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

Algorithms and Parallel Computing

Professor: Danilo Ardagna

Last updated: January 13, 2025

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:

```
#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:

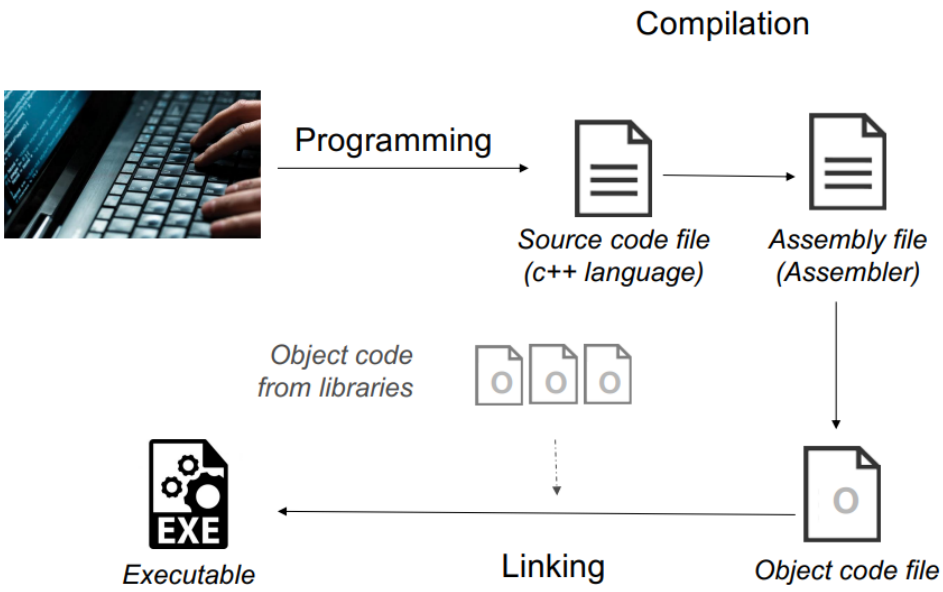

Figure 1.1: Compilation and Linking

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:

```
#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.

```
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:

```cpp
#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 π. This is an accepted practice because the value of π 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 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 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.

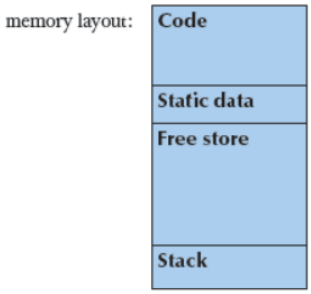

Figure 3.1: Memory layout

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
```

3.8 Smart pointers

We have seen that `C++` lets us allocate objects dynamically, using the memory space called the free store. To do so, we use the `new` operator. However, we must remember to deallocate the memory when we are done with it, using the `delete` operator.

We have also seen that properly freeing dynamic objects turns out to be a surprisingly rich source of bugs:

- If we forget to free the memory, we have a memory leak.
- If we free the memory when there there are still pointers referring to it, we have a pinter that refers to memory that is no longer valid **(dangling pointer)**.
- If we subsequently delete the other pointers, then the free store may be corrupted.

To avoid these problems, `C++11` introduced smart pointers. Smart pointers ensure that the objects to which they point are automatically freed when it is appropiate to do so. This allows us to avoid the problems associated with manual memory management.

There are 3 main types of smart pointers:

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

In this course, we will focus on `std::shared_ptr`. Note that these pointers work as templates and are defined in the `<memory>` header.

3.8.1 Shared pointers

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several shared pointers may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- The last remaining shared pointer owning the object is destroyed.
- The last remaining shared pointer owning the object is assigned another pointer via the `reset()` method.

`std::shared_ptr` is a reference-counted smart pointer. It keeps track of how many shared pointers are pointing to the same object. When the last shared pointer pointing to an object is destroyed, the object is automatically deallocated.

`std::shared_ptr` is a template class, so we have to specify the type of the object it points to. For example:

```
#include <memory>

std::shared_ptr<int> p1; // p1 is a shared pointer to an integer
std::shared_ptr<std::vector<int>> p2; // p2 is a shared pointer to a vector of
    integers
```

To create a shared pointer, we can use the `std::make_shared` function. This function creates a shared pointer that owns a dynamically allocated object. For example:

```
#include <memory>

std::shared_ptr<int> p = std::make_shared<int>(10);
// p points to an integer with value 10
std::shared_ptr<std::vector<int>> q = std::make_shared<std::vector<int>>(10, 20);
// q points to a vector of 10 integers with value 20
```

We have the following operations for shared pointers:

`shared_ptr<T> sp`	Null smart pointer that can point to objects of type T
`p`	Use p as a condition; true if p points to an object
`*p`	Dereference p to get the object to which p points
`p->mem`	Same as (*p).mem
`p.get()`	Returns the pointer in p. **Be very careful!**
`swap(p, q)`	Swap the pointers in p and q
`p.swap(q)`	Swap the pointers in p and q
`make_shared<T> args`	Returns a shared_ptr pointing to a dynamically allocated object of type T; use args to initialize that object
`shared_ptr<T> p(q)`	p is a copy of the shared_ptr q; increments the count in q. The pointer in q must be convertible to T*

`p = q`	p and q are shared_ptrs holding pointers that can be converted to one another. Decrements p reference count and increments q count; deletes p existing memory if p count goes to 0
`p.unique()`	Returns true if p count is one; false otherwise
`p.use_count()`	Returns the number of objects sharing with p; slow, use for debugging

Table 3.1: Operations and functions related to `shared_ptr`

Note that the copy, assignment, and destruction of `shared_ptr` objects work as follows:

- When we copy a `shared_ptr`, the reference count is incremented.
- When we assign a `shared_ptr`, the reference count of the left-hand side is decremented, and the reference count of the right-hand side is incremented.
- When a `shared_ptr` goes out of scope, the reference count is decremented. If the reference count is zero, the object is deallocated.

Be wary of shared ownership!

Do not design your program to use shared ownership by default. Instead, use shared ownership only when you have to. One such reason is to avoid expensive copies of large objects, but only if the performance benefits are significant, and the underlying object is immutable. In many cases, copies can be avoided by using references.

If you have to use shared ownership, always prefer to use `std::shared_ptr` over other forms of shared ownership.

3.8.2 `std::shared_ptr` vs built-in pointers

Note that `std::shared_ptr` is a smart pointer, and it is not the same as a built-in pointer. We cannot inplicitly convert a built-in pointer to a `std::shared_ptr`. For this reason, a function that returns a `std::shared_ptr` cannot use a built-in pointer as a return value. For example:

```
#include <memory>

std::shared_ptr<int> create_int() {
    int *p = new int(10);
    return p; // error: cannot convert 'int*' to 'std::shared_ptr<int>'
}
```

To fix this, we have to use the `std::make_shared` function. For example:

```
#include <memory>

std::shared_ptr<int> create_int() {
    return std::make_shared<int>(10); // ok
}
```

We should never mix ordinary pointers and smart pointers. A `std::shared_ptr` can coordinate destructions only with other `std::shared_ptr` objects that are copies of itself. It is dangerous to use a built-in pointer to access an object owned by a smart pointer, as we may not know when the object is destroyed. For example:

```
#include <memory>

void f() {
    std::shared_ptr<int> p = std::make_shared<int>(10);
    int *q = p.get(); // q points to the object owned by p
} // p is destroyed, but q still points to the object

int main() {
    f();
    // q is a dangling pointer
    return 0;
}
```

In this case, `q` is a dangling pointer, as it points to an object that has been destroyed. To avoid this, we should never use built-in pointers to access objects owned by smart pointers.

Which pointer to use?

We should use raw pointers:

- When you need to store addresses of existing variables.

```
int x = 10;
int *p = &x;
```

We should use smart pointers:

- When you want to declare a new dynamic object.

```
#include <memory>

std::shared_ptr<int> p = std::make_shared<int>(10);
```

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-Ω notation, and big-Θ 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-Ω Notation

The big-Ω 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-Ω 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-Ω notation are the same as those used in the big-O notation.

4.2.3 Big-Θ Notation

The big-Θ 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-Θ 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-Θ 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:

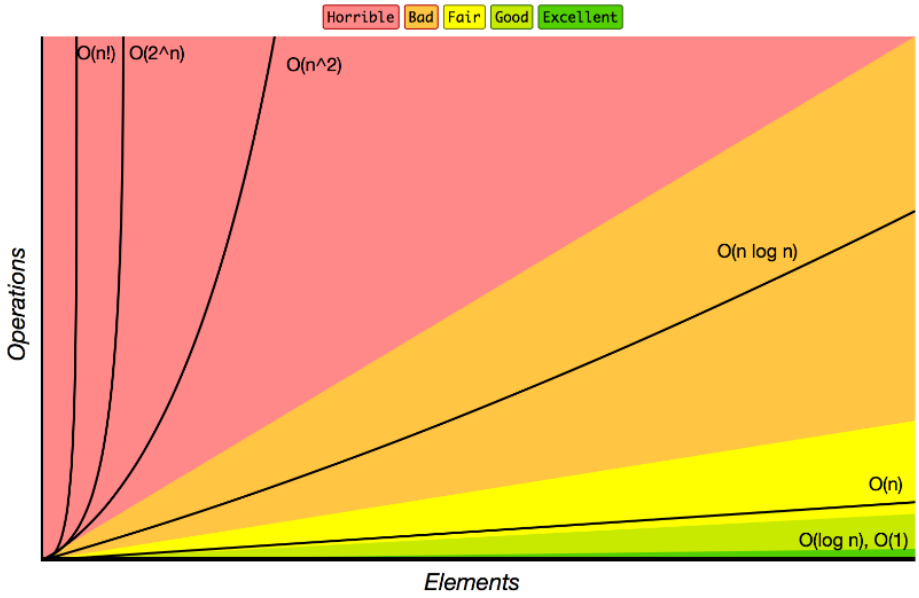

Figure 4.1: Big-O complexity chart

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;
```

5.8 Functions overloading

In `C++`, we can define multiple functions with the same name, as long as they have different parameter lists. This is called function overloading. Function overloading allows us to define multiple functions with the same name that perform different operations depending on the arguments that they receive.

When we call an overloaded function, the compiler selects the function that best matches the arguments that we pass to the function. The selection is based on the number and types of the arguments that we pass to the function.

Note that the return type of a function is not considered when selecting an overloaded function. Therefore, we cannot define two functions with the same name and parameter list that differ only in their return type, as this would result in a compilation error. To overload a function, we must define it with a different parameter list. Let us see an example:

```
int f(int x);
int f(int x, int y); // This is ok, because the number of parameters is different

int g(int x);
void g(int x); // This is not ok, because we are only changing the return type
```

Note also that the compiler will always choose the most specific function that matches the arguments that we pass to the function. If there is no exact match, the compiler will try to find the best match by performing implicit conversions on the arguments. Look at the following example:

```
void f(int x);
void f(double x);

f(1); // Calls f(int x)
f(1.0); // Calls f(double x)
```

In this example, we define two overloaded functions `f` that take an integer and a double as arguments. When we call `f(1)`, the compiler selects the function `f(int x` because the argument is an integer. When we call `f(1.0)`, the compiler selects the function `f(double x` because the argument is a double.

It is important to notice that sometimes we can have ambiguity when calling an overloaded function. This happens when the compiler cannot determine the best match for the arguments that we pass to the function. In this case, the compiler will issue a compilation error. For example, look at the following code:

```
void f(int x, int y);
void f(double x, double y);

f(1, 2.6); // Ambiguity, in both cases we have to perform a conversion
```

We could also define default arguments for a function. This is useful when we want to provide default values for some of the arguments of a function. For example, we can define a function `f` with default arguments as follows:

```
void f(int x, int y = 0, int z = 0) {
    // ... Implementation ...
}
```

In this example, we define a function `f` with three arguments `x`, `y` and `z`, where `y` and `z` have default values of 0. This means that we can call the function `f` with one, two or three arguments, and the missing arguments will be replaced by their default values. For example, we can call the function `f` as follows:

```
f(1); // Calls f(1, 0, 0)
f(1, 2); // Calls f(1, 2, 0)
f(1, 2, 3); // Calls f(1, 2, 3)
```

5.8.1 Overloading and `const` parameters

When we are overloading a function, we have to be careful with the `const` qualifier. If we define two functions with the same name and parameter list, but one of them takes a `const` parameter, the compiler will consider them as the same function. In other words, a parameter that has a top-level `const` is indistinguishable from one that does not have it. For example:

```
void f(int x);
void f(const int x); // This is not ok, because the compiler will consider them as
    the same function
```

Nonetheless, there is a distinction when the `const` qualifier is applied to a pointer or a reference. In this case, the compiler will consider them as different functions. For example:

```
void f(int &x);
void f(const int &x); // This is ok
```

5.8.2 Overloading a member function

As with non-member functions, member functions can also be overloaded. The same function-matching process is used for calls to member functions. The compiler will select the most specific function that matches the arguments that we pass to the function.

In this case, we can overload member functions based on whether they are `const` or not. For example, we can define a class `Point` with two member functions `getX` that differ in their `const` qualifier as follows:

```
class Point {
    public:
        int getX() { return x; }
        int getX() const { return x; }
    private:
        int x;
}
```

In this example, we define a class `Point` with two member functions `getX` that differ in their `const` qualifier. The first function `getX` is a non-const member function that returns the x-coordinate of the point, while the second function `getX` is a `const` member function that returns the x-coordinate of the point.

When we call the function `getX` on a `const` object of the class `Point`, the compiler will select the `const` member function `getX`. When we call the function `getX` on a non-`const` object of the class `Point`, the compiler will select the non-`const` member function `getX`.

5.9 Constructors

A constructor is a special member function that is called when an object of a class is created. It is used to initialize the object and set its initial state. A constructor has the same name as the class and no return type.

As other functiones in `C++`, constructors have a (possibly empty) parameter list, and a (possibly empty) function body. A constructor may also be overloaded, meaning that we can define multiple constructors for a class with different parameter lists. Unlike other functions, constructors cannot be declared `const`.

5.9.1 Default constructors

Classes control default initialization by defining a special constructor known as the default constructor. This constructor takes no arguments, and if it is not explicitly defined by the programmer, the compiler will generate one.

The compiled-generated constructor is known as the synthesized default constructor. For most cases, this constructor initializes each data member of the class as follows:

- If there is an in-class initializer, it is used to initialize the data member.
- Otherwise, default-initialize the data member.

Note that the synthesized default constructor is generated only if no other constructors are defined for the class. If we define any constructor for the class, the synthesized default constructor will not be generated.

We cannot always rely on the synthesized default constructor, because it may not initialize the data members of the class as we expect. For example, objects of built-in or compound types (such as arrays and pointers) have undefined values when default-initialized. Therefore, we should initialize those members inside the class or provide a user-defined default constructor.

Sometimes, the compiler is unable to generate a default constructor. This happens when the class has a reference member, a const member, or a member of a class that has no default constructor.

