Ch05-UserDefinedFunctions

August 7, 2020

1   5 User Defined Functions

1.1   Topics

- adding new functions
- using functions
- function types
- ways to pass data to functions
- function prototypes and their purpose
- debugging and automatically testing functions
- scopes and name resolution process

1.2   5.1 Adding new functions

- we've used a several functions provided in various standard libraries
- **main( )** is a function that is required for any C++ program
    - indicates where the execution of the program begins
- one can add as many functions as required in the program
- function is a block of code that execute as a group
- the ultimate goal of the function is to break the problem into smaller sub-problems
- if you have many tasks in algoritm steps, function can be used to solve a particular task/step
- using function is a two-step process
    1. define a function
    2. call or use the function

1.2.1   Why functions?

- dividing a program into functions or sub-programs have several advantages
    - each sub-problem can be solved using a function
    - makes it easier to design a solution to a big problem
    - makes the solution modular
    - helps reuse the code
        * function once defined can be used many times!
    - makes the solution concise
    - helps test and update a part of program without affecting the rest
    - makes it easier to work in a team where each member can focus on a function

1.2.2 syntax to define a function

```
type NAME ( ) {
    // body
}
```

- new function name conventions:
    - can't name it main()
    - can't reuse identifiers or keywords
    - same as variable naming conventions!

1.2.3 syntax to call a function

- functions are called by their name

```
NAME();
```

- typically functions are defined outside **main( )** and called inside **main( )**
- however, one function can call another function
- the function that calls another function is called **caller**
- the function being called is called **callee**

```
[3]:  #include <iostream>
      using namespace std;
```

```
[28]:  //1. define a function that prints Hello World!
       void sayHello() {
           cout << "Hello World!" << endl;
       }
```

```
[29]:  // 2. call sayHello
       sayHello();
```

```
Hello World!
```

```
[30]:  // define a function that prints Hello World! three times
       // caller function
       void sayHelloThrice() {
           // caller calls other functions
           sayHello(); // callee
           sayHello(); // callee
           cout << "Hello World!" << endl;
       }
```

```
[31]:  // call sayHelloThrice
       sayHelloThrice();
```

```
Hello World!
Hello World!
Hello World!
```

1.3 5.2 Parameters and arguments

- we've used some built-in functions that take arguments
  - float(val), int(val), char(val), setw(arg), etc.
- parameter is way to send/pass data to function so the function can do its job
  - placeholders for data to be copied into
  - also called **formal parameters**
- in algebra we may define equations as:

```
y = f(x) = x^2 + x + 2
find y if x = 1: f(1) = 1^2 + 1 + 2 = 4
find y if x = -10: f(-10) = (-10)^2 + (-10) + 2 = 92
```

- programming functions burrow the notion of algebraic functions
- parameters help us define generic functions that computes answer based on the given data value
- the data/value that is passed into the parameter is called argument
- x is parameter and 1 and -10 are arguments
- functions with parameters are more useful because they can work on varities of different values
- syntax to define function with parameters:

```
type NAME(type PARAMETER1, type PARAMETER2, ...) {
    // body
}
```

- parameters are variables (have types and names)
  - eventually will have values of same type when the function is called
- syntax to call function with parameters:

```
NAME(argument1, argument2, ...);
```

- function must be called passing the same number of arguments as the no. of parameters
- types of arguments and parameters must match correspondingly left to right
- arguments can be literal values or variables
- by default values of arguments are copied to corresponding parameters
  - value/argument1 copied to PARAMETER1
  - value/argument2 copied to PARAMETER2, etc.
- arguments passed to functions are also called **actual parameters**
  - represent the actual data that are acutally passed

```
[32]:  // define a function that greets a person by their name
       #include <string>

       // name is the only parameter of type string
       void greeting(string name) {
           cout << "Hello there, " << name << endl;
       }
```

```
[33]:  // wrong way... calling greeting without argument will generate error!
       greeting();
```

```
input_line_58:3:1: error: no matching function for call
to 'greeting'
greeting();
^~~~~~~~

input_line_57:2:6: note: candidate function not viable:
requires single argument 'name', but no arguments were provided
void greeting(string name) {
     ^
```

