The Zen of C++

2nd Edition

Adnan Zejnilovic

Chapter 6

Functions

What is a Function?

- Top-Down Design
  - Stepwise refinement
  - Divide and Conquer
- Built-in (Predefined) Functions
- Programmer defined functions

Benefits of Using Functions

- Eliminate repetitive statements
- Make code modular
- Easier to read/understand the code
- Easier to isolate software defects if there are any

Some “Rules”

- Function Calls
- Function Body
- Function Definition
- Function Header
- Function Prototype/Declaration
- “Caller” Program

Function Calls

```
void goodEvening()
{
    cout << "Hello, good evening"
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

Function Body

```
void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

- Starts with **{**
- Ends with a **}**

Function Definition

```
void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

Function Definition

Function Definition

```
void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

Return data type

Function Definition

Function definition consists of:

- Function return data type
  - “void”
- Function name
- Parameter List

Function Definition

```
void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

Function Definition

Function definition consists of:

- Function return data type
- Function name “goodEvening”
- Parameter List

Function Definition

```
void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

Function Definition

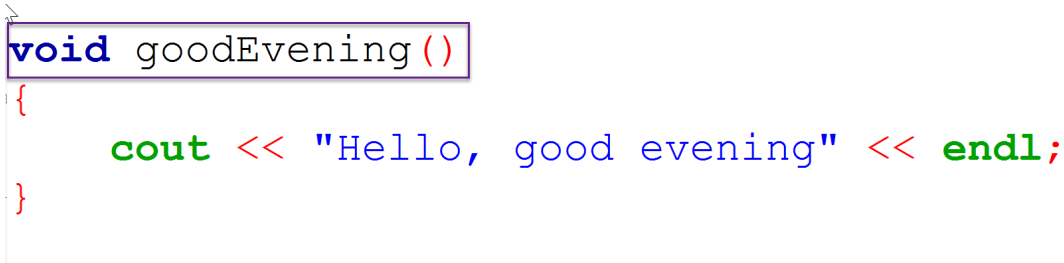
Function definition consists of:

- Function return data type
- Function name
- Parameter List
  - In this case, the list is empty
  - Hence nothing in parenthesis
    - “Empty” parameter list

Function Header

Function Header contains:

- Function return data type
- Function name
- Parameter list



```
void goodEvening()
{
    cout << "Hello, good evening" << endl;
}
```

Write Functions Before the int main()

```cpp
#include <iostream>

using namespace std;

void goodMorning()
{
    cout << "Hello, good morning" << endl;
}

void goodAfternoon()
{
    cout << "Hello, good afternoon" << endl;
}

void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

Write Functions after the int main()

```cpp
#include <iostream>

using namespace std;

// function prototypes
void goodMorning();
void goodAfternoon();
void goodEvening();

int main()
{
    goodMorning();  // function calls
    goodMorning();
    goodAfternoon();
    goodEvening();
    goodEvening();
    return 0;
}

void goodMorning()
{
    cout <<"Good morning!" << endl;
}

void goodAfternoon()
{
    cout <<"Good Afternoon!" << endl;
}

void goodEvening()
{
    cout <<"Good Evening!" << endl;
}
```

- In this case, declare function prototypes
- Let the compiler know (forward declaration)
- No difference between two approaches
- Using prototypes is the preferred approach
  - More complex programs are split into
    - Header files (prototypes)
    - Implementation files

```cpp
#include <iostream>

using namespace std;

void goodMorning()
{
    cout << "Hello, good morning" << endl;
}

void goodAfternoon()
{
    cout << "Hello, good afternoon" << endl;
}

void goodEvening()
{
    cout << "Hello, good evening" << endl;
}

int main()
{
    goodMorning();
    goodMorning();
    goodMorning();
    goodAfternoon();
    goodEvening();
    return 0;
}
```

```cpp
#include <iostream>

using namespace std;

