More on Functions
Command Line Arguments

CS 16: Solving Problems with Computers I
Lecture #8

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Homework #7 due today**

- **Lab #4 is due on Monday at 8:00 AM!**

- Homework Solutions are now online at:

http://cs.ucsb.edu/~zmatni/cs16/hwSolutions/

Lecture Outline

- Overloading function names in C++
- *void* functions
- Getting arguments from the OS command line

Overloading Function Names

- C++ **allows more than one definition** for the **same function** name
  - Very convenient for situations in which the "same" function is needed for different numbers or types of arguments

- ***Overloading a function name:***

  providing more than one declaration and definition using the same function name

Overloading Examples

```
double ave(double n1, double n2)
{
          return ((n1 + n2) / 2);
}
```

```
double ave(double n1, double n2, double n3)
{
     return (( n1 + n2 + n3) / 3);
}
```

- Compiler checks the **number and types of arguments** in the function call & then decides which function to use.

- So, with a statement like:

```
cout << ave( 10, 20, 30);
```

the compiler knows to use the second definition

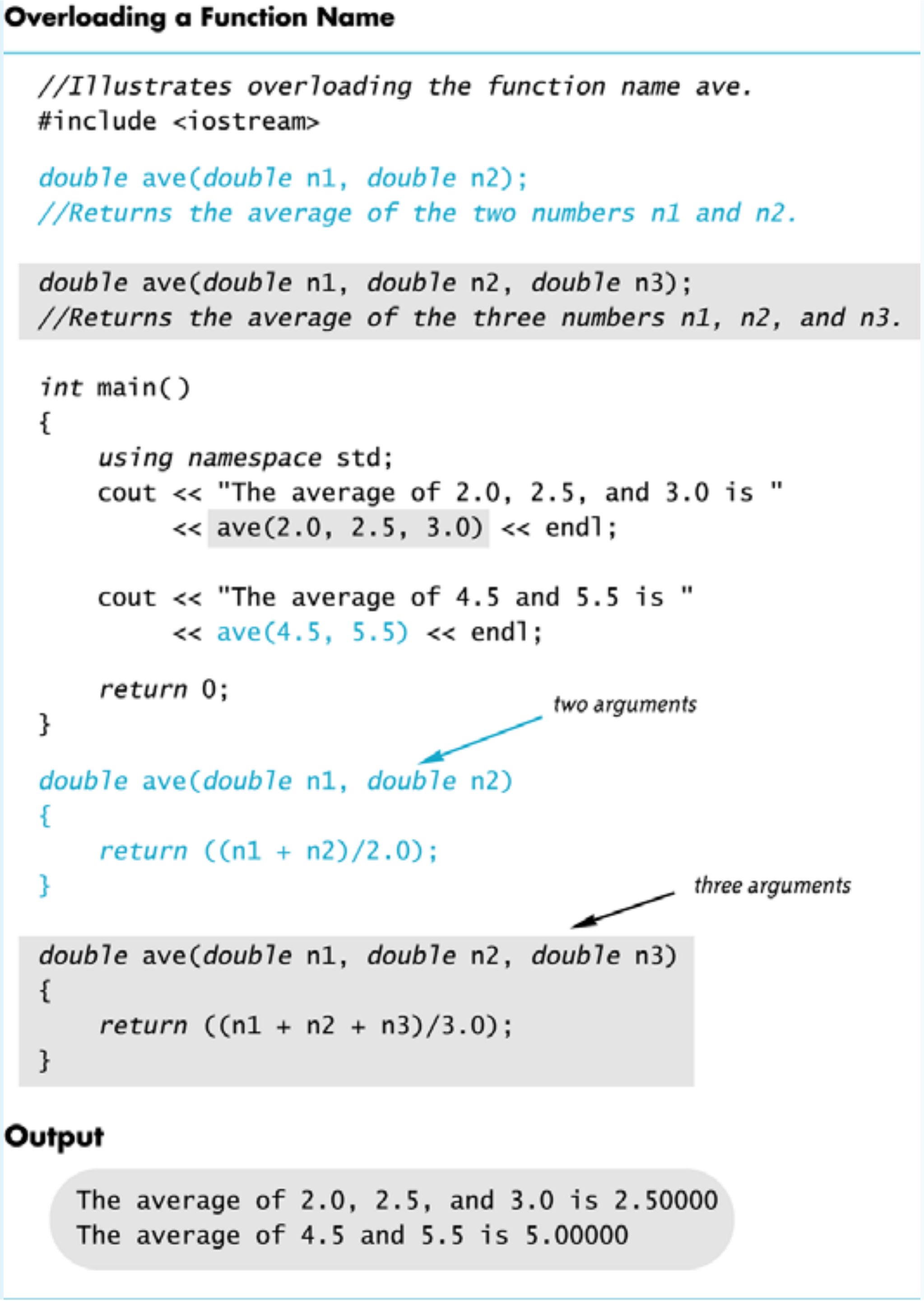
Overloading Details

- Overloaded functions
  - Must have ***different numbers*** of formal parameters

    AND / OR

  - Must have at least ***one different type*** of parameter

  - Must return a value of the ***same type***



Overloading a Function Name

```
//Illustrates overloading the function name ave.
#include <iostream>

double ave(double n1, double n2);
//Returns the average of the two numbers n1 and n2.

double ave(double n1, double n2, double n3);
//Returns the average of the three numbers n1, n2, and n3.

int main()
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;

