Lecture #6 Functions

chapter: 6

Opening Problem

Find the sum of integers from <u>1</u> to <u>10</u>, from <u>20</u> to <u>37</u>, and from <u>35</u> to <u>49</u>, respectively.

Problem

```
int sum{0};
for (int i = 1; i <= 10; i++)
   sum += i;
cout << "Sum from 1 to 10 is " << sum << endl;

sum{0};
for (int i = 20; i <= 37; i++)
   sum += i;
cout << "Sum from 20 to 37 is " << sum << endl;

sum{0};
for (int i = 35; i <= 49; i++)
   sum += i;
cout << "Sum from 35 to 49 is " << sum << endl;
```

Solution

```
int main()
{
  cout << "Sum from 1 to 10 is " << sum(1, 10) << endl;
  cout << "Sum from 1 to 10 is " << sum(20, 37) << endl;
  cout << "Sum from 1 to 10 is " << sum(35, 49) << endl;
  return 0;
}
```

```
int sum(int i1, int i2)
{
  int sum = 0;
  for (int i = i1; i <= i2; i++)
    sum += i;
  return sum;
}
```

Benefits of Functions

- Write a function once and reuse it anywhere.
- Information hiding. Hide the implementation from the user.
- Reduce complexity.

Objectives

- To declare and define functions with formal parameters.
- To define/invoke value-returning and void functions.
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain.
- To use function overloading and understand ambiguous overloading.
- To use function prototypes to declare function headers.
- To define functions with default arguments.
- To improve runtime efficiency for short functions using inline functions.
- To determine the scope of local and global variables .
- To pass arguments by reference and understand the differences between pass-by-value and pass-by-reference.
- To declare const parameters to prevent them from being modified accidentally.

Function

1) Prototype
2) Definition
3) Call

Function Prototypes

Before a function is called, it must be declared first. One way to ensure it is to place the declaration before all function calls. Another way to approach it is to declare a function prototype before the function is called. A function prototype is a function declaration without implementation. The implementation can be given later in the program.

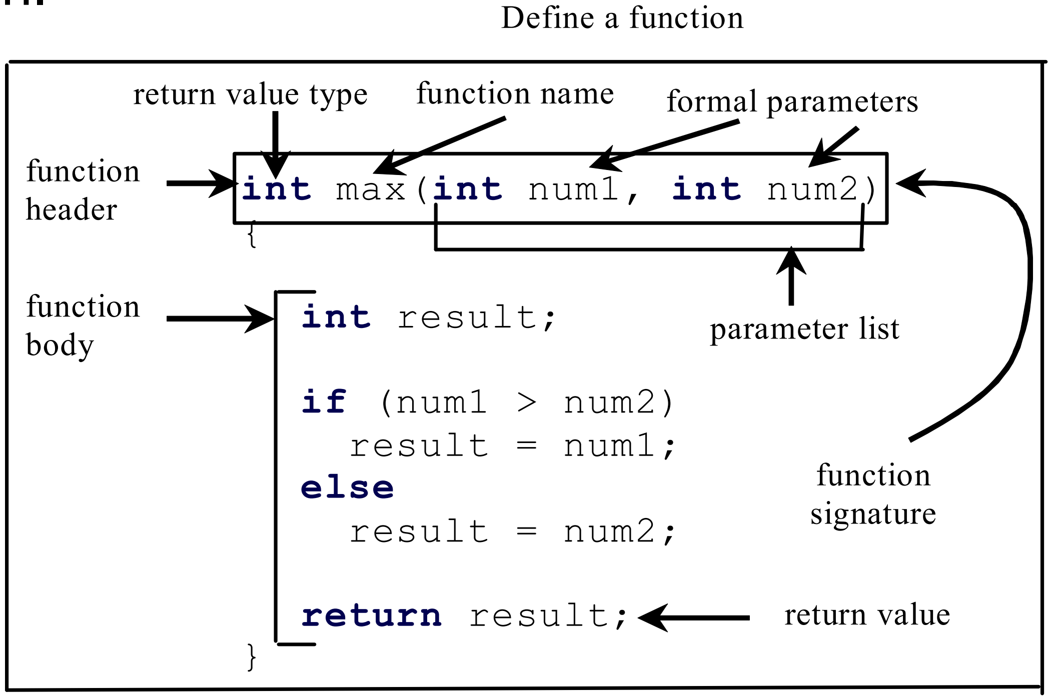
```
//This is function-prototype
int max(int num1, int num2);
```