// function prototypes
void goodMorning();
void goodAfternoon();
void goodEvening();

int main()
{
    goodMorning();  // function calls
    goodMorning();
    goodAfternoon();
    goodEvening();
    goodEvening();
    return 0;
}

void goodMorning()
{
    cout <<"Good morning!" << endl;
}

void goodAfternoon()
{
    cout <<"Good Afternoon!" << endl;
}

void goodEvening()
{
    cout <<"Good Evening!" << endl;
}
```

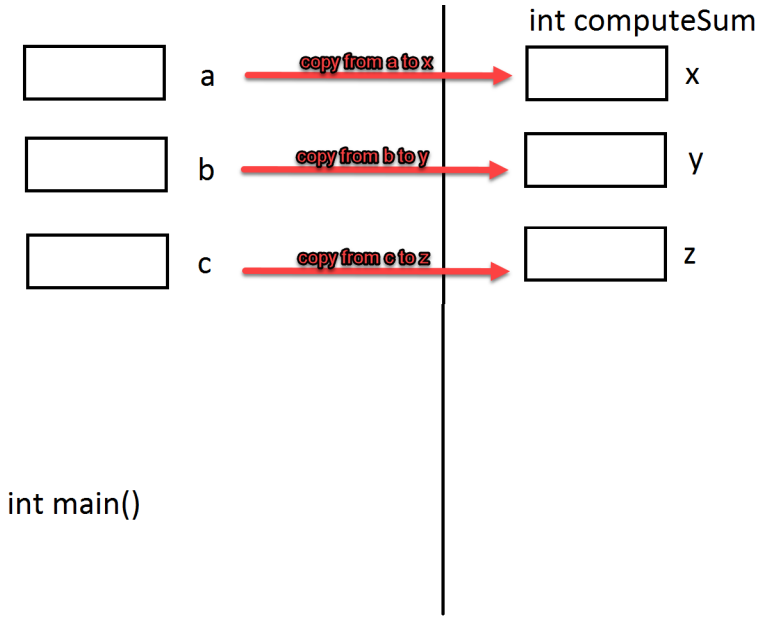
Passing by Value

- Functions receive data through parameters
- Parameters are local (to the function) variables designed to receive data from a caller program
- Distinguish between an *argument* and a *parameter*
- ***a***, ***b***, and ***c*** *are* ***arguments***
- ***x***, ***y***, and ***z*** are ***parameters***

```
#include <iostream>

using namespace std;

// function prototypes
int computeSum(int, int, int);

int main()
{
    //myFunction();
    int a, b, c, aSum;
    int nNumbers = 3;
    //double average;

    cout << "Enter an integer: ";
    cin >> a;

    cout << "Enter an integer: ";
    cin >> b;

    cout << "Enter an integer: ";
    cin >> c;

    // function call, returns result of its calculation
    // capture that and store it in a local variable
    aSum = computeSum(a, b, c);
    cout << a << " + " << b << " + " << c << " = " << aSum << endl;
    return 0;
}

int computeSum(int x, int y, int z)
{
    int sum;
    sum = x + y + z;
    return sum;
}
```

Passing by Value – Order Matters!

- Contents of a are copied to x
- Contents of b are copied to y
- Contents of c are copied to z




```
#include <iostream>

using namespace std;

// function prototypes
int computeSum(int, int, int);

int main()
{
    //myFunction();
    int a, b, c, aSum;
    int nNumbers = 3;
    //double average;

    cout << "Enter an integer: ";
    cin >> a;

    cout << "Enter an integer: ";
    cin >> b;

    cout << "Enter an integer: ";
    cin >> c;

    // function call, returns result of its calculation
    // capture that and store it in a local variable
    aSum = computeSum(a, b, c);
    cout << a << " + " << b << " + " << c << " = " << aSum << endl;
    return 0;
}

int computeSum(int x, int y, int z)
{
    int sum;
    sum = x + y + z;
    return sum;
}
```

Pass by Reference

- A reference parameter is denoted by an “&”
- When passing by reference the data is NOT copied from arguments to parameters
- Instead, the function parameter is given the address of the data