    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;

    return 0;
}

double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}

double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

two arguments

three arguments

Output

```
The average of 2.0, 2.5, and 3.0 is 2.50000
The average of 4.5 and 5.5 is 5.00000
```

More Overloading Functions Examples

See textbook, Ch. 4.6, pp. 235 – 237 (Pizza buying program)

- There are two types of pizzas: circular and rectangle
- One overloaded function to calculate the unit price: ***unitprice***
  - Returns the unit price of a slice of pizza

- If you want to calculate the unit price of a circular pizza, call **unitprice (diameter, price)**

- If you want to calculate the unit price of a rectangular pizza, call **unitprice (length, width, price)**

Automatic Type Conversion

- C++ will automatically converts types between int and double in multiple examples
  - Eg. If I divide integers, I get integers: 13/2 = 6
  - Eg. If I make on these a double, I a double: 13/2.0 = 6.5

- It does the same with overloaded functions, for example, given the definition:

```
    double mpg(double miles, double gallons) {
        return (miles / gallons);      }
```

  what will happen if **mpg** is called in this way?

```
    cout << mpg(45, 2) << " miles per gallon";
```

- The values of the arguments will automatically be converted to type double (45.0 and 2.0)

Type Conversion Problem

- Let’s keep the previous mpg function and then ADD the following definition in the same program (we’ll overload the **mpg** function):

```
int mpg(int goals, int misses)
     // returns the Measure of Perfect Goals
     {
     return (goals – misses);   }
```

- What happens if **mpg** is called this way now?

