

# C++ Functions

## Contents

<b>1</b>	<b>Overview of Functions</b>	<b>2</b>
<b>2</b>	<b>Note: Functions and Top-Down Design</b>	<b>4</b>
<b>3</b>	<b>Function Prototypes</b>	<b>7</b>
<b>4</b>	<b>Pass by Value vs. Pass by Reference</b>	<b>9</b>
4.1	Motivation: Saving Overhead . . . . .	9
4.2	Motivation: Procedural/Memory Intensive . . . . .	10
<b>5</b>	<b>Local Values &amp; Scope</b>	<b>11</b>
5.1	Global Scope . . . . .	11
<b>6</b>	<b>Function Overloading</b>	<b>13</b>
<b>7</b>	<b>Function Templates</b>	<b>14</b>
<b>8</b>	<b>Storage Classes for Local Variables</b>	<b>15</b>
8.1	Storage Modifier: <code>auto</code> . . . . .	16
8.2	Storage Modifier: <code>static</code> . . . . .	16
8.3	Storage Modifier: <code>mutable</code> . . . . .	17
8.4	Storage Modifier: <code>extern</code> . . . . .	17
8.5	Storage Modifier: <code>register</code> . . . . .	18
<b>9</b>	<b>A Few Comments about Functions</b>	<b>18</b>

# 1 Overview of Functions

- We *finally* get to start talking about functions!
- *As you have all already read Chapter 6, this should really be just a review, right?*

- Functions in C++ (indeed, any language) are powerful, and a necessary tool in any modern/good software engineering methodology
- Since they're so powerful, they can be a little tricky at times
- Plus, as you've read in Chapter 6, there are many variations of how to implement/use functions in C++
- (E.g. it's not as simple as the minor variations of `if/else` or `while & for`)
- We'll be spending quite some time talking about functions

- First, a quick overview of how/why we have functions. . .
- In mathematics, we frequently employ functions (things like  $+$ ,  $-$ , etc.), and in general, a function in mathematics is some *mapping* from things in a *domain* to things in a *range*
- Functions are usually represented as

$$f : D \rightarrow R \tag{1}$$

where  $D$  is the domain (some set of things) and  $R$  is the range (some other, although potentially the same, set of things)

- Given some element  $x \in D$ ,  $f$  does something to  $x$  and returns a value  $f(x)$  in  $R$
- In the above example. . .
  - Call “ $f$ ” the *name* of the function (that's easy!)
  - Call  $x$  (the item in the domain) the *parameter* to  $f$
  - Call  $f(x)$  the *return value* of  $f$  given parameter  $x$
- Now, when we define functions in mathematics, we generally give some definition that tells us *how*, given  $x$ , to calculate  $f(x)$  (without such a definition, we can't really do much with  $f$ !)
- Call such a definition the *function body*

- Functions in C++ aren't much different
- *examples...*

- In C++, functions are defined as follows...

```
returnType SomeFunctionName( parameterList )
{
    // Where the function calculates its value (for the given
    // parameters
}
```

or, following the example we were working with above

```
TypeOfRange f ( TypeOfDomain )
{
    // function body
}
```

- The `returnType` must be some valid C++ type...either a simple type (`bool`, `int`, `float`, `double`, `char`) or complex (e.g. `string`)
- Similarly, the `parameterList` is some sequence of types (again, either simple or complex) which are named (very similar to variable declaration)

- *Let's look again at that last example...*

- Remember that C++ is a *typed* language, so any parameter (variable) must have a corresponding type
- Similarly, since functions return *something*, that something must have a type (the *return type*)

- The *function body* is (almost) any set of C++ code (variable declarations, control/looping structures, etc.) that calculates the return value
- The return value is actually *returned* from the function using the `return` keyword

- We'll see more about the implementation details in a few minutes, but first...