Let us see an example of a class `Point` with a constructor:

```
class Point {
    public:
        Point(int xx, int yy) {
            x = xx;
            y = yy;
        }
    private:
        int x;
        int y;
}
```

In this example, we define a class `Point` with a constructor that takes two integers `xx` and `yy` as arguments and initializes the data members `x` and `y` with those values. When we create an object of the class `Point`, the constructor is called to initialize the object. If we do not define a constructor for the class, the compiler will generate a default constructor that initializes the object with undefined values. In this case, because the data members are `int`, they will be initialized with garbage values.

5.9.2 Initialization lists

In `C++`, we can initialize the data members of a class in the constructor using an initialization list. An initialization list is a comma-separated list of data members of the class followed by their initial values enclosed in parentheses.

The syntax for an initialization list is to define the data members of the class followed by a colon and the initial values of the data members enclosed in parentheses. For example, we can define a class `Point` with a constructor that initializes the data members `x` and `y` using an initialization list as follows:

```
class Point {
    public:
        Point(int xx, int yy) : x(xx), y(yy) {}
    private:
        int x;
        int y;
}
```

This is specially useful when we have const members, or references, or when we want to initialize the members with a value that is not the default one. In that case, we need to use the initialization list, because we cannot assign a value to a const member in the constructor body. For example:

```
class ConstRef {
    public:
        ConstRef(int ii);
    private:
        const int i;
        int &ri;
}

// This is not ok
ConstRef::ConstRef(int ii) {
    i = ii; // Error, we cannot assign a value to a const member
    ri = ii; // Error, we cannot assign a value to a reference member
}

// This is ok
ConstRef::ConstRef(int ii) : i(ii), ri(i) {} // Ok
```

5.9.3 Delegating constructors

In C++11, we can define a constructor that delegates its initialization to another constructor of the same class. This is known as a delegating constructor. A delegating constructor is useful when we want to avoid code duplication by reusing the initialization logic of another constructor.

The syntax for a delegating constructor is to call another constructor of the same class using the constructor initializer syntax. For example, we can define a class `Point` with two constructors that delegate their initialization to a third constructor as follows:

```
class Point {
    public:
        Point() : Point(0, 0) {} // Delegating constructor
        Point(int xx, int yy) : x(xx), y(yy) {}
    private:
        int x;
        int y;
}
```

In this example, we define a class `Point` with two constructors: a default constructor that delegates its initialization to another constructor that takes two integers as arguments, and a

constructor that takes two integers as arguments and initializes the data members `x` and `y` with those values. When we create an object of the class `Point` using the default constructor, the delegating constructor is called to initialize the object with the values `0` and `0`.

5.9.4 Copy, assignment and destruction

When we define a class in `C++`, we must consider how the objects of that class are copied, assigned and destroyed. By default, the compiler will generate a copy constructor, an assignment operator and a destructor for the class.

The copy constructor is a special member function that is called when an object is copied. It is used to create a new object that is a copy of an existing object. The assignment operator is a special member function that is called when an object is assigned. It is used to assign the value of one object to another object. The destructor is a special member function that is called when an object is destroyed. It is used to clean up the resources that the object has acquired.

5.9.5 Defining a type member

A type alias is a name that is a synonym for another type. We can define a type alias in one of two ways:

- Using the `typedef` keyword:

```
    typedef double d; // d is a synonym for double
    typedef double *dp; // dp is a synonym for double*
```

- Using the `using` keyword:

```
    using d = double; // d is a synonym for double
    using dp = double*; // dp is a synonym for double*
```

We can define a type member inside a class to define a type alias that is specific to that class. A type member is a type alias that is a member of a class. For example:

```
class Screen {
    public:
        using pos = std::string::size_type;
    private:
        pos cursor = 0;
        pos height = 0, width = 0;
}
```

In this example, we define a class `Screen` with a type member `pos` that is a synonym for `std::string::size_type`. We can use the type member `pos` to define the type of the data members `cursor`, `height` and `width` of the class `Screen`. This way, we can ensure that the data members have the same type and are consistent with each other.

5.10 `static` members

Classes sometimes need members that are associated with the class itself, rather than with individual objects of the class. For example, a bank account class might need a data member that holds the interest rate for all accounts. In that case, we would want to associate the rate with the class, not with each individual account.

These members are called `static` members. When we declare a member of a class as `static`, it means that the member is shared by all objects of the class. This means that there is only one copy of the `static` member, regardless of how many objects of the class are created.

Like any other member, static members can be either public or private. The type of a static data member can also be `const`, reference, array, class type, and so on, and we can also define static methods.

Let us see an example of a class `Account` with a static data member `interestRate`:

```
class Account {
    public:
        void calculate() { amount += amount * interestRate; }
        static double rate() { return interestRate; }

    private:
        std::string owner;
        double amount;
        static double interestRate;
}
```

Note that static member functions are not associated with any object. Therefore, they do not have a `this` pointer. This means that they cannot access non-static members of the class. So, for this same reason, declaring a static member function as `const` is meaningless.

Even though static members are not part of the objects of its class, we can use an object, reference or pointer of the class to access the static members. For example:

```
Account myAccount;
double r = myAccount.rate(); // Ok
```

Because static data members are not part of individual objects of the class type, they are not defined when we create objects of the class. As a result, they are not initialized by the class cosntructors. We may not initialize a static data member inside the class, instead we must define and initialize each static data member outside the class body, and like any other object, a static data member may be defined only once.

Like global objects, static data members are defined outside any function. Once they are defined, they continue to exist until the program completes.

5.11 Copy constructors and destructors

Each class defines a new type and the operations that objects of that type can perform. When we copy, assign or destroy objects of a class, this class controls how these operations are performed. Collectively, these operations are known as the **copy-control** operations.

IF a class does not define all the copy-control operations, the compiler will define them for us. As a result, many classes can ignore the copy-control operations. However, there are some classes that must define these operations, as relying on the compiler-generated versions would lead to incorrect behavior. Usually, the hardest part of implementing copy-control operations is recognizing when they are needed.

5.11.1 Copy-assigment operator

Just as a class controls how the objects are initialized, it also controls how they are assigned. The copy-assignment operator is a special member function that is called when an object is assigned to another object. It is represented by the `=` operator.

Just as it does for the copy constructor, the compiler defines a synthesized copy-assignment operator for classes that do not define their own. The synthesized copy-assignment operator assigns each non-static member of the right-hand object to the corresponding member of the left-hand object. If some members cannot be copy-assigned, the synthesized copy-assignment operator will not be defined. Array members are assigned by assigning each element of the array.

The synthesized copy-assignment operator returns a reference to its left-hand object. This allows assignments to be chained.

We can define our own copy-assignment operator for a class. The copy-assignment operator is a member function that takes a single parameter of the class type. Let us see an example of a class `SalesData` with a copy-assignment operator:

```
class SalesData {
    public:
        SalesData &operator=(const SalesData &);
    private:
        std::string bookNo;
        unsigned unitsSold = 0;
        double revenue = 0.0;
}

SalesData &SalesData::operator=(const SalesData &rhs) {
    bookNo = rhs.bookNo;
    unitsSold = rhs.unitsSold;
    revenue = rhs.revenue;
    return *this;
}
```

In this example, we could also want that when we copy an object of the class `SalesData`, everything is copied except the `bookNo`. In that case, we could define the copy-assignment operator

as follows:

```
SalesData &SalesData::operator=(const SalesData &rhs) {
    if (this != &rhs) {
        unitsSold = rhs.unitsSold;
        revenue = rhs.revenue;
    }
    return *this;
}
```

5.11.2 Copy initialization

When we use direct initialization, i.e., when we use the `Object(values)` syntax, we are asking the compiler to use ordinary function matching to select the constructor that best matches the arguments we provide.

When we use copy initialization, i.e., when we use the `Object = values` syntax, we are asking the compiler to copy the right-hand object into the left-hand object, converting that operand if necessary.

Copy initialization ordinarily uses the **copy constructor**. This happens not only when we define variables using an `=`, but also when we:

- Pass an object as an argument to a parameter of non-reference type
- Return an object from a function that has a non-reference return type
- Brace initialize the elements in an array or the members of an aggregate class

Also, some class types use copy initialization for the objects they allocate. The library containers copy initialize their elements when we initialize the container, or when we call an insert or push member.

5.11.3 Copy constructor

A constructor is called the **copy constructor** if its first parameter is a reference to the class type and any additional parameters have default values. For example:

```
class Foo {
    public:
        Foo(); // Default constructor
        Foo(const Foo &); // Copy constructor
}
```

Not that the first parameter must be a reference type: almost always a **reference to const**, although we can define the copy constructor to take a reference to non-const.

When we do not define a copy constructor, the compiler tries to synthesize one for us. Unlike the synthesized default constructor, the synthesized copy constructor is defined even if we define

other constructors.

Note that if all members of a class can be copied (i.e., they have copy constructors), the synthesized copy constructor member-wise copies each member of its argument into the corresponding member of the object being created. If some members cannot be copied, the synthesized copy constructor is unavailable (implicitly deleted).

We can define our own copy constructor for a class. For example, we can define a class `SalesData` with a copy constructor as follows:

```
class SalesData {
    public:
        SalesData(const SalesData &);
    private:
        std::string bookNo;
        unsigned unitsSold = 0;
        double revenue = 0.0;
}

SalesData::SalesData(const SalesData &orig) :
    bookNo(orig.bookNo),
    unitsSold(orig.unitsSold),
    revenue(orig.revenue)
{}
```

An important remark: container elements are copies. When we use an object to initialize a container, or insert an object into a container, a copy of that object is placed in the container, not the object itself. Just as when we pass an object to a non-reference parameter, there is no relationship between the element in the container and the object from which that value originated. Subsequent changes to the element in the container have no effect on the original object, and vice versa.

5.11.4 Destructors

The destructore does whatever operations the class designer wishes to have executed after the last use of an object. Typically, the destructor frees the resources that the object acquired during its lifetime.

The destructor operates inversely to the constructors. And just as a constructor has an initialization part and a function body, a destructor has a function body and a destruction part.

In a destructor, there is nothing akin to the constructor initializer list to control how members are destroyed. The destruction part is implicit.

Let us compare the constructor and destructor of a class:

- **Constructors** initialize the non-static data members of an object and may do other work.
  - Members are initialized before the function body is executed.

  - Members are initialized in the same order as ther appear in the class.

- **Destructors** do whatever work is needed to free the resources used by an object and destroy the non-static data members of the object.
  - The function body is executed first and then the members are destroyed.
  - Members are destroyed in the reverse order from which they were initialized.

What happens when a member is destroyed depends on the type of the member. Members of class type are destroyed by running the member's own destructor. The built-in types do not have destructors, so nothing is done t destroy members of built-in type.

The destructor is a member function with the name of the class prefixed by a tilde (~). It has no return value and takes no parameters. Because it takes no parameters, a destructor cannot be overloaded, so there is always only one destructor for a class. The syntax is as follows:

```
class Foo {
    public:
        ~Foo(); // Destructor
        // ...
}
```

The destructor is called automatically whenever an object of its type is destroyed. This happens on the following situations:

- Variables are destroyed when they go out of scope.
- Members of an object are destroyed when the object of which they are a part is destroyed.
- Elements in a container, whether a library container or an array, are destroyed when the container is destroyed.
- Dynamically allocated objects are destroyed when the delete operator is applied to a pointer to the object.
- Temporary objects are destroyed at the end of the full expression in which they were created.

The compiler synthesizes a destructor for a class if the class does not define one. As with the copy constructor and copy-assignment operator, for some classes, the default destructor cannot be synthesized.

A synthesized destructor works by calling the destructors of the members in reverse order of their declaration. These members are destroyed as part of the implicit destruction phase tha follows the destructor body.

5.12 Copy control

There are three basic operations to control copies of class objects:

- Copy constructor
- Copy-assignment operator
- Destructor

There is no requirement that we define all these operations, but ordinarily we should think of these operations as a unit. If we define one of these operations, we should define all three. This is known as the **Rule of three**.

Although many classes need to define all (or none of) the copy-control members, some classes have work that need to be done to copy or assign objects but has no need for the destructor.

For example, consider a class that gives each object its own, unique serial number:

- Such a class would need a copy constructor to generate a new, distinct serial number for the object being created.
- That constructor would copy all the other data members from the given object.
- This class would also need its own copy-assignment operator to avoid assigning to the serial number of the left-hand object.
- This class would have no need for a destructor.

This example gives rise to a second rule of thumb: if a class needs a copy constructor, it almost surely needs a copy-assignment operator as well, and vice versa. Nevertheless, needing either the copy constructor or the copy-assignment operator does not (necessarily) indicate the need for a destructor.

Another rule of thumb to use when you decide whether a class needs to define its own versions of the copy-control members is to decide first whether the class needs a destructor. Often, the need for a destructor is more obvious that the need for the copy constructor or assigment operator. If the class **needs a destructor**, it almost surely **needs a copy constructor** and a **copy-assignment** operator as well.

5.12.1 Using default and delete

In `C++11`, we can use the `default` and `delete` keywords to control the synthesized copy-control members. The `default` keyword tells the compiler to generate the default version of a member function. This can prevent custom implementation and improves code readability. For example:

```
1 class SalesData {
2     public:
3     // copy control; use defaults
4     SalesData() = default;
5     SalesData(const SalesData&) = default;
6
7     SalesData& operator=(const SalesData&);
8     ~SalesData() = default;
```

```
9
10      // other members
11  };
12
13  SalesData& SalesData::operator=(const SalesData&) = default;
```

For some classes, there really is no sensible meaning for create a copy, so copies or assigments must be denied. For example, the `iostream` classes prevent copying to avoid multiple objects to write or read from the same IO buffer.

To prevent (deny) the copyong behavior, in `C++` we use **deleted functions**. To define a function as deleted, we use the same syntax as default, but using `delete` afte the = operator. This prevents the compiler to automatically synthesize the functions declared as deleted. For example:

```
1   struct NoCopy{
2       NoCopy() = default;
3
4       // disallow copy
5       NoCopy(const NoCopy&) = delete;
6       // disallow assignment
7       NoCopy &operator=(const NoCopy&) = delete;
8
9       ~NoCopy() = default;
10
11      // other members
12  }
```

5.12.2 Resource management

Classes that manage resources that do not reside in the class must define the copy-control members. In order to define these members, we first have to decide what copying an object of our type will mean. In general, we have two choices:

- **Like-a-value:** the class behaves like a value
- **Like-a-pointer:** the class behaves like a pointer

Like-a-value classes have their own state. When we copy a like-a-value object, the copy and the original are independent of each other. Changes made to the copy have no effect on the original, and viceversa. **Like-a-pointer** classes share part of the state. When we copy objects of such classes, the copy and the original use the same underlying data, so changes made to the copy also change the original, and viceversa.

