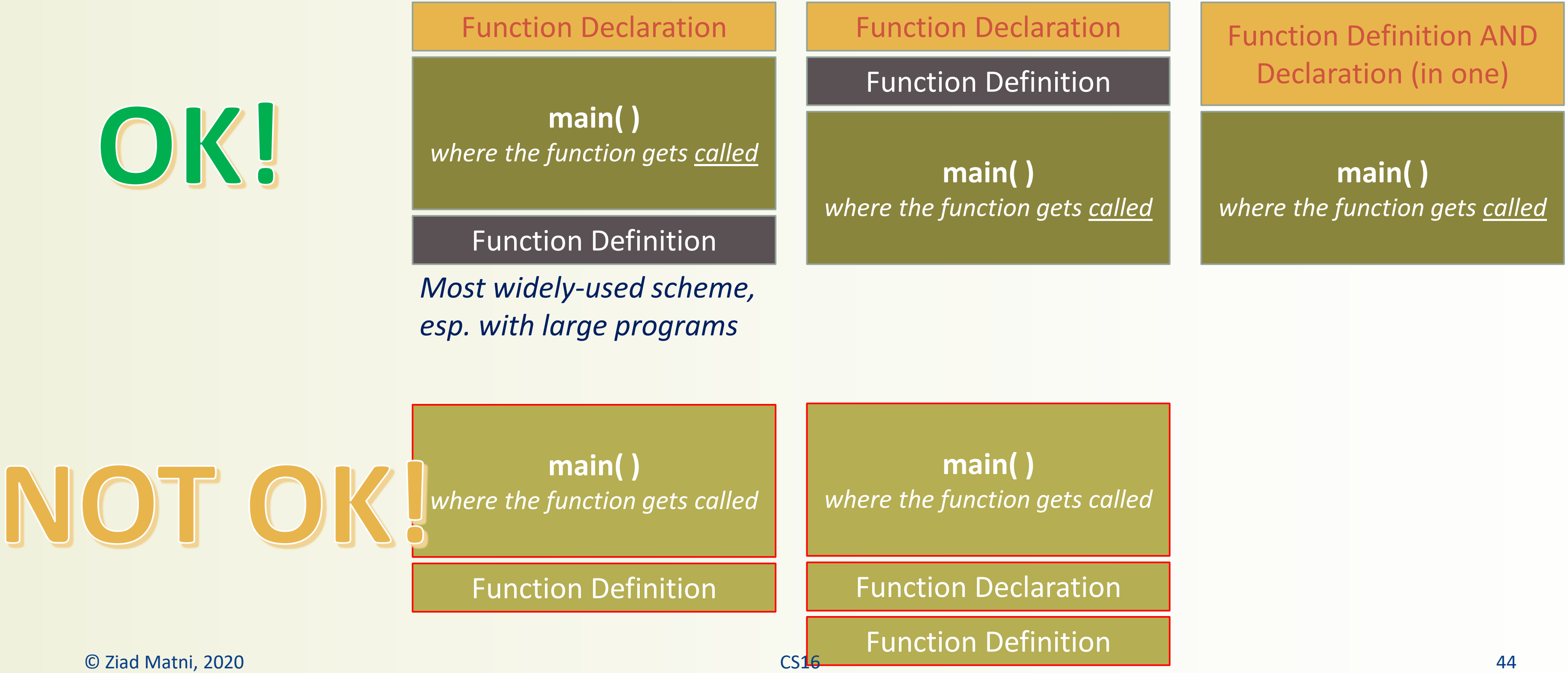
```cpp
#include <iostream>
using namespace std;

int sum2nums(int num1, int num2); // returns the sum of 2 numbers

int main ( )
{
    int a(3), b(5);
    int sum = sum2nums(a, b);
    cout << sum << endl;
    return 0;
}

int sum2nums(int num1, int num2)
{
    return (num1 + num2);
}
```

Declaration

Call

Definition

Block Placements for Functions



OK!