## 2 Note: Functions and Top-Down Design

- When we talked about good software engineering practices, we briefly mentioned “top-down design”
  - We didn’t go into much detail regarding what top-down design was, simply because we didn’t have the *tools* to really see what it was
  - Functions are one such tool
  - Now that we have functions covered, we can see in detail how to employ proper *top-down design* when solving problems
- 
- A classic first example of using functions to employ top-down design is the *Blackjack* assignment I gave on Friday (well...I *meant* to give it on Friday!)
  - At first, the problem seems daunting
  - With a little thought, and using top-down design, we’ll see how to reduce the problem to only a handful of *extremely simple pieces*
- 
- *You’ve all read Chapter 6 by now*, so I’ll give a hint about how to do this, before we cover functions in class today...
  - If we want to write a program to play Blackjack with the user, we might start with the following pseudo-code
    1. Deal two cards to user and dealer (self)
    2. DO UNTIL both user and dealer “stay”
      - IF user hasn’t yet “stayed” THEN...
        - (a) ASK user if they want another card
        - (b) IF they do THEN deal another card AND add to score
        - (c) OTHERWISE the user “stays”
      - IF dealer (self) hasn’t yet “stayed” THEN...
        - (a) DETERMINE if dealer (self) wants another card
        - (b) IF dealer does THEN deal another card to dealer AND add to score
        - (c) OTHERWISE dealer “stays”
    3. Now that both users stay, determine winner by the following...
    4. IF both users bust THEN dealer wins
    5. *Rest of winning-conditions here...I’m not going to give all of it away!*
    6. Display the winner

- The above pseudo-code is abstract... it's a high-level overview of what needs to be done
  - *And with functions, we can make it a reality, even in the code!*
- 
- We could, for example, write functions for
    - `userWantsAnotherCard(...)` to prompt the user if they want another card and return `true` if they do, `false` otherwise
    - `dealerWantsAnotherCard(...)` to encapsulate all of the logic used to determine if the dealer (the program) wants another card
    - `dealRandomCard(...)` to deal a random card
    - `scoreIsBust(...)` to determine if a score is bust
    - *And a few more! Which you will have to determine!*
- 
- With functions such as these, we really could translate the above pseudo-code into *real code*
  - Of course, for each function/high-level-abstraction...
  - We need to design that as well!
  - That is, we employ the design practices to *that function*
- 
- This is the heart of top-down design
  - When given a large problem, writing a single algorithm to solve it can be cumbersome and thus error prone
  - Breaking the problem up into smaller pieces makes this *much* easier
  - With top-down design, we start with the highest level of abstraction (like we did above) when laying out our design/algorithm
  - For each abstract/high-level piece, we include a design/algorithm for that
  - Eventually, we end up with a complete design that can be implemented just like we did before

- In general, it's always easier to verify/design/implement solutions to *smaller* problems than larger ones
- So we make use of abstraction (like we did above) at all levels of the design *except* the most basic
- And those basic parts are (if the design went well) easy to implement

### 3 Function Prototypes

- What's wrong with the following C++ code?

```
int main()
{
    cout << "Enter a number: ";
    cin >> number;

    int number;

    return 0;
}
```

- What error does it generate, and why?

- Simple: we tried to use a variable `number` before it was defined!
- *Same thing with functions...*
- The following code will not work for the exact same reason

```
int main()
{
    cout << "The square of 42 is " << square(42) << endl;

    return 0;
}

int square(int base)
{
    return base * base;
}
```

- The problem? At the first line of the `main` function, the function `square` had not *yet* been defined
  - Exactly the same error we had with the variable in the first program
  - The solutions for functions are a little different, however
- 
- The two solutions to this problem are...
    1. Define the function before it's used. This is analogous to defining the variable before it's used, and thus you need to make sure all functions appear *before* the `main` function. (Note: this is considered bad programming practice, which is why the next solution is preferred.)
  
    2. Or use a *function prototype*, which is basically just the type of the function. When the compiler is scanning the code line-by-line and it encounters a function, it just need to know the *type* of the function so that it can check if the parameters given and the return value are being used appropriately. A function prototype is just a duplicate of first line of the functions definition (although names for the parameters aren't required).
- 
- Usually for large software, you'll put a collection of functions (that do the same thing) into their own file, say `foo.cpp`.
  - Then you create a file containing *just* the function prototypes and put those into a file like `foo.h`
  - With that, your programs need only to include the `foo.h` file and then *link* with the compiled `foo.cpp`

## 4 Pass by Value vs. Pass by Reference

- When defining a function, there are two ways to pass parameters to the function
- The first is the way we've worked with function thus far

```
int square( int num )
{
    return num * num;
}
```

- Whenever you call the function from elsewhere in your code, and pass it a number as its parameter, the computer allocates enough memory for a *copy* of the parameter
- The function then works on that *copy*, never touching the original value
- We call this *pass by value* since it duplicates the actual value of the parameter

- There is another way to pass parameters to functions, called *pass by reference*
- With pass by reference, a *reference* to each parameter is passed to the function instead of a copy
- This reference is treated just like a regular parameter to the function, *except that if you change the value of that parameter in the function body, it changes in the calling code as well!*
- *examples...*

