CMPE 180-92

Data Structures and Algorithms in C++

August 31 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

San José State University

Basic Info

- Office hours
  - TuTh 3:00 – 4:00 PM
  - ENG 250

- Website
  - Faculty webpage: http://www.cs.sjsu.edu/~mak/
  - Class webpage: http://www.cs.sjsu.edu/~mak/CMPE180-92/
  - Syllabus
  - Assignments
  - Lecture notes

Assignment #1: Sample Solution

- First start with a “draft” of your program.
  - Test that you can read the individual fields of the input file.
- Then incrementally add to the draft until you have the complete solution.
  - Always build on working code.
- Do not attempt to write an entire program all at once and then try to get it to work.

Predefined Functions

- C++ includes predefined functions.
  - AKA “built-in” functions
  - Example: Math function `sqrt`
- Predefined functions are stored in libraries.
  - Your program will need to include the appropriate library header files to enable the compiler to recognize the names of the predefined functions.
  - Example: `#include <cmath>`
    in order to use predefined math functions like `sqrt`

Predefined Functions, *cont'd*

Savitch_ch_04.ppt: slides 8 – 12, 72

Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
`sqrt`	square root	*double*	*double*	`sqrt(4.0)`	2.0	cmath
`pow`	powers	*double*	*double*	`pow(2.0,3.0)`	8.0	cmath
`abs`	absolute value for *int*	*int*	*int*	`abs(-7)` `abs(7)`	7 7	cstdlib
`labs`	absolute value for *long*	*long*	*long*	`labs(-70000)` `labs(70000)`	70000 70000	cstdlib
`fabs`	absolute value for *double*	*double*	*double*	`fabs(-7.5)` `fabs(7.5)`	7.5 7.5	cmath
`ceil`	ceiling (round up)	*double*	*double*	`ceil(3.2)` `ceil(3.9)`	4.0 4.0	cmath
`floor`	floor (round down)	*double*	*double*	`floor(3.2)` `floor(3.9)`	3.0 3.0	cmath

Random Numbers

- To generate (pseudo-) random numbers using the predefined functions, first include two library header files:

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

- "Seed" the random number generator:

```
srand(time(0));
```

  - If you don't seed, you'll always get the same "random" sequence.

Random Numbers, *cont'd*

- Each subsequent call

```
rand();
```

returns a "random" number ≥ 0
and < **RAND_MAX**.

- Use **+** and **%** to scale to a desired number range.
  - Example: Each execution of the expression

```
rand()%6 + 1
```

returns a random number
with the value 1, 2, 3, 4, 5, or 6.

Type Casting

- Suppose integer variables `i` and `j` are initialized to 5 and 2, respectively.
- What is the value of the division `i/j` ?
- What if we wanted to have a quotient of type double?
  - We want to keep the fraction.

Type Casting, *cont'd*

- One way is to convert one of the operands (say `i`) to double.
  - Then the quotient will be type double.

```
double quotient = static_cast<double>(i)/j;
```

- Why won't the following work?

```
double quotient = static_cast<double>(i/j);
```

Programmer-Defined Functions

- In addition to using the predefined functions, you can write your own functions.
- Programmer-defined functions are critical for good program design.
- In your C++ program, you can call a programmer-defined function only after the function has been declared or defined.

Function Declarations

- A function declaration specifies:
  - The function name.
  - The number, order, and data types of its formal parameters.
  - The data type of its return value.

- Example:

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

Function Definitions

- After you've declared a function, you must define it.
  - Write the code that is executed whenever the function is called.
  - A **return** statement terminates execution of the function and returns a value to the caller.

- Example:

```
double total_cost(double unit_cost, int count)
{
    double total = count*unit_cost;
    return total;
}
```

Function Calls

- Call a function that you wrote just as you would call a predefined function.
- Example:

```
int how_many;
double how_much;
double spent;

how_many = 5;
how_much = 29.99;
spent = total_cost(how_much, how_many);
```

Void Functions

- A void function performs some task but does not return a value.

- Therefore, its **`return`** statement terminates the function execution but does not include a value.
  - A return statement is not necessary for a void function if the function terminates “naturally” after it finishes executing the last statement.

- Example void function definition:

```
void print_TF(bool b)
{
    if (b) cout << "T";
    else   cout << "F";
}
```

Void Functions, *cont'd*

- A call to a void function cannot be part of an expression, since the function doesn't return a value.
- Instead, call a void function as a statement by itself.
- Example:

```
bool flag = true;
print_TF(flag);
```

Top-Down Design, *cont'd*

- Top-down design is an important software engineering principle.
- Start with the topmost subproblem of a programming problem.
  - Write a function for solving the topmost subproblem.
- Break each subproblem into smaller subproblems.
  - Write a function to solve each subproblem.
  - This process is called stepwise refinement.

Top-Down Design, *cont'd*

- The result is a hierarchical decomposition of the problem.
- AKA functional decomposition

Top-Down Design Example

- Write a program that inputs from the user that are positive integer values less than 1000.
- Translate the value into words.
- Example:
  - The user enters 482
  - The program writes "four hundred eighty-two"
- Repeat until the user enters a value ≤ 0.