```
cout << mpg(45, 2) << “ miles per gallon”;
```

- The compiler chooses the function that matches parameter types so the Measure of Perfect Goals will be calculated

This can introduce confusion into the program!

Do not use the same function name for unrelated functions

void Functions

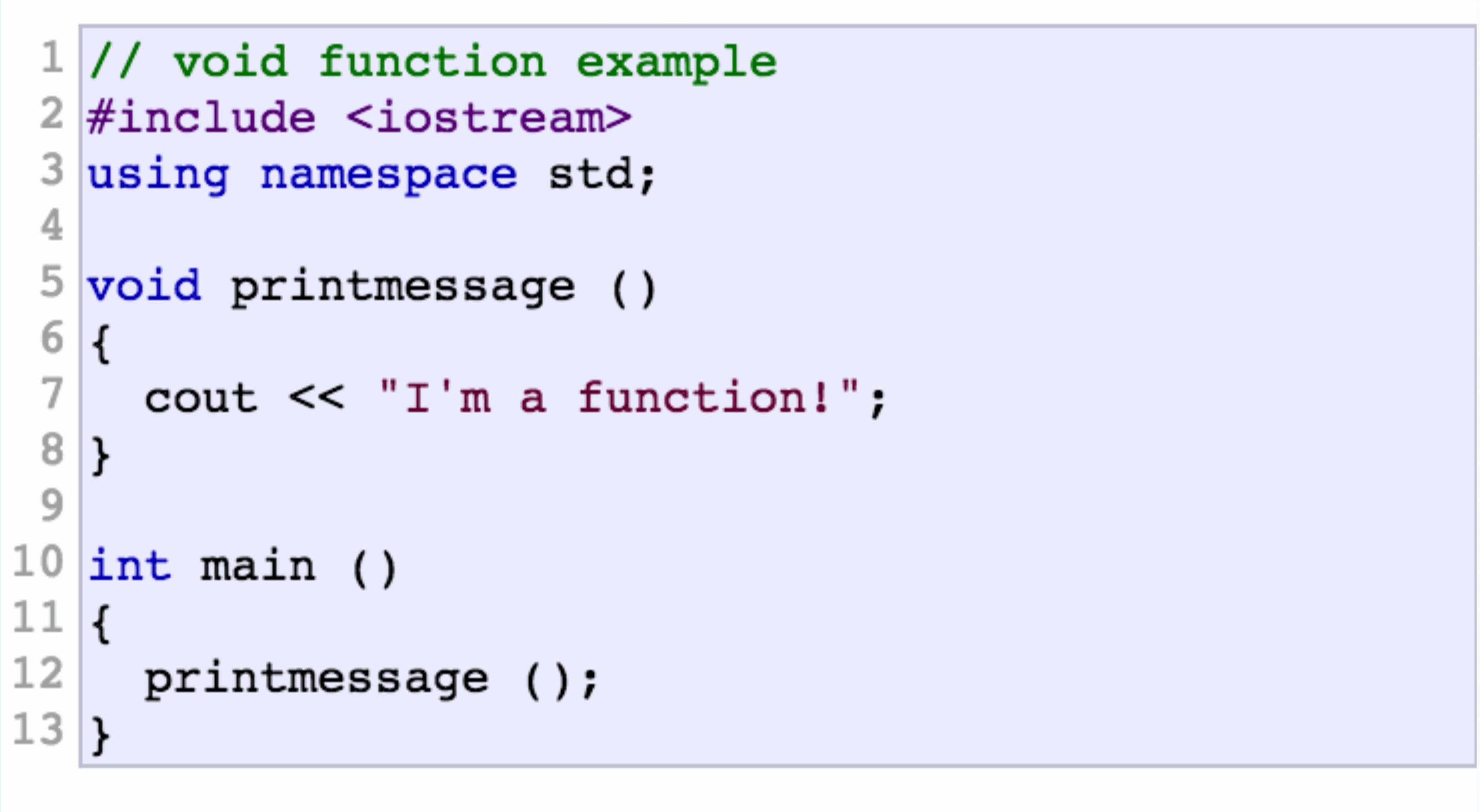
- In a top-down design, we'll want to design subtasks, often implemented as functions.

- A subtask might produce:
  - No value
  - One value
  - More than one value

- We've know how to implement functions that return one value
  - So what about the other cases?

A ***void-function*** implements a subtask that
returns no value ***or*** more than one value

Simple void Function Example



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

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

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

void Function Definition

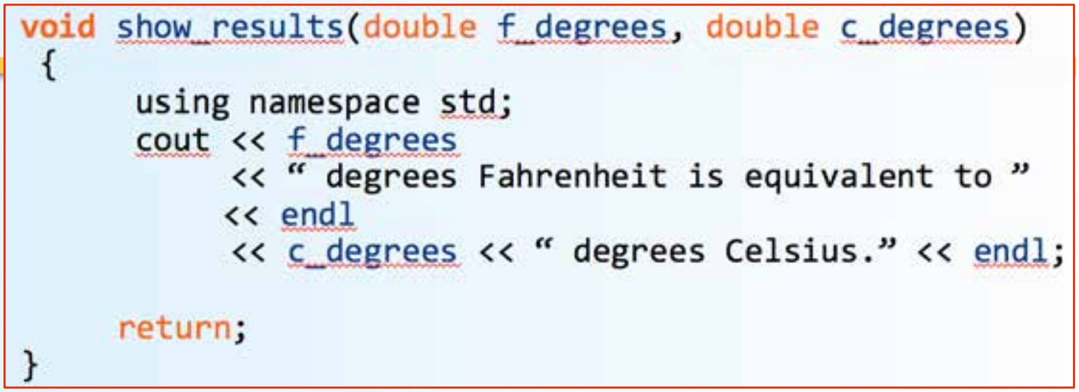
- void function definitions vs. regular function definitions
  - Keyword ***void*** replaces the type of the value returned
  - ***void*** = no value is returned by the function
  - The return statement does **not** include an expression

Example:

```
void show_results(double f_degrees, double c_degrees)
 {
      using namespace std;
      cout << f_degrees
           << “ degrees Fahrenheit is equivalent to ”
           << endl
           << c_degrees << “ degrees Celsius.” << endl;

       return;
}
```

Calling void Functions



```
void show_results(double f_degrees, double c_degrees)
 {
        using namespace std;
        cout << f_degrees
             << “ degrees Fahrenheit is equivalent to ”
            << endl
             << c_degrees << “ degrees Celsius.” << endl;

       return;
}
```

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

- Example:

```
show_results(32.5, 0.3);
```

NOT:

```
cout << show_results(32.5, 0.3);
```

Calling void Functions

- Same as the function calls we have seen so far
  - Argument values are substituted for the formal parameters

- It is fairly common to have no parameters in **void** functions
  - In this case there will be no arguments in the function call

- Statements in the function body are executed

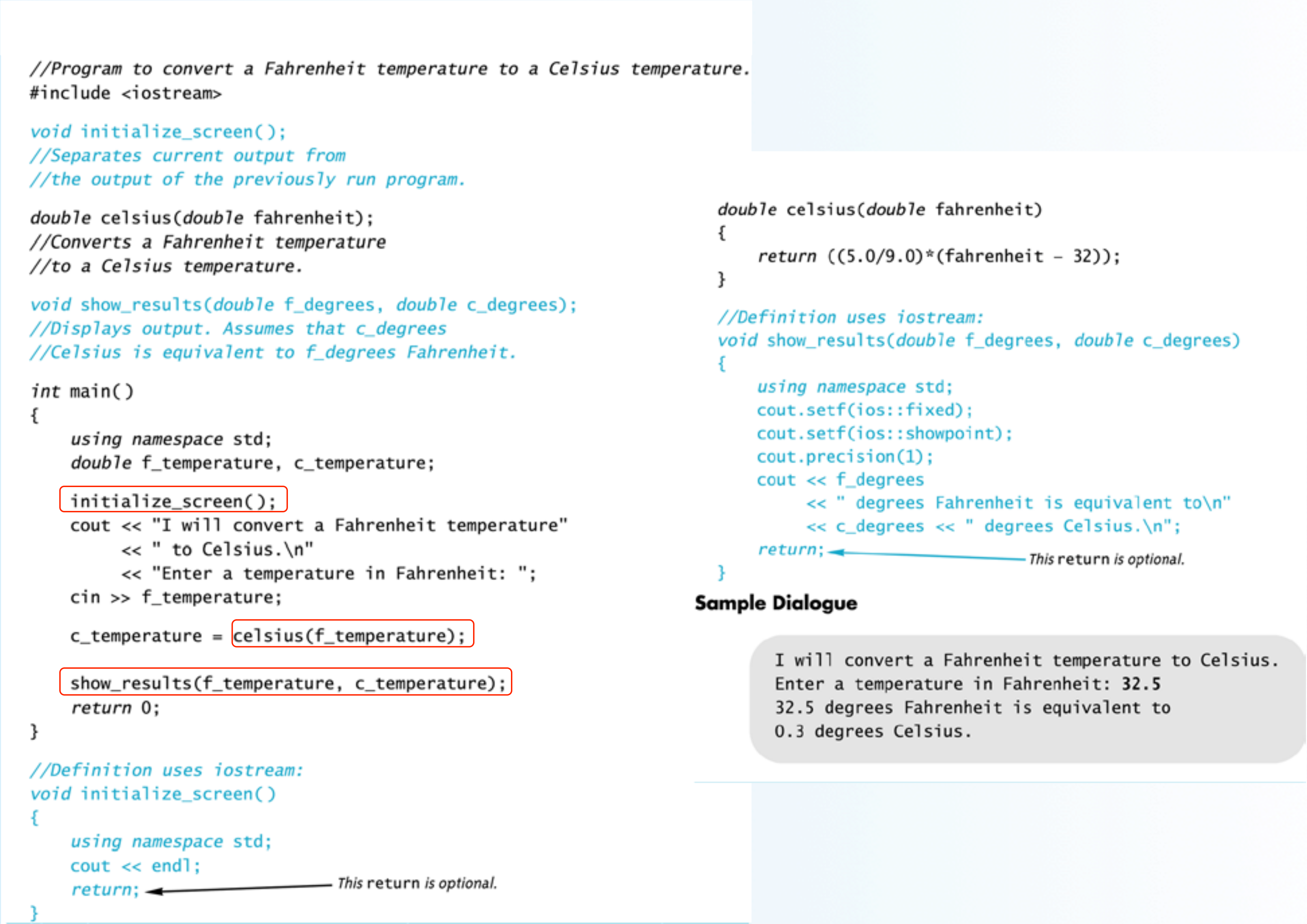
- *Optional* return statement ends the function
  - Return statement does not include a value to return
  - Return statement is implicit if it is not included

Example: Converting Temperatures

- Consider a function in a program that converts Fahrenheit temperatures to Celsius using the formula:

$$\mathbf{C = (5/9)\ (F - 32)}$$

- What's the potential challenge here?
  - Do you see the integer division problem? How do avoid the problem?