Interpreter Error:

[34]: *// right way: calling greeting with one string argument*
greeting("John"); *// passing literal value*

Hello there, John

[35]: *// wrong way to call greeting passing wrong type of argument*
greeting(123);

```
input_line_60:3:1: error: no matching function for call
to 'greeting'
greeting(123);
^~~~~~~~

input_line_57:2:6: note: candidate function not viable:
no known conversion from 'int' to 'std::__1::string' (aka 'basic_string<char,
      char_traits<char>, allocator<char> >') for 1st argument
void greeting(string name) {
     ^
```

Interpreter Error:

[36]: *// calling greeting with passing the data in variable*
// name of the argument has nothing to do with name of the parameter
// only the type needs to match!
string somename;

[37]: somename = "Jake";
greeting(somename); *// access the value of somename and pass/copy to greeting*

Hello there, Jake

4

```
[38]: // define a function that takes two numbers and prints the sum as result
      void sum(int num1, int num2) {
          long total = num1 + num2;
          cout << num1 << " + " << num2 << " = "  << total << endl;
      }
```

```
[39]: // call sum passing to int values
      sum(10, 20);
```

```
10 + 20 = 30
```

1.4   5.3 Types of functions

- functions can be roughly divided into two types:
    1. void functions or fruitless functions
        - functions that do not return any value
        - all the functions defined previously in this notebook are void functions
        - NOTE: printing result/value is NOT the same as returning value
    2. type functions or fruitful functions
        - functions that return some value
- syntax of fruitful functions

```
type NAME(type PARAMETER1, ...) {
    // body
    return someValue;
}
```

- type of the return value must match the type of the function NAME
- fruitful parameterized functions are the most useful functions
    - can use the returned value however you want!
    - can automatically test the results from the functions
    - most library functions are fruitful and parameterized

```
[40]: // define a function that takes two numbers and returns the sum
      long find_sum(int num1, int num2) {
          long total = num1 + num2;
          return total;
      }
```

```
[41]: // call function with values
      find_sum(12, 8);
      // where is the returned value or result?
```

```
[41]: 20
```

```
[42]: // you must find a way to use the returned value
      // print the returned value from find_sum function
      cout << find_sum(12, 8) << endl;
```

```
20
```

[43]:
```
// assign the returned value from find_sum(...) to a variable
long ans = find_sum(99, 1);
```

[44]:
```
// let's see the value of ans
ans
```

[44]: 100

1.5 5.4 Passing data/value or reference

- C++ provides two ways to pass data to functions
    1. pass by value
    2. pass by reference

pass by value - data of argument is copied into parameter - default way the data is passed as we've seen above - easier to understand; no side effect - slower to copy large amount of data - since the data is copied, anything done to the data via parameter doesn't affect the actual argument - if formal parameter is modified, actual parameter or argument is not modified!

[45]:
```
// function to demonstrate pass by value
// num1 and num2 are also called formal parameters
long anothersum(int num1, int num2) {
  num1 += 10; // we don't do this, but only to demonstrate pass by value
  num2 += 20;
  long total = num1 + num2;
  return total;
}
```

[47]:
```
// declare 2 variables to store integers
int n1, n2;
//long ans;
```

[48]:
```
n1 = 20;
n2 = 30;
// n1 and n2 are actual parameters or arguments
ans = anothersum(n1, n2);
cout << n1 << " + " << n2 << " = " << ans << endl;
// gives wrong answer becuase anothersum is not correctly implemented
// at least values of n1 and n2 are remain intact!
```

```
20 + 30 = 80
```

[48]: @0x11227dec0

1.5.1 visualize pass by value in pythontutor.com

1.5.2 pass by reference

- copying data is expensive/slow operation
    - avoid copying with **pass by reference** technique
- pass by reference occurs when the parameter has **&** symbol in-between the type and name
- syntax for pass by reference:

```
// function definition
type NAME(type & PARAMETER1, type & PARAMETER2,...) {
    // body
}
// function call
NAME(argument1, argument2, ...);
```

- data is not copied but the refernce is passed to function
    - meaning actual and formal parameter reference the same memory location
- ONLY variable arguments (NOT literal) can be passed to the reference parameters
- if the formal parameter is modified, argument or actual parameter is also modified!
- efficient, but may have unintended side effect
- provides a way to retrieve data from function!