Following these types of behaviors, we have to ways of implementing the copy logic:

- **Shallow copy (like-a-pointer):** Copy only a pointer so that rhe two pointers now refer to the same object. This is what pointers and references do.
- **Deep copy (like-a-value):** Copy what the pointer points to, so that the two pointers now each refer to a distinct object. This is what vectors, strings, ans other STL objects do. It

requires copy constructors and copy assignments for container classes, and must copy **all the way down** if there are more levels in the object.

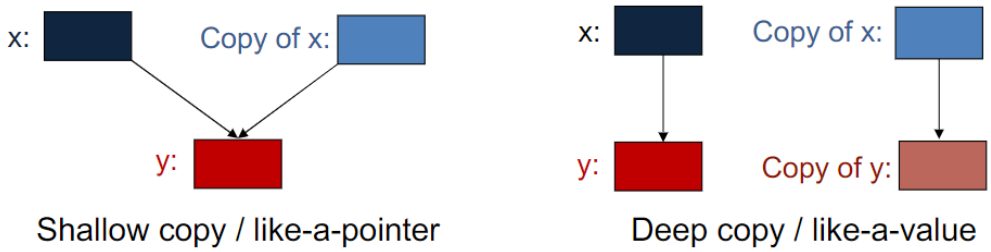

Figure 5.1: Shalow and deep copy

Ordinarily, classes copy members of built-in types (other tha pointers) directly; such members are values and hence ordinarily ought to behave like values. What we do when we copy the pointer member determines whether the class has like-a-value or like-a-pointer behavior.

For example, a like-a-pointer class:

```
class StrLPVector{
    public:
    typedef std::vector<std::string>::size_type size_type;

    StrLPVector();
    StrLPVector(std::initializer_list<std::string> il);

    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }

    // add and remove objects
    void push_back(const std::string &t) { data -> push_back(t); }
    void pop_back() { data-> pop_back(); }

    // element access
    std::string& front() { return data->front(); }
    std::string& back() { return data->back(); }

    private:
    std::shared_ptr<std::vector<std::string>> data;
    // write msg if data[i] isn't valid
};
```

For example, a like-a-value class:

```
class StrLPVector {
    public:
    typedef std::vector<std::string>::size_type size_type;

    StrLPVector();
```

```
    StrLPVector(std::initializer_list<std::string> il);
    size_type size() const { return data.size(); }
    bool empty() const { return data.empty(); }

    // add and remove elements
    void push_back(const std::string &t) { data.push_back(t); }
    void pop_back() { data.pop_back(); };

    // element access
    std::string& front() { return data.front(); }
    std::string& back() { return data.back(); }

    private:
    std::vector<std::string> data;
    // write msg if data[i] isn't valid
};
```

5.13 Implicit class-type conversions

C++ defines several automatic conversions among the built-in types. For example, we can assign an `int` to a `double`, or pass an `int` to a function that takes a `double` parameter.

Classes can define implicit conversion as well. Every constructor that can be called with a single argument defines an implicit conversion to the class type. Such constructors are called **converting constructors**. For example:

```
class MatLabVector {
    vector<double> elem;

    public:
    MatLabVector(unsigned sz) : elem(sz, 0.) {}
    MatLabVector() = default;

    double& opeartor[](unsigned n);
    size_t size() const;

    MatLabVector operator+(const MatLabVector& other) const;
    MatLabVector operator*(double scalar) const;
}
```

In this case, the `MatLabVector` constructor that takes an `unsigned` is a defines implicit conversions from that type to a `MatLabVector`. We can use an `unsigned` wherever a `MatLabVector` is expected. For example:

```
MatLabVector v1(10); // ok: direct initialization
v1 = 20; // ok: implicit conversion to MatLabVector

void f(MatLabVector);
f(10); // ok: implicit conversion to MatLabVector
```

This is very error prone (unless that is the desired behavior). To avoid this, we can use the `explicit` keyword to prevent the compiler from using that constructor for implicit conversions. For example:

```
class MatLabVector {
    vector<double> elem;

    public:
    explicit MatLabVector(unsigned sz) : elem(sz, 0.) {}
    MatLabVector() = default;

    double& opeartor[](unsigned n);
    size_t size() const;

    MatLabVector operator+(const MatLabVector& other) const;
    MatLabVector operator*(double scalar) const;
}

MatLabVector v1(10); // ok: direct initialization
v1 = 20; // error: no implicit conversion to MatLabVector

void f(MatLabVector);
f(10); // error: no implicit conversion to MatLabVector
```

Note that when we declare a constructor as `explicit`, we can only use that constructor for direct initialization. We cannot use that constructor for copy initialization.

Although the compiler will not use an explicit constructor for an implicit conversion, we can use such constructors explicitly to force a conversion. For example:

```
MatLabVector v1(10); // ok: direct initialization
v1 = MatLabVector(20); // ok: explicit conversion to MatLabVector

v1 += 10; // error: no implicit conversion to MatLabVector
v1 += MatLabVector(10); // ok: explicit conversion to MatLabVector
```

IMPORTANT:

Consider the class `SalesData`, with its constructor `SalesData(const std::string)`. Note that in this case, even if it is not explicit, only one implicit conversion is allowed. For instance:

```
// ERROR: requires two user-defined conversions:
// (1) convert "9-999-99999-9" to string
// (2) converst that (temporary) string to SalesData
item += "9-999-99999-9";

// OK: explicit conversion to string, implicit conversion to SalesData
item += string("9-999-99999-9");

// OK: implicit conversion to string, explicit conversion to SalesData
item += SalesData("9-999-99999-9");
```

Chapter 6

Inheritance and polymorphism

6.1 PIE properties

Object Oriented Programming (OOP) is a programming paradigm that uses objects to design applications and computer programs. It is based on several principles, the most important of which are the PIE properties:

- **Polymorphism**: The ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- **Inheritance**: The mechanism by which one class acquires the properties and behavior of another class. It supports the concept of hierarchical classification.
- **Encapsulation**: The mechanism that binds together the code and the data it manipulates, and keeps both safe from outside interference and misuse.

6.2 Inheritance

Inheritance is a mechanism in which one class acquires the properties and behavior of another class. It supports the concept of hierarchical classification. Inheritance is a powerful feature of OOP that allows the creation of a new class that is based on an existing class. The new class inherits the attributes and methods of the existing class, and can also add new attributes and methods of its own.

When a class inherits from another class, there are three main benefits. You can:

- Reuse code: The methods and attributes of the parent class are inherited by the child class, so you don't have to write them again.
- Extend functionality: You can add new methods and attributes to the child class.
- Override functionality: You can override the methods of the parent class in the child class.

In general, inheritance establishes an "is-a" relationship between the parent class and the child class. For example, if you have a class called `Animal`, you could create a child class called `Dog` that

inherits from `Animal`. In this case, you could say that a `Dog` is an `Animal`.

A derived class inherits every data member of the base class, as well as every ordinary member function. However, the derived class does not inherit the constructors, destructor, or assignment operators of the base class, as well as the friend functions of the base class, since these are class-specific.

In C++, inheritance is specified by the colon (`:`) followed by the access specifier (public, protected, or private) and the name of the base class. The access specifier determines the visibility of the base class members in the derived class. The default access specifier is private. In this course we will only use public inheritance.

The syntax for inheritance is as follows:

```
class Base {
    // Base class members
};

class Derived : public Base {
    // Derived class members
};
```

6.2.1 Protected members and class access

In C++, the protected access specifier is similar to the private access specifier, but with one key difference: protected members are accessible in the derived class. This means that a derived class can access the protected members of its base class, but an object (instance) of the derived class cannot.

The protected access specifier is useful when you want to make the data members of the base class accessible to the derived class, but not to the outside world. This allows you to encapsulate the data members of the base class and provide controlled access to them through the derived class.

The syntax for protected members is as follows:

```
class Base {
protected:
    // Protected members
};

class Derived : public Base {
    // Derived class members
};
```

Note that protected members are only accessible for the derived class through the derived class object. They are not accessible through the base class object.

6.2.2 Creation and destruction of derived classes

When a derived class object is created, the following steps occur:

- Space is allocated for the full object, that is, enouch space to store the data members of both the base and derived classes.
- The base class constructor is called to initialize the base class data members.
- The derived class constructor is called to initialize the derived class data members.
- The derived class object is created and is now usable.

When a derived class object is destroyed, the following steps occur:

- The derived class destructor is called to destroy the derived class data members.
- The base class destructor is called to destroy the base class data members.
- The space allocated for the object is deallocated.

6.2.3 Class design principle

In the absence of inheritance, we can think of a class as having two different kinds of developers: the class designer and the class user. The first group is responsible for designing the class, while the second group is responsible for using the class.

When inheritance is used, there is a third group of developers: the class extender. This group is responsible for extending the class by adding new attributes and methods to it. The class extender is also responsible for overriding the methods of the base class.

When implementing inheritance, it is important to be strictest as possible with the access specifiers. The base class should have all its members private, except for the methods that are meant to be overridden by the derived class. These methods should be protected. The derived class should have all its members private, except for the methods that are meant to be used by the class user. These methods should be public.

6.3 Polymorphism

Polymorphism is the ability of objects to respond in different ways to the same message or function call. An object has "multiple identities", based on its class inheritance tree, meaning it can be use in different ways depending on the context.

There are two types of polymorphism: compile-time polymorphism and run-time polymorphism.

Compile-time polymorphism is achieved through function overloading and function redefinition. Function overloading allows you to define multiple functions with the same name but different parameter lists. Function redefinition allows you to define a function with the same name and parameter list in a derived class as in the base class, to change the behavior of the function.

Run-time polymorphism is achieved through virtual functions and inheritance. Virtual functions are functions that are declared in the base class and can be overridden by the derived

class. When a virtual function is called through a base class pointer or reference, the function that is called is determined by the type of the object, not the type of the pointer or reference.

6.3.1 Virtual functions

In `C++`, a base class distinguishes functions that are type dependant from those that it expects its derived classes to inherit withou modification. The former are declared as `virtual` functions. A virtual function is a member function that is declared in the base class using the keyword `virtual`. A virtual function can be overridden by a derived class to provide a different implementation.

A derived class may or may not override a virtual function. If it does not override the virtual function, the base class version of the function is used. If it does override the virtual function, the derived class version of the function is used.

An important thing to notice is that virtual member functions support dynamic binding. This means that the function bounds to the object at runtime, not at compile time. Without virtual member functions, `C++` uses static binding, which means that the function bounds to the object at compile time, and it is only considered function redefinition.

The syntax for virtual functions is as follows:

```
class Base {
public:
    virtual void function() {
        // Base class implementation
    }
};

class Derived : public Base {
public:
    void function() override {
        // Derived class implementation
    }
};
```

Through dynamic binding, we can use the same code to process objects of the base class and objects of the derived class. This is a powerful feature of `C++` that allows us to write more flexible and maintainable code. It is also a key feature of polymorphism.

Note that polymorphic behavior is only possible when using pointers or references to objects. When using objects directly, the function that is called is determined by the type of the object, not the type of the pointer or reference. Look at the following example:

```
#include <iostream>

class Base {
public:
    virtual void function() {
```

```
6           std::cout << "Base class implementation" << std::endl;
7       }
8   };
9
10  class Derived : public Base {
11  public:
12      void function() override {
13          std::cout << "Derived class implementation" << std::endl;
14      }
15  };
16
17  void f(Base& b) {
18      b.function();
19  }
20
21  int main() {
22      Base b;
23      Derived d;
24
25      f(b);
26      f(d);
27
28      return 0;
29  }
30
31  // Output:
32  // Base class implementation
33  // Derived class implementation
```

To summarize, a base class specifies that a member function should be dynamically bound by declaring it as virtual. A derived class specifies that it intends to override a virtual function by using the `override` keyword. Any non-static member function, other than a constructor, can be declared as virtual.

Note that, when a derived class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. This is a good practice, as it makes the code more readable, but once a function is declared as virtual in a base class, it remains virtual in all derived classes.

A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides. If the parameter types differ, the functions are considered to be overloaded, not overridden.

With one exception, the return type of a virtual in the derived class must also match the return type of the base-class function. The exception is that the return type of the derived-class function can be a pointer or reference to a derived class type, even if the return type of the base-class function is a pointer or reference to a base class type. I.e., if `D` is derived from `B`, then a base class virtual can return `B*` and the version of the derived can return `D*`.

6.3.2 Derived-to-base conversion

In C++, a derived class object can be assigned to a base class pointer or reference. This is known as derived-to-base conversion. Derived-to-base conversion is useful when you want to treat a derived class object as a base class object.

When a derived class object is assigned to a base class pointer or reference, only the base class part of the object is accessible. This means that you can only access the base class members of the object, not the derived class members.

For example:

```cpp
#include <iostream>

class Base {
public:
    virtual void function() {
        std::cout << "Base class implementation" << std::endl;
    }
};

class Derived : public Base {
public:
    void function() override {
        std::cout << "Derived class implementation" << std::endl;
    }
};

int main() {
    Derived d;
    Base* b = &d;

    b->function();

    return 0;
}

// Output:
// Derived class implementation
```

The fact that we can bind a reference or pointer to a base class object to a derived class has an important implication: we don't know the actual type of the object to which the pointer or reference is bound. This is a key feature of polymorphism. Tha object can be of the base class type or of any derived class type.

We will deepen about this topic later in this chapter.

6.3.3 Static and dynamic types

In C++, an object has two types: a static type and a dynamic type. The static type of an expression is the type that is known at compile time, it is the the type with which a variable is declared. The

dynamic type of an expression is the type that is determined at runtime, it is the type of the object to which a pointer or reference is bound.

When a base class pointer or reference is bound to a derived class object, the static type of the pointer or reference is the base class type, and the dynamic type of the object is the derived class type. This means that the compiler treats the object as if it were of the base class type, but the object behaves as if it were of the derived class type.

It is important to notice that the dynamic type of an expression that is neither a pointer nor a reference is always the static type.

In the previous example, the static type of the pointer `b` is `Base*`, and the dynamic type of the object to which `b` is bound is `Derived`. This is why the function that is called is the `Derived` class implementation.

6.4 Abstract classes

An abstract class is a class that cannot be instantiated, that is, you cannot create an object of an abstract class. An abstract class is used as a base class for other classes, and it is designed to be inherited by other classes. An abstract class is a class that contains one or more pure virtual functions.

6.4.1 Pure virtual functions

A pure virtual function is a virtual function that is declared in the base class but has no implementation. A pure virtual function is declared using the syntax `virtual void function() = 0;`. A pure virtual function must be overridden by a derived class to provide an implementation.

For example:

```
#include <iostream>

class Base {
public:
    virtual void function() = 0;
};

class Derived : public Base {
public:
    void function() override {
        std::cout << "Derived class implementation" << std::endl;
    }
};

int main() {
    Derived d;

    d.function();

```

```
20      return 0;
21  }
22
23  // Output:
24  // Derived class implementation
```

In this example, the `Base` class is an abstract class because it contains a pure virtual function. The `Derived` class is a concrete class because it provides an implementation for the pure virtual function. The `Derived` class can be instantiated, but the `Base` class cannot.

