# Functions

**T1** Chapters: **30-32**

So far we have been writing code inside the main function, which I said had the special property that a program starts its execution from here. In other words, when your program is run, the first thing it executes is the code inside the main function, and hence the main function is also called the entry point of an application. You must notice what this function is doing - it is *packaging some code that is intended to be used repeatedly*.

Keeping the emphasized words in mind, we can understand functions as things that contain code. In the same way that we create a variable to hold some value, we create functions to hold some code. Also, in exactly the same way that we use the variable to refer to the value it holds, we can use functions to refer to the code they hold. This understanding is going to be extremely useful when we come across the topic of **function pointers** or **pointers to functions**.

## Creating Functions

The syntax for creating a function is as follows

RETURN\_TYPE FUNCTION\_NAME(ARG1, ARG2, ..., ARGN)
  
{
  
 BODY
  
}

where each ARGi, called an **argument**, has the syntax TYPE NAME. A good way to think about these arguments is to notice their similarity with declaration part of the two-step variable definition. In other words, the values for these arguments are assigned at the time or location where the function is actually used. To help visualize this way of thinking, see the example below

#include <cstdio>
  
  
void say(const char\* s)
  
{
  
 printf("s = %s\n", s);
  
}
  
  
int main()
  
{
  
 const char\* hw = "hello world";
  
 say(hw);
  
}

In the program above, we have defined a function called say which takes exactly one argument declared as const char\* s. Ignoring the body of the function for a moment, we can see that the value for this argument is assigned when this function is used (see inside the main function), where we assign the value referred to by the variable hw to s. Also to notice and keep in mind is that the value contained in or referred to by s is valid inside the function say.

Another detail to note would be the new keyword void in the return type of the say function. void in the return type of a function specifies that the function does not produce any value. Here, one might ask, what does it mean for a function to produce a value? To help answer, let us remind ourselves of the concept of a variable.

int y = 4;
  
int x = y;

In the small snippet above, we can see that the value inside y is assigned to the variable x. This assignment can also be thought of in another, more useful, way. More specifically, it can be said that y *produces the value* 4, which is then assigned to x.

With this in mind, we can say that when a function is used or called, and if it is defined to have a non-void return type, it produces a value. This value, in a crude, less accurate and less helpful sense, can be thought of as the output of the function.

#include <cstdio>
  
  
int square(int x)
  
{
  
 return x\*x;
  
}
  
  
int main()
  
{
  
 int x = 4;
  
 int y = square(x);
  
 printf("x = %d, y = %d\n", x, y);
  
}

In the above program, a function called square is defined that takes exactly one argument int x and returns an int, i.e., it produces an int when called. Thus, in the main function it can be seen that the value produced by the square function when given the variable x as its argument is assigned to y.

### Exercises

1. A function can take more than one argument, and the arity of a function is given by the number of arguments it accepts. Hence, a **unary** function is one that accepts only one argument, while a **binary** function is one that accepts two and so on.

* An example of a binary function is as follows,
* int add(const int a, const int b)
    
  {
    
   return a + b;
    
  }
* a. Write a binary function called avg that computes and produces the average of two double arguments. (E)
* b. Write a program that uses avg to compute the average of an array of double elements whose values are entered by the user. (M)

1. Write a program that tests whether a number is prime using a function is\_prime whose signature is given as bool is\_prime(int n). The program will prompt the user to enter a number, then respond with a message indicating whether or not the number is prime. (E)
2. Write a function named power that raises an integer x to the power n. (M)
3. Write a function named split that doesn't return anything, and splits a double-precision value x into its integer and fractional parts. [Hint: See the next section] (M)

## Side-Effects

#include <cstdio>
  
  
// axpy performs the operation y[i] += a\*x[i] for i in [0,N).
  
void axpy(const int a, const int\* x, const int n, int\* y)
  
{
  
 for (int i = 0; i < n; i += 1)
  
 {
  
 y[i] += a\*x[i];
  