[49]:
```
// function to demonstrate pass by reference
// num1 and num2 are also called formal parameters that are passed by reference
long yet_another_sum(int & num1, int & num2) {
  num1 += 10; // for whatever reason we modify formal parameters;
  num2 += 20; // could be a mistake
  long total = num1 + num2;
  return total;
}
```

[52]:
```
// can't pass literals for pass by reference parameters
yet_another_sum(20, 30)
```

input_line_81:3:1: error: no matching function for call

to 'yet_another_sum'
yet_another_sum(20, 30)
^~~~~~~~~~~~~~~

input_line_77:3:6: note: candidate function not viable:
expects an l-value for 1st argument
long yet_another_sum(int & num1, int & num2) {
     ^

          Interpreter Error:

```
[51]:  // n1 and n2 are already declared as integers above
       n1 = 20;
       n2 = 30;
       // n1 and n2 are actual parameters or arguments
       ans = yet_another_sum(n1, n2);
       cout << n1 << " + " << n2 << " = " << ans << endl;
       // gives right answer n1 and n2 values are modified
```

```
30 + 50 = 80
```

[51]: @0x11227dec0

1.5.3 constant parameters

- C++ allows constant parameters
  - prevents the parameters from being changed inside the function
  - thus, prevents the unwanted sideeffects
- constant parameters are READ ONLY

```
[1]:  long sum_with_const_parameters(const int & n1, const int n2){
          // modifying n1 and n2 constants are not allowed
          n1 = 10;
          n2 = 20
          return n1 + n2;
      }
```

```
input_line_7:3:8: error: cannot assign to variable 'n1'

with const-qualified type 'const int &'
    n1 = 10;
    ~~ ^

input_line_7:1:44: note: variable 'n1' declared const
here
long sum_with_const_parameters(const int & n1, const int n2){
                               ~~~~~~~~~~~~^~

input_line_7:4:8: error: cannot assign to variable

'n2' with const-qualified type 'const int'
    n2 = 20
    ~~ ^

input_line_7:1:58: note: variable 'n2' declared const
here
long sum_with_const_parameters(const int & n1, const int n2){
                                                ~~~~~~~~~~^~
```

```
        Interpreter Error:
```

1.5.4 retrieve data from function

- pass by reference can be used to retrieve values/answers from functions

```
[30]: // extracting value from a function with passed by reference technique
      // num1 is passed by value num2 and sum are passed by reference
      void computeSum(int num1, int& num2, long &sum) {
          sum = num1 + num2; // sum is modified
          // notice void function, no return value!
      }
```

```
[32]: n1 = 100;
      n2 = 300;
      ans = 0; // initilize ans to 0; just in case!
      // n1, n2, and ans are actual parameters or arguments
      computeSum(n1, n2, ans);
      cout << n1 << " + " << n2 << " = " << ans << endl;
```

```
100 + 300 = 400
```

[32]: @0x107073ec0

```
[1]: // define a function that computes and returns both area and perimeter
     // C++ doesn't allow returning multiple values from a function
     void area_and_perimeter(const float & l, const float & w, float & area, float &␣
      ↪peri) {
         area = l*w;
         peri = 2*(l+w);
     }
```

```
[2]: float len = 4.5;
     float width = 3.99;
     float area, perimeter;
```

```
[3]: area_and_perimeter(len, width, area, perimeter);
```

```
[4]: area
```

[4]: 17.9550f

```
[5]: perimeter
```

[5]: 16.9800f

1.5.5 visualize pass by reference in pythontutor.com

1.6 5.5 Function prototype

- we know functions must be defined before they're called
- function signature or prototype is used to tell the compiler that the actual function will be defined later
- helps us write the main function towards the top of the source file
- define the actual functions after main without having to worry about the order of definitions
- syntax of function prototype

```
type NAME(type, type, ...);
```

- only the function type, name and parameter types matter
- can provide prameter names but are meaningless
- NOTE: function prototype demostration doesn't work in Jupyter
- see demo_programs/Ch05/func_prototype.cpp