6.4.2 Refactoring

When refactoring a class, it is important to consider the following:

- If a class is not meant to be instantiated, make it an abstract class by adding a pure virtual function.
- If a class is meant to be instantiated, make it a concrete class by providing an implementation for the pure virtual function.
- If a class is meant to be inherited by other classes, make it a base class by adding virtual functions.
- If a class is meant to be used by other classes, make it a utility class by adding static functions.

By following these guidelines, you can create a well-structured class hierarchy that is easy to understand and maintain.

6.5 Derived-to-base conversion

As we said before, because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type as if it were an object of its base type(s). In particular, we can binde a base-class reference or pointer to the base-class part of a derived object.

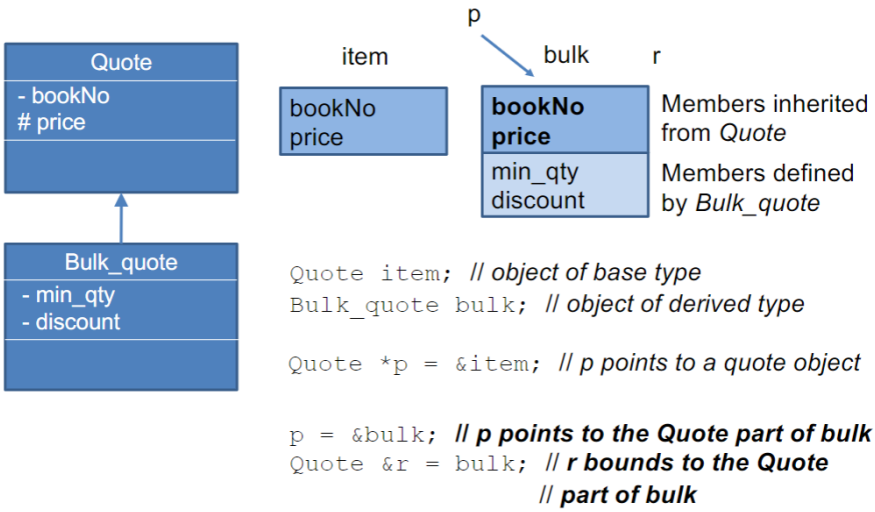

Figure 6.1: Example of derived-to-base conversion

Ordinarily, we can only bind a reference or a pointer to an object that has the same type as the corresponding reference or pointer. Classes related by inheritance are an important exception of this rule.

The conversion from derived to base exists because every derived object contains a base-class part to which a pointer or reference of the base-class type can be bound. Note that there is **no similar guarantee for base-class objects**.

A base-class object can exist either as an independent object or as a part of a derived object. A base object that is not part of a derived object has only the members defined by the class; it doesn't have the members defined by the derived class, i.e, **there is no automatic conversion** from the base class to the derived class.

6.5.1 No conversion between objects

Also note that the automatic derived-to-base conversion applies only for conversions to a reference or pointer type. It is possible to convert an object of a derived class to its base-class type, but such conversions may not behave as we might want.

When we initialize or assign an object of a class type, we are actually calling a function:

- When we **initialize**, we're calling a **copy constructor**
- When we **assign**, we're calling an **assignment operator**
- These members normally have a parameter that is the reference to the const version of the class type.

Because these members take references, the derived-to-base conversion lets us pass a derived object to a base-class copy operation.

IMPORTANT! These operations are **not virtual**. When we pass a derived object to a base-class constructor, the constructor that is run is defined in the base class. If we assign a derived object to a base object, the assignment operator that is run is defined in the base class.

6.6 Containers and Inheritance

When we use a container to store objects from an inheritance hierarchy, we generally must store those objects indirectly. This is because the container must store objects of a single type, and the objects in the hierarchy are of different types.

As an example, assume we want to define a vector to hold several books that a customer wants to buy. We can't use a vector that holds `ScienceBook` objects, as we can't convert a `ScienceBook` object to a `Book` object. We also can't use a vector that holds `Book` objects, because if we add a `ScienceBook` object to the vector, we'll lose the `ScienceBook`-specific information.

The solution is to use a vector of pointers to the base class. This way, we can store pointers to objects of the derived classes in the vector, and we can access the objects through the base-class interface. This is an example of dynamic binding, which is a key feature of polymorphism. Prefer smart pointers to raw pointers.

Chapter 7

Streams & I/O

7.1 Files

We turn our computers on and off. The contents of its main memory are transient, and depend on the power supply. So when we turn it off, we lose all the data that was in memory. Some data needs to be stored permanently, so that we can access it later. This is where files come in. A **file** is a sequence of bytes stored on a permanent storage device, such as a hard disk. Files are used to store data permanently, and to share data between different programs. A file has a name, and the data that is in it is stored in a specific format. We can read and write on a file if we know its name and its format.

At a fundamental level, a file is a sequence of bytes numbered from 0 to $n - 1$, where n is the size of the file in bytes. Other notions can be supplied by programs that interpret a file format. For example, the 6 bytes corresponding to "123.45" might be interpreted as the floating-point number 123.45.

To **read** a file:

- We must know its name.
- We must open it (for reading).
- Then we can read it.
- Once finished, we must close it:
  - That is typically done implicitly (when the stream object is destroyed).

To **write** a file:

- We must name it
- We must open it (for writing), or create a new file of that name.
- Then we can write to it.
- Once finished, we must close it.
  - That is typically done implicitly (when the stream object is destroyed).

7.1.1 Opening and closing files

To open a file, we use 2 main objects: `ifstream` and `ofstream`. The former is used to read from a file, and the latter is used to write to a file. Both are defined in the `fstream` header.

For example, to open a file for reading:

```
#include <fstream>

int main() {
    cout << "Please enter the name of the file you want to read: ";
    string filename;
    cin >> filename;

    ifstream ist{filename};

    if (!ist) {
        cerr << "Can't open input file " << filename << endl;
        return 1;
    }
}
```

To open a file for writing:

```
int main() {
    cout << "Please enter the name of the file you want to write to: ";
    string filename;
    cin >> filename;

    ofstream ost{filename};

    if (!ost) {
        cerr << "Can't open output file " << filename << endl;
        return 1;
    }
}
```

To close a file, we can use the `close()` method. However, this is not necessary, as the file will be closed automatically when the stream object is destroyed. This happens when the stream object goes out of scope. For example:

```
if(read) {
    ifstream input{filename};

    if (input) {
        process(input);
    } else {
        cerr << "Can't open input file " << filename << endl;
        return 1;
    }

} // input goes out of scope, and the file is closed
```

7.1.2 Reading from a file

To read from a file, we can use the >> operator. For example, suppose that a file contains a sequence of pairs represanting hours and temperature readings, like this:

```
10 20.5
11 21.3
12 22.1
```

The hours are numbered from 0 to 23, and the temperatures are floating-point numbers. No further information is assumed, and when we reach end-of-file, or read an unexepected value, we stop. We can read this file like this:

```
struct Reading {
    int hour;
    double temperature;
}

vector<Reading> temps;

int hour;
double temperature;
ifstream ist{fname};

while (ist >> hour >> temperature) {
    if (hour < 0 || 23 < hour) {
        cout << "hour out of range" << endl;
    }
    temps.push_back(Reading{hour, temperature});
}
```

7.1.3 No copy or assign for I/O objects

We cannot copy or assign objects of the IO types:

```
ofstream out1, out2;
out1 = out2; // ERROR: cannot assign stream objects

ofstream print(ofstream); // ERROR: can't initialize the ofstream parameter
out2 = print(out2); // ERROR: cannot copy stream objects
```

Because we can't copy the IO types, we cannot have a parameter of return type that is one of the stream types. Functions that do IO typically pass and return the stream through **references**. Reading or writing an IO object changes its state, so the reference **must not be const**.

7.2 File modes and binary I/O

By default, an `ifstream` opens its file for reading, and an `ofstream` opens its file for writing. We can modify the mode in which a file is opened. The alternatives are:

- `ios_base::app`: Append, output adds to the end of the file.
- `ios_base::ate`: At end, open and seek to the end.
- `ios_base::binary`: Binary mode - beware of system specific behavior.
- `ios_base::in`: For reading.
- `ios_base::out`: For writing.
- `ios_base::trunc`: Truncate file to 0-length

We can optionally specify a file mode after the name of the file:

```
ofstream of1 {name1}; // defaults to ios_base::out
ifstream if1 {name2}; // defaults to ios_base::in
ofstream ofs {name, ios_base::app}; // append rather than overwrite
fstream fs {name, ios_base::in | ios_base::out}; // both in and out
```

7.2.1 Text vs binary files

When we read a file in binary mode, the data is read exactly as it is stored in the file, byte by byte. This is different from text mode, where certain translations may occur, such as converting newline characters to the appropriate format for the operating system (e.g., `\n` to `\r\n` on Windows).

In binary mode, no such translations occur, which makes it suitable for reading and writing non-text data, such as images, audio files, or any other format where the exact byte representation is crucial.

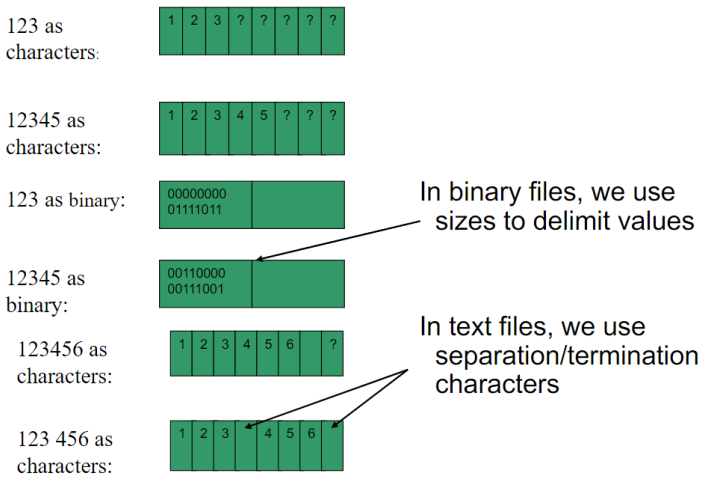

Figure 7.1: Binary vs text reading

In the end, we should use text when we can, as we can read it without a fancy program, we can debug more easily, it is portable across different systems, and most information can be represented reasonably as text. We only use binary when we must, for example, when reading image or sound files.

7.3 String streams

A **stringstream** reads/writes from/to a **string** rather than a file or a keyboard/screen. For example:

```
double str_to_double(string s) {
    istringstream is {s};

    double d;
    is >> d;
    if (!is) error("double format error: ", s);
    return d;
}

double d1 = str_to_double("12.4");
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("twelve point three"); // ERROR
```

String streams are very useful for:

- Formatting into a fixed-sized space (think GUI)
- For extracting typed objects out of a string.

7.3.1 Type vs line

How to read a string:

```
string name;
cin >> name;          // input: Dennis Ritchie
cout << name << '\n'; // output: Dennis
```

How to read a line:

```
string name;
getline(cin, name);   \\ input: Dennis Ritchie
cout << name << '\n'; \\ output Dennis Ritchie

\\ Now what? Maybe:
istringstream ss{name};
ss >> first_name;
ss >> second_name;
```

7.4 Examples

7.4.1 Reading a `.csv` file

An `istringstream` is often used when we have some work to do on an entire line, and other work to do with individual words within a line.

For example, when opening a `.csv` file, we want to read each line, and do some tasks with the information within each line of the file:

```
// input:
// Morgan,2015552368,8625550123
// Drew,9735550130
// Lee,6095550132,2015550175,8005550000

struct PersonInfo {
    string name;
    vector<string> phones;
};

vector<PersonInfo> people;
string line;
ifstream data("data.csv");

while ( getline(data, line) ) {
    PersonInfo info;
    istringstream record(line);

    getline(record, info.name, ',');
    string phone;
    while ( getline(record, phone, ',') ) {
        info.phones.push_back(phone);
    }

    people.push_back(info);
}
```

7.4.2 A word transformation map

Write a program that given one string, transforms it into another. The input to our program is two files. The first file contains rules that we will use to transform the tet in the second file. Each rule consists of a word that might be in the input file and a phrase to use in its place.

Example of files:

Word-transformation file:

```
y why
r are
u you
```

Second file:

```
where r u
```

Output file:

```
where are you
```

The code solution goes as follows:

```
void word_transform(ifstream &map_file, ifstream &input){
    auto trans_map = buildMap(map_file);
    string text;

    while (getline(input, text)) {
        istringstream stream(text);
        string word;
        bool firstword = true;

        while (stream >> word) {
            if (firstword) {
                firstword = false;
            } else {
                cout << " ";
            }

            cout << transform(word, trans_map);
        }

        cout << endl;
    }
}

map<string, string> buildMap(ifstream &map_file) {
    map<string, string> trans_map;
    string key;
    string value;

    while (map_file >> key && getline(map_file, value)) {
        if (value.size() > 1) {
            trans_map[key] = value.substr(1);
        } else {
            cout << "no rule for " + key << endl;
        }
    }
    return trans_map;
}

const string &
transform(const string &s, const map<string, string> &m) {
    auto map_it = m.find(s);

    if (map_it != m.cend()) {
        // if this word is in the transformation map
        return map_it->second;
    } else {
        return s;
    }
}
```

7.5 Readings

We can define our own behavior for the output of a certain object. For example:

```
ostream& operator<<(ostream& os, const Date& d) {
    return os << '(' << d.year()
              << ',' << d.month()
              << ',' << d.day() << ')';
}
```

We often use several different ways of outputtinf a value. Note that tastes for output layout and detail vary.

We can also define our own behavior for the input:

```
istream& operator>>(istream& is, Date &d) {
    int y, d, m;
    if (is >> y >> m >> d) {
        dd = Date{y, m, d}; // Update dd
    }
    return is;
}
```

7.5.1 Read and write on a binary file

To read from a binary file, we can use the `read()` method. For example, suppose that we have a file containing a sequence of integers, like this:

```
10 20 30 40 50
```

We can read this file like this:

```
vector<int> v;
int x;
ifstream ist{fname, ios_base::binary};

while (ist.read(as_bytes(x), sizeof(int))) {
    v.push_back(x);
}
```

To write to a binary file, we can use the `write()` method. For example, suppose that we have a vector of integers, and we want to write them to a file:

```
vector<int> v = {10, 20, 30, 40, 50};
int x;
ofstream ost{fname, ios_base::binary};

for (int x : v) {
    ost.write(as_bytes(x), sizeof(int));
}
```

Warning! Binary files are not portable across different systems. For example, the representation of integers in a binary file may differ between systems.

7.5.2 Positioning in a filestream

To support random access, the system mainatins a marker that determines where the next read or write will occur. We also have two functions:

- One that **repositions the marker** by seeking to a given position.
- One that **reports the current position** of the marker.

