**Functions in C++**

# Function

* A function is a set of statements that take inputs, do some specific computation and produces output.
* The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.
* The function initiating the function call is called the caller, and the function being called is the callee or called function.

**Basic Syntax:**

return-type identifier() // identifier replaced with the name of your function

{

// Your code here

}

// An example function that takes two parameters 'x' and 'y' as input and returns max of two input numbers

#include <stdio.h>

// function definition

int max(int x, int y) { // x, y formal parameters

if (x > y)

return x;

else

return y;

}

// main function

int main(void) {

int a = 10, b = 20;

// Calling function max

int m = max(a, b); // a, b actual parameters

printf("m is %d", m);

return 0;

}

Output:

m is 20

# Return values

First, your function has to indicate what type of value will be returned. This is done by setting the function’s return type, which is the type that is defined before the function’s name.

Second, inside the function that will return a value, we use a return statement to indicate the specific value being returned to the caller. The specific value returned from a function is called the return value.

## Void return value

To tell the compiler that a function does not return a value, a return type of void is used.

## Best practice

* If a function has a non-void return type, it must return a value of that type (using a return statement). Failure to do so will result in undefined behaviour.
* Your main function should return 0 if the program ran normally.
* C++ disallows calling the main function explicitly.

NOTE

The return type of a function cannot be a function type or an array type (but can be a pointer or reference to those).

# Return from void functions in C/C++

Void functions are “void” due to the fact that they are not supposed to return values. True, but not completely.

We cannot return values but there is something we can surely return from void functions.

## A void function can do return

We can simply write return statement in a void fun().

In-fact it is considered a good practice (for readability of code) to write return; statement to indicate end of function.

//In C and C++

#include <stdio.h>

void fun() {

printf("Inside void function\n");

return; // We can write return in void

}

int main() {

fun();

//int retval = fun(); **// CE: void value not ignored as it ought to be**

return 0;

}

Output:

Inside void function

## A void fun() can return another void function

//In C and C++

#include <stdio.h>

void work() { printf("Inside void function work\n"); }

void test() {

printf("Inside void function test\n");

**return work(); // return void function**

}

int main() {

test();

return 0;

}

Output:

Inside void function test

Inside void function work

## A void() can return a void value

A void() cannot return a value that can be used.

But it can return a value which is void without giving an error.

//In C and C++

#include <stdio.h>

void test() {

printf("Inside void function test\n");

return (void)"Void value"; // returning void value

}

int main() {

test();

return 0;

}

Output:

Inside void function test

# Trailing return type (since C++11)

Trailing return type, useful if the return type depends on argument names, such as

template <class T, class U>

auto add(T t, U u) -> decltype(t + u);

or is complicated, such as in

auto fpif(int)->int(\*)(int)

# Nested functions are not supported

Unlike some other programming languages, in C++, functions cannot be defined inside other functions.

Since C++11 Lambda can be used to overcome this limitation.

# Function parameters and arguments

A **function parameter** is a variable used in a function. Function parameters work almost identically to variables defined inside the function, but with one difference: they are always initialized with a value provided by the caller of the function.

An **argument** is a value that is passed from the caller to the function when a function is called.

NOTE:

The C++ specification does not define whether arguments are matched with parameters in left to right order or right to left order.

## Lifetime and scope

Function parameters and variables defined inside function have local scope and lifetime.

Static variable inside a function have local scope but lifetime till execution of program.

NOTE:

Lifetime is a runtime property, and scope is a compile-time property.

Best practice:

Define your local variables as close to their first use as reasonable.

# Forward declarations and definitions

A forward declaration allows us to tell the compiler about the existence of an identifier before actually defining the identifier.

To write a forward declaration for a function, we use a declaration statement called a function prototype. The function prototype consists of the function’s return type, name, parameters, but no function body (the curly braces and everything in between them), terminated with a semicolon.

Here’s a function prototype for the add function:

int add(int x, int y); // function prototype includes return type, name, parameters, and semicolon. No function body!

**What happens if they forward declare a function but do not define it?**

It depends. If a forward declaration is made, but the function is never called, the program will compile and run fine. However, if a forward declaration is made and the function is called, but the program never defines the function, the program will compile okay, but the linker will complain that it can’t resolve the function call.