 }
  
}
  
  
  
int main()
  
{
  
 constexpr int N = 4;
  
 int x[N] = {1,2,3,4};
  
 int y[N] = {0};
  
 axpy(2, x, N, y);
  
  
 // let us print the elements of y
  
 for (int i = 0; i < N; i += 1)
  
 {
  
 printf("%d\n", y[i]);
  
 }
  
}

In the above program, something interesting is happening. The function axpy is defined to be producing nothing, hence its return type is void, and yet it does end up producing *output* inside the array y. From the signature of the function, we can infer that all of the arguments except for y is **read-only**. In other words, the function is forbidden from making changes to all the arguments except y. Given the way we use the function, we intend for the function to put the result of scaling the elements of x by a as into y. One of the limitations of the C programming language is that we cannot produce more than one item from a function, which is why the array that we want the function to put the result into is given to it as an argument. This ability of functions to modify their arguments is possible only because of pointers, in that, a pointer enables a function to access the memory its pointing to indirectly. From one valid perspective, this behavior of functions can be described as a **side-effect** and such functions are impure. Two examples of side-effect producing functions that we have already encountered are printf and scanf. In the case of the former, interaction of the printf function beyond the program with the console is considered to be a side-effect, while with the latter, scanf modifies the memory pointed to by the pointer given to it which is outside the visibility of the function, and hence also a side-effect.

**NOTE**

Side-effect producing functions though sometimes faster, they are significantly more difficult to understand as program complexity increases. It is highly recommended to limit the use of side-effect producing or impure functions to a bare minimum.

### Exercises

1. Write a function that initializes all the elements of an array to 0. (E)
2. Write a function that computes and produces the sum of all the elements of an array. (E)
3. Write a function that computes the sum of two two-dimensional arrays using the axpy function above. (H)

## Function Pointers

As mentioned at the beginning, it is helpful to think of functions as variables referring to code as their value. In fact, functions are not just any variables but pointer variables, called **function pointers** or **pointers to functions** where the name of a function refers to its body of code. It is useful to understand how a function is actually stored in the computer's memory.

A function's body, just like the value of a variable, consumes memory - a specific category of memory called **instruction memory** where CPU instructions are stored. Meaning, a function's name refers to some set of instructions that are executed when the function is called, and just like an array it can be passed as an argument to other functions, but unlike arrays we cannot index into this memory to access the individual instructions.

### Using Function Pointers

#include <iostream>
  
#include <array>
  
  
using bfn\_t = int(\*)(int x, int y);
  
  
int add(int x, int y)
  
{
  
 return x + y;
  
}
  
  
int sub(int x, int y)
  
{
  
 return x - y;
  
}
  
  
int mul(int x, int y)
  
{
  
 return x \* y;
  
}
  
  
int main() {
  
  
 constexpr int N = 3;
  
 const bfn\_t fns[N] = {add, sub, mul};
  
 const int args[2] = {3, 4};
  
  
 for (auto fn : fns)
  
 {
  
 const auto res = fn(args[0], args[1]);
  
 printf("res = %d\n", res);
  
 }
  
}

In the program above, bfn\_t is declaring what is called a **type-alias**, which is referring to the type signature on the right hand side. In this case, bfn\_t is referring to a binary function that takes two integers as arguments and produces an integer. Then, we define three functions add, sub and mul that have an identical signature, whose names we then store inside the fns array defined in the main function. The arguments that we want to pass to each of the functions is stored in the args array and a loop is used to call each function with the arguments. We finally display the results produced by each of the functions.

### Exercises

1. Write a calculator program that works with single-precision numbers entered by the user to perform addition, subtraction, multiplication and division. Each of the operations must be stored, as function pointers, in an array like in the example above. (M)
2. Write a function called apply that applies the given function to each element of an array. The function must accept three arguments for the array, its length and the function to apply. (M)

## Homework

1. **T2** Chapter 11 - Pages 379-387