```
void fahren2Celsius (double &fahren)
{
    double cels;
    cels = 5.0/9 * (fahren -32);

    cout << fahren << " degrees F is "
         << cels << " degrees C" << endl;
}
```

- Note the “&” in front of the fahren parameter

Pass by Reference

```
#include <iostream>

using namespace std;

// function prototype
void fahren2Celsius (double &);

int main()
{
    double degF;
    cout << "Please enter the temperature in degrees F:";
    cin >> degF;
    cout << "DEBUG: degF in main BEFORE the function call: " << degF << endl;
    fahren2Celsius(degF);
    cout << "DEBUG: degF in main AFTER the function call: " << degF << endl;
    return 0;
}

void fahren2Celsius (double &fahren)
{
    double cels;
    cels = 5.0/9 * (fahren -32);

    cout << fahren << " degrees F is "
         << cels << " degrees C" << endl;

    // Purposely add 10 to the fahren
    fahren = fahren + 10;
}
```

```
Please enter the temperature in degrees F:212
DEBUG: degF in main BEFORE the function call: 212
212 degrees F is 100 degrees C
DEBUG: degF in main AFTER the function call: 222

Process returned 0 (0x0)   execution time : 3.627 s
Press any key to continue.
```

- Run this program
- What happened ?
- Why did degF get changed?

Pass by Value

- Let's examine identical function that uses "pass by value" and its output
- The argument(degF) was not affected by the change in the fahren2Celsius function

```
#include <iostream>

using namespace std;

// function prototype
void fahren2Celsius (double );

int main()
{
    double degF;
    cout << "Please enter the temperature in degrees F:";
    cin >> degF;
    cout << "DEBUG: degF in main BEFORE the function call: " << degF << endl;
    fahren2Celsius(degF);
    cout << "DEBUG: degF in main AFTER the function call: " << degF << endl;
    return 0;
}

void fahren2Celsius (double fahren)
{
    double cels;
    cels = 5.0/9 * (fahren -32);

    cout << fahren << " degrees F is "
         << cels << " degrees C" << endl;

    // Purposely add 10 to the fahren
    fahren = fahren + 10;
}
```

```
Please enter the temperature in degrees F:212
DEBUG: degF in main BEFORE the function call: 212
212 degrees F is 100 degrees C
DEBUG: degF in main AFTER the function call: 212

Process returned 0 (0x0)   execution time : 3.656 s
Press any key to continue.
```

Pass by Value

- Let's look at the variable addresses:

```
#include <iostream>

using namespace std;

// function prototype
void fahren2Celsius (double );

int main()
{
    double degF;
    cout << "Please enter the temperature in degrees F:";
    cin >> degF;
    cout << "DEBUG: the address of degF in main: " << &degF << endl;
    fahren2Celsius(degF);
    return 0;
}

void fahren2Celsius (double fahren)
{
    cout << "DEBUG: the address of fahren in fahren2Celsius: " << &fahren << endl;
    double cels;
    cels = 5.0/9 * (fahren -32);

    cout << fahren << " degrees F is "
         << cels << " degrees C" << endl;

    // Purposely add 10 to the fahren
    fahren = fahren + 10;
}
```

- They are different which confirms that the argument is in a different memory location

```
Please enter the temperature in degrees F:212
DEBUG: the address of degF in main: 0x6afef8
DEBUG: the address of fahren in fahren2Celsius: 0x6afeb8
212 degrees F is 100 degrees C

