Pre- and Programmer-Defined Functions
&
Procedural Abstraction

CS 16: Solving Problems with Computers I
Lecture #7

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- **Homework #6 due today**
- **Lab #3 is due on Friday AT NOON!**
- Homework Solutions are now online at:
  http://cs.ucsb.edu/~zmatni/cs16/hwSolutions/
- Grades (finally) up on GauchoSpace!
  - With caveats…

More Announcements

- Please note that 2 of the TAs have amended office hours:

Magzhan Zholbaryssov	Tue. 8-10 am
Dasha Rudneva	Thu. 4-7 pm

- The syllabus is updated to reflect this

MIDTERM IS COMING!

- Material: ***<u>Everything</u>*** we've done**, incl. up to Th. 10/13**
  - Homework, Labs, Lectures, Textbook

- **Tuesday, 10/18** in this classroom
- **Starts at 2:00pm **SHARP****
- **I will chose where you sit!**
- Duration: **1 hour long**
- Closed book: no calculators, no phones, no computers
- Only 1 sheet (single-sided) of written notes
  - Must be no bigger than 8.5" x 11"
  - You have to turn it in with the exam
- **You will write your answers on the exam sheet itself.**

Lecture Outline

- More about Pre-Defined Functions in C++
  - Type casting
- Programmer-Defined Functions in C++
- Procedural Abstraction

Type Casting

- Recall the problem with integer division in C++:

```
int total_candy = 9, number_of_people = 4;
double candy_per_person;
candy_per_person = total_candy / number_of_people;
```

  – candy_per_person = 2,   not   2.25!

- A **Type Cast** produces a value of one type from another
  – **static_cast<double>(total_candy)**
  produces a double representing
  the integer value of total_candy

Type Cast Example

```
int total_candy = 9, number_of_people = 4;
double candy_per_person;
candy_per_person =
    static_cast<double>(total_candy)/number_of_people;
```

- **candy_per_person now is 2.25!**

- The following would also work:

```
candy_per_person =
        total_candy / static_cast<double>(number_of_people);
```

- This, however, would not!

```
candy_per_person = static_cast<double>
                        (total_candy / number_of_people);
```

Integer division occurs before type cast!

Question

- Can you determine the value of d?

double d = 11 / 2;

- What about this value of d?

double d = 11.0 / 2.0;

Programmer-Defined Functions

- 2 components of a function definition
  - **Function declaration** (or function prototype)
    - Shows how the function is called from main() or other functions
    - Declares the type of the function
    - Must appear in the code *before* the function can be called
    - Syntax:

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

;

Only needed for declaration statement

  - **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
   }
```

Function Declaration

- Declares:
  - The return type
  - The name of the function
  - How many arguments are needed
  - The types of the arguments
  - The formal parameter names
    - Formal parameters are like placeholders for the actual arguments used when the function is called
    - Formal parameter names can be any valid identifier

- Example:

```
double total_cost(int number_par, double price_par);
// Compute total cost including 5% sales tax on
// number_par items at cost of price_par each
```

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task
- Example:

function header

```
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = price_par * number_par;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

The Return Statement

- Ends the function call
- Returns the value calculated by the function
- Syntax:

```
return expression;
```

  - expression performs a calculation
    or
  - expression is a variable containing the calculated value

- Example:

```
return subtotal + subtotal * TAX_RATE;
```

The Function Call

- Tells the name of the function to use
- Lists the arguments
- Is used in a statement where the returned value makes sense
- Example:
```
double bill = total_cost(number, price);
```

A Function Definition (*part 1 of 2*)

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

double total_cost(int number_par, double price_par);
//Computes the total cost, including 5% sales tax,
//on number_par items at a cost of price_par each.

int main()
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = total_cost(number, price);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

function declaration

function call

function heading

function body

function definition

Function Call Details

- The values of the arguments are plugged into the formal parameters
  - Call-by-value mechanism with call-by-value parameters
- The first argument is used for the first formal parameter,
  the second argument for the second formal parameter,
  and so forth.
- The value plugged into the formal parameter is used in all instances of the formal parameter in the function body
- **In other words, make sure everything matches, esp. your data types!**

Alternate Declarations

- There are two forms for function declarations
  - List formal parameter names
  - List types of formal parameters, but not their names
  - The 1st aids the description of the function in comments

- Examples:

```
double total_cost(int number_par, double price_par);
```

vs.

```
double total_cost(int, double);
```

- **Function headers**, however, must **always** list formal parameter names!