Defining Functions

- *Function signature* is the combination of the function name and the parameter list.
- The variables defined in the function header are known as *formal parameters*.
- When a function is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.
- A Function may return a value. The <u>returnValueType</u> is the data type of the value the function returns. If the function does not return a value, the <u>returnValueType</u> is the keyword <u>void</u>.

Defining a Function

A function is a collection of statements that are grouped together to perform an operation.



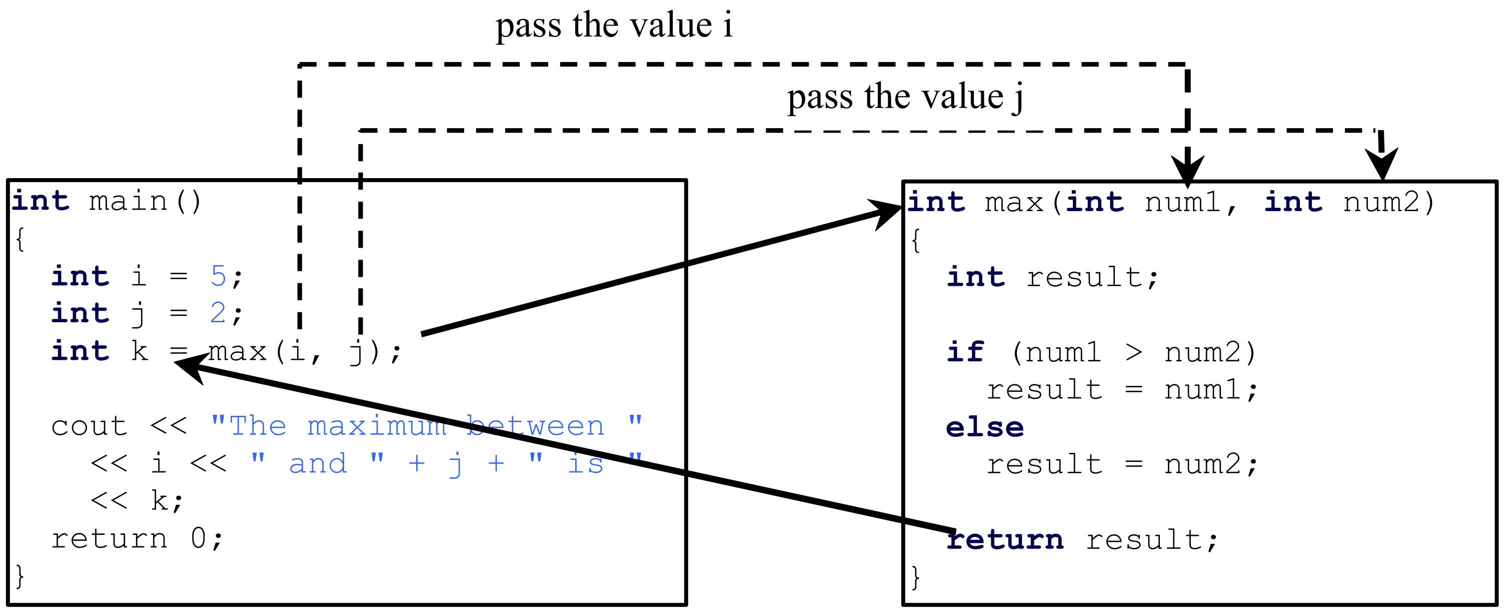
```
int max(int num1, int num2)
{
    int result;

    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
}
```