Process returned 0 (0x0)   execution time : 3.968 s
Press any key to continue.
```

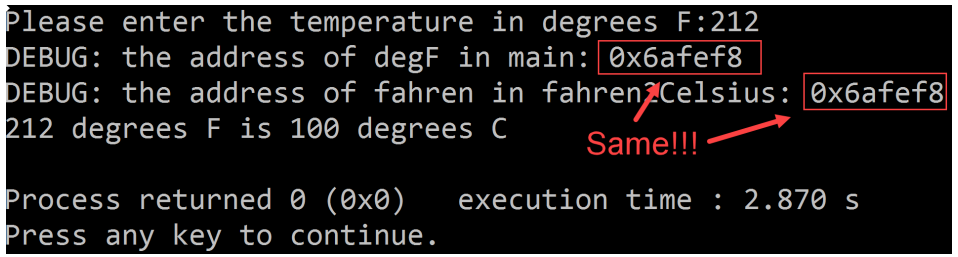
Different

Pass by Reference

- Let's look at the variable addresses when passing by reference

```
#include <iostream>

using namespace std;

// function prototype
void fahren2Celsius (double &);

int main()
{
    double degF;
    cout << "Please enter the temperature in degrees F:";
    cin >> degF;
    cout << "DEBUG: the address of degF in main: " << &degF << endl;
    fahren2Celsius(degF);
    return 0;
}

void fahren2Celsius (double &fahren)
{
    cout << "DEBUG: the address of fahren in fahren2Celsius: " << &fahren << endl;
    double cels;
    cels = 5.0/9 * (fahren -32);

    cout << fahren << " degrees F is "
         << cels << " degrees C" << endl;

    // Purposely add 10 to the fahren
    fahren = fahren + 10;
}
```

- The addresses of ***degF*** and ***fahren*** are the same:



- Which proves that ***fahren*** is an "alias" for ***degF*** as they are the same memory location

Pass by Constant Reference

- Passing by reference is faster as we are only passing the address
- This is especially true if we were to pass a large object
  - Making a copy of it would be time consuming
  - Passing by reference is passing only the address of the argument variable
- However, the value of the argument variable can be changed in the function so it is dangerous
- What if we wanted to pass by reference and prevent the function from changing the value of the argument?
- The answer is pass by ***constant reference***

Pass by Constant Reference

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  // function prototype
7  void fahren2Celsius (const double &);
8
9  int main()
10 {
11     double degF;
12     cout << "Please enter the temperature in degrees F:";
13     cin >> degF;
14     cout << "DEBUG: the address of degF in main: " << &degF << endl;
15     fahren2Celsius(degF);
16     return 0;
17 }
18
19 void fahren2Celsius (const double &fahren)
20 {
21     cout << "DEBUG: the address of fahren in fahren2Celsius: " << &fahren << endl;
22     double cels;
23     cels = 5.0/9 * (fahren -32);
24
25     cout << fahren << " degrees F is "
26          << cels << " degrees C" << endl;
27
28     // Purposely add 10 to the fahren
29     fahren = fahren + 10;
30 }
31
```

Error

```
Logs & others
Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck | CppCheck messages | Cscope | Debugger | DoxyBlocks | Fortran info | Closed
File      L.. Message
            === Build: Debug in CH06-passByConstReference (compiler: GNU GCC Compiler) ===
C:\Dev...   In function 'void fahren2Celsius(const double&)':
C:\Dev... 29 error: assignment of read-only reference 'fahren'
            === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

- When passing by const reference, any attempt to modify the parameter inside the function would result in an error

void Functions and the return Statement

- void functions do not return any values
- It is possible to place a “return” statement in a void function
- The moment the return statement executes, the program control is transferred to the caller program
- Any statements after the return statement are not going to execute as the function immediately goes out of scope following the execution of the return statement
- Avoid using return statements in void functions
- Instead use value returning functions that return bool or some other value

void Functions and the return Statement

```
#include <iostream>

using namespace std;

// function prototypes
void computeSum(int , int , int );

int main()
{
    int a=10,
        b=20,
        c=35;

    computeSum(a,b,c);
    return 0;
}

void computeSum(int x, int y, int z)
{
    int sum;

    sum = x + y + z;
    cout << "Sum is: " << sum << endl;
}
```

```
Sum is: 65

Process returned 0 (0x0)   execution time : 0.132 s
Press any key to continue.
```

void Functions and the return Statement

