C++ Programming Language

Hello World

Compile Command

Using GNU, the compilation command is `g++` followed by the file name. Here, the name of the source file is **hello.cpp**.

```
g++ hello.cpp
```

Execute Command

The execution command is `./` followed by the file name. Here, the name of the executable file is **a.out**.

```
./a.out
```

Single-line Comments

Single-line comments are created using two consecutive forward slashes. The compiler ignores any text after `//` on the same line.

```
// This line will denote a comment in C++
```

Multi-line Comments

Multi-line comments are created using `/*` to begin the comment, and `*/` to end the comment. The compiler ignores any text in between.

```
/*
This is all commented out.
None of it is going to run!
*/
```

Program Structure

The program runs line by line, from top to bottom:

- The first line instructs the compiler to locate the file that contains a library called `iostream`. This library contains code that allows for input and output.
- The `main()` function houses all the instructions for the program.

```
#include <iostream>

int main() {

    std::cout << "1\n";
    std::cout << "2\n";
    std::cout << "3\n";
}
```

Basic Output

`std::cout` is the "character output stream" and it is used to write to the standard output. It is followed by the symbols `<<` and the value to be displayed.

```
std::cout << "Hello World!\n";
```

New Line

The escape sequence `\n` (backward slash and the letter n) generates a new line in a text string.

```
std::cout << "Hello\n";
std::cout << "Hello again\n";
```

Getting Started

Why C++?

- C++ is developed by Bjarne Stroustrup at Bell Labs in 1979.
- C++ adds object-oriented programming and many other new features to C.

- C++ is fast, flexible, and well-supported across multiple platforms.
- Being one of the most popular languages today, C++ has a wide range of applications across many fields.

C++ Compile and Execute

C++ a compiled language, which means a compiler needs to first translate your C++ *source code* into *machine code* before it can be run. There are two common ways for running C++ programs: using the command line or an IDE.

- On the command line, type `g++` and the filename to compile your program, then execute it with `./` and the name of the executable.
- On an IDE, explore external resources because the process is different depends on which IDE is being used.

```
g++ hello.cpp -o hello
./hello
```

C++ Style Guide

Style is what we call the conventions that govern our C++ code. These rules exist to keep the code base manageable and readable.

Here are some basic tips from Google's C++ Style Guide:

- `#include` statements are mostly written **at the beginning** of any C/C++ program.
- Names can never start with a digit or be the same as a predefined C++ keyword.
- Types, variable, operators, and literal values should be separated by **one space horizontally**.
- Classes, functions, and global variables should be separated by **one space vertically**.
- All indentations should be **two spaces** at a time.

```
#include <iostream>
#include <string>
using namespace std;
```

```
// This program print out “Hello World!”
int main() {
  string message = "Hello World!\n";
  cout << message;
  return 0;
}
```

Variables and Syntax

User Input

`std::cin`, which stands for “character input”, reads user input from the keyboard.

Here, the user can enter a number, press enter, and that number will get stored in `tip`.

```
int tip = 0;

std::cout << "Enter amount: ";
std::cin >> tip;
```

Variables

A variable refers to a storage location in the computer’s memory that one can set aside to save, retrieve, and manipulate data.

```
// Declare a variable
int score;
// Initialize a variable
score = 0;
```

Arithmetic Operators

C++ supports different types of arithmetic operators that can perform common mathematical operations:

- `+` addition
- ` ` subtraction

- `*` multiplication
- `/` division
- `%` modulo (yields the remainder)

```
int x = 0;
x = 4 + 2;  // x is now 6
x = 4 - 2;  // x is now 2
x = 4 * 2;  // x is now 8
x = 4 / 2;  // x is now 2
x = 4 % 2;  // x is now 0
```

`double` Type

`double` is a type for storing floating point (decimal) numbers. Double variables typically require 8 bytes of memory space.

```
double price = 8.99;
double pi = 3.14159;
```

Chaining the Output

`std::cout` can output multiple values by chaining them using the output operator `<<`.

Here, the output would be `I'm 28.`

```
int age = 28;

std::cout << "I'm " << age << ".\n";
```

`int` Type

`int` is a type for storing integer (whole) numbers. An integer typically requires 4 bytes of memory space and ranges from -2^{31} to $2^{31}-1$.

```
int year = 1991;
int age = 28;
```

`char` Type

`char` is a type for storing individual characters. Characters are wrapped in single quotes `'`. Characters typically require 1 byte of memory space and range from -128 to 127.

```
char grade = 'A';
char punctuation = '?';
```

`string` Type

`std::string` is a type for storing text strings. Strings are wrapped in double quotes `"`.

```
std::string message = "good nite";
std::string user = "codey";
```

`bool` Type

`bool` is a type for storing `true` or `false` boolean values. Booleans typically require 1 byte of memory space.

```
bool organ_donor = true;
bool late_to_work = false;
```

Conditionals & Logic

`if` Statement

An `if` statement is used to test an expression for truth.

- If the condition evaluates to `true`, then the code within the block is executed; otherwise, it will be skipped.

```
if (a == 10) {
    // Code goes here
}
```

`else` Clause

An `else` clause can be added to an `if` statement.

- If the condition evaluates to `true`, code in the `if` part is executed.
- If the condition evaluates to `false`, code in the `else` part is executed.

```
if (year == 1991) {
    // This runs if it is true
    }
else {
    // This runs if it is false
}
```

Relational Operators

Relational operators are used to compare two values and return `true` or `false` depending on the comparison:

- `==` equal to
- `!=` not equal to
- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

```
if (a > 10) {
    // ☝ means greater than
}
```

`else if` Statement

One or more `else if` statements can be added in between the `if` and `else` to provide additional condition(s) to check.

```
if (apple > 8) {
  // Some code here
}
```

```
else if (apple > 6) {
  // Some code here
}
else {
  // Some code here
}
```

Conditional Statements

Conditional statements are used to control the flow of code execution by testing for a condition for truth.

- `if` statements execute code only if the provided condition is `true`.
- `else` statements execute code only if the provided condition in the `if` statement is `false`.
- one or more `else if` statements can be added in between the `if` and `else` to provide additional condition(s) to check.

Some useful tricks with conditional statements:

- It is possible to condense an `ifelse` expression into a single statement using the following syntax:

```
variable = (condition) ? condition_is_true : condition_is_f
alse;
```

```
int temperature = 60;

if (temperature < 65) {
  std::cout << "Too cold!";
}
else if (temperature > 75) {
  std::cout << "Too hot!";
}
else    // brackets may be omitted here
  std::cout << "Just right...";
```

- Curly brackets `{ }` may be omitted if there is only a single statement inside a conditional statement.

`while` Loop

A `while` loop statement repeatedly executes the code block within as long as the condition is `true`. The moment the condition becomes `false`, the program will exit the loop.

Note that the `while` loop might not ever run. If the condition is `false` initially, the code block will be skipped.

```
while (password != 1234) {

  std::cout << "Try again: ";
  std::cin >> password;

}
```

`for` Loop

A `for` loop executes a code block a specific number of times. It has three parts:

- The initialization of a counter
- The continue condition
- The increment/decrement of the counter

This example prints 0 to 9 on the screen.

```
for (int i = 0; i < 10; i++) {

  std::cout << i << "\n";

}
```

`switch` Statement

A `switch` statement provides a means of checking an expression against various `case`s. If there is a match, the code within starts to execute. The `break` keyword can be used to terminate a case.

`default` is executed when no case matches.

```
switch (grade) {
  case 9:
    std::cout << "Freshman\n";
    break;
  case 10:
    std::cout << "Sophomore\n";
    break;
  case 11:
    std::cout << "Junior\n";
    break;
  case 12:
    std::cout << "Senior\n";
    break;
  default:
    std::cout << "Invalid\n";
    break;
}
```

Logical Operators

Logical operators can be used to combine two different conditions.

- `&&` requires both to be true (`and`)
- `||` requires either to be true (`or`)
- `!` negates the result (`not`)

```
if (coffee > 0 && donut > 1) {
  // Code runs if both are true
}

if (coffee > 0 || donut > 1) {
  // Code runs if either is true
}

if (!tired) {
  // Code runs if tired is false
}
```

Break and Continue

In C++, the `break` keyword is used to exit a switch or loop.

The `continue` keyword is used to skip an iteration of a loop.

```
// Prints: 0123
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  std::cout << i;
}

// Prints: 012356789
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  std::cout << i;
}
```

Functions

Function Overloading

With *function overloading*, C++ functions can have the same name but handle different input parameters.

At least one of the following criteria must be true in order for functions to be properly overloaded:

- Each function has different types of parameters.
- Each function has a different number of parameters.

The function return type is NOT used to differentiate overloaded functions.

```
#include <iostream>

int add(int a, int b) {
  return a + b;
```

```
}

double add(double a, double b) {
  return a + b;
}

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

int main() {
  std::cout << add(3, 2); // Calls add(int, int)
  std::cout << "\n";
  std::cout << add(5.3, 1.4); // Calls add(double, double)
  std::cout << "\n";
  std::cout << add(2, 6, 9);  // Calls add(int, int, int)
}
```

Function Parameters

When calling a function with multiple parameters, the number and order of the arguments must match with the parameters.

Default parameters initialize to a default value if an argument is not provided in the function call.

Pass by reference lets the function modify the arguments variables. Use the `&` operator to indicate that a parameter is passed by reference.

```
#include <iostream>

double totalPrice(int items, double price = 9.99) {
  return items * price;
}

// Pass by reference
void addOne(int &i) {
  i += 1;
}
```

```
int main() {
  std::cout << totalPrice(10) << "\n";  // Output: 99.9

  int num = 2;
  addOne(num);
  std::cout << num; // Output: 3

  return 0;
}
```

Command Line Arguments

Command line arguments are optional arguments passed to the `main()` function of a C++ program.

Passing command line arguments is as easy as appending the arguments after the executable name. For example:

```
./greeting Hello World
```

In order to access command line arguments, the new form of `main()` takes two arguments:

- `argc`: the number of command line arguments.
- `argv`: an array containing the values of command line arguments.

```
#include <iostream>

int main(int argc, char* argv[]) {
  std::cout << argc << "\n";

  for(int i = 0; i < argc; i++) {
    std::cout << argv[i] << "\n";
  }
  return 0;
}
```

Introduction to Functions

A *function* in C++ contains a set of instructions that are executed when it is called.

A function declaration is composed of three parts:

1. Function return type
2. Function name
3. Function parameters

A function can be called by specifying its name followed by a pair of parentheses `()`.

```
#include <iostream>

void printTitle() {
  std::string msg = "Codecademy\n";
  std::cout << msg;
}

int main(){
  printTitle();

  return 0;
}
```

Object Oriented Programming

Class Members

A class is comprised of class members:

- *Attributes*, also known as member data, consist of information about an instance of the class.
- *Methods*, also known as member functions, are functions that can be used with an instance of the class.

```
class City {

  // Attribute
```

```
  int population;

public:
  // Method
  void add_resident() {
    population++;
  }

};
```

Constructor

For a C++ class, a *constructor* is a special kind of method that enables control regarding how the objects of a class should be created. Different class constructors can be specified for the same class, but each constructor signature must be unique.

```
#include "city.hpp"

class City {

  std::string name;
  int population;

public:
  City(std::string new_name, int new_pop);

};
```

Objects

In C++, an *object* is an instance of a class that encapsulates data and functionality pertaining to that data.

```
City nyc;
```

Class

A C++ class is a user-defined data type that encapsulates information and behavior about an object. It serves as a blueprint for future inherited classes.

```
class Person {};
```

Access Control Operators

C++ classes have access control operators that designate the scope of class members:

- `public`
- `private`

`public` members are accessible everywhere; `private` members can only be accessed from within the same instance of the class or from friends classes.

```
class City {

  int population;

public:
  void add_resident() {
    population++;
  }

private:
  bool is_capital;

};
```

Constructors

For a C++ class, a *constructor* is a special kind of method that enables control regarding how the objects of a class should be created. Different class constructors can be specified for the same class, but each constructor signature must be unique.

A constructor can have multiple parameters as well as default parameter values.

In order to initialize `const` or reference type attributes, use *member initializer lists* instead of normal constructors.

```
#include <iostream>

using namespace std;

class House {
private:
  std::string location;
  int rooms;

public:
  // Constructor with default parameters
  House(std::string loc = "New York", int num = 5) {
    location = loc;
    rooms = num;
  }

  // Destructor
  ~House() {
    std::cout << "Moved away from " << location << "\n";
  }
};

int main()
{
  House default_house;  // Calls House("New York", 5)
  House texas_house("Texas");   // Calls House("Texas", 5)
  House big_florida_house("Florida", 10);   // Calls House("F

  return 0;
}
```

Inheritance

In C++, a class can inherit attributes and methods from another class. In an inheritance relationship, there are two categories of classes:

- *Base class*: The class being inherited from.
- *Derived class*: The class that inherits from the base class.

It’s possible to have multi-level inheritance where classes are constructed in order from the “most base” class to the “most derived” class.

```
#include <iostream>

class Base {
public:
  int base_id;

  Base(int new_base) : base_id(new_base) {}
};

class Derived: public Base {
public:
  int derived_id;

  Derived(int new_base, int new_derived)
    : Base(new_base), derived_id(new_derived) {}

  void show() {
    std::cout << base_id << " " << derived_id;
  }
};

int main() {
  Derived temp(1, 2);

  temp.show(); // Outputs: 1 2

  return 0;
}
```

Access Specifiers

Access specifiers are C++ keywords that determine the scope of class components:

- `public`: Class members are accessible from anywhere in the program.
- `private`: Class members are only accessible from inside the class.

Encapsulation is achieved by declaring class attributes as `private`:

- Accessor functions: return the value of `private` member variables.
- Mutator functions: change the value of `private` member variables.

```
#include <iostream>

class Computer {
private:
  int password;

public:
  int getPassword() {
    return password;
  }

  void setPassword(int new_password) {
    password = new_password;
  }
};

int main()
{
  Computer dell;

  dell.setPassword(12345);
  std::cout << dell.getPassword();

  return 0;
}
```

Classes and Objects

A C++ *class* is a user-defined data type that encapsulates information and behavior about an object.

A class can have two types of *class members*:

- *Attributes*, also known as member data, consist of information about an instance of the class.
- *Methods*, also known as member functions, are functions that can be used with an instance of the class.

An *object* is an instance of a class and can be created by specifying the class name.

```
#include <iostream>

class Dog {
public:
  int age;

  void sound() {
    std::cout << "woof\n";
  }
};

int main() {
  Dog buddy;

  buddy.age = 5;

  buddy.sound();    // Outputs: woof
}
```

Polymorphism

In C++, *polymorphism* occurs when a derived class overrides a method inherited from its base class with the same function signature.

Polymorphism gives a method many “forms”. Which form is executed depends on the type of the caller object.

```
#include <iostream>

class Employee {
public:
```

```
  void salary() {
    std::cout << "Normal salary.\n";
  }
};

class Manager: public Employee {
public:
  void salary() {
    std::cout << "Normal salary and bonus.\n";
  }
};

int main() {
  Employee newbie;
  Manager boss;

  newbie.salary(); // Outputs: Normal salary.
  boss.salary(); // Outputs: Normal salary and bonus.

  return 0;
}
```

Built in Data Structures

vectors

In C++, a vector is a data structure that stores a sequence of elements that can be accessed by index.

Unlike arrays, vectors can dynamically shrink and grow in size.

The standard `<vector>` library provide methods for vector operations:

- `.push_back()`: add element to the end of the vector.
- `.pop_back()`: remove element from the end of the vector.
- `.size()`: return the size of the vector.
- `.empty()`: return whether the vector is empty.

```
#include <iostream>
#include <vector>

int main () {
  std::vector <int> primes = {2, 3, 5, 7, 11};

  std::cout << primes[2];   // Outputs: 5

  primes.push_back(13);
  primes.push_back(17);
  primes.pop_back();

  for (int i = 0; i < primes.size(); i++) {
      std::cout << primes[i] << " ";
  }
  // Outputs: 2 3 5 7 11 13

  return 0;
}
```

Stacks and Queues

In C++, *stacks* and *queues* are data structures for storing data in specific orders.

Stacks are designed to operate in a **Last-In-First-Out** context (LIFO), where elements are inserted and extracted only from one end of the container.

- `.push()` add an element at the top of the stack.
- `.pop()` remove the element at the top of the stack.

Queues are designed to operate in a **First-In-First-Out** context (FIFO), where elements are inserted into one end of the container and extracted from the other.

- `.push()` add an element at the end of the queue.
- `.pop()` remove the element at the front of the queue.

```
#include <iostream>
#include <stack>
#include <queue>

int main()
{
  std::stack<int> tower;

  tower.push(3);
  tower.push(2);
  tower.push(1);

  while(!tower.empty()) {
    std::cout << tower.top() << " ";
    tower.pop();
  }
  // Outputs: 1 2 3

  std::queue<int> order;

  order.push(10);
  order.push(9);
  order.push(8);

  while(!order.empty()) {
    std::cout << order.front() << " ";
    order.pop();
  }
  // Outputs: 10 9 8

  return 0;
}
```

Sets

In C++, a *set* is a data structure that contains a collection of unique elements. Elements of a set are index by their own values, or *keys*.

A set cannot contain duplicate elements. Once an element has been added to a set, that element cannot be modified.

The following methods apply to both `unordered_set` and `set`:

- `.insert()`: add an element to the set.
- `.erase()`: removes an element from the set.
- `.count()`: check whether an element exists in the set.
- `.size()`: return the size of the set.

```
#include <iostream>
#include <unordered_set>
#include <set>

int main()
{
  std::unordered_set<int> primes({2, 3, 5, 7});

  primes.insert(11);
  primes.insert(13);
  primes.insert(11);  // Duplicates are not inserted

  primes.erase(2);
  primes.erase(13);

  // Outputs: primes does not contain 2.
  if(primes.count(2))
    std::cout << "primes contains 2.\n";
  else
    std::cout << "primes does not contain 2.\n";

  // Outputs: Size of primes: 4
  std::cout << "Size of primes: " << primes.size() << "\n";

  return 0;
}
```

arrays

Arrays in C++ are used to store a collection of values of the same type. The size of an array is specified when it is declared and cannot change afterward.

Use `[]` and an integer index to access an array element. Keep in mind: array indices start with `0`, not `1`!.

A multidimensional array is an "array of arrays" and is declared by adding extra sets of indices to the array name.

```
#include <iostream>

using namespace std;

int main()
{
  char vowels[5] = {'a', 'e', 'i', 'o', 'u'};

  std::cout << vowels[2];   // Outputs: i

  char game[3][3] = {
    {'x', 'o', 'o'} ,
    {'o', 'x', 'x'} ,
    {'o', 'o', 'x'}
  };

    std::cout << game[0][2];    // Outputs: o

  return 0;
}
```

Hash Maps

In C++, a *hash map* is a data structure that contains a collection of unique elements in the form of *key-value* pairs. Elements of a hash map are identified by key values, while the *mapped values* are the content associated with the keys.

Each element of a `map` or `unordered_map` is an object of type `pair`. A `pair` object has two member variables:

- `.first` is the value of the key

- `.second` is the mapped value

The following methods apply to both `unordered_map` and `map`:

- `.insert()`: add an element to the map.
- `.erase()`: removes an element from the map.
- `.count()`: check whether an element exists in the map.
- `.size()`: return the size of the map.
- `[]` operater:
  - If the specified key matches an element in the map, then access the mapped value associated with that key.
  - If the specified key doesn't match any element in the map, add a new element to the map with that key.

```
#include <iostream>
#include <unordered_map>
#include <map>

int main() {
  std::unordered_map<std::string, int> country_codes;

  country_codes.insert({"Thailand", 65});
  country_codes.insert({"Peru", 51});
  country_codes["Japan"] = 81;      // Add a new element
  country_codes["Thailand"] = 66; // Access an element

  country_codes.erase("Peru");

  // Outputs: There isn't a code for Belgium
  if (country_codes.count("Belgium")) {
    std::cout << "There is a code for Belgium\n";
  }
  else {
   std::cout << "There isn't a code for Belgium\n";
  }

  // Outputs: 81
```

```
  std::cout << country_codes["Japan"] << "\n";

  // Outputs: 2
  std::cout << country_codes.size() << "\n";

  // Outputs: Japan 81
  //          Thailand 66
  for(auto it: country_codes){
    std::cout << it.first << " " << it.second << "\n";
  }

  return 0;
}
```

References and Pointers

Pass-By-Reference

In C++, *pass-by-reference* refers to passing parameters to a function by using references.

It allows the ability to:

- Modify the value of the function arguments.
- Avoid making copies of a variable/object for performance reasons.

```
void swap_num(int &i, int &j) {

  int temp = i;
  i = j;
  j = temp;

}

int main() {

  int a = 100;
  int b = 200;
```

```
  swap_num(a, b);

  std::cout << "A is " << a << "\n";
  std::cout << "B is " << b << "\n";

}
```

`const` Reference

In C++, pass-by-reference with `const` can be used for a function where the parameter(s) won't change inside the function.

This saves the computational cost of making a copy of the argument.

```
int triple(int const &i) {

  return i * 3;

}
```

Pointers

In C++, a *pointer* variable stores the memory address of something else. It is created using the `*` sign.

```
int* pointer = &gum;
```

References

In C++, a *reference* variable is an alias for another object. It is created using the `&` sign. Two things to note:

1. Anything done to the reference also happens to the original.
2. Aliases cannot be changed to alias something else.

```
int &sonny = songqiao;
```

Memory Address

In C++, the *memory address* is the location in the memory of an object. It can be accessed with the "address of" operator, `&`.

Given a variable `porcupine_count`, the memory address can be retrieved by printing out `&porcupine_count`. It will return something like: `0x7ffd7caa5b54`.

```
std::cout << &porcupine_count << "\n";
```

Dereference

In C++, a *dereference reference operator*, `*`, can be used to obtain the value pointed to by a pointer variable.

```
int gum = 3;

// * on left side is a pointer
int* pointer = &gum;

// * on right side is a dereference of that pointer
int dereference = *pointer;
```

Vectors

Index

An index refers to an element's position within an ordered list, like a vector or an array. The first element has an index of 0.

A specific element in a vector or an array can be accessed using its index, like `name[index]`.

```
std::vector<double> order = {3.99, 12.99, 2.49};

// What's the first element?
std::cout << order[0];

// What's the last element?
std::cout << order[2];
```

Vectors

In C++, a vector is a dynamic list of items, that can shrink and grow in size. It is created using `std::vector<type> name;` and it can only store values of the same type.

To use vectors, it is necessary to `#include` the `vector` library.

```
#include <iostream>
#include <vector>

int main() {

  std::vector<int> grades(3);

  grades[0] = 90;
  grades[1] = 86;
  grades[2] = 98;

}
```

`.push_back()` & `.pop_back()`

The following functions can be used to add and remove an element in a vector:

- `.push_back()` to add an element to the “end” of a vector
- `.pop_back()` to remove an element from the “end” of a vector

```
std::vector<std::string> wishlist;

wishlist.push_back("Oculus");
wishlist.push_back("Telecaster");

wishlist.pop_back();

std::cout << wishlist.size();
// Prints: 1
```

Vector Type

During
the creation of a C++ vector, the data type of its elements must be specified. Once the vector is created, the type cannot be changed.

`.size()` Function

The `.size()` function can be used to return the number of elements in a vector, like `name.size()`.

```
std::vector<std::string> employees;

employees.push_back("michael");
employees.push_back("jim");
employees.push_back("pam");
employees.push_back("dwight");

std::cout << employees.size();
// Prints: 4
```

Functions

Scope of Code

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.

```
#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";
}
```

Function Declarations in Header file

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.

```
// ~~~~~~ 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 defintion
```

```
std::string say_hi(std::string name) {

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

}
```

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.

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.

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.

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.

Inline Functions

An *inline* function is a function definition, usually in a header file, qualified by the inline keyword, which advises the compiler to insert the function's body where the function call is. If a modification is made in an inline function, it would require all files containing a call to that function to be recompiled.

Return Values

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.

```
#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);
}
```

Parameters

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.

```
#include <iostream>

void print(int);

int main() {
  print(10);
}

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

Functions

A *function* is a set of statements that are executed together when the function is called. Every function has a name, which is used to call the respective function.

```
#include <iostream>

// Declaring a function
void print();

int main() {
  print();
}

// Defining a function
void print() {
  std::cout << "Hello World!";
}
```

Built-in Functions

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

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

int main() {

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

}
```

Calling a Function

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 `()`.

```
// calling a functionprint();
```

`void` Functions

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.

```
#include <iostream>

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

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

Function Declaration & Definition

A C++ function has two parts:

- Function declaration
- Function definition

The declaration includes the function's name, return type, and any parameters.

The definition is the actual body of the function which executes when a function is called. The body of a function is typically enclosed in curly braces.

```
#include <iostream>

// function declaration
void blah();

// main function
int main() {
  blah();
```

```
}

// function definition
void blah() {
  std::cout << "Blah blah";
}
```

Function Arguments

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

```
#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;
}
```