- So a function can modify its parameters when passed by reference!
- Why on earth would we want something like this?
- There are two main reasons

### 4.1 Motivation: Saving Overhead

- For functions that take many parameters, or take parameters of large data types (e.g. arrays or non-simple C++ data types), allocating this additional memory can be costly
- Using pass-by-reference parameters can reduce this overhead
- *However*, anytime you choose to pass parameters by reference for this reason...
- *You should pass the parameters as const!*
- This ensures that the parameters passed to the function will *NOT* be changed
- *examples...*

## 4.2 Motivation: Procedural/Memory Intensive

- Sometimes we'll want to do simple procedural tasks with memory
  - Classic example: swapping
  - Sometimes we'd like to swap the value of two variables (we'll see why when we come to sorting)
  - *examples...*
- 
- A note about the use of reference parameters...
  - With pass by value, you can use numbers as parameters just fine, e.g. `square(42)`. However if `square` uses pass by reference instead, and tries to overwrite the value of that parameter *you will receive a compile error!* Obviously, you don't want to redefine the number 42 in your program!



- As such, putting variables into the global scope is considered a poor programming practice since *any code, anywhere in your project could change that variable!*
- Global variables can make debugging a living nightmare at times, and in general you should stay away from using them for now
- **Note:** Global *constants* are perfectly acceptable, since they are constant

## 6 Function Overloading

- Sometimes we'd like functions to work on different sets of types
- For example, the `swap` function we saw before swapped the value of two integers
- It would be nice if we could define that same function for different types *without* using a different name for each
- C++ provides this functionality, through *overloaded functions*

- In C++, you can write multiple functions with the same name as long as each definition differs in either the types of the parameters *OR* the number of parameters
- These restrictions are imposed because C++ needs to know *which* function to call *for each* set-of/types of parameters
- Without such restrictions, C++ would not know which function to use when

- As such, if there are two versions of a function, say

```
int  max( int  num1, int  num2 );
float max( float num1, float num2 );
```

- And you try something like

```
int i = 10;
float f = 3.2;

someMax = max(i, f);
```

where you pass both a `float` and an `int`, C++ *may* have difficulty figuring out which version of `max` to use

- Some have you have already encountered this problem in your homeworks when trying to use the `cmath` function `pow`

- Overloading functions can be handy when we would like functionality such as `swap` for different types, but with the same name
- (Maintaining the same name greatly helps improve code-readability and overall abstraction)
- *examples...*

## 7 Function Templates

- Function overloading are nice... but wouldn't it be nice if we could write a single function that would work for all appropriate types?
- This is where *function templates* are useful
- A function template can be used to declare a function using a *type variable* (or set of type variables)
- Simple example...

```
template <class TypeVariable>
TypeVariable myFunction( TypeVariable param1, TypeVariable param2 )
{
    // statements;
}
```

- When you write a C++ function template, the template doesn't actually get compiled
- Rather, every call to `myFunction` with a variable of type `int` will create a new, overloaded version of `myFunction` where all instances of `TypeVariable` in the function definition is replaced with `int`
- Thus, every call of a new type creates a new version of the function!

- This may seem a little complicated, but it's rather easy, and can save some time
- *examples...*

- Note the following...
  1. If you were to use some function within a function template, and there are some types for which this function is not defined, that template cannot be created for such types

## 8 Storage Classes for Local Variables

- *We're only touching on these here... but you should read what the book has to say about this in §6.9.*
  - When declaring variables, we've seen that we have to include a *type* (such as `int`, `bool`, `string`, `double`, etc.)
  - We've also seen a *type modifier* like `const`, which (when added before the type in a variable declaration) makes the variable *constant*
  - Now, we'll look at the *storage specifiers* C++ provides
- 
- Storage modifiers describe during what period of time a variable is allocated and thus “exists”
  - We've talked about *scope* in class, and the book has described it in detail
  - Normally, when a new scope is entered (such as a function), the variables passed to that function and any variables declared in that function<sup>1</sup> are allocated, used in the function, and once the function `returns`, the memory for those variables is deallocated
  - So, those variables *only exist while the program is executing within their scope*
- 
- Storage modifiers allow one to change this behavior
  - **NOTE:** Storage modifiers do *not* modify the scope of variables. Rather, they simply change how/when the variables are *allocated*
  - Also, only one storage modifier may be applied to each variable. Declaring more than one will result in a compiler error.
  - A quick overview of each modifier...