```
#include <iostream>

using namespace std;

// function prototypes
void computeSum(int , int , int );

int main()
{
    int a=10,
        b=20,
        c=35;

    computeSum(a,b,c);
    return 0;
}

void computeSum(int x, int y, int z)
{
    int sum;

    return;
    sum = x + y + z;
    cout << "Sum is: " << sum << endl;
}
```

unreachable

- Placing a “return” statement in a void function will return the control to the caller program
- Statements after the return statement are not going to execute
- In fact, the function will go out of scope following successful return to the caller program
- Below is the output of the program run
  - sum is never computed nor displayed

```
Process returned 0 (0x0)   execution time : 0.084 s
Press any key to continue.
```

Default Arguments

```
#include <iostream>

using namespace std;

void displayValues(int=0, int=5000 );

int main()
{
    int x = 10;
    int y = 20;
    displayValues();
    return 0;
}

void displayValues(int a, int b)
{
    cout << "This is the 1st value: " << a << endl;
    cout << "This is the 2nd value: " << b << endl;
}
```

- Note the function prototype – both parameters have default values
  - Syntax: datatype = value
- Should a programmer forget to pass an argument (or both as is the case in this example) when calling the function, the default argument will be assigned
- Following is the output of the program run:

```
This is the 1st value: 0
This is the 2nd value: 5000

Process returned 0 (0x0)   execution time : 0.085 s
Press any key to continue.
```

- Note the function call was made without any parameters and default arguments were assigned

Default Arguments

```
#include <iostream>

using namespace std;

void displayValues(int=0, int=5000 );

int main()
{
    int x = 10;
    int y = 20;
    displayValues(y);
    return 0;
}

void displayValues(int a, int b)
{
    cout << "This is the 1st value: " << a << endl;
    cout << "This is the 2nd value: " << b << endl;
}
```

- The function was called with only one argument “y”
- Following is the output of program run:

```
This is the 1st value: 20
This is the 2nd value: 5000

Process returned 0 (0x0)   execution time : 0.162 s
Press any key to continue.
```

- Notice that the passed argument’s value was copied to the first parameter and the second parameter got its value from the default argument
- When skipping default arguments, you cannot skip arguments on the left
  - For example, if there are three default arguments, if the function is called with only two arguments, then 1st and 2nd parameters will receive the values and the 3rd (skipped) argument will receive the default parameter
  - displayValues(a); is correct
  - displayValues(, b); is incorrect

Overloading Functions

- Great feature of the language
- Simplifies source code
- Functions have the same name but different “signature”
- Function signature
  - Number of parameters and their data types

```
#include <iostream>

using namespace std;

int mySum(int a, int b);
int mySum(int a, int b, int c);
double mySum(double a, double b, double c);

int main()
{
    int x = 1, y = 10, z = 5;
    cout << "\nCalling cout << "\nCalling sum with double arguments " << endl; with integer arguments " << endl;
    cout << x << " + " << y << " + " << z << " = " << mySum (x, y, z) << endl;

    double a = 1.2, b = 10.5, c = 5.8;
    cout << "\nCalling cout << "\nCalling sum with double arguments " << endl; with double arguments " << endl;
    cout << a << " + " << b << " + " << c << " = "<< mySum (a, b, c);
    cout << "\nCalling mySum with 2 integer arguments " << endl;
    cout << x << " + " << y << " + " << " = " << mySum (x, y);

    return 0;
}

double mySum(double a, double b, double c)
{
    return a + b + c;
}

(int a, int b)

n a + b;

(int a, int b, int c)

n a + b + c;
```

```
Calling sum with 3 integer arguments
1 + 10 + 5 = 16

Calling sum with 3 double arguments
1.2 + 10.5 + 5.8 = 17.5
Calling mySum with 2 integer arguments
1 + 10 +  = 11
Process returned 0 (0x0)   execution time : 0.164 s
Press any key to continue.
```

Overloading Functions - Example