```
//Program to convert a Fahrenheit temperature to a Celsius temperature.
#include <iostream>

void initialize_screen();
//Separates current output from
//the output of the previously run program.

double celsius(double fahrenheit);
//Converts a Fahrenheit temperature
//to a Celsius temperature.

void show_results(double f_degrees, double c_degrees);
//Displays output. Assumes that c_degrees
//Celsius is equivalent to f_degrees Fahrenheit.

int main()
{
    using namespace std;
    double f_temperature, c_temperature;

    initialize_screen();
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> f_temperature;

    c_temperature = celsius(f_temperature);

    show_results(f_temperature, c_temperature);
    return 0;
}

//Definition uses iostream:
void initialize_screen()
{
    using namespace std;
    cout << endl;
    return;                    This return is optional.
}
```

```
double celsius(double fahrenheit)
{
    return ((5.0/9.0)*(fahrenheit - 32));
}

//Definition uses iostream:
void show_results(double f_degrees, double c_degrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << f_degrees
         << " degrees Fahrenheit is equivalent to\n"
         << c_degrees << " degrees Celsius.\n";
    return;                    This return is optional.
}
```

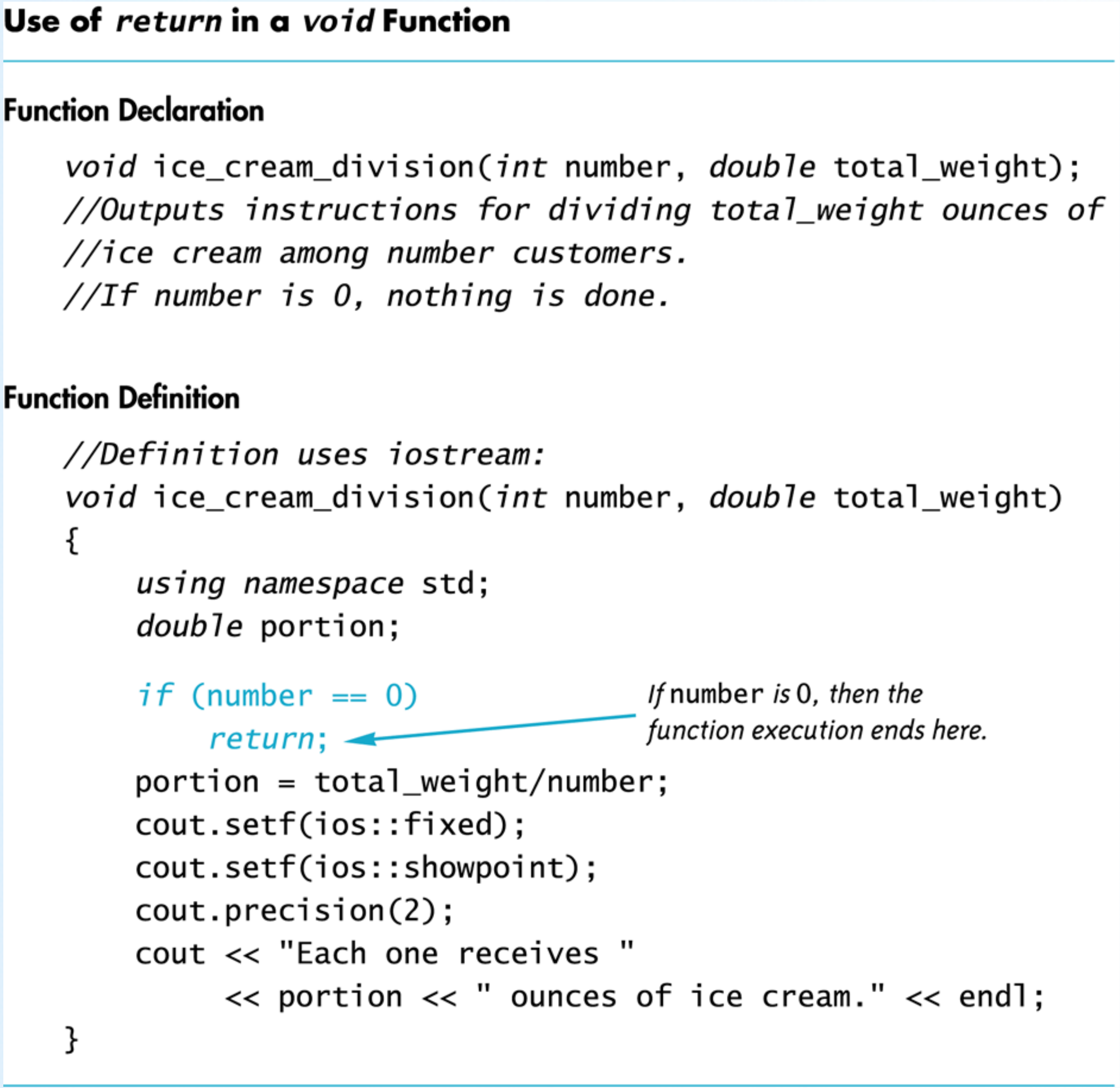
Sample Dialogue