The library actually defines two pairs of seek and tell functions:

- One pair is used by input streams, the other by output streams.
- The input and output versions are distinguished by a suffix that is eithr a **g** (for get) or a **p** (for put).

```
1 ifstream ist{fname};
2 ist.seekg(0); // go to the beginning of the file
3 ist.seekg(4); // go to the 5th reading position of the file
4
5 ofstream ost{fname};
6 ost.seekp(0); // go to the beginning of the file
7 ost.seekp(4); // go to the 5th writing position of the file
```

We can only use the **g** versions on an input stream, and on the types tha inherit from it, like `ifstream` and `istringstream`. We can only use the **p** versions on an output stream, and on the types that inherit from it, like `ofstream` and `ostringstream`.

An `iostream`, `fstream`, or `stringstream` can be used for both read and write operations. We can use either the **g** or **p** versions on these types.

There is only one marker!

The fact that the library defines two pairs of seek and tell functions is a bit misleading. There is only one marker, and it is used for both reading and writing. The library keeps track of whether the marker is being used for reading or writing, and adjusts its behavior accordingly.

For example, if we seek to a position in a file, and then read from it, the library will read from the position we seeked to. If we then write to the file, the library will write to the position we seeked to.

Repositioning the marker

The `seekg()` and `seekp()` functions reposition the marker. They take two arguments:

- The first argument is the **offset** from the specified position.

- The second argument is the **position** to which the offset is applied.

The position can be one of the following:

- `ios_base::beg`: The beginning of the file.
- `ios_base::cur`: The current position of the marker.
- `ios_base::end`: The end of the file.

7.5.3 Reading and writing to the same file

We can read and write to the same file, but we must be careful. For example, suppose that we have a file, and we want to write the number of accumulated characters for each row, at the end of the file. We can do this like this:

```
int main() {
    // open for input and output and preposition file pointers to end-of-file
    // file mode argument
    fstream inOut("copyOut", fstream::ate | fstream::in | fstream::out);

    if (!inOut) {
        cerr << "Unable to open file!" << endl;
        return EXIT_FAILURE;//EXIT_FAILURE
    }

    // inOut is opened in ate mode, so it starts out positioned at the end
    auto end_mark = inOut.tellg();// remember original end-of-file

    // position
    inOut.seekg(0, fstream::beg); // reposition to the start of the file

    size_t cnt = 0; // accumulator for the byte count string line;
    string line; // hold each line of input

    // while we haven't hit an error and are still reading the original
    // data and can get another line of input
    while (inOut && inOut.tellg() != end_mark
    && getline(inOut, line))
    {
        cnt += line.size() + 1; // add 1 to account for the newline
        auto mark = inOut.tellg(); // remember the read position
        inOut.seekp(0, fstream::end); // set the write marker to the end
        inOut << cnt; // write the accumulated length
        // print a separator if this is not the last line
        if (mark != end_mark) inOut << " ";
        inOut.seekg(mark); // restore the read position
    }
    inOut.seekp(0, fstream::end); // seek to the end
    inOut << "\n"; // write a newline at end-of- file
    return 0;
}
```

In general, whenever you can, use simple streaming. It is easier to understand (streams/streaming is a powerful metaphor). In this case, you should write most of your code in terms of plain `istreams` and `ostreams`. Positioning is more error prone, and harder to understand. Handling of the end of file position is system dependant and basically unchecked.

7.6 Using ostringstream

An `ostringstream` is a stream that writes to a string. It is useful when we need to build up our output a little at a time, but we don't want to print the output until later. For example, we might want to validate and reformat the phone numbers we read in a previous example:

- If all the numbers are valid, we want to print a new file containing the reformatted numbers.
- If a person has any invalid numbers, we won't put the in the new file. Instead, we'll write an error message containing the person's name and a list of their invalid numbers

```
for (const auto &entry : people) { // for each entry in people
    ostringstream formatted, badNums; // objects created on each loop
    for (const auto &nums : entry.phones) { // for each number
        if (!valid(nums)) {
            badNums << " " << nums; // string in badNums
        } else
        // ''writes'' to formatted's string
            formatted << " " << format(nums);
    }
    if (badNums.str().empty()) // there were no bad numbers
        os << entry.name << " " // print the name
            << formatted.str() << endl; // and reformatted numbers
    else
    // otherwise, print the name and bad numbers
        cerr << "input error: " << entry.name
            << " invalid number(s) " << badNums.str() << endl;
}
```

Chapter 8

STL: Standard Template Library

8.1 STL overview

STL is a software library for the `C++` programming language that provides four main components:

- Algorithms
- Containers
- Functional (or functors)
- Iterators

STL provides a ready-made set of containers that can be used with any built-in and user-defined types that supports some elementary operations (copying and assignment, which are synthetized for us by the compiler if we don't define). Containers implement a **like-a-value** semantic.

When we use an object to initialize a container, or insert an object into a container, a copy of that object value is placed in the container, not the object itself. Just as when we pass an object to a non-reference parameter (pass-by-value), there is no relationship between the element in the container and the object that was used to initialize it, so subsequent changes to the object will not affect the element in the container, and vice versa.

STL algorithms are independent of containers, which significantly reduces the complexity of the library. This is obtained also by the use of iterators, which are used to access the elements of a container.

STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern `C++` compilers are tuned to minimize abstraction penalties arising from the heavy use of the STL.

8.2 STL containers classes

Containers classes share a common interface, which each of the containers extends in its on way. This common inteface make the library easier to learn, and also make it easier to change the con-

tainer type used in a program, as we need limited changes to the code.

Each kind of container offers a different set of performance and functionality trade-offs. There are two main categories of containers:

- **Sequential containers:** Let the programmer control the order in which the elements are stored and accessed. That order does not depend on the values of the elements but on their position.
- **Associative containers:** Store their elements based on the value of a key. Elements are retrieved efficiently according to their key value.

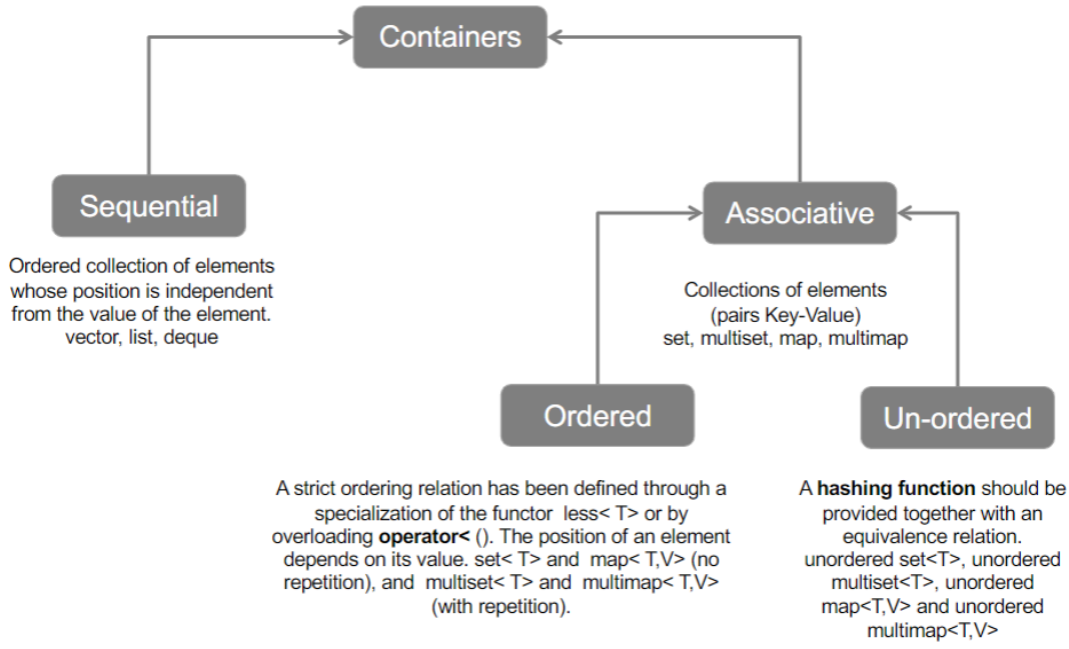

Figure 8.1: STL containers classes

8.2.1 Sequential containers

The sequential containers provide fast sequential access to their elements. However, they offer different performance trade-offs relative to:

- the costs to add or delete elements in the container
- the costs to perform non-sequential access to elements in the container

Some of the most important sequential containers are:

Container	Description
`vector`	Flexible-size array. Supports **fast random access**. Inserting or deleting elements other than the back is slow.
`deque`	Double-ended queue. Supports **fast random access**. **Fast insert/delete at front or back**.
`list`	Doubly linked list. Supports only **bidirectional sequential access**. **Fast insert/delete at any point**.
`forward_list`	Singly linked list. Supports only **sequential access in one direction**. **Fast insert/delete at any point**.
`array`	Fixed-size array. Supports **fast random access**. Cannot add or remove elements.
`string`	Specialized container (characters only), similar to `vector`. **Fast random access**. **Fast insert/delete at the back**.

Table 8.1: Characteristics of different sequential containers in C++.

8.3 How are `vectors` implemented?

As we know, a `vector` can hold an arbitrary number of elements of the same type, up to the limit of available memory. The size of a vector can grow and shrink dynamically, as elements are added or removed. The vector size changes by 3 mechanisms:

- **Pushing elements:** When we add an element to the vector, the vector size grows.
- **Resizing:** This is done by the method `resize()`.
- **Assigning:** When we assign a vector to another vector, the size of the destination vector changes.

To support random access, the vector elements are stored in a contiguous block of memory. Given that elements are contiguous, and that the size of the container is flexible, when we add an element if there is no room for it:

- the container must **allocate new memory** to hold the existing elements plus the new one.
- **copy** the elements from the old location into the new space.
- **add** the new element.
- **deallocate** the old memory.

To avoid these costs, library implementators use allocation strategies that reduce the number of times the container is reallocated. One of them is, when new memory is needed, allocate capacity beyond what is immediately needed. The container holds this storage in reserve and uses it to allocate new elements as they are added. This allocation strategy is dramatically more efficient that reallocating the container every time a new element is added.

8.3.1 Adding elements to a vector

There are 3 main operations that occur when we add an element to a vector:

- **Reserve**
- **Resize**
- **Push back**

`reserve(unsigned newalloc)`

This method reserves memory for `newalloc` elements. If the requested size is less than or equal to the current capacity, the method does nothing. Otherwise, it allocates memory for `newalloc` elements, copies the existing elements into the new memory, and deallocates the old memory.

The complexity of this method is linear in the number of elements in the vector.

`resize(unsigned newsize)`

Given `reserve`, `resize` is easy. While the first deals with the memory allocation, `resize` deals with the element values. The `resize` goal is to resereve `newsize` elements, and fill the elements with indeces between sz and `newsize-1` with the default value of the element type.

The complexity of this method is linear in the argument `newsize`.

`push_back(T val)`

This method adds a new element with value `val` to the end of the vector. If there is enough room, it simply increments the size of the vector and copies the new element into the pointer after the last previous element. If there is not enough room, it calls `reserve` with twice the current capacity, copies the elements into the new memory, and adds the new element.

The complexity of this method is linear in the number of elements in the vector.

8.3.2 Vector complexity: final considerations

Working with vectors implies that `push_back` worst case complexity is linear in the number of elements in the vector. However, the amortized (average) complexity of `push_back` is constant. This is because the cost of reallocation is spread over the number of insertions.

Also, random access on vectors is constant time $O(1)$, as the elements are stored in a contiguous block of memory. Insert in the middle worst and average cases are $O(N)$.

8.3.3 Range checking

Ideally, we would like to have range checking on vectors. However, STL does not provide it by default. Why is this? Because range checking is expensive, and the STL designers wanted to keep

the cost of using vectors low.

In general:

- Range checking costs in speed and code size.
- Some projects need optimal performance: think huge (e.g., Google) and tiny (e.g., embedded systems).
- The standard must serve everybody. You can build checked on top of optimal, but not the other way around.

8.4 Sequential contaniers overview

As we saw before, sequential containers provide efficient, flexible memory managememt. The strategies that containers use for storing their elements have inheren, and sometimes significant, impact on the efficiency of operations like adding or removing elements. In some cases, these strategies also affect whether a particular container supplies a particular operation.

Based on amortized complexity, the following table provides a comparison of each sequential container:

	Random Access	Add element at back	Add element in front	Add element in the middle
`vector`	+	+	N.A. (-)	-
`deque`	+	+	+	-
`list`	N.A.	++	++	++
`forward_list`	N.A.	N.A. (-)	++	++
`array`	+	N.A.	N.A.	N.A.

Figure 8.2: A comparison of the STL sequential containers

Note: the operation **add in the middle** assumes you have access (an iterator) to the element before the one you will insert.

The `list` container implements a doubly-linked list, while the `forward_list` implements a singly-linked list.Both are designed to make it fast to add or remove an element anywhere in the container, but in exchange, these types do not support random access to elements. Also, the memory overhead for these containers is significant.

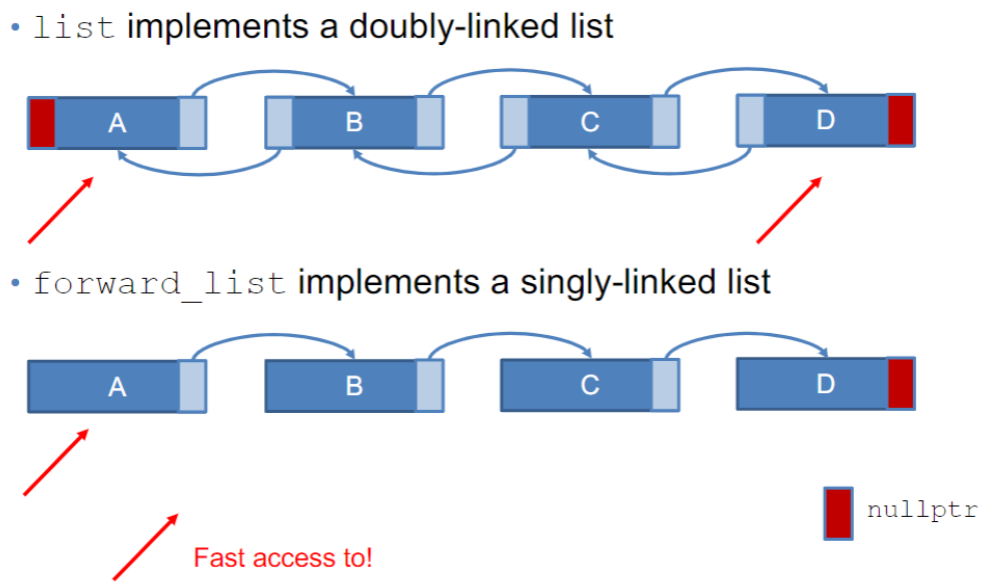

Figure 8.3: `list` and `forward_list`

A `deque` is a more complicated data structure. Like `string` and `vector`, it supports **fast random access**, and **adding** or **removing** elements in the **middle** of a `deque` is an expensive operation. Adding or removing elements at either the **front or end** is a fast operation, comparable to a list or **forward_list**.

The `forward_list` and `array` types were added by `C++ 11`. The first one is comparable with a `list`, but it does not have the `size` operation and is more memory efficient. In exchange, it can only be accessed from the **begin to end** (cannot move backwards). On the other hand, an `array` is a safer, easier-to-use alternative to built-in arrays and has a fixed size, but it does not support operations to add and remove elements or to resize.

8.4.1 Which one to use?

It is best to use `vector` unless there is a good reason to prefer another container. If you need lots of small elements and space overhead matters, don't use `list` or `forward_list`. If the program requires random access to elements, use `vector` or `deque`. If the program needs to insert or delete elements in the middle of the container, use a `list` or a `forward_list`. If the program needs to insert or delete elements at the front and the back, but not in the middle, use `deque`.

There are some gray cases. If the program needs to insert elements in the middle of the container only while reading input (e.g., to keep them in order), and subsequently needs random access to the elements:

- First decide whether you actually need to add elements in the middle of the container. It is often easier to append to a `vector` and then call the library `sort` function to reorder the container when you are done with the input.

- If you must insert into the middle, consider using a `list` for the input phase. Once the input is complete copy the `list` into a `vector`.

If the program needs random access and needs to insert and delete elements in the middle, evaluate the relative cost of accessing the elements in a `list` of `forward_list` versus the cost of inserting or deleting elements in a `vector` or `deque`.

In general, the predominant operation of the application (whether it does more access, more insertion or more deletion) will determine the choice of the container type. Application performance testing is usually needed.

ADVICE: If ypu are not sure which container to use, write your code using only operations common to `vector` and `list`:

- Use iterators, not subscripts.
- Avoid random access to elements.

This way, the code could be changed easily!

8.4.2 Container commont types

The following table shows some common types between containers:

`iterator`	Type of the iterator for the considered container type
`const_iterator`	Iterator type that can read but cannot change its elements
`size_type`	Uns. int. large enough to hold the largest possible container size
`difference_type`	Sign. int. large enough to hold the distance between two iterators
`value_type`	Element type
`reference`	Element lvalue reference type, synonymous for `value_type &`
`const_reference`	Element const lvalue type, (i.e., `const value_type &`)

Figure 8.4: Container common types

8.4.3 Container common operations

The following table shows the containers common operations:

Operation	Description
`C c;`	Default constructor, empty container
`C c1(c2);`	Construct c1 as a copy of c2
`C c(b, e);`	Copy elements from the range denoted by the iterators b and e (no for `array`)
`C c{a, b, c, ...};`	List initialize c
`c.size()`	Number of elements in c (no for `forward_list`)
`c.max_size()`	Maximum number of elements c can hold
`c.empty()`	`true` if c has no elements, `false` otherwise
`c.insert(args)`	Copy element(s) as specified by args in c
`c.emplace(inits)`	Use inits to construct an element in c
`c.erase(args)`	Remove element(s) specified by args
`c.clear()`	Remove all elements from c
`==`, `!=`	Equality
`<`, `>`, `<=`, `>=`	Relationals (no for unordered associative containers)
`c.begin()`, `c.end()`	Return iterator to the first/one past element in c
`c.cbegin()`, `c.cend()`	Return `const_iterator`
`reverse_iterator`	Iterator that addresses elements in reverse order
`const_reverse_iterator`	Reverse iterator read-only
`c.rbegin()`, `c.rend()`	Iterator to the last, one past the first element in c
`c.crbegin()`, `c.crend()`	Return `const_reverse_iterator`

Table 8.2: Container operations in STL

Since `C++ 20`, the very same iterators can be obtained through the `<ranges>` library:

- `ranges::begin(c)`, `ranges::end(c)`
- `ranges::cbegin(c)`, `ranges::cend(c)`
- `ranges::rbegin(c)`, `ranges::rend(c)`
- `ranges::crbegin(c)`, `ranges::crend(c)`

These are more helpful with functions that use both the begin and end iterators. For example, consider the `sort` function:

```
vector<double> v = {3, 7, 9.2, 44.3};
sort(v.begin(), v.end());
```

This now can be written as:

```
#include <ranges>

vector<double> v = {3, 7, 9.2, 44.3};
ranges::sort(v);  // sort is a function of <algorithm>
```

8.4.4 Creating a container

Each container is defined in a header file with the same name as the type of container. Note that containers are class templates, so you must specify the type of elements that the container will hold. For example:

```
#include <vector>
#include <list>
#include <deque>

vector<int> vi;  // vector of integers
list<string> ls; // list of strings
deque<double> vd; // deque of doubles
```

Almost any type can be used as element type of a sequential container. Some container operations impose requirements of their own on the element type. In other words, we can define a container for a type that does not support an operation-specific requirememt, but we can use an operation only if the element type supports it.

8.4.5 Iterators

Iterators have also a common interface. All the iterators let access an element from a container providing the dereference operator and allow to move from one element to the next through the increment operator.

The following table shows the common operations of 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 8.5: Common operations of iterators

Also for the containers that support random access, we have the following operations:

iter + n iter - n	Adding (subtracting) an integral value *n* from the iterator *iter* yields an iterator *n* elements forward of backward than *iter* within the container
iter1 +=n iter1 -=n	Assign to *iter1* the value of adding (subtracting) 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 8.6: Random access operations

The previous table only apply to iterators for `string`, `vector`, `deque` and `array`.

Iterator ranges

These are denoted by a pair of iterators, each of which referers to an element, or one past the element, in the same container. They are often referred to as begin and end, or first and last. In an iterator range we have a left-closed, right-open interval $[begin, end)$.

We have some nice properties of iterator ranges:

- If begin equals end, the range is empty.
- If begin is not equal to end, there is at least one element in the range. and begin refers to the first element in that range.
- We can increment begin som number of times until it equals end.

For example:

```
1  vector<int> v = {1, 2, 3, 4, 5};
2
3  for (auto p = v.begin(); p != v.end(); ++p)
4      cout << *p << endl;
5
6  // or
7
8  p = v.begin();
9  while (p != v.end())
10     cout << *p++ << endl;
```

Reverse iterators

Most containers provide reverse iterators, i.e., an iterator that goes backward through a container and inverts the meaning of the iterator operations. For example, incrementing a reverse iterator moves it to the previous element, and decrementing it moves it to the next element.

8.4.6 Assignment operator

The assignment operator replaces the entire range of elements in the left-hand container with copies of the elements in the right-hand container. The left-hand container must be the same type as the right-hand container, and the element type must be assignable. After an assignment, the left-hand container is equal to the right-hand container.

The following table shows operations related to the assignment operator in containers:

c1=c2	Replace the elements in c1 with copies form c2. c1 and c2 must have the same type
c={a, b, c, ...}	Replace the elements in c1 with copies of elements in the initializer list
swap(c1,c2)	Exchanges elements in c1 with those in c2. c1 and c2 must have the same type
c1.swap(c2)	
seq.assign(b,e)	Replaces elements in seq with those in the range denoted by the iterators b and e. b and e **must not be** iterators belonging to seq
seq.assign(i1)	Replaces elements in seq with those in the initializer list i1
seq.assign(n,t)	Replaces elements in seq with n elements with value t

Figure 8.7: Operations related to the assignment operator

Using `swap`

The `swap` operation exchanges the elements in two containers. The containers must be of the same type and have the same element type. After the operation, the elements in the two containers are exchanged, but the containers themselves are not changed.

For example:

```
1 vector<int> v1 = {1, 2, 3, 4, 5};
2 vector<int> v2 = {6, 7, 8, 9, 10};
3
4 swap(v1, v2);  // v1 = {6, 7, 8, 9, 10}, v2 = {1, 2, 3, 4, 5}
```

With the exception of `array`, the `swap` operation is a constant-time operation. Swapping two arrays does exchange the elements, and it requires time proportional to the number of elements in the array, complexity $O(N)$.

The fact that elements are not moved means that iterators, reference and pointers into the containers are not invalidated by the `swap` operation. They will refer to the same elements as they

did before the operation, but after the swap, those elements will be in a different container.

In C++ 11, the containers offer both a member function and a non-member function to perform the swap operation. The member function is `c1.swap(c2)`, and the non-member function is `swap(c1, c2)`. As a matter of habit, is better to use the non-member function, as it is more flexible and can be used with any two containers of the same type.

8.4.7 Container size operations

Container types have three size-related operations:

- `size()` returns the number of elements in the container.
- `empty()` returns a bool that is true if the size is zero, and false otherwise.
- `max_size()` returns the maximum number of elements that the container can hold.

Note that `forward_list` does not have a `size()` operation, but it does have `max_size()` and `empty()` operations.

8.4.8 Relational operators

Every container type supports the equality operators `==` and `!=`. With the exception of unordered associative containers, all container types also support the inequality operators `<`, `>`, `<=`, and `>=`. The right-hand and left-hand containers must be of the same type, and must hold elements of the same type.

Comparing two containers performs a pairwise comparison of the elements in the containers. If both containers are the same size and all the elements in the containers are equal, the containers are equal, otherwise, they are not. If the containers have different sizes, but every element of the smaller one is equal to the corresponding element of the larger one, then the smaller container is less than the larger one. If neither container is an initial subsequence of the other, then the comparison is based on the first element that differs. This is called **lexicographical comparison**.

Note that instead of relying on `==` and `!=`, it is often better to use the `equal` algorithm from the `<algorithm>` header. This algorithm is more flexible and can be used with any two containers of the same type. In this case, you might also specify a binary function to compare the elements.

8.4.9 Adding elements to a sequential container

Excepting `array`, all sequential containers provide flexible memory management. We can add of remove elements dynamically changing the size of the container at run time.

Operation	Description
`c.push_back(t)`	Creates an element with value `t` or constructed from arguments at the end of `c`.
`c.emplace_back(args)`	Creates an element with value `t` or constructed from arguments at the end of `c`.
`c.push_front(t)`	Creates an element with value `t` or constructed from arguments at the front of `c`.
`c.emplace_front(args)`	Creates an element with value `t` or constructed from arguments at the front of `c`.
`c.insert(p, t)`	Creates an element with value `t` or constructed from arguments **before** the element denoted by iterator `p`.
`c.emplace(p, args)`	Creates an element with value `t` or constructed from arguments **before** the element denoted by iterator `p`.
`c.insert(p, n, t)`	Creates `n` elements with value `t` **before** the element denoted by iterator `p`.
`c.insert(p, b, e)`	Inserts the elements from the range denoted by the iterators `b` and `e` **before** the element denoted by iterator `p`. `b` and `e` may not be in `c`.
`c.insert(p, {i1})`	`i1` is a braced list of element values. Inserts the elements **before** the element denoted by iterator `p`.

Table 8.3: Adding Elements to a Sequential Container

When we use these operations, we must remember that the containers use different sttrategies for allocating elements and that these strategies affect performance:

- Adding elements anywhere but at the end of a vector or string, or anywhere but the beginning or end of a deque, requires elements to be moved.
- Adding elements to a vector or string may cause the entire object to be reallocated.
- Reallocating an object requires allocating new memory and moving elements from the old space to the new space.

How a `deque` works

A `deque` is a double-ended queue. It supports fast random access, and fast insert and delete at the front or back. The `deque` organizes its elements in chunks of memory referred by a sequence of pointers. It guarantees amortized constant time for adding or removing elements at the front or back, but inserting or removing elements in the middle is slow.

The worst case complexity for `push_back` and `push_front` is $O(N)$.

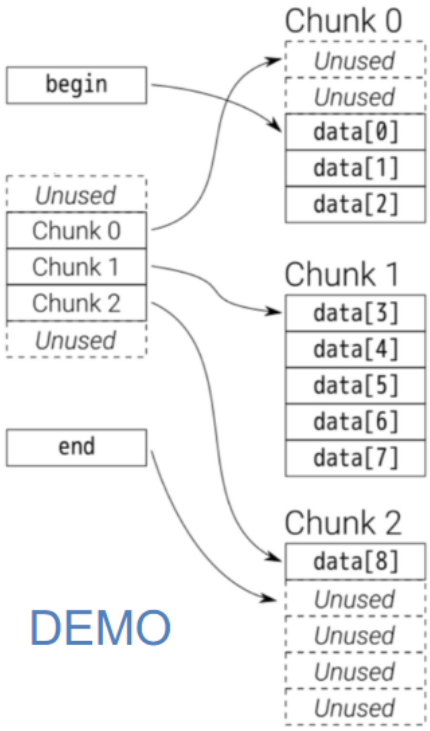

Figure 8.8: How a `deque` works

Adding elements in the middle of a container

Adding elements in the middle of a container is an expensive operation for all containers except `list` and `forward_list`.

To insert an element in the middle of a container, we rely on the `insert` operation. It lets us insert one or more elements at any point in the container. The `insert` is supported for `vector`, `string`, `deque`, and `list`, and `forward_list` provides specialized versions.

The operator `insert` takes an iterator as its first argument, which indicates where in the container to put the element(s) (any position, including one past the end). Element(s) is/are inserted before the position denoted by the iterator (the iterator might refer to a non-existant element off the end of the container). It return an iterator to the first element inserted.

For example:

```
1  vector<int> v = {1, 2, 3, 4, 5};
2
3  auto it = v.begin();
4  ++it;  // it points to the second element
5
6  v.insert(it, 42);  // v = {1, 42, 2, 3, 4, 5}
```

We can use the return value of `insert` to insert multiple elements:

```
list<string> lst;
auto iter = lst.begin();

while (cin >> word)
    iter = lst.insert(iter, word);
```

This is equivalent to using `push_front`.

8.4.10 Accessing elements

We can access elements of a container using the following operations:

- `c.back()`: returns a reference to the last element in `c`. It is undefined if `c` is empty.
- `c.front()`: returns a reference to the first element in `c`. It is undefined if `c` is empty.
- `c[n]` and `c.at(n)`: returns a reference to the element at position `n` in `c`. It is undefined if `n` is out of range.

For example:

```
vector<int> v = {1, 2, 3, 4, 5};

if (!v.empty()) {
    auto val1 = v.front();  // val1 = 1, equivalent to *v.begin()
    auto val2 = v.back();   // val2 = 5, equivalent to *(v.end() - 1)
    auto val3 = v[2];       // val3 = 3
    auto val4 = v.at(2);    // val4 = 3
}
```

Note that the operators `[]` and `at()` are only supported for containers that provide random access to their elements. Also, the `c.back()` is not supported for `forward_list`.

IMPORTANT: The members that access elements in a container return **references**. This means that we can change the value of the element using the returned reference, but only if the container is not `const`. For example:

```
vector<int> v = {1, 2, 3, 4, 5};

v[2] = 42;  // v = {1, 2, 42, 4, 5}

auto &val = v.front();
val = 42;   // v = {42, 2, 42, 4, 5}

auto val2 = v.front();
val2 = 42;  // This does not change the value of the first element in v,
            // but the value of val2, as it is a copy of the first element.
```

8.4.11 Removing elements

We can remove elements from a container using the following operations:

c.pop_back()	Removes the last element in c. Undefined if c is empty
c.pop_front()	Removes the first element in c. Undefined if c is empty
c.erase(p)	Removes the element denoted by the iterator p. Undefined if p is the off-the-end iterator
c.erase(b,e)	Removes the range of elements denoted by the iterators b and e
c.clear()	Removes all elements in c

Figure 8.9: Removing elements from a container

`pop_front` and `pop_back`

These functions remove the first and last elements of a container, respectively, and return `void`. There is no `pop_front` for `vector` and `string`, and no `pop_back` for `forward_list`. We also cannot use `pop_front` or `pop_back` on an empty container.

`erase`

This member function removes elements at a specified position in the container. We can delete a single element or a range of elements. The function returns an iterator to the element that follows the last element removed. For example:

```
list<int> lst = {1, 2, 3, 4, 5};
auto it = lst.begin();

while (it != lst.end())
    if (*it % 2)
        it = lst.erase(it);
    else
        ++it;
```

8.4.12 Iterator invalidation

When we add or remove elements from a container, the iterators that refer to elements in the container may be invalidated. An iterator is invalidated if the element to which it refers is removed from the container. Using an invalidated iterator is a **serious error** that can lead to undefined behavior.

After an operation that adds or removes elements from a container, iterators, pointers and references to a vector or string are invalid if the operation causes the container to reallocate. If no reallocation happens indirect references to elements before the insertion remain valid; those to elements after the insertion are invalid. It is **very risky** to rely on this!

- Iterators, pointers and references to a `deque` are invalidated if we add elements anywhere but at the front or back of the container.

- If we add at the front of back, iterators are invalidated, but references and pointers to existing elements remain valid.
- Iterators, pointers and references to a `list` or `forward_list` remain valid after an insertion. This includes the off-the-end and before-the-beginning iterators.

When we loop through a container and modify it, we must be careful to avoid using an invalidated iterator. For example:

```
vector<int> v = {1, 2, 3, 4, 5};
auto it = v.begin();

// This is ok, as we are refreshing the end iterator
while (it != v.end())
    if (*it % 2)
        it = v.erase(it);
    else
        ++it;

// This is not ok, as we are not refreshing the end iterator
auto end = v.end();
while (it != end)
    if (*it % 2)
        it = v.erase(it);
    else
        ++it;
```

8.5 Adaptors

Container adaptors ae interfaces created on top of a (limited) set of functionalities of a pre-existing sequential container, which provide a different API. When you declare the container adaptors, you have an option of specifying which sequential container to use as underlying container.

We have three main container adaptors:

- `stack`:
  - This container provides a **last-in, first-out** (LIFO) access.
  - You remove (pop) elements in the reverse order you insert (push) them. You cannot get any elements in the middle of the stack.
  - Usually, this goes on top of a `deque`.
- `queue`:
  - Container providing a **first-in, first-out** (FIFO) access.
  - You remove (pop) elements in the same order you insert (push) them. You cannot get any elements in the middle of the queue.
  - Usually, this goes on top of a `deque`.

- `priority_queue`:
  - Container providing **sorted-order** access to elements.
  - You can insert (push) elements in any order, and then retrieve (pop) the highest-priority element.
  - Priority queues in `C++` STL use a heap structure internally, which in turn is basically **array-backed**; thus, usually goes on top of a `vector`.

8.6 Associative containers

Associative containers support the general container operations, but they do not support the sequential-container position-specific operations, such as `push_front`. Because the elements are stored based on their **keys**, these operations would be meaningless for the associative containers. Nevertheless, they have:

- Type aliases
- **Bidirectional iterators**
- Specific operations
- Hash functions (for unordered associative containers)

The main associative containers are:

- For the header `<map>`:
  - `map`: a collection of key-value pairs, where the keys are unique.
  - `multimap`: a collection of key-value pairs, where the keys may not be unique.
- For the header `<set>`:
  - `set`: a collection of unique keys.
  - `multiset`: a collection of keys that may not be unique.
- For the header `<unordered_map>`:
  - `unordered_map`: a map organized by a hash function, where the keys are unique.
  - `unordered_multimap`: a map organized by a hash function, where the keys may not be unique.
- For the header `<unordered_set>`:
  - `unordered_set`: a set organized by a hash function, where the keys are unique.
  - `unordered_multiset`: a set organized by a hash function, where the keys may not be unique.

8.6.1 `map`

A `map` is a collection of key-value pairs, where the keys are unique. It is often referred to as an associative array. An associative array is like a "normal" array, but instead of using integers as indices, you can use any type of key.

Values in a `map` are found by a key and not by their position. For example, given a map of names to phone numbers, we'd use a person's name as a subscript to fetch their phone number:

```
map<string, string> phone_book;

phone_book["John"] = "123-456-7890";

cout << phone_book["John"] << endl;  // prints 123-456-7890
```

Implementation of a `map`

A `map` is implemented by red-black trees, self-balancing binary search trees. The `map` is always sorted by its key according to the comparison function. The complexity of the operations is logarithmic ($O(\log(n))$) in the size of the container.

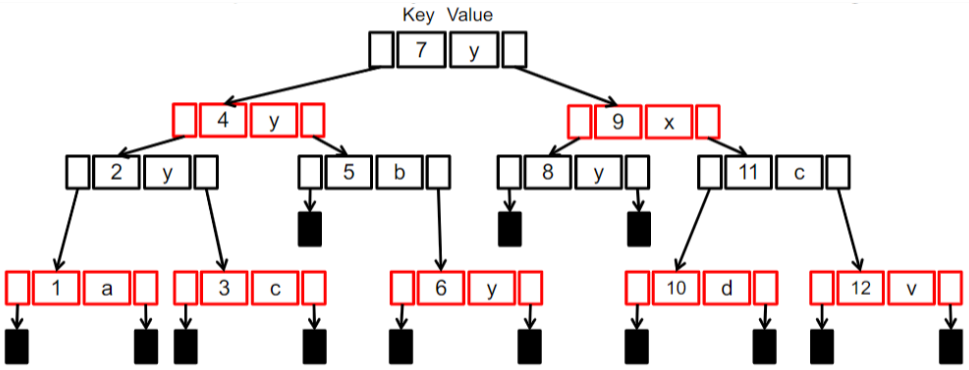

Figure 8.10: How a `map` works

8.6.2 `set`

A `set` is simply a collection of objects. A `set` is most useful when we simply want to know whether an object is present in the collection or not.

For example, a business might have a set to hold the names of individuals sho have written bad checks. When a new check is received, the business can quickly check the set to see if the person has a history of writing bad checks:

```
set<string> bad_check_writers;

if (bad_check_writers.find(name) != bad_check_writers.end())
    cout << "Do not accept check from " << name << endl;
```

Implementation of a `set`

A `set` is implemented by red-black trees, self-balancing binary search trees. The `set` is always sorted by its key according to the comparison function. The complexity of the operations is logarithmic ($O(\log(n))$) in the size of the container.

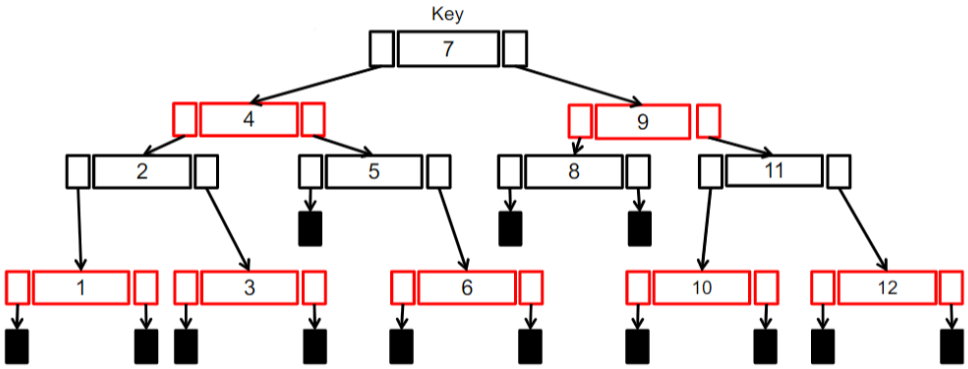

Figure 8.11: How a `set` works

8.6.3 Associative vs sequential containers

There are two main differences between associative and sequential containers:

- The associative containers do not support the position-specific operations, such as `push_front` and `push_back`. Also they do not support constructors or insert operations that take an element value and a count.
- The asociative containers iterators are always bidirectional. Accessing `begin()` and `end()` is always $O(1)$, and incrementing or decrementing an iterator is always $O(1)$.

8.6.4 Requirements on the key type

For the **ordered containers**, the key type must define a way to compare the elements. By default, the library uses the `<` operator to compare the keys, but we can also supply our own `<` operation to use a strict weak ordering over the key type.

It has to be a strict weak ordering, which means that the relation must be:

- **Irreflexive**: $a < a$ is always false.
- **Transitive**: If $a < b$ and $b < c$, then $a < c$.
- **Antisymmetric**: If $a < b$, then $b < a$ is false.

8.6.5 The `pair` type

The `pair` type is a simple way to combine two values into a single object. It is a template class that takes two type parameters. It is defined in the `utility` header. For example:

```
pair<string, int> p1;  // default initialized

pair<string, int> p2("hello", 42);  // initialized to "hello" and 42

pair<string, int> p3 = make_pair("world", 42);  // initialized to "world" and 42
```

The `pair` type is used to return two values from a function. For example, the `insert` operation on a `map` returns a `pair` of an iterator to the element inserted and a bool that indicates whether the insertion was successful.

The data members of a `pair` are public, and can be accessed using the `first` and `second` members. Note that the elements in a `map` are `pair`s. We have a limited set of operations for `pair`:

`pair<T1, T2> p;`
`pair<T1, T2> p(v1, v2);`
`pair<T1, T2> p={v1, v2};`
	Pair definition with or without initialization

`make_pair(v1, v2)`	Pair definition. Type of pair is inferred from v1 and v2 type.
`p.first / p.second`	Returns first or second member of p
`p1(<,>,<=,>=,==,!=)p2`	Relational operators and equality. For example `p1<p2 if p1.first<p2.first or !(p2.first<p1.first) && p1.second < p2.second`

Figure 8.12: Operations for `pair`

8.6.6 Associative container type aliases

We have the following type aliases for the associative containers:

- `key_type`: the type of the key.
- `mapped_type`: the type of the value.
- `value_type`: it is the same as `key_type` for `set` and a `pair` of `key_type` and `mapped_type` for `map`.

Note that the key is `const`, meaning that we cannot change the key of an element in an associative container. For example, when we retrieve an element from a `map`, we can change the value of the element, but not the key. Look at the following example:

```
map<string, int> m;

m["hello"] = 42;

auto it = m.find("hello");
if (it != m.end())
    it->second = 43;  // ok
it->first = "world";  // error
```

8.6.7 Iterating over an associative container

When we use an iterator to travers a map, multimap, set or multiset, the iterators yield elements in **ascending order** according to the key. For example, consider the following code:

```
map<string, int> m = {{"hello", 42}, {"world", 43}};
for (auto it = m.begin(); it != m.end(); ++it)
    cout << it->first << " " << it->second << endl;

// prints:
// hello 42
// world 43
```

8.6.8 Adding elements to an associative container

Because (unordered) `map` and (unordered) `set` contain unique keys, inserting elements that are already present, has no effect:

`c.insert(v)` `c.emplace(args)`	v `value_type` object. *args* are used to construct an element. Insert in **map** or **set** only if an element with the given key already is not in c. **Return a pair of an iterator referring to the element with the given key and a bool indicating whether the element was inserted**. For multimap and multiset it returns an iterator to the new element.
`c.insert(b, e)`	b and e are iterators denoting a range of `c::value_type` elements
`c.insert(il)`	il is a braced list of values
`c.insert(p, v)` `c.emplace(p, args)`	Like the first two, but uses p as a hint for where to begin the search for where the new element should be stored. Returns an iterator to the element with the given key

Figure 8.13: Operations for adding elements on associative containers

For example:

```
vector<int> ivec = {2, 4, 6, 8, 2, 4, 6, 8};

set<int> set2;
set2.insert(ivec.cbegin(), ivec.cend()) // set2 has 4 elements
```

8.6.9 Erasing elements

We can erase one element or a range of elements by passing erase or an iterator pair:

`c.erase(k)`	Removes every element with key k from c. Returns `size_type` indicating the number of removed elements
`c.erase(p)`	Removes the element denoted by the iterator p. Returns an iterator to the element after p
`c.erase(b,e)`	Removes elements in the range from b to e, and returns e

Figure 8.14: Erasing ops on associative containers

8.6.10 Subscritping a `map`

The `map` and `unordered_map` containers provide the susbcript operator and a corresponding `at` function. The `set` types do not support subscripting because there is no value associated with a key. We also cannot subscript a `multimap` or an `unordered_multimap` because there may be more than one value associated with a given key:

`c [k]`	Returns the element with key k; **if k is not in c, adds a new, value initialized element with key k.**
`c.at(k)`	Checked access to the element with key k; out_of_range error if k is not in c.

Figure 8.15: Subscripting a map

8.6.11 Accessing elements

We can use these operations to access elements on an associative container:

`c.find(k)`	Returns an iterator to the **first element with key k,** or the off-the-end iterator if k is not in the container
`c.count(k)`	Returns the number of elements with key k. For the containers with unique keys, the result is always zero or one
`c.lower_bound(k)`	Return an iterator to the **first element with key not less than k**
`c.upper_bound(k)`	Return an iterator to the **first element with key greater than k**

Figure 8.16: Accessing elements on associative containers

Note that lower and upper bound are not valid for unordered containers, in that case you can rely on `equal_range`.

8.6.12 `unordered_map` & `unordered_set`

The unordered associative containers are just a collection of buckets, each containing a variable number of items. These containers use a **hash function** to map elements to buckets. Given the item key, this identifies the proper bucket to store such item. All of the elements with a given hash value are stored in the same bucket, and all the elements with the same key (in the multi-version) will be in the same bucket.