1.7 5.6 Debugging and Unittesting with assert( )

- if a problem is broken into sub-problems, it can be easily debugged
- functions can be used to solve sub-problems that can be designed and tested independently
- testing code at the functional level is called unittesting
- we'll learn the basics of unittesting by automatically testing functions
- 3rd party frameworks such as googletest is used for comprehensive unittesting
- we use **assert()** macro function provided in **<cassert>** library
- syntax to use assert

```
assert(condition);
```

- condition expression is created comparing two data values using **==** operator
  - more on comparison operator in conditionals chapter
- if condition evaluates to true (two values are indeed equal); assertion passes
  - otherwise, assertion fails and the program halts immidiately
- more on assert: https://en.cppreference.com/w/cpp/error/assert
- **NOTE: assert( ) doesn't work in Jupyter notebook**; see the following demo programs

1.7.1 debugging demo

- debugging with assert function `demo_programs/Ch05/assertdebug.cpp` or at repl.it

1.7.2 unittesting demos

- using this concept, we can automatically test if the returned results from functions are correct or not

- see unittesting function with assert here `demo_programs/Ch05/unittesting.cpp` or at repl.it

- see improved version 2.0 unittesting example `demo_programs/Ch05/unittesting_v2.cpp`

1.8 5.7 Function overloading and templates

- often times we do similar operations on different data types

- adding numbers are not same as adding integers or strings, e.g.
- e.g. we want to add two numbers using functions called add( )
  - add(10, 20) // adding two integers
  - add(20.5, 5) // adding float and int
  - add(5.6f, 6.1f) // adding two floats
  - add(56.66, 4646.444) // adding two doubles
- two solutions:
  1. function overloading
  2. function templating

[5]:
```
// just this function doesn't correctly add all data types
int addNumbers(int n1, int n2) {
    return n1+n2;
}
```

[6]:
```
// correct answer!
addNumbers(10, 20)
```

[6]: 30

[8]:
```
// wrong answer!
addNumbers(10.5f, 30)
```

```
input_line_18:3:12: warning: implicit conversion from

'float' to 'int' changes value from 10.5 to 10 [-Wliteral-conversion]
addNumbers(10.5f, 30)
~~~~~~~~~~ ^~~~~
```

[8]: 40

1.8.1 function overloading

- functions can be redefined as long as the prototypes are different
  - same name can be used with different types

[9]:
```
// function overloading for float type
float addNumbers(float n1, float n2) {
    return n1+n2;
}
```

[10]:
```
// overaloading for double type
double addNumbers(double n1, double n2) {
    return n1 + n2;
}
```

```
[15]:  // now we can add two floating point numbers
       addNumbers(10.5f, 30.0f)
```

```
[15]:  40.5000f
```

```
[17]:  // we can also add two doubles
       addNumbers(10.1, 8.9)
```

```
[17]:  19.000000
```

```
[18]:  // how about adding int and float?
       // compiler gets confused, as we don't have addNumbers defined for int and float
       addNumbers(5, 10.5)
```

input_line_32:3:1: error: call to 'addNumbers' is

ambiguous
addNumbers(5, 10.5)
^~~~~~~~~~

input_line_11:2:5: note: candidate function
int addNumbers(int n1, int n2) {
    ^

input_line_21:2:8: note: candidate function
double addNumbers(double n1, double n2) {
      ^

input_line_20:2:7: note: candidate function
float addNumbers(float n1, float n2) {
     ^

Interpreter Error:

1.8.2 Function templates

- ways to create functions that work with **generic types**
- better approach as we write one template function that works for any types
    - without having to repeat the function for each type
- can be achieved using **template parameters**
- template parameter allows us to pass types to a function
- syntax to define a template function:

```
template <class type1, class type2,...>
type1 functionName(type1 PARAMETER1, type2 PARAMETER2, ...) {
    // body
```

```
}
```

- type1 and type2 in definitions are identifiers
  - you can choose any names
- syntax to call template function:

```
functionName<type1, type2...>(arg1, arg2, ...);
```

- type1 and type2 in function call are actual data types that C++ understands

[7]:
```cpp
// define a function that adds two values
// let's assume T2 is the bigger type if T1 and T2 are differnt types
template<class T1, class T2>
T2 addTwo(T1 val1, T2 val2) {
    return val1 + val2;
}
```

[8]:
```cpp
// call addTwo providing types and arguments
addTwo<int, int>(10, 5)
```

[8]: 15

[9]:
```cpp
addTwo<int, float>(5, 10.9)
```

[9]: 15.9000f

[10]:
```cpp
addTwo<int, double>(56, 99.90)
```

[10]: 155.90000

[11]:
```cpp
// can also add two chars
addTwo<char, char>('A', 'A')
```

[11]: '0x82'

[26]:
```cpp
// let's add two strings
//#include <string>
//using namespace std;
addTwo<string, string>("hello", "world")
```

[26]: "helloworld"

1.9   5.8 Scope and name resolution

- scope is the area of visibility of identifiers such as variables and constants within the program
  - scope determines where the variables and identifiers are valid
- scopes can be conceptualized as box frames
- C++ provides 3 common types of identifiers' scopes

1. global scope
2. local scope
3. block scope

1.9.1 global scope

- any identifiers (variables, constants, functions) decalared outside any function have global scope
- almost all functions have global scopes
- can be used anywhere in the program

1.9.2 local scope

- any variables and constants defined inside function
  - including parameters have local scope
- local identifiers can be used anywhere within the function scope

1.9.3 block scope

- smallest region that can is enclosed inside curly braces { }
- any identifiers declared inside curly braces can be used only withing that block
- very limited scope and rarely used

1.9.4 name resolution

- program needs to know and resolve what different names are

- where they're declared and their scope

- compiler first looks into the current scope or fame then the frame it's immidiately inside and so on until global frame to try to resolve any name used in the program

- the following figure depicts the various scopes and name resolution in C++

1.9.5 visualize identifiers' scopes in pythontutor.com

1.10 5.9 Exercises

1. Kattis hello problem
   - must solve the hello problem: https://open.kattis.com/problems/hello using a function and a test case.
   - a sample solution can be found here `demo_programs/Ch05/helloworld.cpp`
2. Write a C++ program including algorithm steps that calculates area and circumference of a circle.
   - must write functions to compute area and perimeter and automatically test each function with atleast 3 test cases
3. Write a C++ program including algorithm steps that calculates Body Mass Index (BMI) of a person.
   - must use as many functions as possible
   - write at least 3 test cases for each function
   - more info on BMI - https://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm

- Formula here.
- a sample solution is provided at `demo_programs/Ch05/BMI_v2.cpp`
4. Write a C++ program including algorithm steps that calculates area and perimeter of a triangle given three sides.
   - must write and use separate functions to calculate area and perimeter
   - write at least 3 test cases for each function
   - Hint: use Heron's formula to find area with three sides.
5. Write a C++ program that converts hours into seconds.
   - must write and use function(s) to computer answer(s)
   - must write at least 3 test cases for each function
   - e.g. given 2 hours, program should print 7200 as answer.
6. Write a C++ program that converts seconds into hours, minutes and seconds.
   - must define and use function(s)
   - write at least 3 test cases for each function
   - sample input: 3600 sample output: 1 hour, 0 minute and 0 second
   - sample input: 3661 sample output: 1 hour, 1 minute and 1 second
   - Hint: use series of division and module operations

1.11 5.10 Kattis Problems

- functions are not required to solve Kattis problems
- however, it's best practices to use function to learn to be able to sovle hard and large problems

1. Hello World!: https://open.kattis.com/problems/hello
   - solve the Hello World! problem using function(s)
2. Solving for Carrots - https://open.kattis.com/problems/carrots
   - solve the problem using function(s)

1.12 5.11 Summary

- this chapter covered concepts on functions; how to create new functions and use them
- went over various types of functions (fruitful and fruitless)
- learned about why and how to pass data to functions
- learned about debugging using assert(), cerr (stderr stream)
- learned about important foundation concept of automatically testing functions
- covered function overloading and templating
- finally, exercises and sample solutions

[ ]: