void Functions
Call-by-Reference Functions
Overloading Functions

CS 16: Solving Problems with Computers I
Lecture #7

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Midterm will be graded by next week.**

- **Homework #6 due today**
- Lab #3 is due Monday, 5/1
- Lab #4 is due Tuesday, 5/2

- **Don't forget your TAs' and Instructor's office hours!! ☺**

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

```cpp
// 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
- It is fairly common to have no parameters in **void** functions
  - In this case there will be no arguments in the function call
- *Optional* **return** statement ends the function
  - Return statement does not include a value to return
  - Return statement is implicit if it is not included

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? <u>Stick to what's legal</u>: it's ALWAYS **int main ( )**

- 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? <u>Always</u> include **return 0** in the **main** function to be safe.

Call-by-Value vs Call-by-Reference

- When you call a function, your arguments are getting passed on as *values*
  – At least, with what we've seen so far…
  – The call `func(a, b)` passes on the *values* of **a** and **b**

- You can also call a function with your arguments used as *references* to the actual variable location in memory
  – Why would we want to do that?

Call-by-Reference Parameters

- "Call-by-reference" parameters allow us to change the **variable** used in the function call
  - "Normally", you wouldn't change the variable in the parameter (this is called call-by-value)

- Note: Arguments for call-by-reference parameters must be variables, not numbers
  - i.e. fn(var), not fn(5)

- We use the ampersand symbol (**&**) to distinguish a variable as being called-by-reference, in a function definition

Call-by-Reference Parameters

- Recall that, up until now, we have changed the values of formal parameters in a function body, **but we have not changed the arguments found in the function call!**
  - Example: when you call **func(a, b, c)**, you might get a returned value for **func**, but **a**, **b**, and **c** do not change after the call.

- So if you want to get an input (via `cin`) <u>inside</u> a function using call-by-value, it wouldn't "stick" outside that function!

Call-by-Reference Example

```
void get_input(double& f_variable)
 {
   cout << " Convert a Fahrenheit temperature"
        << " to Celsius.\n"
        << " Enter a temperature in Fahrenheit: ";
    cin >> f_variable;
}
```

Now you can use f_variable's new value from the main()
(or from wherever the function got called)

- '**&**' symbol (ampersand) identifies **f_variable** as a call-by-reference parameter
  - Note: Has to be used in both declaration and definition!

Call-By-Reference Details

- Call-by-reference works almost as if the argument ***variable itself*** is substituted for the formal parameter, not the argument's ***value***

- In reality, it's the ***memory location of the argument variable*** that is given to the formal parameter
  - Whatever is done to a formal parameter in the function body, is ***actually done to the value at the memory location of the argument variable***

Call-by-Reference Behavior

- Assume variables **first** and **second** are assigned memory addresses **1010** and **1012** by the compiler, respectively

- Now a function call executes: **get_numbers(first, second);**
- The function is defined as:

```
void get_numbers(int& first, int& second) {
    cout << “Enter two integers: ”
    cin >> first >> second; }
```

- The function may as well say:

```
void get_numbers(the int variable at memory location 1010,
the int variable at memory location 1012) {
    cout << “Enter two integers: ”
    cin >> the variable at memory location 1010;
       >> the variable at memory location 1012; }
```

Call by-Reference vs by-Value

- **<u>Call-by-reference</u>**
  - The function call:

f(age);

- **<u>Call-by-value</u>**
  - The function call:

f(age);

Memory

Name	Location	Contents
age	**1001**	**34**
initial	1002	A
hours	1003	23.5
	1004	

void f(int& ref_par);

void f(int var_par);

Example: swap_values

```
void swap(int& variable1, int& variable2)
{
     int temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
```

- If called with **`swap(first_num, second_num);`**
  - The values of **`first_num`** and **`second_num`** are swapped
  - Can ONLY do this if the function is call-by-reference

Mixed Parameter Lists

- Call-by-value and call-by-reference parameters **<u>can</u>** be mixed in the same function

- Example:

```
void good_stuff(int& par1, int par2, double& par3);
```

  - par1 and par3 are call-by-reference formal parameters
    - Changes in par1 and par3 change the argument variable

  - par2 is a call-by-value formal parameter
    - Changes in par2 do not change the argument variable

Caution!
Inadvertent Local Variables

- Forgetting the ampersand (&) creates a call-by-value parameter
  - The value of the variable will not be changed

- The formal parameter (i.e. when called-by-value) is a local variable **that has no effect outside the function**
  - Hard error to debug/find… because it looks right!

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

Do not use the same function name for unrelated functions

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

```
double average(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 << average( 10, 20, 30);
```

the compiler knows to use the second definition

Overloading Details

- Overloaded functions
  - Must have ***different numbers*** of formal parameters, but Must all be the same type
    - e.g.: **double average(int a, int b)** vs**. double average(int a, int b, int c)**

OR

  - They can have the same number of parameters, but Must have at least one of them be of a ***different type***
    - e.g.: **void print(int a)** vs. **void print(double a)** vs. **void print(char a)**

- You can not overload function declarations that differ only by return type.

Example from Textbook, Ch. 4

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
```

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 get 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)

</LECTURE>