Politecnico di Milano
School of Industrial and Information Engineering
Academic Year 2024-2025

# Algorithms and Parallel Computing

Professor: Danilo Ardagna

*Last updated: October 8, 2024*

**This document is intended for educational purposes only.**
**These are unreviewed notes and may contain errors.**
Made by Roberto Benatuil Valera

# Contents

Chapter 1

# Introduction to C++ Programming

## 1.1 Why C++ and Object-Oriented Programming (OOP)?

### 1.1.1 Why C++?

The purpose of a programming language is to allow you to express your ideas in code. C++ is the language tha most directly allows you to do this from the largest number of application areas, especially engineering and scientific computing. Also C++ is the most widely used language in the world, and it is the language of choice for systems programming and high-performance computing.

C++ is a general-purpose programming language that was developed by Bjarne Stroustrup as an extension of the C language. It is precisely and comprehensively defined by the ISO standard, and that standard is almost universally accepted. Programming concepts that you learn using C++ can be used fairly directly in other languages, such as C, Java, C#, and Python. It is also faster than most other object-oriented languages. Where is this needed? Some examples are:

- In banking and trading systems, where latency is critical.
- In scientific computing, where performance is very important.
- in tiny embedded systems, where memory is very limited, or in large systems, where speed and energy consumption are critical.

We can find some real-world examples of the importance of speed and efficiency:

→ More than 10 years ago, Amazon found that every 100ms of latency cost them 1% in sales.

→ In 2006, Google found that an extra 0.5 seconds in search page generation time dropped traffic by 20%.

→ A broker could lose $4 million in revenues per millisecond if their electronic trading platform is 5 milliseconds behind the competition.

### 1.1.2 C++ in the context of Programming Paradigms

Programming languages have a level of abstraction that allows you to express your ideas in code. The level of abstraction is the amount of detail that you have to deal with when you are writing code. The higher the level of abstraction, the less detail you have to deal with. One can think of the level of abstraction as the distance between the code that you write and the machine code that the computer executes. The history of computer programming is a steady move away from machine-oriented views of programming towards concepts and metaphors that more closely reflect the way in which we ourselves understand the world.

We also have multiple programming paradigms. C++ is a multi-paradigm programming language, which means that it supports several different programming paradigm, which are just ways to think about and approach problems when writing code. Some of the most common ones are:

#### Procedural Programming

Procedural programming is the most basic programming paradigm. It is based on the concept of the procedure call. A procedure is a group of statements that are executed sequentially. Procedural programming is about writing procedures or functions that perform operations on data.

It has a top-down approach, where the problem is broken down into smaller parts, and each part is solved by writing a procedure or function. The main program calls these procedures to perform the required operations. C is a procedural programming language, and C++ is a superset of C, so it supports procedural programming. Some of the advantages of procedural programming are:

- It is easier to comprehend the solution of a smaller and less complicated problem, than to understand the solution of a larger and more complex problem.
- It is easier to test segments of solutions, rather than the whole solution at once. This method allows one to test the solution of each sub-problem separately until the whole solution is tested.
- It is often possible to simplify the logical steps of each sub-problem, so that when taken as a whole, the solution is easier to develop.
- A simplified solution takes less time to develop, and is easier to maintain.

In the end, this paradigm is very useful for writing small programs, but it becomes difficult to manage as the program grows in size and complexity.

#### Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function. In OOP, the data is an object, and the functions that operate on the data are methods.

It has a bottom-up approach, where lower-level tasks are solved first, and then are integrated into higher-level tasks, to provide the solution of a single program. The main program is divided into objects, and the problem is solved by writing methods for these objects. This paradigm promotes code reusability, and favors software modularization.

To use this approach, we need to identify the main abstractions that characterize the application domain, and then define classes that represent these abstractions. We then assemble the various components by identifying the mechanisms that allow the objects to work together to implement the desired functionality.

In this way, applications are easier to understand, maintain, and extend.

### 1.1.3 How is C++ structured?

This language has mainly 3 parts:

- Low-level language features, largely inherited from C. Some of these features are: pointers, data types, flow control, functions, arrays, etc.
- Advanced language features, to define our own data types, such as classes, inheritance, polymorphism, templates, exceptions, etc.
- Standard Library, which is a collection of classes and functions that are part of the C++. It has some useful data structures and algorithms.

## 1.2 Our first C++ program: Hello World!

Let us take a look at the following code:

```cpp
#include <iostream>  // Include the iostream library
using namespace std;  // This allows us to use cout instead of std::cout

int main() {  // The main function is the entry point of the program
    cout << "Hello World!" << endl;  // Print Hello World! to the console
    return 0;  // Return 0, indicating that the program has ended successfully
}
```

`Hello World!` is the first program that people write when they are learning a new programming language. It is a very important program, as it helps you understand the basic structure of this language.

Its purpose is to help you get used to C++ tools, such as the compiler, the program delelopment environment, and the program execution environment. Its almost all "boiler plate", that is, notation, libraries and other support that makes our code simple, comprehensible, trustworthy, and efficient.

### 1.2.1 Standard I/O objects

C++ does not define any input/output operations in the language itself. Instead, it relies on a set of standard I/O objects that are defined in the standard library. This library is included with the line `#include <iostream>`. The most important I/O objects are:

- `cin`: Standard input stream, used to read input from the console.
- `cout`: Standard output stream, used to write output to the console.
- `cerr`: Standard error stream, used to write error messages to the console.
- `clog`: Standard log stream, used to write log messages to the console.
- `endl`: Standard end line, used to end the current line and flush the buffer.

Let us take a look at the following code:

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

int main() {
    cout << "Please enter your name and age:\n";
    string name;
    int age;
    cin >> name >> age;
    cout << "Hello, " << name << " (age " << age << ")\n" << endl;
    return 0;
}
```

In this code, we are using the `cin` object to read the name and age of the user, and the `cout` object to print a greeting message. The `cin` object is used with the `>>` operator, which is called the extraction operator. The `cout` object is used with the `<<` operator, which is called the insertion operator.

We observe that the same operator is used with different types of data, and it is able to handle them correctly. This is called operator overloading, and it is an useful feature of C++. Other examples of operator overloading are seen in this table:

| **Strings** | **Integers and floating-point numbers** |
|---|---|
| `cin >>` reads a word | `cin >>` reads a number |
| `cout <<` writes | `cout <<` writes |
| + concatenates | + adds |
| += s adds the string s at end | += n increments by the int n |
| ++ is an error | ++ increments by 1 |
| - is an error | - subtracts |

Table 1.1: Operations on Strings vs. Numbers in C++

Another useful feature of `<iostream>` is seen in the following code:

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

int main() {
    int sum = 0; value = 0;
    while (cin >> value) {
        sum += value;
    }

    cout << "Sum is: " << sum << endl;
    return 0;
}
```

In this code, we are using the `cin` object in the condition of the `while` loop. When we use an `istream` object as a condition, the effect is to test the state of the stream. If the stream is valid (i.e., the stream hasn't encountered an error), the condition is true. An `istream` becomes invalid when it reaches the end of the file, or when it encounters an invalid input (in this case, any non-integer value). This will cause the condition to be false, and the loop will terminate.

### 1.2.2 Naming variables in C++

In C++, variable names must start with a letter, and can only contain letters, digits, and underscores. They are case-sensitive, and cannot be a reserved word, such as `int`, `double`, `string`, etc.

It is a good practice to use meaningful names for variables, as it makes the code easier to read and understand. For example, the variable `sum` is a better name than `s`, as it is more descriptive.

Short names are acceptable for variables that are used in a small scope, such as loop counters, or when used conventionally, such as `i` for a loop counter, or `x` and `y` for coordinates, for example. It is also not recommended to use names that are too long, as they can make the code harder to read.

### 1.2.3 Simple arithmetic in C++

C++ has the usual arithmetic operators, such as `+`, `-`, `*`, `/`, and `%`. It also has the increment and decrement operators, `++` and `--`.

Let us take a look at the following code:

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

int main() {
    int x = 5;
    int y = 2;
    cout << "x + y = " << x + y << endl;
    cout << "x - y = " << x - y << endl;
    cout << "x * y = " << x * y << endl;
    cout << "x / y = " << x / y << endl;
    cout << "x % y = " << x % y << endl;
```

```
    cout << "x++ = " << x++ << endl;
    cout << "++x = " << ++x << endl;
    cout << "x-- = " << x-- << endl;
    cout << "--x = " << --x << endl;
    return 0;
}

// Output:
// x + y = 7
// x - y = 3
// x * y = 10
// x / y = 2
// x % y = 1
// x++ = 5
// ++x = 7
// x-- = 7
// --x = 5
```

In this code, we are using the arithmetic operators to perform simple arithmetic operations. We are also using the increment and decrement operators to increment and decrement the value of a variable. The difference between the pre-increment and post-increment operators is that the pre-increment operator increments the value of the variable before it is used, while the post-increment operator increments the value of the variable after it is used.

## 1.3 Namespaces in C++

A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

We have already seen the `std` namespace, which is the namespace that contains the standard library. The `using namespace std;` directive tells the compiler to use the `std` namespace for the identifiers that are part of the standard library.

Let us take a look at the following code:

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


namespace foo {
    int value() { return 5; }
}

namespace bar {
    const double pi = 3.1416;
    double value() { return 2 * pi; }
}

int main() {
    cout << foo::value() << endl;
```

```
    cout << bar::value() << endl;
    cout << bar::pi << endl;
    return 0;
}

// Output:
// 5
// 6.2832
// 3.1416
```

In this code, we are defining two namespaces, `foo` and `bar`, that contain the `value` function and the `pi` constant, respectively. We are then using the scope resolution operator (::) to access the identifiers that are part of the namespaces.

The scope resolution operator is used to define the scope of the identifiers. It is used with namespaces, classes, and structures. It is also used to define the scope of the identifiers that are part of the standard library.

To avoid having to use the scope resolution operator every time we want to access an identifier that is part of a namespace, we can use the `using` directive. This directive tells the compiler to use the namespace for the identifiers that are part of the namespace.

We can have two types of `using` directives:

- `using namespace foo;`: This directive tells the compiler to use the `foo` namespace for the identifiers that are part of the namespace. This is properly called a `using` directive.
- `using foo::value;`: This directive tells the compiler to use the `value` identifier that is part of the `foo` namespace. This is called a `using` declaration.

For example, let us take a look at the following codes:

```
// Code 1: example of using directive
#include <iostream>
using namespace std;

namespace foo {
    int value() { return 5; }
}

using namespace foo;

int main() {
    cout << value() << endl;
    return 0;
}

// Output:
// 5
// 6.2832
// 3.1416
```

```cpp
// Code 2: example of using declaration
#include <iostream>
using namespace std;

namespace foo {
    int value() { return 5; }
    int value2() { return 10; }
}

using foo::value;

int main() {
    cout << value() << endl;
    cout << foo::value2() << endl;
    return 0;
}

// Output:
// 5
// 10
```

In the first code, we are using the `using` directive to tell the compiler to use the `foo` namespace for the identifiers that are part of the namespace. In the second code, we are using the `using` declaration to tell the compiler to use the `value` identifier that is part of the `foo` namespace.

## 1.4 Built-in data types in C++

C++ has several built-in data types that are used to define variables. These data types are used to store different types of data, such as integers, floating-point numbers, characters, and booleans.

The type of a variable determines the size and layout of the variable's memory, the range of values that can be stored in the variable, and the set of operations that can be applied to the variable. In C++, all variables must be declared with a type before they can be used, and the type of a variable cannot be changed once it has been declared.

The built-in data types in C++ can be classified into the following categories:

- Integer types: Used to store integer values.
- Floating-point types: Used to store floating-point values.
- Character types: Used to store characters.
- Boolean type: Used to store boolean values.
- Void type: Used to indicate that a function does not return a value.

Let us take a look at the following table:

| S. No | DATA TYPE | Size (in bytes) | RANGE |
|---|---|---|---|
| 1 | short int | 2 | -32768 to +32767 |
| 2 | unsigned short int | 2 | 0 to 65535 |
| 3 | long int | 4 | -2147483648 to 2147483647 |
| 4 | float | 4 | 3.4e-38 to 3.4e+38 |
| 5 | char | 1 | -128 to 127 |
| 6 | unsigned char | 1 | 0 to 255 |
| 7 | unsigned long int | 4 | 0 to 4294967295 |
| 8 | double | 8 | 1.7e-308 to 1.7e+308 |
| 9 | long double | 10 | 1.7e-308 to 1.7e+308 |

Table 1.2: Data Types, Sizes, and Ranges

### 1.4.1 Type checking

C++ is a statically-typed language, which means that the type of a variable is known at compile time. This allows the compiler to perform type checking, which is the process of verifying that the types of the variables in the program are used correctly.

It is not always possible to determine the type of a variable at compile time. Some errors can only be detected at runtime, such as dividing by zero, or accessing an array out of bounds. These errors are called runtime errors, and they can cause the program to crash. The compiler cannot detect these errors, as they depend on the input values that are provided at runtime. Sometimes it can warn you about potential runtime errors, but it cannot guarantee that they will not occur.

Some examples of runtime errors can be seen in the following code:

```cpp
// Implicit narrowing conversion
#include <iostream>
using namespace std;


int main() {
    int x = 1000;
    char c = x;
    cout << "c = " << c << endl;
    return 0;
}


// Output:
// c = 232
```

In this code, we are assigning an integer value to a character variable. The integer value is implicitly converted to a character value, which causes the value to be truncated. This is called an implicit narrowing conversion, and it can cause the loss of data. In this case, the value of the integer is 1000, which is too large to be stored in a character variable, so it is truncated to 232.

Another example of a runtime error can be seen in the following code:

```cpp
// Uninitialized variables
#include <iostream>

int main() {
    int x;
    std::cout << "x = " << x << std::endl;
    return 0;
}

// Output:
// x = 0
```

In this code, we are using an uninitialized variable. The value of the variable is undefined, as it has not been initialized with a value. This can cause the program to behave unpredictably, as the value of the variable depends on the memory location that it is stored in. In this case, the value of the variable is 0, as it is stored in a memory location that is initialized to 0, but this is not guaranteed.

## 1.5 Control structures in C++

Control structures are used to control the flow of the program. They allow you to execute different blocks of code based on certain conditions. There are three main types of control structures in C++:

- Selection structures: Used to execute different blocks of code based on certain conditions.
- Iteration structures: Used to execute the same block of code multiple times.
- Functions: Used to divide the program into smaller parts that can be reused.

### 1.5.1 Selection structures

Selection structures are used to execute different blocks of code based on certain conditions. There are two main types of selection structures in C++:

- `if` statement: Used to execute a block of code if a condition is true.
- `switch` statement: Used to execute different blocks of code based on the value of an expression.

Let us take a look at the following code:

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

int main() {
    int x = 10;
    if (x > 5) {
        cout << "x is greater than 5" << endl;
    } else {
        cout << "x is less than or equal to 5" << endl;
    }
```

```
    return 0;
}

// Output:
// x is greater than 5
```

In this code, we are using the `if` statement to execute a block of code if the value of the variable `x` is greater than 5. If the condition is true, the block of code inside the `if` statement is executed. If the condition is false, the block of code inside the `else` statement is executed.

The `if` statement can also be used without an `else` statement, as seen in the following code:

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

int main() {
    int x = 10;
    if (x > 5) {
        cout << "x is greater than 5" << endl;
    }
    return 0;
}

// Output:
// x is greater than 5
```

In this code, we are using the `if` statement without an `else` statement. If the condition is true, the block of code inside the `if` statement is executed. If the condition is false, the block of code is not executed.

The `switch` statement is used to execute different blocks of code based on the value of an expression. It is similar to a series of `if` statements, but it is more concise and easier to read. Let us take a look at the following code:

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

int main() {
    int x = 2;
    switch (x) {
        case 1:
            cout << "x is 1" << endl;
            break;
        case 2:
            cout << "x is 2" << endl;
            break;
        case 3:
            cout << "x is 3" << endl;
            break;
        default:
            cout << "x is not 1, 2, or 3" << endl;
    }
    return 0;
}

```

```
// Output:
// x is 2
```

In this code, we are using the `switch` statement to execute different blocks of code based on the value of the variable `x`. If the value of the variable is 1, the block of code inside the `case 1` statement is executed. If the value of the variable is 2, the block of code inside the `case 2` statement is executed. If the value of the variable is 3, the block of code inside the `case 3` statement is executed. If the value of the variable is not 1, 2, or 3, the block of code inside the `default` statement is executed.

The `break` statement is used to exit the `switch` statement. If the `break` statement is not used, the control will fall through to the next `case` statement.

### 1.5.2 Iteration structures

Iteration structures are used to execute the same block of code multiple times. There are three main types of iteration structures in `C++`:

- `for` loop: Used to execute a block of code a fixed number of times.
- `while` loop: Used to execute a block of code as long as a condition is true.
- `do-while` loop: Used to execute a block of code at least once, and then as long as a condition is true.

Let us take a look at the following code:

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

int main() {
    for (int i = 0; i < 3; i++) {
        cout << "i = " << i << endl;
    }
    return 0;
}

// Output:
// i = 0
// i = 1
// i = 2
```

In this code, we are using the `for` loop to execute a block of code three times. The loop has three parts: the initialization part, the condition part, and the update part. The initialization part is executed once before the loop starts. The condition part is evaluated before each iteration of the loop. If the condition is true, the block of code inside the loop is executed. The update part is executed after each iteration of the loop.

The `while` loop is used to execute a block of code as long as a condition is true. Let us take a look at the following code:

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

int main() {
    int i = 0;
    while (i < 3) {
        cout << "i = " << i << endl;
        i++;
    }
    return 0;
}

// Output:
// i = 0
// i = 1
// i = 2
```

In this code, we are using the `while` loop to execute a block of code as long as the value of the variable `i` is less than 3. The condition is evaluated before each iteration of the loop. If the condition is true, the block of code inside the loop is executed. The update part is executed inside the loop.

The `do-while` loop is used to execute a block of code at least once, and then as long as a condition is true. Let us take a look at the following code:

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

int main() {
    int i = 0;
    do {
        cout << "i = " << i << endl;
        i++;
    } while (i < 3);
    return 0;
}

// Output:
// i = 0
// i = 1
// i = 2
```

In this code, we are using the `do-while` loop to execute a block of code at least once, and then as long as the value of the variable `i` is less than 3. The block of code inside the loop is executed once before the condition is evaluated. If the condition is true, the block of code inside the loop is executed. The update part is executed inside the loop.

### 1.5.3 Functions

Functions are used to divide the program into smaller parts that can be reused. They allow you to write a block of code once, and then call it from different parts of the program. Functions can

take parameters, and they can return a value.

Let us take a look at the following code:

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

int add(int x, int y) {
    return x + y;
}

int main() {
    int result = add(3, 4);
    cout << "3 + 4 = " << result << endl;
    return 0;
}

// Output:
// 3 + 4 = 7
```

In this code, we are defining a function called `add` that takes two parameters, `x` and `y`, and returns the sum of the two parameters. We are then calling the function with the values 3 and 4, and storing the result in a variable called `result`. We are then printing the result to the console.

Functions can also be defined without a return type, as seen in the following code:

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

void print_hello() {
    cout << "Hello, World!" << endl;
}

int main() {
    print_hello();
    return 0;
}

// Output:
// Hello, World!
```

In this code, we are defining a function called `print_hello` that does not return a value. We are then calling the function to print the message "Hello, World!" to the console.

## 1.6 Arrays and Structs in C++

Arrays and structs are used to store multiple values in a single variable. Arrays are used to store multiple values of the same type, while structs are used to store multiple values of different types.

### 1.6.1 Arrays

Arrays are used to store multiple values of the same type in a single variable. They are used to represent collections of values, such as a list of numbers, a list of names, or a list of grades. Arrays are indexed starting at 0, so the first element of the array is at index 0, the second element is at index 1, and so on.

Let us take a look at the following code:

```
1
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int numbers[3] = {1, 2, 3};
7      for (int i = 0; i < 5; i++) {
8          cout << "numbers[" << i << "] = " << numbers[i] << endl;
9      }
10     return 0;
11 }
12
13 // Output:
14 // numbers[0] = 1
15 // numbers[1] = 2
16 // numbers[2] = 3
```

In this code, we are defining an array called `numbers` that can store three integer values. We are then initializing the array with the values 1, 2, and 3. We are then using a `for` loop to iterate over the array and print the values to the console.

Arrays can also be initialized without specifying the size, as seen in the following code:

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int numbers[] = {1, 2, 3};
6      for (int i = 0; i < 5; i++) {
7          cout << "numbers[" << i << "] = " << numbers[i] << endl;
8      }
9      return 0;
10 }
11
12 // Output:
13 // numbers[0] = 1
14 // numbers[1] = 2
15 // numbers[2] = 3
```

In this code, we are defining an array called `numbers` and initializing it with the values 1, 2, and 3. The size of the array is automatically determined by the number of values that are provided in the initialization list.

### Arrays and memory allocation

Arrays are stored in contiguous memory locations, which means that the elements of the array are stored one after the other in memory. The memory location of the first element of the array is the base address of the array, and the memory location of the last element of the array is the base address plus the size of the array.

The size of the array is determined at compile time, and it cannot be changed once the array has been declared. The size of the array is part of the type of the array, so two arrays with the same type but different sizes are considered to be different types.

### Multidimensional arrays

Arrays can have more than one dimension, which allows you to represent tables of values, such as a matrix of numbers, a list of names and ages, or a list of grades for different subjects. Multidimensional arrays are stored in contiguous memory locations, with the elements of the array arranged in rows and columns.

Let us take a look at the following code:

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

int main() {
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << "matrix[" << i << "][" << j << "] = " << matrix[i][j] << endl;
        }
    }
    return 0;
}

// Output:
// matrix[0][0] = 1
// matrix[0][1] = 2
// matrix[0][2] = 3
// matrix[1][0] = 4
// matrix[1][1] = 5
// matrix[1][2] = 6
```

In this code, we are defining a two-dimensional array called `matrix` that can store two rows and three columns of integer values. We are then initializing the array with the values 1, 2, 3, 4, 5, and 6. We are then using two nested `for` loops to iterate over the array and print the values to the console.

## 1.6.2 Structs

Structs are used to store multiple values of different types in a single variable. They are used to represent complex data types, such as a person with a name, age, and address, a car with a

make, model, and year, or a point with x and y coordinates. Structs are defined using the `struct` keyword, and they can contain multiple fields of different types.

Let us take a look at the following code:

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

struct Student {
    string name;
    int age;
    double gpa;
};

int main() {
    Student student = {"Alice", 20, 3.5};
    cout << "Name: " << student.name << endl;
    cout << "Age: " << student.age << endl;
    cout << "GPA: " << student.gpa << endl;
    return 0;
}

// Output:
// Name: Alice
// Age: 20
// GPA: 3.5
```

In this code, we are defining a struct called `Student` that contains three fields: `name`, `age`, and `gpa`. We are then defining a variable called `student` of type `Student` and initializing it with the values "Alice", 20, and 3.5. We are then printing the values of the fields to the console.

Structs can also contain arrays and other structs, as seen in the following code:

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

struct Point {
    int x;
    int y;
};

struct Rectangle {
    Point top_left;
    Point bottom_right;
};

int main() {
    Rectangle rect = {{0, 0}, {10, 10}};
    cout << "Top left: (" << rect.top_left.x << ", " << rect.top_left.y << ")" <<
        endl;
    cout << "Bottom right: (" << rect.bottom_right.x << ", " << rect.bottom_right.
        y << ")" << endl;
    return 0;
}

// Output:
```

```
// Top left: (0, 0)
// Bottom right: (10, 10)
```

In this code, we are defining two structs: `Point` and `Rectangle`. The `Point` struct contains two fields: `x` and `y`. The `Rectangle` struct contains two fields of type `Point`: `top_left` and `bottom_right`. We are then defining a variable called `rect` of type `Rectangle` and initializing it with the values {(0, 0), (10, 10)}. We are then printing the values of the fields to the console.

## 1.7 Declarations and Definitions in C++

In C++, a declaration is a statement that introduces a name into the program. It tells the compiler that the name exists, but it does not allocate memory for the name. A declaration also is used to define the type of the name, so that the compiler can perform type checking. A name must be declared before it can be used in the program.

A definition is a statement that provides a value for the name. It tells the compiler how much memory to allocate for the name, and it initializes the memory with a value. A definition also is used to define the behavior of the name, so that the compiler can generate code for the name.

In a program it is not necessary to define a name before it can be used, but it is necessary to declare it. It is also not possible to define a name more than once, but it is possible to declare it more than once.

Let us take a look at the following code:

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

double square(double x);  // Declaration

int main() {
    double x = 3.0;
    double y = square(x);
    cout << "The square of " << x << " is " << y << endl;
    return 0;
}

double square(double x) {  // Definition
    return x * x;
}

// Output:
// The square of 3 is 9
```

In this code, we are declaring a function called `square` before it is used in the program. The declaration tells the compiler that the function exists, and it defines the type of the function. We are then defining the function later in the program.

### 1.7.1 Why do we need declarations and definitions?

Declarations and definitions are used to separate the interface of a program from its implementation. The interface of a program is the part of the program that is visible to the user, while the implementation is the part of the program that is hidden from the user.

Declarations are used to define the interface of a program, while definitions are used to define the implementation of a program. This separation allows the user to use the program without knowing how it is implemented, and it allows the implementation to be changed without affecting the user.

Declarations and definitions are also used to improve the readability and maintainability of a program. They allow the program to be divided into smaller parts that can be developed and tested independently. They also allow the program to be reused in different parts of the program.

To refer to something, we only need to declare it. To use it, we need to define it. We often want the definition to be in a different file from the declaration. When using a library, we only need to include the header file, which contains the declarations, and link the library, which contains the definitions.

## 1.8 Header files in C++

Header files are used to separate the interface of a program from its implementation. They contain the declarations of the names that are used in the program, but they do not contain the definitions of the names. Header files are used to define the interface of a program, while source files are used to define the implementation of a program.

Header files have the extension `.h`, and they are included in the source files using the `#include` directive. The `#include` directive tells the compiler to include the contents of the header file in the source file.

Let us take a look at the following code:

```
// square.h
#ifndef SQUARE_H
#define SQUARE_H

double square(double x);

#endif
```

```
// square.cpp
#include "square.h"

double square(double x) {
    return x * x;
}
```

```cpp
// main.cpp
#include <iostream>
#include "square.h"
using namespace std;

int main() {
    double x = 3.0;
    double y = square(x);
    cout << "The square of " << x << " is " << y << endl;
    return 0;
}

// Output:
// The square of 3 is 9
```

In this code, we are defining a function called `square` in the `square.cpp` file. We are then declaring the function in the `square.h` file. We are then including the `square.h` file in the `main.cpp` file using the `#include` directive.

As we said before, header files are used to define the interface of a program, while source files are used to define the implementation of a program. They allow the program to be divided into smaller parts that can be developed and tested independently. They also allow the program to be reused in different parts of the program.

### 1.8.1 Header files and the preprocessor

Header files are processed by the preprocessor before the source files are compiled. The preprocessor is a program that runs before the compiler, and it is used to process the `#include` directive, as well as other directives, such as `#define`, `#ifdef`, and `#endif`.

The `#include` directive tells the preprocessor to include the contents of the header file in the source file. The `#define` directive is used to define a macro, which is a name that is replaced with a value before the program is compiled. The `#ifdef` and `#endif` directives are used to conditionally include code in the program.

The preprocessor is used to process the header files and the source files separately. The preprocessor is used to process the header files first, and then the source files. This allows the compiler to know the interface of the program before the implementation is compiled.

When the preprocessor processes a header file, it replaces the `#include` directive with the contents of the header file. This allows the compiler to see the declarations of the names that are used in the program. When the preprocessor processes a source file, it replaces the macros with their values. This allows the compiler to generate code for the names that are used in the program.

### 1.8.2 Compilation and linking

The compilation process in C++ consists of two main steps: compilation and linking. The compilation step is used to generate object files from the source files, while the linking step is used to generate an executable file from the object files.

The compilation step is used to generate object files from the source files. The object files contain the machine code for the names that are used in the program. The compilation step is performed by the compiler, which is a program that translates the source files into object files.

The linking step is used to generate an executable file from the object files. The executable file contains the machine code for the names that are used in the program, as well as the machine code for the standard library. The linking step is performed by the linker, which is a program that combines the object files into an executable file.

Let us take a look at the following diagram:

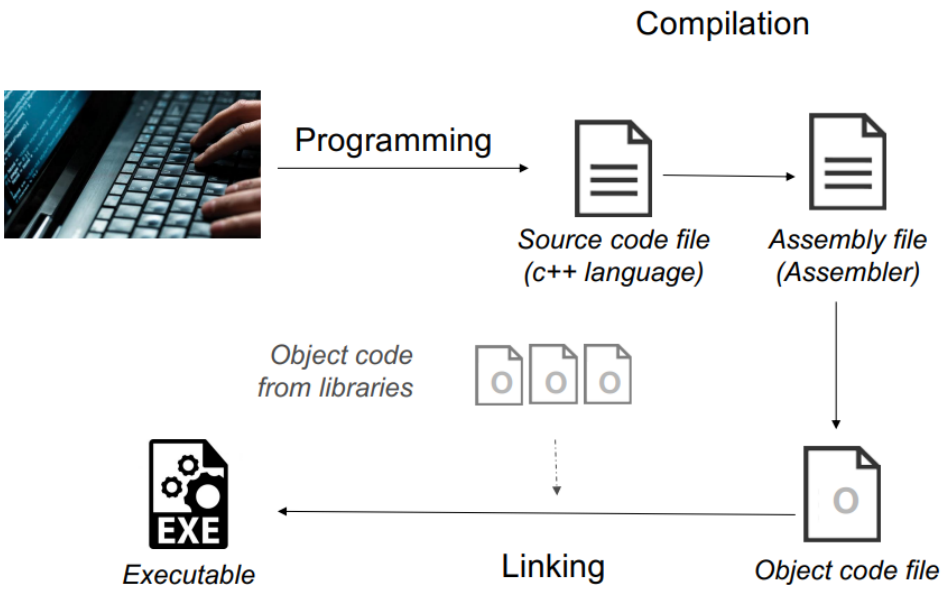


Figure 1.1: Compilation and Linking

The procedure is as follows:

- The main file includes headers of the function/declarations it uses.
- Source files (.cpp) include the corresponding header (+ all headers for the functions/declarations they use).
- Source files are compiled separately and transformed (through the assembler!) into object code files.
- The linker merge together all object code files to build the executable.

## 1.9 Classes and objects in C++

Classes and objects are used to represent real-world entities in a program. They allow you to define the properties and behaviors of an entity, and then create instances of the entity in the program. Classes are used to define the structure of an entity, while objects are used to create instances of the entity.

A class is a (used-defined) data type that specifies how objects of its type can be created and used. A class directly represents a concept in a program. In C++, as in most modern languages, a class is the key building block for large programs. Classes implement a very important concept: Abstract Data Types (ADTs).

An Abstract Data Type (ADT) is a data type that is defined by its behavior (its methods) rather than its implementation (its data members). An ADT is a high-level view of a data type that specifies what operations can be performed on the data type, but not how they are implemented. It offers a simple interface to the user, hiding the details of the implementation. This is known as data encapsulation.

A class is an ADT characterized by:

- Interface (public part): The interface of a class is the set of operations that can be performed on the objects of the class.
- Implementation (private part): The implementation of a class is the set of data members and methods that are used to implement the operations of the class.

An object is an instance of a class. It is a concrete realization of the class, with its own data members and methods. An object is created by instantiating a class, which means allocating memory for the object and initializing its data members.

Let us take a look at the following code:

```
\\stack.h
#ifndef STACK_H
#define STACK_H

class Stack {
public:
    Stack(int size);
    ~Stack();
    void push(int value);
    int pop();
    bool is_empty();
    bool is_full();
private:
    int* data;
    int size;
    int top;
};

#endif
```

```cpp
\\stack.cpp
#include "stack.h"

Stack::Stack(int size) {
    data = new int[size];
    this->size = size;
    top = -1;
}

Stack::~Stack() {
    delete[] data;
}

void Stack::push(int value) {
    if (!is_full()) {
        top++;
        data[top] = value;
    }
}

int Stack::pop() {
    int value = -1;
    if (!is_empty()) {
        value = data[top];
        top--;
    }
    return value;
}

bool Stack::is_empty() {
    return top == -1;
}

bool Stack::is_full() {
    return top == size - 1;
}
```

```cpp
\\main.cpp
#include <iostream>
#include "stack.h"
using namespace std;

int main() {
    Stack stack(3);
    stack.push(1);
    stack.push(2);
    cout << "pop: " << stack.pop() << endl;
    cout << "pop: " << stack.pop() << endl;
    return 0;
}

// Output:
// pop: 3
// pop: 2
```

In this code, we are defining a class called `Stack` that represents a stack data structure. The class contains four public methods: `push`, `pop`, `is_empty`, and `is_full`. The class also contains three private data members: `data`, `size`, and `top`. We are then defining the methods of the class in the `stack.cpp` file. We are then using the class in the `main.cpp` file. Note that the class definition is in the header file, while the class implementation is in the source file.

## 1.10 Vectors

Vectors are a sequence container that encapsulates dynamic size arrays. They can resize themselves automatically when inserting elements. To use a vector, you need to include the `vector` header file.

A vector is a class template, which means that you can create vectors of any assignable type. It is provided by the Standard Template Library (STL), which is a collection of classes and functions that are used to implement common data structures and algorithms. Templates are not themselves classes or functions, but they instead can be thought of as the instructions to the compiler for generating classes or functions. This process is called instantiation.

Let us take a look at the following code:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> numbers;
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);
    for (int i = 0; i < numbers.size(); i++) {
        cout << "numbers[" << i << "] = " << numbers[i] << endl;
    }
    return 0;
}

// Output:
// numbers[0] = 1
// numbers[1] = 2
// numbers[2] = 3
```

In this code, we are creating a vector called `numbers` that can store integer values. We are then adding the values 1, 2, and 3 to the vector using the `push_back` method. We are then using a `for` loop to iterate over the vector and print the values to the console.

### 1.10.1 Ways to initialize a vector

There are several ways to initialize a vector in `C++`:

| | |
|---|---|
| `vector <T> v1` | Vector that holds objects of type `T`. Default initialization, `v1` is empty |
| `vector <T> v2(v1)` | `v2` has a copy of each element in `v1` |
| `vector <T> v2 = v1` | As above |
| `vector <T> v3(n, val)` | `v3` has `n` elements with value `val` |
| `vector <T> v4(n)` | `v4` has `n` copies of a value-initialized object |
| `vector <T> v5 = {a, b, c,...}` | `v5` has as many elements as there are initializers; elements are initialized by corresponding initializers |
| `vector <T> v5 = {a, b, c,...}` | As above |

Table 1.3: Vector Initialization in C++

Note that we can only create vectors that contain values with assignable types. That is, values that can be reassignable throughout the code. Types like references and `const` are non-assignable, so they cannot be used to construct vectors.

Vectors grow efficiently, and because of this, it is often unnecessary to specify the size of a vector when it is created, since it can lead to poorer performance. The exception of this is when you know the size of the vector in advance.

The fact that we can easily and efficiently add elements to a vector is one of the reasons why vectors are so popular in `C++`, since it greatly simplifies many programming tasks. This simplicity imposes a new obligation on the programmer: we must ensure that any loops we write are correct, even if the loop changes the size of the vector.

### 1.10.2 Summary of Vectors operations

| **Operation** | **Description** |
|---|---|
| `v.empty()` | Returns true if `v` is empty, false otherwise |
| `v.size()` | Returns the number of elements in `v` |
| `v.push_back(t)` | Adds a new element with value `t` to the end of `v` |
| `v[n]` | Returns a reference to the element at position `n` in `v` |
| `v1 = v2` | Replaces the elements in `v1` with a copy of the elements in `v2` |
| `v1 == v2` | Returns true if `v1` and `v2` have the same size and the elements in corresponding positions are equal, false otherwise |
| `v1 != v2` | Returns true if `v1` and `v2` have different sizes or the elements in corresponding positions are not equal, false otherwise |
| `<, <=, >, >=` | Lexicographical comparison of `v1` and `v2` |

Table 1.4: Vector Operations in C++

Chapter 2

# Pointers, references and function parameters

## 2.1 Pointers

Declaring a variable means reserving a memory area, including several locations. The number of locations depends on the type of the variable. For example, an `int` variable occupies 4 bytes, a `float` variable occupies 4 bytes, and a `double` variable occupies 8 bytes.

Each memory location has a physicall address, which is a number that identifies it. The address of a variable can be obtained by using the `&` operator. For example, the following code prints the address of the variable `a`:

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

int main() {
    int a = 10;
    cout << &a << endl;
    return 0;
}

// Output: 0x7fffbf7b3b7c (a mem address, it may vary)
```

The `&` operator is called the **address-of operator**.

A pointer is a variable that stores the address of another variable. The type of the pointer must be the same as the type of the variable whose address it will store. For example, the following code declares a pointer to an integer variable and assigns the address of the variable `a` to it:

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

int main() {
    int a = 10;
    int *p = &a;
    cout << p << endl;
    cout << *p << endl;
    return 0;
}

// Output:
```

```
0x7fffbf7b3b7c (a mem address, it may vary)
10
```

The `*` operator is called the **dereference operator**. It is used to access the value stored in the memory location pointed by the pointer. In the example above, `*p` is equivalent to `a`. Note that the operator is used to declare a pointer, but it is also used to access the value stored in the memory location pointed by the pointer. This is a common source of confusion.

The following code shows how to declare a pointer to a `float` variable:

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

int main() {
    float b = 3.14;
    float *q = &b;
    cout << q << endl;
    cout << *q << endl;
    return 0;
}

// Output:
0x7fffbf7b3b7c (a mem address, it may vary)
3.14
```

As you can see, the code is very similar to the previous example. The only difference is that the type of the pointer is `float *` instead of `int *`. Pointers can store the address of any variable, regardless of its type, but the type of the pointer must match the type of the variable whose address it will store.

### 2.1.1 Pointers to pointers

There are no limits to how many type modifiers can be applied to a declarator. When there is more than one modifier, they combine in ways that are logical, but not always obvious. The **dereference operator** is a type modifier when it is used in a declaration. As such, it can be used multiple times.

A pointer is an object in memory, so like any other object, it has an address. Therefore, we can store the address of a pointer on another pointer. For this, we use two dereferencing operators `**`. There is no limit to how many of `*` we can stack, and each one will refer to a new pointer level.

```cpp
int val = 1024;
int *pi = &ival;
int **ppi = &pi;

cout << "direct value: " << val << "\n";
cout << "indirect value: " << *pi << "\n";
cout << "double indirect value: " << **ppi << "\n" << endl;

// Output
// direct value: 1024
// indirect value: 1024
// double indirect value: 1024
```

## 2.2 Function parameters

The general form of a function is:

```
// function declaration
return_type function_name(type1 parameter1, type2 parameter2, ...)

// function definition
return_type function_name(type1 parameter1, type2 parameter2, ...)
{
    // function body
}
```

In a function definition, we use formal parameters represeinting a symbolic reference (identifier) to objects used within the function. The initial value of formal parameters is defined when the function is called using the actual parameters specified by the caller. Let us see an example:

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

double circ(double radius) {
    double res;
    res = 2 * 3.1416 * radius;
    return res;
}

int main() {
    double r = 5.0;
    double c = circ(r);
    cout << c << endl;
    return 0;
}

// Output: 31.416
```

In the example above, the function `circ` calculates the circumference of a circle given its radius. Here, `radius` is a formal parameter, while `r` is an actual parameter. The function is called with the actual parameter `r`, and the value of `r` is copied to the formal parameter `radius`.

The exchange of information with the passing of parameters between the caller and callee can take place in two ways:

- **Passing by value**
- **Passing by reference**

### 2.2.1 Passing by value

When passing by value, the actual parameter is copied to the formal parameter. This means that any changes made to the formal parameter do not affect the actual parameter. For example:

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

void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;
    swap(a, b);
    cout << a << " " << b << endl;
    return 0;
}

// Output: 10 20
```

In the example above, the function `swap` is supposed to swap the values of the variables `x` and `y`. However, the function does not work as expected because the parameters are passed by value. The function receives copies of the actual parameters, so any changes made to the formal parameters do not affect the actual parameters.

### 2.2.2 Passing by reference

When passing by reference, the address of the actual parameter is passed to the formal parameter. This means that the formal parameter is an alias of the actual parameter. In other words, the formal and the actual parameters refer to the same memory location. Any changes made to the formal parameter affect the actual parameter. For example:

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

void swap(int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;
    swap(&a, &b);
    cout << a << " " << b << endl;
    return 0;
}

// Output: 20 10
```

In the example above, the function `swap` receives the addresses of the variables `x` and `y`. The function swaps the values of the variables by dereferencing the pointers. The function works as

expected because the parameters are passed by reference.

An important observation: arrays are always passed by reference. This means that any changes made to an array in a function affect the original array. This is because the name of an array is a pointer to the first element of the array. For example:

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

void change(int* arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = arr[i] * 2;
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    change(arr, n);
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

// Output: 2 4 6 8 10
```

In the example above, the function `change` receives the address of the array `arr` and the size of the array `n`. The function multiplies each element of the array by 2. The function works as expected because the array is passed by reference.

### 2.2.3 Passing by value vs passing by reference

When you pass by value:

- The actual parameter is copied to the formal parameter.
- It requires a lot of time to perform the copy if the parameter is large.
- Actual parameters and formal parameters are different.
- You cannot return a value to the caller without a return statement.

When you pass by reference:

- The address of the actual parameter is passed to the formal parameter.
- It requires less time to pass the address than to copy the parameter, since the address has a fixed size.
- Actual parameters and formal parameters are the same.
- You can return a value to the caller by changing the value of the formal parameter.

## 2.3 References

A reference is an alias for a variable. It is a constant pointer that is automatically dereferenced. A reference is declared by using the `&` operator. It must be always initialized when declared, and it cannot be changed after initialization. For example:

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

int main() {
    int a = 10;
    int &b = a;
    cout << a << " " << b << endl;
    b = 20;
    cout << a << " " << b << endl;
}

// Output:
10 10
20 20
```

In the example above, the reference `b` is an alias for the variable `a`. This means that `b` and `a` refer to the same memory location. Any changes made to `b` affect `a`, and vice versa.

When we initialize a variable, the value of the inititializer is copied into the object we are creating. When we define a reference, instead of copying the initializer value, we bind the reference to its initializer. Once initialized, a reference cannot be reseated to refer to a different object, and that is why references must be initialized when declared. This is what we call a non-assignable type.

When we fetch the value of a reference, we are actually fetching the value of the object to which the reference is bound. When we use a reference as an initializer, we are actually using the value of the object to which the reference is bound. For example:

```cpp
int i = 7;
int &r = i; // r is a reference to i
int &r2 = r; // r2 is a reference to the object to which r is bound
int j = r; // j is initialized by the value in i
```

### 2.3.1 References vs pointers

References and pointers are similar in that they both provide an alternative way to access an object. However, there are some differences between them:

- A pointer is a compound type that holds a memory address. A reference just is an alias for an already existing object.
- Like references, pointers can be used for indirect access to an object.
- Unlike a reference, a pointer is an object that has its own address and can be assigned and copied to other pointers. A single pointer can point to different objects during its lifetime.

- Like other built-in types, pointers have undefined values if they are not initialized, so we have to be careful when using them.
- A reference is an alias for an object, and once a reference is initialized, it cannot be made to refer to a different object. A reference must be initialized when it is defined.

Note that the operators `*` and `&` are used as both an operator in an expression and as part of a declaration. The context in which the operator is used determines what the symbol means.

### 2.3.2 References as function parameters

`C++` relies on references to implement pass by reference mechanism. This simplifies the syntax for the caller and the callee. When a reference is passed to a function, the function receives an alias to the actual parameter. This means that any changes made to the formal parameter affect the actual parameter.

Let us see an example:

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

void swap(int &x, int &y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;
    swap(a, b);
    cout << a << " " << b << endl;
    return 0;
}

// Output: 20 10
```

In the example above, the function `swap` receives references to the variables `x` and `y`. The function swaps the values of the variables. The function works as expected because the parameters are passed by reference.

### 2.3.3 `const` references

We might want to define a variable whose value cannot be changed, for example, to refer to the size of a buffer. We can make a variable unchangeable by using the `const` keyword. For example:

```
const int bufSize = 512;
bufSize = 512; // error: cannot assign to a const object
```

Since we can't change the value of a `const` object, it must be always initialized when declared, i.e., this is also a non-assignable type. By default, `const` objects are local to the file in which they

are defined. When a `const` object is initialized from a compile-time constant, our compiler will usually substitute the uses of the variable with the value of the constant during compilation.

We can also define a reference to a `const` object. To do so, we use a reference to `const` type. Unlike an ordinary reference, a reference to `const` cannot be used to change the value of the object to which it is bound.

This is useful when we want to pass an argument to a function by reference, but we don't want the function to change the value of the argument. In other words, `const` references can be used to pass large objects to functions in read-only mode, obtaining the benefits of pass by reference without the risk of changing the value of the object.

Let us see an example:

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

double circ(const double &radius) {
    double res;
    res = 2 * 3.1416 * radius;
    //radius = 10;  --> f we try to change the value of radius, we get an error
    return res;
}

int main() {
    double r = 5.0;
    double c = circ(r);
    cout << c << endl;
    return 0;
}

// Output: 31.416
```

In the example above, the function `circ` receives a reference to a `const` variable. The function calculates the circumference of a circle given its radius. In this case, we protect the value of the radius from being changed by using a reference to a `const` variable.

An important thing we need to notice is that the reference to `const` restricts only what we can do through that reference. Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`, and because of this, it might be changed by other means:

```
int i = 42;
int &r1 = i;
const int &r2 = i;

r1 = 0; // r1 is not const, so i is now 0
r2 = 0; // error: r2 is a reference to const
```

## 2.4 Guidelines for passing parameters

When passing parameters to a function, we have to decide whether to pass by value, by reference, or by `const` reference. Here are some guidelines:

- If the parameter is small and we don't want the function to change its value, we can pass by value.
- If the parameter is large and we don't want the function to change its value, we can pass by `const` reference.
- If the parameter is large and we want the function to change its value, we can pass by reference.

In other words:

- Use call-by-value for very small objects (base types).
- Use call-by-const-reference for large objects.
- Use call-by-reference only when you must return a result, rather than modify an object through a reference argument.

## 2.5 Variables scope

The scope of an identifier (to a variable, a function, etc) is the region of the program in which the identifier can be referenced. Some identifiers can be referenced throughout the program, while others can only be referenced within a specific portion of the program. For example, when we declare a local variable in a block (e.g., in a for loop), it can be referenced only in that block or blocks nested within it.

### 2.5.1 Methods of variable creation

There are three methods of creating variables in `C++`:

#### Global variables

Global variables are declared outside of all functions. They can be referenced throughout the whole program. They are stored in the static data. For example:

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

int x = 10; // global variable (bad practice)

int main() {
    cout << x << endl;
    return 0;
}

// Output: 10
```

These variables are difficult to debug and maintain, so in general, it is considered a bad practice to use them. However, global constants are an exception to this rule. For example:

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

const double PI = 3.1416; // global constant

int main() {
    cout << PI << endl;
    return 0;
}

// Output: 3.1416
```

In this case, the global constant `PI` is used to store the value of $\pi$. This is an accepted practice because the value of $\pi$ is constant and will not change throughout the program, and will probably be used in many parts of the program.

### Local on the fly variables

This type of variables are simply created right when they are needed. They are only available within the block in which they were created, and are stored in the stack. For example:

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

int main() {
    for (int i = 0; i < 5; i++) { // i is a local on the fly variable
        cout << i << " ";
    }
}

// Output: 0 1 2 3 4
```

### Local defined variables

These variables are created at the beginning of the block in which they are defined, and are available throughout the block. They are stored in the stack. This is the most common way of creating variables. For example:

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

int main() {
    int x = 10; // local defined variable
    cout << x << endl;
    return 0;
}

// Output: 10
```

### 2.5.2 Global and local declarations

When a local variable has the same name as a global variable, the local variable takes precedence. For example:

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

int x = 10; // global variable

int main() {
    int x = 20; // local variable
    cout << x << endl;
    return 0;
}

// Output: 20
```

In the example above, the local variable `x` takes precedence over the global variable `x`. Be aware that function parameters are also local variables, so the same rule applies. For example:

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

int x = 10; // global variable

void f(int x) {
    cout << x << endl;
}

int main() {
    f(20); // local variable
    return 0;
}

// Output: 20
```

### 2.5.3 Lifetime of variables

The lifetime of a variable is the period during which the variable exists in memory. The lifetime of a global variable is the entire execution of the program, while the lifetime of a local variable is the period during which the function is executed. When you call a function, memory will be allocated for all local variables defined inside the function. When the function returns, the memory will be deallocated.

## 2.6 `auto` specifier and range-for loops

It is not uncommon to want to store the value of an expression in a variable declaration. But to declare the variable, we have to know the type of that expression. Since `C++ 11`, we can let the compiler figure out the type for us, using the `auto` type specifier.

Unlike other type specifiers, such as `double` or `int`, that name a specific type, `auto` tells the compiler to deduce the type from the initializer. That means that a variable using `auto` as its type specifier must have an initializer. Let us see an example:

```
auto item = val1 + val2;
// item will be initialized to the result type of val1 + val2
```

### 2.6.1 Range-for loops

A range-for loop is a very useful tool to iterate over a container of elements (e.g., an array or a vector). Its sytax is similar to the simple for loop, but the iterator variable gets the value of each element in the container, instead of each position index. For example:

```
std::vector<int> v = {10, 20, 30};

for (int i: v) {
    std::cout << i << " ";
}
// Output: 10 20 30
```

Note that in this case, every element of `v` is copied into `i`. Sometimes, we might want to avoid the copy of each element, since they could be large. For that, we can use references as such:

```
// Assume that v is a vector of Strings
for (string &s : v) {
    cout << s << "\n";
}
```

In this way, we avoid the copy of each element to the iteration variable. We might even want to avoid modyfing any element of the vector inside the range-for loop. In that case, we could use a reference to `const`, instead of just a reference.

Sometimes, we might not be sure of the type of the objects inside a certain vector, but we still want to iterate over them. For that, we can use the `auto` specifier in the same way as before:

```
for (auto e : v) {cout << e << "\n"} // e is a copy of all v[i]

for (auto &e : v) {cout << e << "\n"} // no copy

for (const auto &e : v) {cout << e << "\n"} // unmodifiable
```

## 2.7 Iterators

Until now, we have been using subscripts (`[n]`) to access the elements of a container. There is a more general mechanism, the **iterators**, that we can use for the same purpose.

The STL library defines several kinds of containers, and all of them have iterators, but only a few support the subscript operator. We can think of an iterator as a pointer to access any container. Using iterators instead of subscripts allows us to change the container type without changing our code.

Like pointers, iterators give us indirect access to an object, and they have also operations to move from one element to another.

Types that have iterators have also members to return those iterators. These are `begin()` and `end()`. The first one returns an iterator that denotes the first element of the container, if there is one, while the second one returns an iterator positioned "one past the end" of the associated container. Let us see an example:

```
std::vector<int> v = {1, 2, 3};

auto b = v.begin();
auto e = v.end();

// The type should be vector::iterator, but we let the compiler deduce it
```

If the container is empty, the iterators returned by both methods are equal, they are both off-the-end iterators. Here is another example:

```
string s("some string");

for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it) {
    *it = toupper(*it);
}
```

### 2.7.1 Standard container iterator operations

These operations apply to all STL iterators:

| | |
|---|---|
| *iter | Returns a reference to the element denoted by the iterator *iter* |
| iter->memb | Dereferences *iter* and fetches the member *memb* from the underlying element |
| (*iter).memb | |
| ++iter | Increments *iter* to refer to the next element in the container |
| --iter | Decrements *iter* to refer to the previous element in the container |
| iter1==iter2 | Compares two iterators. Two iterators are equal if they denote the same element or if they are the **off-the-end** iterator for **the same container** |
| iter1!=iter2 | |

Figure 2.1: STL iterator operations

These operations apply only to vectors and strings:

| | |
|---|---|
| iter + n<br>iter - n | Adding (subctracting) an integral value *n* from the iterator *iter* yields an iterator *n* elements forward of backward than *iter* within the container |
| iter1 +=n<br>iter1 -=n | Assign to *iter1* the value of adding (subctrating) n to *iter1* |
| iter1-iter2 | Compute the number of elements between *iter1* and *iter2* |
| >,>=,<,<= | One iterator is less than another if it denotes an element that appears in the container before the one referred to |

Figure 2.2: Vector and String iterator operations

### 2.7.2 Constant iterators

These type of iterators, called `const_iterator`, are used when we need to read but not write to an object. To access these, the standard containers have the methods `cbegin()` and `cend()`. They are used in the same way as the other iterators, but the only constraint is that we cannot use them to modify the elements inside the container.

```
1 string s = "hello world";
2
3 for (auto it = s.cbegin(); it != s.cend(); ++it) {
4     cout << *it;
5 }
```

Chapter 3

# Pointers and memory allocation

## 3.1 Classifications of pointers

Pointers are classified into two main categories:

- **Raw pointers**: They are the most basic form of pointers, already present in `C` language. They manually manage the memory allocation and deallocation. When using them, we have to be extremely careful to avoid memory leaks.
- **Smart pointers**: They are a part of the `C++11` standard, and are used to automate the memory management process. We wil come back to this later.

## 3.2 Computer's memory

The computer's memory is divided into four main sections:

- **Code section**: It contains the program's executable code.
- **Static Data section**: It contains the global and static variables.
- **Heap section**: It is used for dynamic memory allocation. It is also called the *free store*. It is managed by the `new` and `delete` operators.
- **Stack section**: It is used for local variables and function calls.

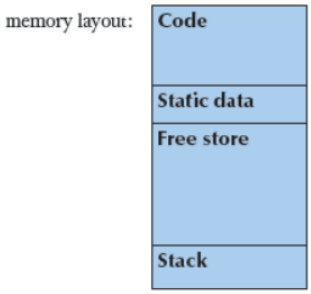


Figure 3.1: Memory layout

### 3.2.1 Heap section (free store)

You request memory to be allocated in the heap using the `new` operator. This operator returns a pointer to the allocated memory, which is just the address of the first byte of that memory block. For example:

```
int *p = new int; // allocate one uninitialized integer
int *q = new int[10]; // allocate an array of 10 uninitialized integers
double *r = new double[4]; // allocate an array of 4 uninitialized doubles
```

A pointer points to an object of its specific type, but it does not know how many elements it points to.

To deallocate memory in the heap, you use the `delete` operator. For example:

```
delete p; // deallocate the memory pointed by p
delete[] q; // deallocate the memory pointed by q
delete[] r; // deallocate the memory pointed by r
```

## 3.3 Pointer states

The value (i.e., the address) stored in a pointer can be in one of the following four states:

1. It can point to an object.
2. It can point to the location just past the end of an object.
3. It can be a null pointer, indicating that it does not point to any object.
4. It can be an invalid pointer; that is, values that are not any of the above.

It is an error to copy or try to access the value of an invalid pointer. As when we use an uninitialized pointer, this error is one that the compiler is unlikely to detect. The result of using an invalid pointer is undefined, so we must always ensure that a pointer is valid before using it.

### 3.3.1 Null pointers

A null pointer is a pointer that does not point to any object. It is represented by the literal `nullptr`. The code can check if a pointer is null by comparing it to `nullptr`. For example:

```
int *p = nullptr; // p is a null pointer
if (p == nullptr) {
    std::cout << "p is a null pointer" << std::endl;
}
```

The `nullptr` is a keyword that represents a null pointer. It is a pointer literal that can be converted to any pointer type. Be aware that a better practice is to rely on short-circuit evaluation to check if a pointer is null. For example:

```
// some code...
if (p != nullptr && *p == 10) {
    std::cout << "p is not null and points to 10" << std::endl;
}
// some more code...
```

## 3.4 Pointer arithmetic and array access

We can perform arithmetic operations on pointers. For example, we can increment or decrement a pointer, or add or subtract an integer from a pointer. When doing so, the compiler automatically scales the integer by the size of the type the pointer points to. This is the way that arrays are accessed in C++. For example:

```
int *arr = new int[5];
*arr = {1, 2, 3, 4, 5};
int *p = arr; // p points to the first element of arr
std::cout << *p << std::endl; // prints 1
p++; // p now points to the second element of arr
std::cout << *p << std::endl; // prints 2
```

Note that when we try to access a specific element of an array, we can use the subscript operator `[]`. For example:

```
int *arr = new int[5];
*arr = {1, 2, 3, 4, 5};
std::cout << arr[0] << std::endl; // prints 1
std::cout << arr[1] << std::endl; // prints 2
```

This is equivalent to the following code:

```
int *p = arr; // p points to the first element of arr
std::cout << *p << std::endl; // prints 1
std::cout << *(p + 1) << std::endl; // prints 2
```

We have to be careful, as the pointer itself does not know how many elements it points to. So, although it is valid to subscript a negative index, or an out of bounds index (since it is just an arithmetic operation), it is not safe to do so, and it leads to undefined behavior and undetectable errors. Let us see an example:

```
arr[-1] = 10; // undetected error
arr[5] = 10; // undetected error
```

Note that `arr[-1]` is equivalent to `*(arr - 1)`, and `arr[5]` is equivalent to `*(arr + 5)`, but they both are pointers to memory that has not been allocated for the array `arr`.

A pointer does know the size of the type it points to, and that is why we can use pointer arithmetic to access the elements of an array. Unlike other types (e.g. `int` and `double`), there is no implicit conversion between pointers to different types.

### 3.4.1 `begin()` and `end()` on arrays

To avoid errors and make it easier for us to use pointers and arrays, `C++ 11` library includes 2 functions: `begin()` and `end()`. These allow us to obtain the pointers to the first and just past the last element of an array, respectively.

```
int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *beg = begin(arr);
int *last = end(arr);

for (int *b = beg; b != last; ++b) {
    cout << *b << endl;
}

// Output: 0 1 2 3 4 5 6 7 8 9
```

It can also be used to compute the length of an array:

```
int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *beg = begin(arr);
int *last = end(arr);

auto n = last - beg;
cout << n << endl;

// Output: 10
```

## 3.5 Why use pointers and free store?

With the `C` language, we use the heap memory when we don't know a priori the size of the data structure and we don't want to waste memory. For example, when we want to create a linked list, we don't know how many elements it will have, so we use the heap memory to allocate memory for each element.

Note that with pointers and arrays, we are "touching" the hardware directly, so we have to be careful when using them. Here is where serious programming errors can most easily occur, resulting in malfuntioning programs, or even worse, programs that appear to work correctly but are subtly incorrect (obscure bugs). If we get "segmentation fault", "bus error", or "core dumped", it is likely that we are using pointers incorrectly.

In C++, we have the STL containers, like `std::vector`, which automatically manage the memory for us. We will come back to this later.

We mainly use free store to allocate objects that have to outlive the scope in which they are created. For example, when we want to return a pointer to an object from a function, we have to allocate memory in the heap, as the stack memory is deallocated when the function returns. This is an example of this case:

```
// some code...
double *create_array(int n) {
    double *arr = new double[n];
    return arr;
}
// some more code...
```

We mainly use raw pointers when we want to share large data structures and avoid multiple copies of them. Note that copies not only waste memory, but also need to be kept in sync, introducing additional overhead.

## 3.6 Memory leaks

A memory leak occurs when a program allocates memory in the heap and does not deallocate it. This memory is not available for other programs, and it is lost. This is a serious problem, as it can lead to the exhaustion of the available memory, and the program can crash.

Let us see an example:

```
double *calc(int result_size, int max){
    int *p = new double[max];
    double *result = new double[result_size];
    // use p to calculate results to be put in result...
    return result;
}

int main(){
    double *res = calc(10, 100); // we forgot to deallocate p
    return 0; // we forgot to deallocate res
}
```

In this case, we have a memory leak, as we forgot to deallocate the memory pointed by `p`. To avoid memory leaks, we have to always deallocate the memory we allocate in the heap, by using the `delete` operator. Here is the fixed code:

```
double *calc(int result_size, int max){
    int *p = new double[max];
    double *result = new double[result_size];
    // use p to calculate results to be put in result...
    delete[] p; // deallocate p
    return result;
}
```

```
8
9  int main(){
10     double *res = calc(10, 100);
11     delete[] res; // deallocate res (this is easy to forget)
12     return 0;
13 }
```

A program that needs to run for a long time can't afford any memory leaks, as they accumulate over time. An example of this is an operating system, which has to run for a long time without crashing.

Nonetheless, programs that run to completion with predictable memory usage may leak without causing problems, i.e., memory leaks aren't "good/bad", but they can be a major problem in specific circumstances.

Another way of getting memory leaks is when we overwrite the pointer to the allocated memory before deallocating it. For example:

```
1 double *p = new double[10];
2 p = new double[20]; // we lost the pointer to the first memory block
3 delete[] p; // we deallocate the memory pointed by p, but it is not the memory we
      allocated
```

In this case, we have a memory leak, as we lost the pointer to the first memory block. To avoid this, we have to always deallocate the memory before overwriting the pointer.

### 3.6.1 How to avoid memory leaks

To systematically avoid memory leaks, we can follow these rules:

- Don't mess directly with `new` and `delete`, unless you have to. Try to use the STL containers.
- Use a garbage collector. This is a program that keeps track of all the memory you allocated dynamically.
- In `C++`, use smart pointers. They are a part of the `C++11` standard, and are used to automate the memory management process (more on this later).

Unfortunately, not even garbage collectors or smart pointers can prevent all memory leaks.

## 3.7 Free store summary

- Allocate using `new`:
  `new` allocates an object on the free store, sometimes initializes it, and returns a pointer to it.

```
1         int *p = new int; // allocate one uninitialized integer
2         char *q = new char('a'); // allocate one initialized char
3         double *r = new double[10]; // allocate an array of 10 uninitialized
              doubles
```

- Deallocate using `delete` and `delete[]`:
  `delete` deallocates an object on the free store, and `delete[]` deallocates an array of objects on the free store.

```
1         delete p; // deallocate p
2         delete q; // deallocate q
3         delete[] r; // deallocate array r
```

- Delete of null pointers does nothing:

```
1         int *p = nullptr;
2         delete p; // harmless
```

Chapter 4

# Algorithms Complexity

## 4.1 What is an algorithm?

An algorithm is a sequence of steps to solve a problem. It is a finite set of instructions that, when followed, accomplish a particular task. An algorithm is a tool for solving a well-specified computational problem. It is a recipe for computation, a sequence of steps that specifies how to solve a problem.

The branch of computer science that studies algorithms is called **algorithm analysis** or **computational complexity**. It is concerned with the theoretical study of computer-program performance and resource usage.

Why do we need to study algorithms in a formal way? Algorithm analysis helps us to understand the performance of an algorithm. It provides a way to compare different algorithms for the same problem. It also provides a way to understand the limitations and scalability of a program.

## 4.2 Asymptotic Notation

The performance of an algorithm is measured in terms of the input size. The input size is the number of bits required to represent the input. The performance of an algorithm is also measured in terms of the number of operations performed by the algorithm. The number of operations performed by an algorithm is a function of the input size.

To analyze the performance of an algorithm, we need to study the growth rate of this function. For this purpose, we use the **asymptotic notation**. The asymptotic notation is a mathematical notation that describes the limiting behavior of a function as the input size approaches infinity. We will define three asymptotic notations: big-O notation, big-$\Omega$ notation, and big-$\Theta$ notation.

### 4.2.1 Big-O Notation

The big-O notation is used to describe the upper bound of a function. Formally, we say that $f(n)$ is $O(g(n))$ if there exist positive constants $c$ and $n_0$ such that:

$$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0 \tag{4.1}$$

In other words, $f(n)$ is $O(g(n))$ if there exists a constant $c$ such that $f(n)$ is bounded above by $c \cdot g(n)$ for all $n$ greater than some threshold $n_0$.

For example, the function $f(n) = 3n^2 + 2n + 1$ is $O(n^2)$ because $3n^2 + 2n + 1 \leq 6n^2$ for all $n \geq 1$. In this case, $c = 6$ and $n_0 = 1$.

The big-O notation is used to describe the worst-case performance of an algorithm. It provides an upper bound on the number of operations performed by the algorithm. The main classes of functions used in the big-O notation are:

- Constant functions: $O(1)$
- Logarithmic functions: $O(\log n)$
- Linear functions: $O(n)$
- Linearithmic functions: $O(n \log n)$
- Quadratic functions: $O(n^2)$
- Cubic functions: $O(n^3)$
- Exponential functions: $O(2^n)$

### 4.2.2 Big-$\Omega$ Notation

The big-$\Omega$ notation is used to describe the lower bound of a function. Formally, we say that $f(n)$ is $\Omega(g(n))$ if there exist positive constants $c$ and $n_0$ such that:

$$0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0 \tag{4.2}$$

In other words, $f(n)$ is $\Omega(g(n))$ if there exists a constant $c$ such that $f(n)$ is bounded below by $c \cdot g(n)$ for all $n$ greater than some threshold $n_0$.

For example, the function $f(n) = 3n^2 + 2n + 1$ is $\Omega(n^2)$ because $3n^2 + 2n + 1 \geq n^2$ for all $n \geq 1$. In this case, $c = 1$ and $n_0 = 1$.

The big-$\Omega$ notation is used to describe the best-case performance of an algorithm. It provides a lower bound on the number of operations performed by the algorithm. The classes of functions used in the big-$\Omega$ notation are the same as those used in the big-O notation.

### 4.2.3 Big-$\Theta$ Notation

The big-$\Theta$ notation is used to describe the tight bound of a function. Formally, we say that $f(n)$ is $\Theta(g(n))$ if there exist positive constants $c_1$, $c_2$, and $n_0$ such that:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0 \tag{4.3}$$

In other words, $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1$ and $c_2$ such that $f(n)$ is bounded above and below by $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$, respectively, for all $n$ greater than some threshold $n_0$.

For example, the function $f(n) = 3n^2 + 2n + 1$ is $\Theta(n^2)$ because $n^2 \leq 3n^2 + 2n + 1 \leq 6n^2$ for all $n \geq 1$. In this case, $c_1 = 1$, $c_2 = 6$, and $n_0 = 1$.

The big-$\Theta$ notation is used to describe the average-case performance of an algorithm. It provides a tight bound on the number of operations performed by the algorithm. It is useful when we want to analyze the behavior of an algorithm in a more precise way, although it is more difficult to determine. The classes of functions used in the big-$\Theta$ notation are the same as those used in the big-O notation.

## 4.3 Sorting Algorithms

Sorting is the process of arranging a list of elements in a specific order. The most common orders are ascending and descending. Sorting is a fundamental operation in computer science. It is used in many applications, such as databases, search engines, and operating systems. There are many sorting algorithms, each with its own advantages and disadvantages. In this section, we will study some of the most popular sorting algorithms.

### 4.3.1 Selection Sort

Selection sort is a simple sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted part of the list and moving it to the sorted part of the list. It is considered the naive sorting algorithm because it is easy to implement and understand.

Its implementation is as follows:

```
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        swap(arr[i], arr[min_index]);
    }
}
```

The time complexity of selection sort is $O(n^2)$ in the worst-case and average-case scenarios. The best-case time complexity is also $O(n^2)$ because the algorithm always performs $n$ swaps, even if the list is already sorted.

### 4.3.2 Insertion Sort

Insertion sort is a simple sorting algorithm that works by repeatedly inserting an element from the unsorted part of the list into its correct position in the sorted part of the list. Its implementation is as follows:

```c
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

The time complexity of insertion sort is $O(n^2)$ in the worst-case scenario, $O(n)$ in the best-case scenario, and $O(n^2)$ in the average-case scenario. The best-case time complexity is $O(n)$ because the algorithm performs $n - 1$ comparisons and no swaps when the list is already sorted.

### 4.3.3 Merge Sort

Merge sort is a divide-and-conquer sorting algorithm that works by dividing the list into two halves, sorting each half recursively, and then merging the two sorted halves. Its pseudocode is as follows:

**Algorithm 1** Merge Sort

```
function MergeSort(arr[], l, r)
    if l < r then
        m ← (l + r)/2
        MergeSort(arr, l, m)
        MergeSort(arr, m + 1, r)
        Merge(arr, l, m, r)
    end if
end function
```

The `MERGE` function merges two sorted subarrays into a single sorted array. Its main steps are:

1. Compare the first element of the left subarray with the first element of the right subarray.
2. If the first element of the left subarray is smaller, copy it to the output array
3. Otherwise, copy the first element of the right subarray to the output array
4. Repeat the process until all elements are copied to the output array

The time complexity of merge sort is $O(n \log n)$ in all scenarios. This is because the operation of recursively dividing the list into two halves reduces the size of the problem by a factor of 2 in each step. The amout of times we can divide the list by 2 is $\log_2 n$, where $n$ is the size of the list. The merge operation has a time complexity of $O(n)$, where $n$ is the size of the list. Therefore, the total time complexity of merge sort is $O(n \log n)$.

The divide-and-conquer paradigm usually results in algorithms with a time complexity with a logarithmic factor.

## 4.4 Big-O complexity chart

The following chart shows the comparison of the most common time complexities in computer science:

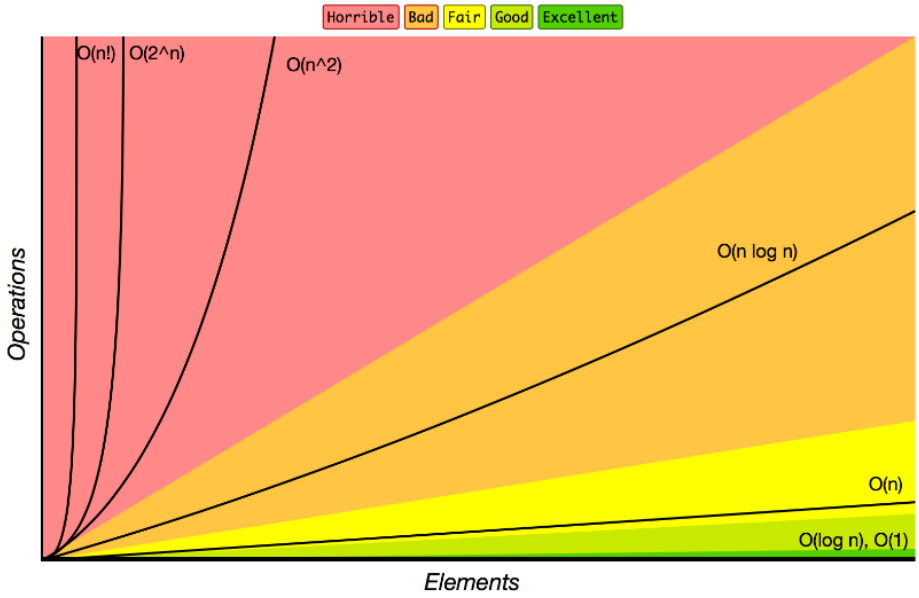


Figure 4.1: Big-O complexity chart

Chapter 5

# Classes

## 5.1 What is a class?

A class is a user-defined type. It specifies a blueprint for the creation of objects. A class consists of a set of members:

- Data members: variables that store the state of the object.
- Member functions (methods): functions that operate on the object.

The methods of a class can define the meaning of creation (constructor), initialization, assignment, copy and cleanup (destructor), among other operations, that determine the behavior of the objects of that class.

## 5.2 Classes: `C++` general syntax

In order to define a class in `C++`, we use the following syntax:

```
class ClassName {
    public: // Accesible by all
        // functions
        // types
        // data (often best kept private!)

    private: // Accessible only by members of the class
        // functions
        // types
        // data
};
```

Notice that the members of a class can be public or private. Public members are accessible from outside the class, while private members are only accessible from within the class. A good practice is to keep the data members private and provide public member functions to access and modify them. In that way, we can control the access to the data and ensure that it is always in a valid state.

For example, we can define a class `Point` as follows:

```
class Point {
    public:
        Point(int xx, int yy) : x(xx), y(yy) {} // Constructor
        int getX() { return x; }
        int getY() { return y; }
        void setX(int xx) { this->x = xx; }
        void setY(int yy) { this->y = yy; }
    private:
        int x;
        int y;
};
```

In this example, we define a class `Point` with two data members `x` and `y` and four member functions: a constructor, two getter functions `getX` and `getY`, and two setter functions `setX` and `setY`.The members of a class can be accessed using the dot operator `.` for objects, and the arrow operator `->` for pointers to objects. Also, common operators such as `+`, `-`, `*`, `/`, among others, can be defined for a class.

Note that the public members of a class provide the class interface, while the private members provide the class implementation. The class interface defines the operations that can be performed on the objects of that class, while the class implementation defines how those operations are performed.

Generally, we want to define the class interface, with all the public and private member declarations, on a `.h` header file, while having the implementation of each function (including constructor and destructor) on a source `.cpp` file that includes this header. For example:

```
// Point.h
#pragma once

class Point {
    public:
        Point(int xx, int yy); // Constructor
        int getX();
        int getY();
        void setX(int xx);
        void setY(int yy);
    private:
        int x;
        int y;
};
```

```
// Point.cpp
#include "Point.h"

Point::Point(int xx, int yy) : x(xx), y(yy) {}
Point::getX() { return x; };
Point::getY() { return y; };
Point::setX(int xx) { x = xx };
Point::setY(int yy) { y = yy };
```

## 5.3 Structs vs Classes

In C++, the only difference between a `struct` and a `class` is that the members of a `struct` are public by default, while the members of a `class` are private by default. In general, we use `structs` to define simple data structures, while we use `classes` to define more complex data structures with associated operations.

### 5.3.1 Structs in C++

Structs are the simplest user-defined data structure that we have. As we said before, all its members are public by default. This structure is inherited from the `C` language. Its main syntax is:

```
struct StructName {
    // data
    // Constructor
    // functions
};
```

For example, we can define a struct `Point` as follows:

```
struct Point {
    int x;
    int y;
};
```

In general, we use structs to define simple data structures that group related data together. Unlike classes, structs cannot define private members, so all the date members of a struct are accessible from outside the struct.

### 5.3.2 Public/private benefits

The main benefit of using private members is that we can control the access to the data and ensure that it is always in a valid state. For example, we can define a class `Point` with private data members `x` and `y` and provide public member functions to access and modify them. In that way, we can ensure that the `x` and `y` coordinates of a point are always non-negative.

In general, we use the public/private paradigm to:

- Provide a clean interface to the users of the class.
- Hide the implementation details of the class.
- Allow the class to change its implementation without affecting its users.
- Easier to support code evolution.
- Mantain the class **invariants.**

### 5.3.3 Invariants

An invariant is a condition that must always be true for an object of a class. It helps to ensure that the object is always in a valid state. For example, if we define a class `Date` with data members `day`, `month` and `year`, we can define, for example, the following invariants:

- The day must be between 1 and 31.
- The month must be between 1 and 12.
- The year must be greater than 0.

We can enforce these invariants by defining the data members as private and providing public member functions to access and modify them. In that way, we can ensure that the `day`, `month` and `year` of a date are always in a valid state.

In general, invariants help to ensure that the data that an object stores is always correct and meaningful for its context. They help to prevent bugs and make the code easier to understand and maintain.

If we can't think of a good invariant for our data structure, we are probably dealing with plain data, and if so, we may use a `struct` instead. But generally, we should try to find good invariants for our user-defined types, so we can regularize the behavior of our objects and prevent buggy code.

## 5.4 `this` parameter

When we call a member function, we do so on behalf of an specific object (an instance of our class). Member functions access the object on which they were called through an extra, implicit parameter named `this`. This parameter will be initialized with the address of that object, so its data will be accesible from within the member function.

For example, let us define a class that stores a date:

```
1  class Date {
2      public:
3          Date(int dd, int mm, int yy) : d(dd), m(mm), y(yy) {}
4          int day() { return d; }
5          int month() { return m; }
6          int year() { return y; }
7      private:
8          int d;
9          int m;
10         int y;
11 }
```

Then, when we call `object.month()` on a certain `object`, the compiler automatically passes the address of that object to the method. Its if like the method were defined as:

```
Date::month(Date *this) // Just a representation of what is happening
```

So, we have:

```
Date my_birthday(26, 2, 2001);
int m = my_birthday.month()
// It is like we were writing Date::month(&my_birthday)
```

Note that inside the member functions, we are refering directly to the members of the object on which the function was called, without using `this`. Any direct use of a member of the class is assumed to be an implicit reference through `this`. That is, when `month` uses `m`, it is as if we had written `this -> m`.

It is then obvious that, when we define members methods of a class, it is forbidden to use the keyword `this` for naming a parameter or a variable. Note that `this` is a `const` pointer, meaning that we cannot change the address that it holds.

## 5.5 `const` member functions

When we define a member function of a class, we can specify that it is a `const` member function by appending the `const` keyword to the function declaration. A `const` member function is a member function that cannot modify the object on which it was called.

For example, we can define a class `Date` with a `const` member function `year` as follows:

```
class Date {
    public:
        Date(int dd, int mm, int yy) : d(dd), m(mm), y(yy) {} // Constructor

        // ... Non-const member functions ...

        int year() const {
            ++y // (Imagine we do this by mistake)
            return y;
        }
    private:
        int d;
        int m;
        int y;
}

Date my_birthday(26, 2, 2001);
int y = my_birthday.year(); // Will result in an error
```

In this example, we define a class `Date` with a `const` member function `year` that tries to increment the year of the date. Since `year` is a `const` member function, it cannot modify the object on which it was called. Therefore, the statement `++y` will result in a compilation error.

In general, we use `const` member functions to ensure that the object on which they were called is not modified. This helps to prevent bugs and make the code easier to understand and maintain.

## 5.6 Helper functions

Helper functions are functions that are not members of a class, but that operate on objects of that class. They are useful to define operations that are not part of the class interface, but that are related to the class.

Usually, we want to keep the class interface clean and simple, and as minimal as posible, so we define helper functions as non-member functions (outside the class) to avoid cluttering the class interface, and when we need to define more complex operations.

For examplee, if we continue with the `Date` class, we can define a helper function `next_sunday()` that returns the next Sunday after a given date. We can define this function as follows:

```
class Date {
    // ... previous implementation ...
}

Date next_sunday(const Date &d) {
    // ... Implementation ...
    // returns a new Date object
}
```

Usually, we declare helper functions in the header file as the class, and define them in the source file that includes the header. This way, we can keep the class interface clean and simple, and provide the implementation details in the source file.

## 5.7 Operator overloading

In `C++`, we can define the behavior of operators for a class by overloading them. Operator overloading allows us to define the meaning of operators such as `+`, `-`, `*`, `/`, among others, for objects of a class.

When defining an operator for a class, we must define a member function or a non-member function that specifies the behavior of that operator for objects of that class. The operator must have at leat one operand of the class for which it is defined.

Some advices for operator overloading are:

- Define operators only when they make sense for the class.
- Define operators only with their conventional meaning.
- Don't overload operators like `*`, `&&`, `||`, `!`

### 5.7.1 Operators as member functions

When we define an operator as a member function, the object on which the operator was called is the left operand of the operator, and the right operand is passed as an argument to the member function.

For example, we can define the operator `+` for a class `Point` as a member function as follows:

```
class Point {
    public:
        // ... previous implementation ...

        Point operator+(const Point &p) {
            return Point(x + p.x, y + p.y);
        }

    private:
        // ... previous implementation ...
}
```

Note that the syntax for overloading an operator as a member function is to define a member function with the name `operator` followed by the operator that we want to overload. In this case, we define the operator `+` for the class `Point` that adds two points.

We can also define operators between our class and other types. For example, when defining the `[]` operator for a class, we can define it as a member function that takes an integer as an argument. Let us see an implementation of this operator for a class `Vector`:

```
class Vector {
    public:
        // ... some implementation ...

        double &operator[](size_t i) const;

    private:
        std::vector<double> data;
}

double &Vector::operator[](size_t i) const { // Obs: returning a reference
    while (data.size() <= i) {
        data.push_back(0.);
    }
    return data[i];
}
```

In this example, we define the operator `[]` for the class `Vector` that returns a reference to the element at the given index. If the index is out of bounds, we push back zeros to the vector until the index is valid.

For operators as member functions, the left operand is always bounded to `this`.

### 5.7.2 Operators as non-member functions

When we define an operator as a non-member function, the object on which the operator was called is passed as an argument to the function. This overloading is useful when we want to define operators that are symmetric between two objects of the same class.

The syntax for overloading an operator as a non-member function is to define a non-member function that takes two arguments of the class for which we want to overload the operator.

For example, we can define the operator `+` for a class `Point` as a non-member function as follows:

```
class Point {
        // ... some implementation ...
}

Point operator+(const Point &p1, const Point &p2) {
    return Point(p1.x + p2.x, p1.y + p2.y);
}
```

In this example, we define the operator `+` for the class `Point` as a non-member function that adds two points. We have a problem here, because the operator `+` is not a member of the class `Point`, so it cannot access the private members of the class (in this case, `x` and `y`). To solve this, we can declare the operator `+` as a friend function of the class `Point`. It goes as follows:

#### Friend functions

When we define an operator as a non-member function, we can specify that the function is a friend of the class. A friend function is a function that is not a member of the class, but that has access to the private members of the class.

The syntax for defining a friend function is to declare the function as a friend of the class in the class definition, and to define the function as a non-member function.

For example, we can define the operator `+` for a class `Point` as a friend function as follows:

```
class Point {
    public:
        // ... some implementation ...

        friend Point operator+(const Point &p1, const Point &p2);
}

Point operator+(const Point &p1, const Point &p2) {
    return Point(p1.x + p2.x, p1.y + p2.y);
}
```

In this example, we define the operator `+` for the class `Point` as a friend function that adds two points. Now the operator `+` has access to the private members of the class `Point`, so we can add two points without any problem:

```
    Point p1(1, 2);
    Point p2(3, 4);

    Point p3 = p1 + p2;
```

### 5.7.3 Member vs non-member

When we define an operator for a class, we can define it as a member function or as a non-member function. The choice between the two depends on the context and the semantics of the operator.

Here are some general guidelines for choosing between a member function and a non-member function:

- Must be member: `= [] () ->`
- Should be member:
  - Compound assignments: `+= -= *= /=`, etc.
  - Modify operator: `++ --`
- Better non-member:
  - Arithmetic operators: `+ - * /`
  - Bitwise operators: `& | ^`, etc.
  - Comparison operators: `== != < >`, etc.
- Better not overloaded: `* && || !`
- Cannot be overloaded: `::  ?:  .* .`

### 5.7.4 Returning `this` object

When we define an operator for a class, we can return the object on which the operator was called by reference. This allows us to chain multiple operators together.

This is usually done when overloading the assignment operator `=` or the compound assignment operators such as `+=`, `-=`, `*=`, `/=`. For example, we can define the operator `+=` for a class `Point` as a member function that returns the object on which the operator was called by reference:

```
class Point {
    public:
        // ... some implementation ...

        Point &operator+=(const Point &p) {
            x += p.x;
            y += p.y;
            return *this;
        }
}
```

In this example, we define the operator `+=` for the class `Point` as a member function that adds a point to another point and returns the object on which the operator was called by reference. This allows us to chain multiple operators together:

```
Point p1(1, 2);
Point p2(3, 4);
Point p3(5, 6);

p1 += p2 += p3;
```