# Naming collisions and an introduction to namespaces

## Naming collision (or naming conflict)

C++ requires that all identifiers be non-ambiguous. If two identical identifiers are introduced into the same program in a way that the compiler or linker can’t tell them apart, the compiler or linker will produce an error. This error is generally referred to as a naming collision (or naming conflict).

Most naming collisions occur in two cases:

1. Two (or more) definitions for a function (or global variable) are introduced into separate files that are compiled into the same program. This will result in a linker error.
2. Two (or more) definitions for a function (or global variable) are introduced into the same file (often via an #include). This will result in a compiler error.

## What is a namespace?

A namespace is a region that allows you to declare names inside of it for the purpose of disambiguation. The namespace provides a scope (called namespace scope) to the names declared inside of it -- which simply means that any name declared inside the namespace won’t be mistaken for identical names in other scopes.

## The global namespace

In C++, any name that is not defined inside a class, function, or a namespace is considered to be part of an implicitly defined namespace called the global namespace (sometimes also called the global scope).

## Accessing an identifier that is defined in a namespace

### Scope resolution operator :: (Explicit namespace qualifier)

The :: symbol is an operator called the scope resolution operator. The identifier to the left of the :: symbol identifies the namespace that the name to the right of the :: symbol is contained within. If no identifier to the left of the :: symbol is provided, the global namespace is assumed.

**Best practice:** *Use explicit namespace prefixes to access identifiers defined in a namespace.*

### using directive

A using directive tells the compiler to check a specified namespace when trying to resolve an identifier that has no namespace prefix. So in the above example, when the compiler goes to determine what identifier cout is, it will check both locally (where it is undefined) and in the std namespace (where it will match to std::cout).

**Warning:** *Avoid using directives (such as using namespace std;) at the top of your program. They violate the reason why namespaces were added in the first place.*

#include <iostream>

using namespace std; // this is a using directive telling the compiler to check the std namespace when resolving identifiers with no prefix

int main() {

std::cout << "Hello world!\n"; // when we say cout, we mean the cout defined in the std namespace

cout << "Hello world!\n"; // cout has no prefix, so the compiler will check to see if cout is defined locally or in namespace std

return 0;

}

# Default Arguments in C++

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn’t provide a value for the argument with default value.

Once default value is used for an argument, all subsequent arguments must have default value.

One interesting note: **Default parameters won’t work for functions called through function pointers**.

#include <stdio.h>

int sum(int x, int y, int z=0, int w=0);

// A function with default arguments, it can be called with 2, 3 or 4 arguments

int sum(int x, int y, int z, int w) {

return (x + y + z + w);

}

int main() {

printf("sum : %d\n",sum(10, 15));

printf("sum : %d\n",sum(10, 15, 25));

printf("sum : %d\n",sum(10, 15, 25, 30));

return 0;

}

G:\coding\test>g++ -Wall -g cppmain.cpp -o cppout

G:\coding\test>cppout.exe

sum : 25

sum : 50

sum : 80

// int sum(int x, int y, int z=0, int w)

G:\coding\test>g++ -Wall -g cppmain.cpp -o cppout

cppmain.cpp: In function 'int sum(int, int, int, int)':

cppmain.cpp:7:5: error: default argument missing for parameter 4 of 'int sum(int

, int, int, int)'

int sum(int x, int y, int z=0, int w)

^

G:\coding\test>

In C,

CE: expected ';', ',' or ')' before '=' token

int sum(int x, int y, int z=0, int w=0)

^

# Returning multiple values from a function using Tuple and Pair in C++

One method to do the same is by using pointers, structures or global variables, already discussed in functions in C.

Another method is using tuples (for returning multiple values) and pair (for two values).

We can declare the function with return type as pair or tuple and can pack the values to be returned and return the packed set of values.

The returned values can be unpacked in the calling function.

**std::tuple**

A tuple is an object capable to hold a collection of elements where each element can be of a different type.

Class template std::tuple is a fixed-size collection of heterogeneous values

**std::pair**

This class couples together a pair of values, which may be of different types

A pair is a specific case of a std::tuple with two elements

Note: Tuple can also be used to return two values instead of using pair .