Function call

TestMax



```
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);

  cout << "The maximum between "
    << i << " and " + j + " is "
    << k;
  return 0;
}
```

```
int max(int num1, int num2)
{
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Passing Arguments by Value

By default, the arguments are passed by value to parameters when invoking a function.

The power of a function is its ability to work with parameters. You can use **max** to find the maximum between any two **int** values. When calling a function, you need to provide arguments, which must be given in the same order as their respective parameters in the function signature. This is known as *parameter order association*.

void Functions

The preceding section gives an example of a non-void function. This section shows how to declare and invoke a void function.

```
void printGrade(double score); //Prototype
```

```
printGrade(score);  //Call of a void function
```

TestVoidFunction

Scope of Variables

A local variable: a variable defined inside a function.

Scope: the part of the program where the variable can be referenced.

The scope of a variable starts from its declaration and continues to the end of the block that contains the variable.

Global Variables

C++ also allows you to use *global variables*. They are declared outside all functions and are accessible to all functions in its scope. Local variables do not have default values, but global variables are defaulted to zero.

VariableScopeDemo

Unary Scope Resolution

If a local variable name is the same as a global variable name, you can access the global variable using ::globalVariable. The :: operator is known as the *unary scope resolution*. For example, the following code:

```
#include <iostream>
using namespace std;
int v1 = 10;
int main()
{
  int v1 = 5;
  cout << "local variable v1 is " << v1 << endl;
  cout << "global variable v1 is " << ::v1 << endl;
  return 0;
}
```

Overloading Functions

The max function that was used earlier works only with the int data type. But what if you need to find which of two floating-point numbers has the maximum value? The solution is to create another function with the same name but different parameters, as shown in the following code:

TestFunctionOverloading

Remember when overloading!

Do NOT overload functions with almost the same type for in-parameters.

For example, int and double!

WrongOverloading

Default Arguments

C++ allows you to declare functions with default argument values. The default values are passed to the parameters when a function is invoked without the arguments.

```
void printArea(double radius=1); //prototype

void print(int tal,double radius=1); //prototype
```

Default values always start from right to left

DefaultArgumentDemo

Inline Functions

Implementing a program using functions makes the program easy to read and easy to maintain, but function calls involve runtime overhead (i.e., pushing arguments and CPU registers into the stack and transferring control to and from a function). C++ provides *inline functions* to avoid function calls. Inline functions are not called; rather, the compiler copies the function code *in line* at the point of each invocation. To specify an inline function, precede the function declaration with the inline keyword.

```
inline void f(int month, int year) //Definition
{
  std::cout << "month is " << month << std::endl;
  std::cout << "year is " << year << std::endl;
}
```

InlineDemo

Static Local Variables

After a function completes its execution, all its local variables are destroyed. Sometimes, it is desirable to retain the value stored in local variables so that they can be used in the next call. C++ allows you to declare static local variables. Static local variables are permanently allocated in the memory for the lifetime of the program. To declare a static variable, use the keyword static.

```
void t1()  //Definition
{
  static int x = 1;
  //Do more!
}
```

StaticVariableDemo

Reference Variables

C++ provides a special type of variable, called a *reference variable*, which can be used as a function parameter to reference the original variable. A reference variable is an alias for another variable. Any changes made through the reference variable are actually performed on the original variable. To declare a reference variable, place the ampersand (&) in front of the name.

```
int count = 1;
int& r = count;
```

TestReferenceVariable

Pass by Reference

Pass-by-value has serious limitations. TestPassByValue is a program that shows the effect and limitation of passing by value. The program creates a function for swapping two variables. The swap function is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the function is invoked.

TestPassByValue

Constant Reference Parameters

```
// Return the max between two numbers
int max(const int& num1, const int& num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```

Thank you