

Function Overloading

In C++, *function overloading* enables functions to handle different types of input and return different types. It allows multiple definitions for the same function name, but all of these definitions must differ in their arguments.

Calling a Function

```
// calling a function  
print();
```

In C++, when we define a function, it is not executed automatically. To execute it, we need to “call” the function by specifying its name followed by a pair of parentheses `()`.

Return Values

```
#include <iostream>

int sum(int a, int b);

int main() {
    int r = sum(10, 20);
    std::cout << r;
}

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

A function that returns a value must have a `return` statement. The data type of the return value also must match the method's declared return type.

On the other hand, a `void` function (one that does not return anything) does not require a `return` statement.

Built-in Functions

```
#include <iostream>
#include <cmath>

int main() {

    // sqrt() is from cmath
    std::cout << sqrt(10);

}
```

C++ has many built-in functions. In order to use them, we have to import the required library using `#include`.

Parameters

```
#include <iostream>

void print(int);

int main() {
    print(10);
}

void print(int x) {
    std::cout << x;
}
```

Function parameters are placeholders for values passed to the function. They act as variables inside a function.

Here, `x` is a parameter that holds a value of 10 when it's called.

Default Arguments

In C++, *default arguments* can be added to function declarations so that it is possible to call the function without including those arguments. If those arguments are included the default value is overwritten. Function parameters are read from left to right, so default parameters should be placed from right to left.

Scope of Code

```
#include <iostream>

void print();

int i = 10;      // global variable

int main() {
    std::cout << i << "\n";
}

void print() {
    int j = 0;    // local variable
    i = 20;
    std::cout << i << "\n";
    std::cout << j << "\n";
}
```

The *scope* is the region of code that can access or view a given element:

- Variables defined in *global scope* are accessible throughout the program.
- Variables defined in a function have *local scope* and are only accessible inside the function.

`void` Functions

```
#include <iostream>

void print() {
    std::cout << "Hello World!";
}

int main() {
    print();
}
```

In C++, if we declare the type of a function as `void`, it does not return a value. These functions are useful for a set of statements that do not require returning a value.

Function Arguments

```
#include <iostream>

void print(int);

int main() {
    print(10);
    // the argument 10 is received as input value
}

// parameter a is defined for the function print
void print(int a) {
    std::cout << a;
}
```

In C++, the values passed to a function are known as arguments. They represent the actual input values.

Function Declarations in Header file

```
// ----- main.cpp -----

#include <iostream>
#include "functions.hpp"

int main() {

    std::cout << say_hi("Sabaa");

}

// ----- functions.hpp -----

// function declaration
std::string say_hi(std::string name);

// ----- functions.cpp -----

#include <string>
#include "functions.hpp"

// function definition
std::string say_hi(std::string name) {

    return "Hey there, " + name + "!\n";

}
```

C++ functions typically have two parts: declaration and definition.

Function declarations are generally stored in a *header file* (.hpp or .h) and function definitions (body of the function that defines how it is implemented) are written in the .cpp file.

Function Template

A *function template* is a C++ tool that allows programmers to add data types as parameters, enabling a function to behave the same with different types of parameters. The use of *function templates* and *template parameters* is a great C++ resource to produce cleaner code, as it prevents function duplication.

Functions Definitions

In C++, it is common to store function definitions in a separate `.cpp` file from the `main()` function. This separation results in a more efficient implementation.

Note: If the file containing the `main()` function needs to be recompiled, it is not necessary to recompile the files containing the function definitions.