Function Declaration

main( )
where the function gets *<u>called</u>*

Function Definition

Most widely-used scheme, esp. with large programs

Function Declaration

Function Definition

main( )
where the function gets *<u>called</u>*

Function Definition AND Declaration (in one)

main( )
where the function gets *<u>called</u>*

NOT OK!

main( )
where the function gets called

Function Definition

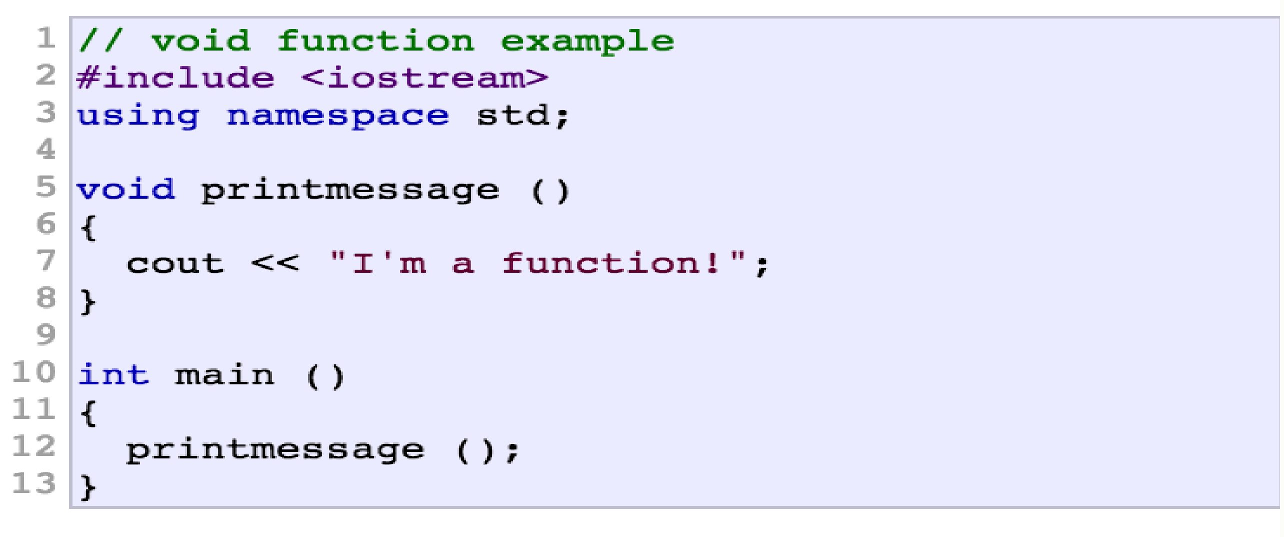
main( )
where the function gets called

Function Declaration

Function Definition

void Functions

- Sometimes, we want ***design subtasks*** to be implemented as functions.
  - Repetition involved, like printing some variable over and over again
  - We may not want to return anything



```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

void Function: Simple Example

- Let’s say, you want to pass a number to a function and then have it always ***print*** out its triple value (i.e. var * 3)

```
void tripleIt(double number)
{
   cout << number << "x 3 = " << number*3 << endl;
   return;
}
```

NOTE: the ‘return’ instruction here is OPTIONAL (why?)

Calling **void** Functions

- void-function calls are, essentially, *executable statements*
  - They do not need to be part of another statement
  - They end with a semi-colon

- Example from previous slide:

Call it inside of main() with: `tripleIt(32.5);`

NOT with: `cout << tripleIt(32.5);`

Will not compile!!!!

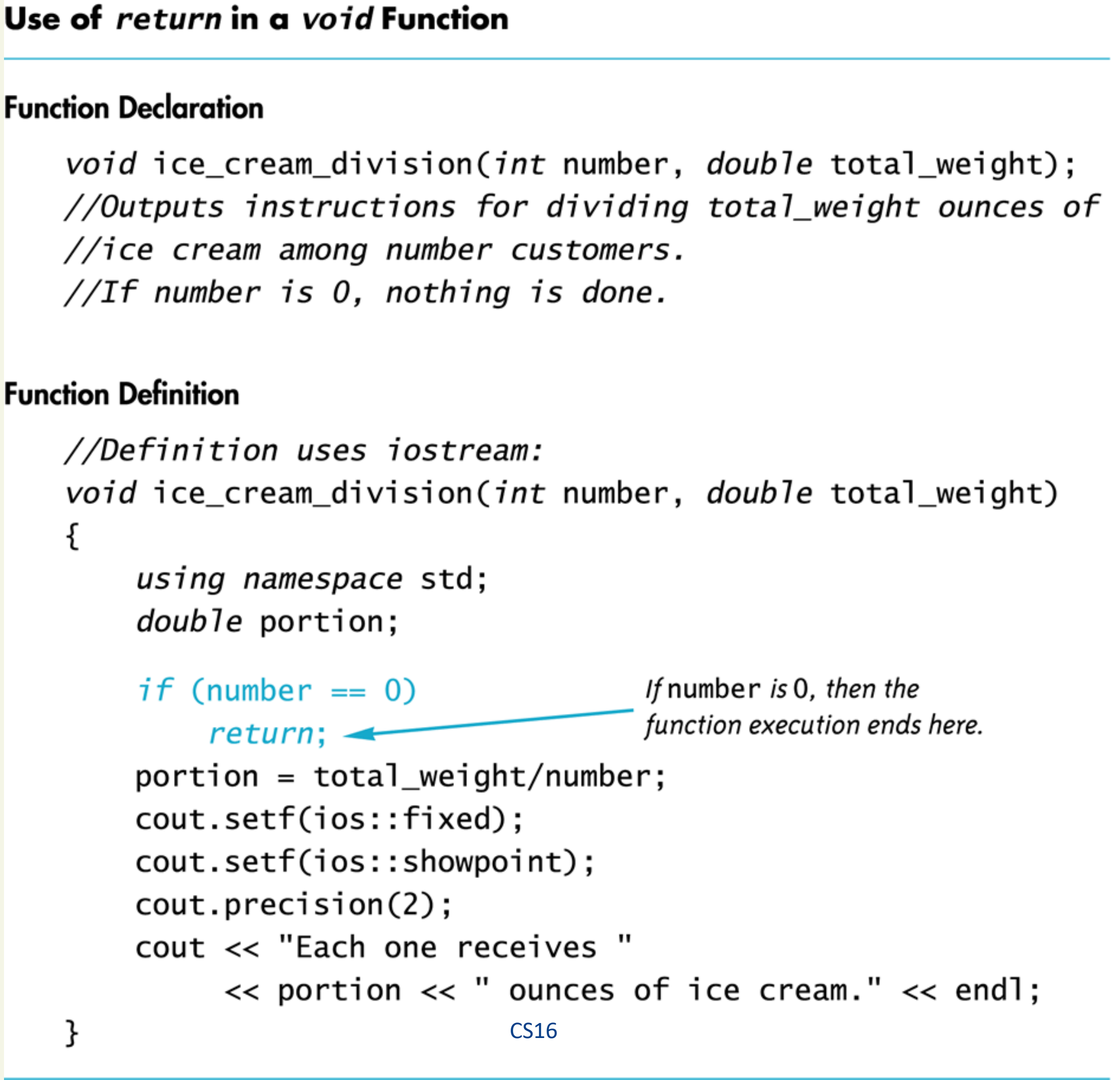
This distinction is important and a typical rookie mistake to make!!!

void Functions: To Return or Not Return?

- In void functions, we need "return" to indicate the end of the function
  - Is it strictly necessary for that? *No, it's optional*

- Can we use "return" to signal an "interrupt" to the function…
  - …and end it prematurely? *Yes you can do that!*

- Example: What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?
  - See example on next page of a void function that avoids division by zero with a return statement



Use of *return* in a *void* Function

Function Declaration

```
void ice_cream_division(int number, double total_weight);
//Outputs instructions for dividing total_weight ounces of
//ice cream among number customers.
//If number is 0, nothing is done.
```

Function Definition

```
//Definition uses iostream:
void ice_cream_division(int number, double total_weight)
{
    using namespace std;
    double portion;

    if (number == 0)
        return;
    portion = total_weight/number;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Each one receives "
         << portion << " ounces of ice cream." << endl;
}
```

If number *is* 0, *then the function execution ends here.*

The **main** Function in C++ :
*Why is it an **int** type, not a **void** type???*

- The **main** function in a program **is used like a void function**
  - So why do we have to end the program with a return statement?
  - And why isn't it DEFINED as a void function?

- The **main** function is defined to return a value of type **int**, therefore a return is needed
  - It's a matter of what is "legal" and "not legal" in C++
  - **void main ( )** is not legal in C++ !! (this ain't Java)
  - Most compilers **will not accept a void main** *(none of the ones we're using, anyway…)*
  - Solution? **Stick to what's legal**: it's ALWAYS **int main ( )**

</LECTURE>