Order of Arguments

- Compiler checks that the types of the arguments are correct and in the correct sequence
  - Typical compile errors occur when we don't pay attention to detail…

- Compiler cannot check that arguments are in the correct logical order

- Example: Consider this function declaration – ***where's the error?***

```
char grade(int received_par, int min_score_par);

int received = 95,  min_score = 60;

cout <<  grade( min_score, received);
```

This produces a faulty result because the **arguments are not in the correct logical order**. The compiler will not catch this!

Function Definition Syntax

Within a function definition:

- Variables must be declared before they are used
- Variables are typically declared before the executable statements begin
- At least one return statement must end the function
  - Each branch of an if-else statement might have its own return statement

Syntax for a Function That Returns a Value

Function Declaration

Type_Returned Function_Name (*Parameter_List*) ;
Function_Declaration_Comment

Function Definition

```
Type_Returned  Function_Name(Parameter_List)  <-- function header
{
        Declaration_1
        Declaration_2
              . . .
        Declaration_Last
        Executable_Statement_1
        Executable_Statement_2
              . . .
        Executable_Statement_Last
}
```

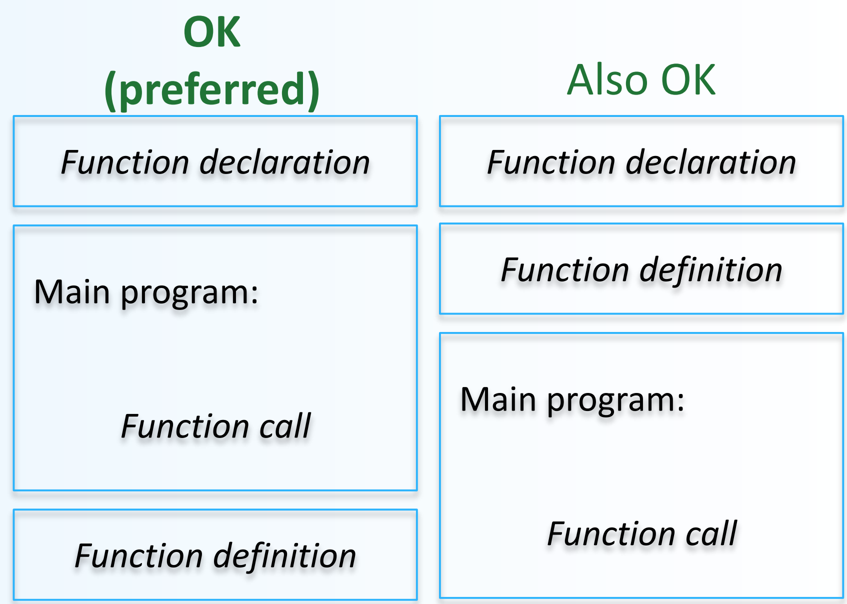
body

Must include one or more `return` *statements.*

Placing Definitions

- A function call must be preceded by either
  - The function's declaration
    or
  - The function's definition
    - If the function's definition precedes the call, a declaration is not needed

- *ProTip*:
  Placing the function declaration *prior* to the **main** function and the function definition *after* the **main** function leads naturally to building your own libraries in the future



bool Return Values

- A function can return a Boolean value
  - Such a function can be used where a Boolean expression is expected
    - Makes programs easier to read

- Compare

```
if (((rate >=10) && ( rate < 20)) || (rate == 0))
```

to

```
if (appropriate (rate))
```

  - Which is easier to read!?
    - This works assuming, of course, that function **appropriate** returns a bool value based on the expression above

Function appropriate

- To use function appropriate in the if-statement

```
if (appropriate (rate))
{    …    }
```

appropriate could be defined as

```
bool appropriate(int rate)
{
return (((rate >=10) && ( rate < 20)) || (rate == 0));
}
```

Black Box Abstraction

- A “black box” refers to something that we know how to use, but the method of its internal operation is unknown

- A person *using* a program does not need to know how it is coded

- A person *using* a program needs to know what the program does, not *how* it does it

Procedural Abstraction and C++

- Procedural Abstraction is writing and using functions as if they were “black boxes”

- Procedure is a general term meaning a “function like” set of instructions

- Abstraction implies that when you use a function as a “black box”, you abstract away the details of the code in the function body

Procedural Abstraction and Functions

- Write functions so the declaration and comment is all a programmer needs to use the function
- Function comment should tell all conditions required of arguments to the function
- Function comment should also describe the returned value
- Variables used in the function, other than the formal parameters, should be declared in the function body

Formal Parameter Names

- Functions are designed as **self-contained modules**
- Different programmers may write each function
- Programmers should choose meaningful names for formal parameters
  - i.e. avoid generic parametric names like “x”, or “number”, if possible
  - Formal parameter names may or may not match variable names used in the main part of the program
  - BUT! That does not matter!
- Remember that **only the value of the argument** is plugged into the formal parameter