```cpp
#include <iostream>

using namespace std;

int mySum(int a, int b);
int mySum(int a, int b, int c);
double mySum(double a, double b, double c);

int main()
{
    int x = 1, y = 10, z = 5;
    cout << "\nCalling sum with 3 integer arguments " << endl;
    cout << x << " + " << y << " + " << z << " = " << mySum (x, y, z) << endl;

    double a = 1.2, b = 10.5, c = 5.8;
    cout << "\nCalling sum with 3 double arguments " << endl;
    cout << a << " + " << b << " + " << c << " = "<< mySum (a, b, c);
    cout << "\nCalling mySum with 2 integer arguments " << endl;
    cout << x << " + " << y << " + " << " = " << mySum (x, y);

    return 0;
}

double mySum(double a, double b, double c)
{
    return a + b + c;
}

int mySum(int a, int b)
{
    return a + b;
}

int mySum(int a, int b, int c)
{
    return a + b + c;
}
```

```
Calling sum with 3 integer arguments
1 + 10 + 5 = 16

Calling sum with 3 double arguments
1.2 + 10.5 + 5.8 = 17.5
Calling mySum with 2 integer arguments
1 + 10 +  = 11
Process returned 0 (0x0)   execution time : 0.164 s
Press any key to continue.
```

Scope

- “Lifetime” of a variable
  - Declared
  - Initialized
  - Operations performed on it – i.e. Incremented/decremented
  - Destroyed
- Also it is important to note where in the program a variable has been declared
- This determines the scope (“visibility”) of a variable
- A variable is in scope after its declaration
- Block scope
  - Variable declared inside a block of code
  - In scope only for that block of code
- Global Scope vs Local Scope
- Function Prototype Scope
  - Refers to the scope of function parameters
  - When the function returns to the caller, all of its parameters go out of scope

Global Variables / Global Constants

- Global variables are in scope throughout the entire program
- If not initialized, the compiler will set them to default values – i.e. set an int to zero
- No need to pass them into functions
- However, global variables are a very bad programming practice
  - Code portability is affected as functions that rely on global variables cannot be ported to another program
  - Any function can change global variable
    - The question is, which function changed it?
    - In order to control your code, you should always pass data to functions

Local Variables

- Local variables are in scope for the lifetime of a function
- When the function returns to the caller program, all of its local variables are destroyed along with the data in them
  - Notice the output of the code sample?
  - Each time the function is called, variable x is allocated, initialized, and incremented.
  - Each time the function returns to int main, x (local variable in displayNumber) is destroyed
- Good for program portability
- Good for debugging as they are confined to the function not entire program

```
#include <iostream>

using namespace std;

// Function Prototypes
void displayNumber();

int main()
{
    displayNumber();
    displayNumber();
    displayNumber();
    displayNumber();

    return 0;
}

void displayNumber()
{
    int x = 0;
    x = x + 1;
    cout << "This is x: " << x << endl;
}
```

```
This is x: 1
This is x: 1
This is x: 1
This is x: 1

Process returned 0 (0x0)   execution time : 0.144 s
Press any key to continue.
```

Static Local Variables

- They behave like the global variables
  - The compiler will initialize them if not initialized
  - Notice that x was never initialized?
  - Yet they are local in scope
    - Hint: try printing x in main
- Static local variables keep their values between function calls

```
#include <iostream>

using namespace std;

// Function Prototypes
void displayNumber();

int main()
{
    displayNumber();
    displayNumber();
    displayNumber();
    displayNumber();

    return 0;
}

void displayNumber()
{
    static int x;
    x = x + 1;
    cout << "This is x: " << x << endl;
}
```

```
This is x: 1
This is x: 2
This is x: 3
This is x: 4

Process returned 0 (0x0)   execution time : 0.085 s
Press any key to continue.
```

Stubs & Drivers

- Using stubs and drivers is a very good programming practice when working with functions
- Tests values (data types) passed into a function
- Driver program (usually int main) “drives” the program
  - Calls functions
  - Users get an idea of the program flow