#include <iostream>

#include <tuple> // std::tuple, std::get, std::tie, std::ignore

using namespace std;

// A Method that returns multiple values using tuple in C++.

tuple<int, int, char> foo(int n1, int n2) {

return make\_tuple(n2, n1, 'a'); // Packing values to return a tuple

}

// A Method returns a pair of values using pair

std::pair<int, int> foo1(int num1, int num2) {

return std::make\_pair(num2, num1); // Packing two values to return a pair

}

int main() {

int a,b;

char cc;

// Unpack the elements returned by foo

tie(a, b, cc) = foo(5, 10);

// Storing returned values in a tuple

tuple<int, int, char> tp = foo(6, 13);

// Storing returned values in a pair

pair<int, int> p = foo1(5,2);

cout << "Values returned by tuple: ";

cout << a << " " << b << " " << cc << endl;

cout << "Values returned by tuple: ";

cout << get<0>(tp) << " " << get<1>(tp) << " " << get<2>(tp) << endl;

cout << "Values returned by Pair: ";

cout << p.first << " " << p.second << endl;

return 0;

}

Output:

Values returned by tuple: 10 5 a

Values returned by tuple: 13 6 a

Values returned by Pair: 2 5

# Function Call Puzzle

Predict the output of this when compiled with C and C++ compilers.

#include <stdio.h>

void func() { /\* definition \*/ }

int main() {

func();

func(2); // In C++, CE: too many arguments to function 'void func()'

}

The above program compiles fine in C, but doesn’t compiler in C++.

In C++, func() is equivalent to func(void)

In C, func() is equivalent to func(…)

# Function Pointer

we can have pointers to functions

#include <stdio.h>

// A normal function with an int parameter and void return type

void fun(int a) {

printf("Value of a is %d\n", a);

}

int main() {

void (\*fun\_ptr)(int) = &fun; // fun\_ptr is a pointer to function fun()

/\* The above line is equivalent of following two

void (\*fun\_ptr)(int);

fun\_ptr = &fun;

\*/

(\*fun\_ptr)(10); // Invoking fun() using fun\_ptr

return 0;

}

Output:

Value of a is 10

Following are some interesting facts about function pointers.

1. Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
2. Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
3. A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment.

void (\*fun\_ptr)(int) = fun; // & removed

fun\_ptr(10); // \* removed // implicit dereference

1. Like normal pointers, we can have an array of function pointers. Below example shows syntax for array of pointers.

void (\*fun\_ptr\_arr[])(int, int) = {add, subtract, multiply};

1. Function pointer can be used in place of switch case.
2. Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
3. Many object oriented features in C++ are implemented using function pointers in C. For example virtual functions. Class methods are another example implemented using function pointers.

One interesting note: **Default parameters won’t work for functions called through function pointers**. Default parameters are resolved at compile-time (that is, if you don’t supply an argument for a defaulted parameter, the compiler substitutes one in for you when the code is compiled). However, function pointers are resolved at run-time. Consequently, default parameters cannot be resolved when making a function call with a function pointer. You’ll explicitly have to pass in values for any defaulted parameters in this case.

## typedef or type aliases

typedefs can be used to make pointers to functions look more like regular variables

typedef bool (\*validateFcn)(int, int);

This defines a typedef called “validateFcn” that is a pointer to a function that takes two ints and returns a bool.

Now instead of doing this

bool validate(int x, int y, bool (\*fcnPtr)(int, int)); // ugly

You can do this

bool validate(int x, int y, validateFcn pfcn) // clean

Which reads a lot nicer! However, the syntax to define the typedef itself can be difficult to remember.

**In C++11**, you can instead use type aliases to create aliases for function pointers types:

using validateFcn = bool(\*)(int, int); // type alias

This reads more naturally than the equivalent typedef, since the name of the alias and the alias definition are placed on opposite sides of the equals sign.