A Function Definition (*part 1 of 2*)

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

double total_cost(int number_par, double price_par);
//Computes the total cost, including 5% sales tax,
//on number_par items at a cost of price_par each.

int main()
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = total_cost(number, price);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

function declaration

function call

function heading

function body

function definition

Example Function: Factorial

- **n!** Represents the factorial function
- n! = 1 x 2 x 3 x … x n
- We need this function to:
  - Require one argument of type int, call it "n"
  - Return a value of type int
  - Use a local variable to store the running product
  - Decrement n each time it does another multiplication:

n * n-1 * n-2 * … * 1

```
#include <iostream>

using namespace std;

int main(){

    int n(0);
    int factorial (int n);
    //Returns the factorial of input n (must be non-negative)

    cout << "Enter a number: " << endl;
    cin >> n;

    cout << "The factorial of this number is: " << endl << factorial(n) << endl;

    return 0;
} // end main()


int factorial (int k)
{
    int product = 1;
    while (k > 0)
    {
        product *= k;
        k--;
    }

    return product;
} // end factorial()
```

Global Constants

- Global Named Constant
  - Available to more than one function as well as the main part of the program
  - Declared outside any function body
  - Declared outside the main function body
  - Declared before any function that uses it

- Example:

```
const double PI = 3.14159;
double volume(double);
    int main()
        {…}
```

  - PI is available to the main function and to function **volume**

Global Variables

- Rarely used
- When more than one function must use a common variable
- Declared just like a global constant except **const** is not used
- Generally make programs more difficult to understand and maintain, so it’s not considered “good practice”

Formal Parameters are Local Variables

- Formal parameters are actually variables that are local to the function definition
  - They are used just as if they were declared in the function body
  - Do NOT re-declare the formal parameters in the function body, they are declared in the function declaration
- When a function is called, the formal parameters are initialized to the values of the arguments in the function call

Block Scope

- Local and global variables conform to the rules of Block Scope
- The code block (generally defined by the **{ }**) where an identifier like a variable is declared determines the scope of the identifier
- Blocks can be nested

Block Scope

Block Scope Revisited

```
1   #include <iostream>
2   using namespace std;
3
4   const double GLOBAL_CONST = 1.0;
5
6   int function1 (int param);
7
8   int main()
9   {
10      int x;
11      double d = GLOBAL_CONST;
12
13      for (int i = 0; i < 10; i++)
14      {
15          x = function1(i);
16      }
17      return 0;
18  }
19
20  int function1 (int param)
21  {
22      double y = GLOBAL_CONST;
23      ...
24      return 0;
25  }
```

Local and Global scope are examples of Block scope. A variable can be directly accessed only within its scope.

Block scope: Variable `i` *has scope from lines 13-16*

Local scope to `main`*: Variable* **x** *has scope from lines 10-18 and variable* **d** *has scope from lines 11-18*

Global scope: The constant **GLOBAL_CONST** *has scope from lines 4-25 and the function* `function1` *has scope from lines 6-25*

Local scope to `function1`*: Variable* `param` *has scope from lines 20-25 and variable* **y** *has scope from lines 22-25*

The Benefits of Namespace

```
#include <iostream>
#include <cmath>

//using namespace std;

int main(){

    int d(1),e(23);
    std::cout << "d " << d << " e " << e << std::endl;

    int f(3), f2(0);
    f2 = std::pow(f,2);
    std::cout << "f squared is: " << f2 << std::endl;

    return 0;
}
```

The calls for **cout** and **endl** go to a block called **std** that is in the **iostream** library.
The calls for **pow()** go to a block called **std** that is in the **cmath** library.

Namespaces Revisited

- We will be eventually be using:

more namespaces than just **std**.

&

different namespaces in different function definitions.

Namespaces Revisited

- The start of a file is not always the best place for
  using namespace std;
- Different functions may use different namespaces
- Placing **using namespace std;** inside the starting brace of a function
  - Allows the use of different namespaces in different functions
  - Makes the "using" directive local to the function

TO DOs

- **<u>Study for your midterm!!!</u>**

- No homework due on Tuesday 10/18 !! !!omg!!!
  - **I will issue new homework at the start of next week that will be due on Thursday 10/20**

- Lab #3
  - Due Friday, 10/14, at noon

- Lab #4 will be posted by the end of the weekend
  - You still have lab on Monday 10/17
  - **The new lab, however, will be due on Monday 10/24 (not Friday 10/21! Yay!)**

</LECTURE>