Top-Down Design Example, *cont'd*

- What is the topmost problem?
  - Read numbers entered by the user until the user enters a value ≤ 0.
  - Translate each number to words.
- How to translate a number into words?
  - Break the number into separate digits.
  - Translate the digits into words such as *one*, *two*, ..., *ten*, *eleven*, *twelve*, ..., *twenty*, *thirty*, etc.

Refinement 1

- Loop to read and print the numbers.
- Call a translate function,
  but it doesn't do anything yet.

A Convention for Functions

- Put function declarations before the main.
  - If you give your functions good names, the declarations show the structure of your program.
- Put function definitions after the main.

Refinement 2

- Refine the translate function to handle some simple cases:
  - **`translateOnes`**: 1 through 9
  - **`translateTeens`**: 11 through 19

Refinement 3

- The translate function takes a 3-digit number and separates out the hundreds digit.
- Translate the hundreds digit.
  - `translateHundreds`
  - Do this simply by translating the hundreds digits as we did a ones digit, and append the word *hundred*.

Refinement 3

- Translate the last two digits:
  - We can already translate a teens number.
  - Otherwise, break apart the two digits into a tens digit and a ones digit.
    - **`translateTens`**: 10, 20, 30, ..., 90
    - We can already translate a ones digit.

Refinement 4

- Add a hyphen between *twenty*, *thirty*, etc. and a ones word.

Refinement 5

- Break a 6-digit number into a 3-digit first part and a 3-digit second part.
- Translate the first part and append the word *thousand*.
- Translate the second part.

Break

Scope and Local Variables

- Any variable declared inside a function is local to that function.
  - The scope of the variable is that function.
  - The variable is not accessible from outside the function.
  - A variable with the same name declared inside another function is a different variable.
- The same is true for any variable declared inside the main function.

Block Scope

- You can declare variables inside of a block.
  - A block of code is delimited by `{` and `}`.
- The variables are local to the block.

Global Constants and Variables

- If a constant or a variable is declared outside of and before the main and the function definitions, then that constant or variable is accessible by the main and any function.

- Global variables are not recommended.

Overloading Function Names

- A function is characterized by both its name and its parameters.
  - Number and data types of the formal parameters.

- You can overload a function name by defining another function with the same name but different parameters.
  - When you call a function with that name, the arguments of the call determine which function you mean.

Overloading Function Names, *cont'd*

- Example declarations:

```
double average(double n1, double n2);
double average(double n1, double n2, double n3);
```

- Example calls:

```
double avg2 = average(x, y);
double avg3 = average(x, y, z);
```

- Be careful with automatic type conversions of arguments when overloading function names.
  - See the Savitch text and slides.

Call-by-Value

- By default, arguments to a function are passed by value.
- A copy of the argument's value is passed to the function.
- Any changes that the function makes to the parameters do not affect the calling arguments.
  - Example: The faulty swap function.

Call-by-Value, *cont'd*

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- Why doesn't this function do what was intended?

Demo

Call-by-Reference

- If you want the function to be able to change the value of the caller's arguments, you must use call-by-reference.

- The address of the actual argument is passed to the function.
  - Example: The proper exchange function.

Call-by-Reference, *cont'd*

```
void exchange(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- Why is this code better?

```
void exchange(int& a, int& b)
```

Demo

Procedural Abstraction

- Design your function such that the caller does not need to know how you implemented it.
- The function is a “black box”.

Procedural Abstraction, *cont'd*

- The function's name, its formal parameters, and your comments should be sufficient for the caller.
- Preconditions: What must be true when the function is called.
- Postconditions: What will be true after the function completes its execution.

Testing and Debugging Functions

- There are various techniques to test and debug functions.
- You can add temporary `cout` statements in your functions to print the values of local variables to help you determine what the function is doing.
- With the Eclipse or the NetBeans IDE, you can set breakpoints, watch variables, etc.

assert

- Use the `assert` macro during development to check that a function's preconditions hold.
  - You must first `#include <cassert>`
  - Example:

```
assert(y != 0);
quotient = x/y;
```

- Later, when you are sure that your program is debugged and you are going into production, you can logically remove all the asserts by defining `NDEBUG` before the include:

```
#define NDEBUG
#include <cassert>
```

Assignment #2: Functional Decomposition

- Practice decomposing a program top-down by using functions.

- The solution for Assignment #1, as suggested by the program outline in CodeCheck, was long main containing much duplicated code.

- For Assignment #2, write a new version of the program, but this time with user-defined functions.

Assignment #2: *cont'd*

- The resulting program should have a hierarchical decomposition.
- Choose good function names and use parameters wisely.
- Your final program should be have correct output and be easy to read.
- The official assignment write-up will appear in Canvas tomorrow.

Week 2 Practice Problems

- Look for practice problems in Canvas.
- They should appear in a day or two.