```
I will convert a Fahrenheit temperature to Celsius.
Enter a temperature in Fahrenheit: 32.5
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

void-Functions
To Return or Not Return?

- Would we ever *need* a return-statement in a void-function if *no value* is returned?
  - Yes: there are cases where we would!

- 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

- 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?

- Because 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, but not all of them…
  - Solution? Stick to what's legal.

- The C++ standard also says the **return 0** can be omitted, but many compilers still require it
  - No compiler will complain if *you have* the return 0 statement in **main**
  - Solution? Always include **return 0** in the **main** function.

Command Line Arguments with C++

- In C++ you can accept **command line arguments**
- These are arguments that are passed into the program from the OS command line
  - See example in Lab 3

- To use command line arguments in your program, you must add 2 special arguments in the **main()** function
  - Argument #1 is the number of elements (**argc**) inside the next argument, which is an *array* (***argv[]**)
  - Argument #2 is a full list of all of the command line arguments: ***argv[]**

- In the OS, to execute the program, the command line form would be:

```
$ program_name  argument1  argument2  …  argumentn
```

example:

```
$ sum_of_squares 4 5 6
```

Setup

- The main() function should be written as either:

```
int main(int argc, char* argv[])
```

or

```
int main(int argc, char** argv)
```

- **`char* argv[]`** means:
  a pointer to an array of characters
  - We'll be discussing pointers in more detail in another lecture…

DEMO:

```
int main ( int argc, char *argv[] ) {
cout << "There are " << argc << " arguments here:" << endl;

for (int i = 0; i < argc; i++)
    cout << "argv[" << i << "] is : " << argv[i] << endl;

return 0; }
```

What If I Want an Argument That’s a Number?

- All you get from the command-line is character arrays
- To treat an argument as another type, you have to ***convert it inside your program***
- **<cstdlib>** library has pre-defined functions to help!
- Examples: **atoi( )**, **atol( )**, and **atof( )**
  Convert a character array to an **int**, **long**, and **double**, respectively.

Example:

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

int main(int argc, char *argv[]) {
   for(int i = 1; i < argc; i++)
        cout << atoi(argv[i]) << endl;
   return 0;  }
```

TO DOs

- Homework #8 due Tuesday 10/25
- Lab #4
  - Due Monday, 10/24, at 8 am
- Lab #5 will be posted by the end of the weekend

</LECTURE>