---

<sup>1</sup>Variables in the outer-most scope of the function, that is.

## 8.1 Storage Modifier: auto

- The `auto` storage class is the default storage class...
- It's the type that we've read about in the book and discussed in class
- As such, it's rarely used since declaring a variable such as

```
double x;
```

is identical to

```
auto double x;
```

and vice-versa

- `auto` variables only exist in the nearest enclosing pair of curly braces within the body of the function in which the declaration appears
- Once program execution enters the scope of the variable, the memory allocation occurs and the variable name is bound (or *linked*) to that location

## 8.2 Storage Modifier: static

- The `static` storage class tells C++ to allocate the memory for the variable as the program is loading, and *before* it even begins to run
- As such, if a variable in a function is declared `static`, it is bound to the same memory location for *every* call to the function
- Thus, the variable will retain its value between function calls!
- Note that since the allocation of a `static` variable occurs when the program loads, the declaration (*and any assignment in that declaration*) only occurs once
- *examples...*

- Global variables are always `static`, since they exist in every scope and thus must exist (be allocated) once the program starts
- So, all global variables are automatically declared `static`, whether you add the `static` keyword or not
- The reverse, however, is *not* true

- That is, a static variable declared in a function is *not* a global variable. . . it's scope is still subject to the normal rules

- **Common and Acceptable Uses:**

1. **Cutting Overhead.** If a specific function is called *many* times throughout the course of a programs execution, allocating and deallocating memory for its variables may become costly. Declaring them **static** reduces this overhead. Do **NOT** get **static**-happy and declare all of your function variables **static**! Like global variables, they can make debugging *enormously difficult*!
2. **Profiling.** Sometimes, we'll want to *profile* a given program. That is, we want to measure how much time an average run of the program spends in each function (and perhaps how much memory is allocated in each function). Use of **static** variables to count function calls is frequently done.

### 8.3 Storage Modifier: mutable

- The **mutable** storage class is specific to C++ (it doesn't exist in C), and it is used specifically for class members
- As such, we'll skip **mutable** for now and once you cover classes next quarter, you'll read about this

### 8.4 Storage Modifier: extern

- With large software, code is distributed across hundreds or even thousands of files. Frequently one will have variables declared in one file and used in other files.
- *Example. . .*
- Suppose we had the following variable declaration in a file called `MyLib.cpp`

```
int someVar;
```

- If we have another file, say `Foo.cpp`, which is compiled and then *linked* with `MyLib.cpp` (but doesn't include `MyLib.cpp`), we would like to be able to use that variable
- If we just used `someVar` in `Foo.cpp` with no declaration, the compiler would generate an error
- We could declare another variable named `someVar` in `Foo.cpp`, but *that would be a different variable!*
- To solve this, we could place the following in `Foo.cpp`

```
extern int someVar;
```

- This tells the compiler that there exists some variable named `someVar` of type `int`, and no allocation of memory will actually take place
- Instead, when `Foo.cpp` is compiled and then linked against the compiled version of `MyLib.cpp`, the linker will *bind* occurrences of `someVar` in `Foo.cpp` to the variable in `MyLib.cpp`

## 8.5 Storage Modifier: `register`

- The last storage modifier is `register`, and it specifies *how* (more specifically, *where*) to store a variable
- CPU's have *registers*, which hold bits of memory that the CPU actually works with
- For most operations of a CPU, the data to be worked with *must* be loaded into these registers. Data stored in registers is immediately available to the processor.
- So, when a CPU performs some action, such as an addition like

```
x = y + z;
```

the values of `y` and `z` are first retrieved from main memory and then loaded into the *registers* of the CPU

- The CPU then adds the values in the registers these values were loaded into, and stores the result in some other register
- Afterwards, the resulting value in a register is then copied back to main memory (in this case, wherever `y` is stored)

- If you have variable that is used *heavily* in a function, it's sometime beneficial to specify that the variable should be stored in a register since the CPU doesn't have to fetch the value from memory
- Placing `register` before a variable declaration *requests* that the program stores the variable in a register
- This is only a *request* since there are finitely many registers for any CPU, and if they are all used the request cannot be made
- As the book mentions, most optimizing compilers are very good at figuring out which variables should be stored in registers, and will automatically do so
- As such, use of `register` is usually unnecessary

## 9 A Few Comments about Functions

- Many people have been stuck trying to write functions that always *output* a value, rather than *returning* a value
- They should