Using a type alias is identical to using a typedef:

bool validate(int x, int y, validateFcn pfcn) // clean

## Using std::function in C++11

Introduced in **C++11**, an alternate method of defining and storing function pointers is to use std::function, which is part of the standard library <functional> header. To define a function pointer using this method, declare a std::function object like so:

#include <functional>

bool validate(int x, int y, std::function<bool(int, int)> fcn);

// std::function method that returns a bool and takes two int parameters

#include <functional>

#include <iostream>

int foo() {    return 5; }

int goo() {    return 6; }

int main() {

    std::function<int()> fcnPtr;

// declare function pointer that returns an int and takes no parameters

    fcnPtr = goo; // fcnPtr now points to function goo

    std::cout << fcnPtr(); // call the function just like normal

    return 0;

}

## Pointers to member functions in C++

This is how C++ uses function pointers when dealing with member functions of classes or structs. These are invoked using an object pointer or a this call. They are type safe in that you can only call members of that class (or derivatives) using a pointer of that type. This example also demonstrates the use of a typedef for the pointer to member function add for simplicity. Function pointers to static member functions are done in the traditional 'C' style because there is no object pointer or this call required.

#include <iostream>

using namespace std;

class Foo {

public:

int add(int i, int j) { return i+j; }

int mult(int i, int j) { return i\*j; }

static int negate(int i) { return -i; }

};

int bar1(int i, int j, Foo\* pFoo, int(Foo::\*pfn)(int,int)) {

return (pFoo->\*pfn)(i,j);

}

typedef int(Foo::\*Foo\_pfn)(int,int);

int bar2(int i, int j, Foo\* pFoo, Foo\_pfn pfn) {

return (pFoo->\*pfn)(i,j);

}

typedef int(\*PFN)(int);

int bar3(int i, PFN pfn) {

return pfn(i);

}

int main() {

Foo foo;

cout << "Foo::add(2,4) = " << bar1(2,4, &foo, &Foo::add) << endl;

cout << "Foo::mult(3,5) = " << bar2(3,5, &foo, &Foo::mult) << endl;

cout << "Foo::negate(6) = " << bar3(6, &Foo::negate) << endl;

return 0;

}

# Functors in C++

**Functors (Not Functions)**

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

**Functors** are objects that can be treated as though they are a function or function pointer.

**A functor (or function object) is a C++ class that acts like a function**. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the operator().

MyFunctor(10);

Is same as

MyFunctor.operator()(10);

// C++ program to demonstrate working of functors.

#include <iostream>

using namespace std;

#include <algorithm>

// A Functor

class increment {

private:

int num;

public:

increment(int n) : num(n) { }

//operator overloading enables calling operator function () on objects of increment

int operator () (int arr\_num) const {

return num + arr\_num;

}

};

// Driver code

int main() {

int arr[] = {1, 2, 3, 4, 5};

int n = sizeof(arr)/sizeof(arr[0]);

int to\_add = 5;

transform(arr, arr+n, arr, increment(to\_add));

for (int i=0; i<n; i++)

cout << arr[i] << " ";

return 0;

}

Output:

6 7 8 9 10

transform(arr, arr+n, arr, increment(to\_add));

is the same as writing below two lines,

increment obj(to\_add); // Creating object of increment

transform(arr, arr+n, arr, obj); // Calling () on object

Functors can be used effectively in conjunction with C++ STLs.

# Return statement vs exit() in main()

In C++, what is the difference between exit(0) and return 0 ?

**exit(0)** is used to exit from program, destructors for locally scoped non-static objects are not called.

**return 0;** destructors are called if return 0 is used.

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

using namespace std;

class Test {

public:

Test() { printf("Inside Test's Constructor\n"); }

~Test() {printf("Inside Test's Destructor\n"); }

};

int main() {

Test t1; // destrutor for t1 will not be called

static Test t2;

// using exit(0) to exit from main

exit(0);

}

Output:

Inside Test's Constructor

Inside Test's Constructor

Inside Test's Destructor // static object will always be destroyed

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

using namespace std;

class Test {

public:

Test() { printf("Inside Test's Constructor\n"); }

~Test() {printf("Inside Test's Destructor\n"); }

};

int main() {

Test t1;

static Test t2;

// using return 0 to exit from main

return 0;

}

Output:

Inside Test's Constructor

Inside Test's Constructor

Inside Test's Destructor

Inside Test's Destructor

# Return type deduction

For this section visit:

<https://en.cppreference.com/w/cpp/language/function>

# Member functions in C++

Check CPP OOP classes and objects

# Inline Functions in C++

Check document on Inline function

# Virtual Functions

Check multiple inheritance in CPP OOP

# References

<https://www.geeksforgeeks.org/c-plus-plus/>

<https://www.learncpp.com/>

<https://en.cppreference.com/w/cpp/language/functions>

# END