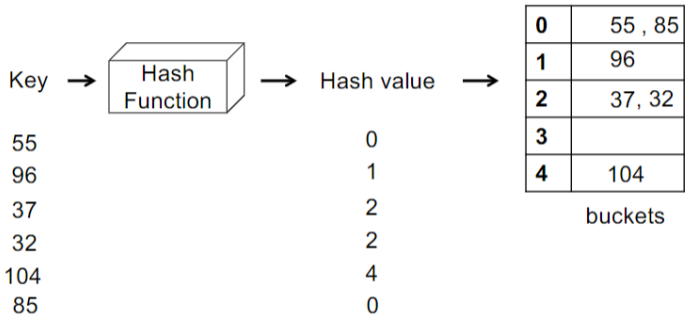

Figure 8.17: How a hash works

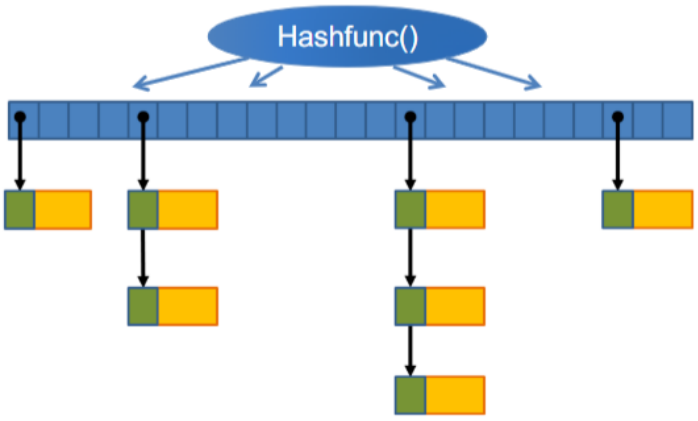

Figure 8.18: Diagram of a map using a hash function

The performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets. The average complexity is constant ($O(1)$) and the worst case has a linear complexity ($O(N)$).

Rather than using comparison operation to organize their elements, these containers use a hash function ans the key type's `==` operator. We should use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve.

8.6.13 Complexities of operations

You can choose the most suitable container in terms of complexity, depending on what operations you need to apply on them:

Container	Operation		
	Insert	Find	Delete
list/ forward_list	O(1)	O(n)	O(1)
set/map	O(Log(n))	O(Log(n))	O(Log(n))
unordered set/map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)

Figure 8.19: Complexities of main operations

8.6.14 `map` vs `unordered_map`

If the most frequent operations of your application are **find, insert or delete** of single elements and you access through a key:

- Use a `map` if you want to optimize the worst case complexity ($O(\log N)$ vs $O(N)$)
- Use an `unordered_map` if you want to optimize the average case complexity ($O(1)$ vs $O(\log N)$)

Chapter 9

Parallel computing

9.1 Introduction and basics

If we want to improve the performance of our applications, i.e., make them run faster, we have 3 main strategies: work harder, work smarter or get some help. The computer engineering recipe to do so is as follows:

- Use **faster hardware**: improve the operating speed of processors and other components. This is only constrained by the speed of light, thermodynamic laws, high financial costs for processor manufacture and other technical constraints.
- **Optimize algorithms** and techniques used to solve computational tasks.
- use **multiple computers/cores** to solve a particular task: connect multiple processors (or cores) together and coordinate their computational efforts.

The last point is the one we will focus on in this chapter. This is the core concept of **parallel computing**.

9.1.1 What is parallel computing?

The concept of **parallelism** refers to applying **multiple processor units (PUs)** to a single problem. To do this, we:

- **Decompose** the computation into many pieces.
- **Assign** these pieces to different PUs.

The goal is to solve the problem faster than with a single PU. A **parallel computer** (or system), is a computer that contains multiple PUs, such that each of them work on its section of the problem, and can exchange information with other PUs.

9.1.2 Parallel computing vs. Serial computing

There are two main advantages of using parallel computing over serial computing. These are:

- **Total performance**

- **Total memory**

Parallel computers enable us to solve problems that benefit from (or require) a fast solution, or that require more memory than a single computer can provide. There are examples of problems that require both conditions to be met, such as: weather forecasting, fluid dynamic simulations, financial simulations, etc.

Some specific benefits of parallel computing include:

- More data points can be processed in the same amount of time: this implies bigger domains, bigger spatial resolution and more particles to consider.
- More time steps: this allows for longer runs and better temporal resolution.
- Faster execution: we can get a lower time-to-solution, and even more solutions in the same amount of time. This allows for larger simulations in real time.

9.1.3 Examples of parallel applications

The following is a list of some fields in which parallel computing is used:

- Artificial intelligence and machine learning.
- Weather forecasting.
- Vehicle design and dynamics.
- Analysis of protein structures.
- Human genome work.
- Astrophysics.
- Earthquake wave propagation.
- Molecular dynamics.
- Climate, ocean and atmospheric modeling.
- Imaging and rendering.
- Petroleum exploration.
- Database query.
- Ozone layer monitoring.
- Natural language understanding.
- Study of chemical phenomena.

And many other scientific and industrial applications. As we can see, parallel computing is a key technology in many fields of science and engineering.

9.2 Flynn's taxonomy

Flynn's taxonomy is a classification of computer architectures based on the number of instruction streams and data streams available in the architecture. It was proposed by Michael J. Flynn in 1966. The taxonomy is based on two concepts:

- **Instruction stream**: the sequence of instructions that are executed by the computer.
- **Data stream**: the sequence of data that is processed by the computer.

On each type of stream, we can have one or multiple streams. This gives us four possible combinations, which are:

- **SISD**: Single Instruction, Single Data (sequential computers).
- **SIMD**: Single Instruction, Multiple Data.
- **MISD**: Multiple Instruction, Single Data (non-existent in practice).
- **MIMD**: Multiple Instruction, Multiple Data (most common and general parallel machine).

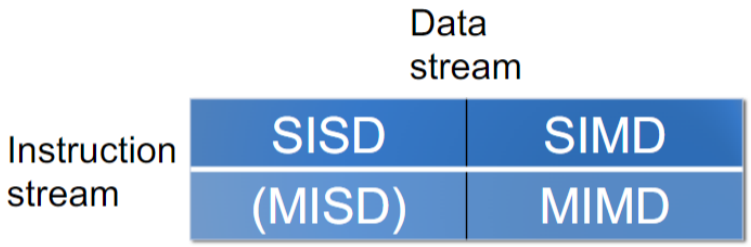

Figure 9.1: Flynn's taxonomy

9.2.1 SISD: conventional computers

SISD stands for Single Instruction, Single Data. This is the most common type of computer architecture, and it is the one we have been using so far. In this type of architecture, a single processor executes a single instruction stream to operate on data stored in a single memory. This is the traditional von Neumann architecture.

Here, the speed is limited by the rate at which the processor can execute instructions, and transfer information internally.

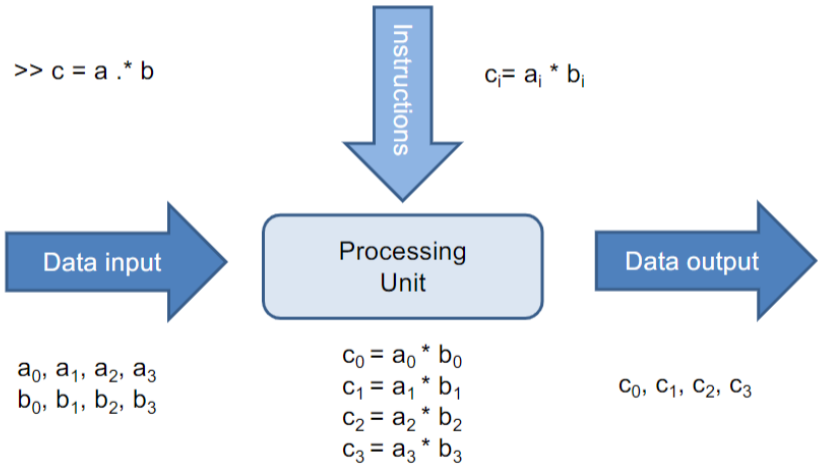

Figure 9.2: SISD architecture

9.2.2 SIMD: vector computers

SIMD stands for Single Instruction, Multiple Data. In this type of architecture, a single instruction stream controls multiple processing elements, which operate on multiple data streams. This is the case of vector computers.

In this architecture, the same operation is performed on multiple data points simultaneously. This is useful for problems that can be parallelized, such as image processing, signal processing, etc. SIMD relies on the regular structure of computations.

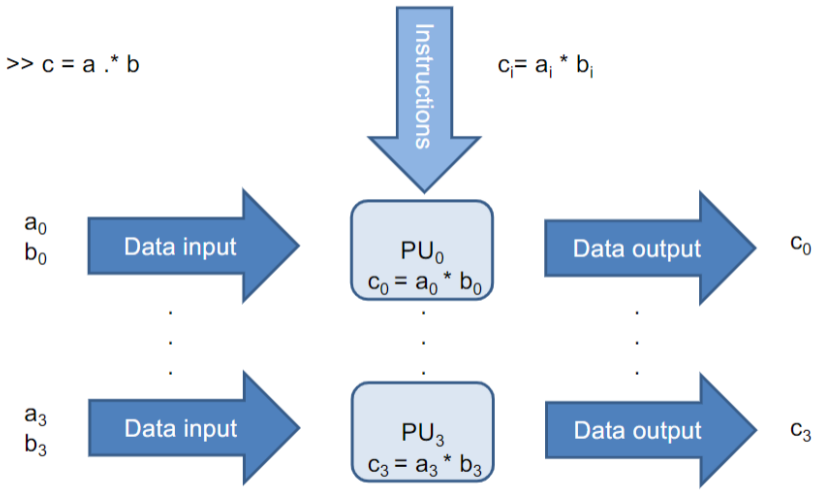

Figure 9.3: SIMD architecture

9.2.3 MIMD: parallel computers

MIMD stands for Multiple Instruction, Multiple Data. In this type of architecture, multiple processors execute multiple instruction streams to operate on multiple data streams. This is the most general type of parallel computer.

In this architecture, each processor can execute different instructions on different data. This is useful for problems that are not easily parallelizable, or that require different operations to be performed on different data points.

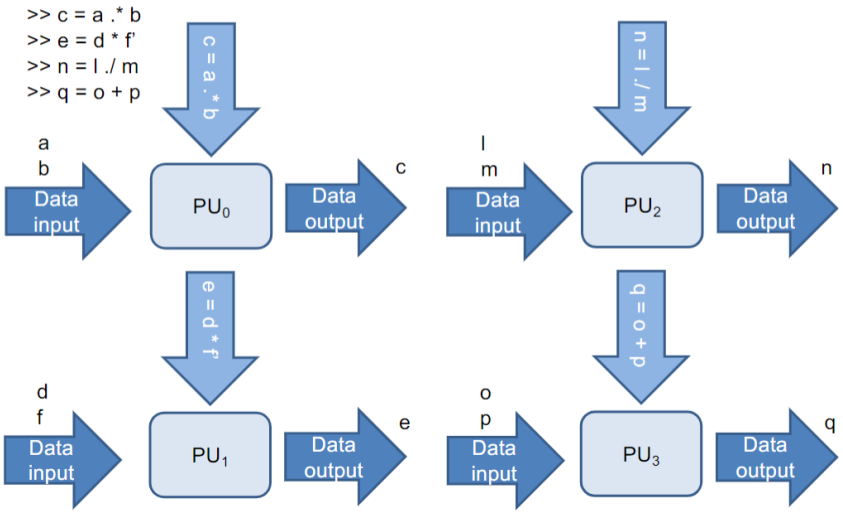

Figure 9.4: MIMD architecture

However, this architecture has some issues, such as:

- Data distribution and dependencies.
- Synchonization.
- Communication cost.

There are two main types of MIMD architectures:

- **Shared memory**: all processors share a common memory space. Processor-to-processor data transfers are done using this shared memory. It has scalability limits. There are 2 methods for memory access: by **bus** or by **crossbar**.

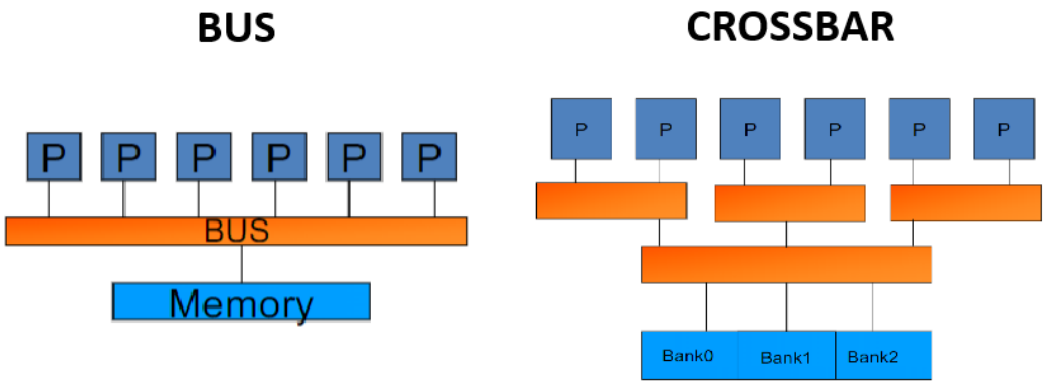

Figure 9.5: Shared memory architecture

- **Distributed memory**: each processor has its own memory space. Processors must do **message passing** to exchange data between them. This architecture is more scalable than shared memory, but has issues like load balancing, and I/O is more complex.

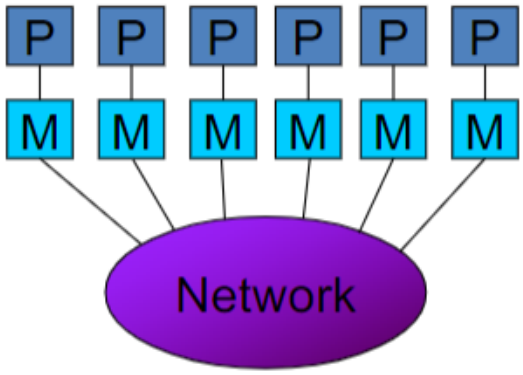

Figure 9.6: Distributed memory architecture

In this course, we will focus on **Message Passing Interface (MPI)**, that works both on shared and distributed memory architectures.

9.3 Limitations of parallel computing

Parallel computing is not a silver bullet. There are some limitations to it, such as:

- Not all problems can be parallelized: some problems are inherently sequential.
- There are theoretical upper limits to parallel speedup: Amdahl's law.
- There are also practical limits, like load balancing and non-computational sections (I/O, system ops, etc.).

- It has a different approach than sequential programming, so we need to rethink the algorithms and re-write the code.

9.3.1 Theoretical upper limits: Amdahl's law

We should notice that all parallel programs contain some sequential sections as well. These sections, as well as duplicated work and no useful work done (waiting for others), limit the parallel effectiveness:

- Lots of serial computation gives bad speedup.
- No serial work "allows" perfect speedup.

The concept of **speedup** is defined as the ratio of the time required to run a code on a single processor to the time required to run the same code on multiple (N) processors. This is formally expressed by **Amdahl's law**:

Law 1 (Amdahl's law)**.** *Let $S(N)$ be the speedup of a program running on N processors. Let f_s be the fraction of the program that is sequential, and $f_p = 1 - f_s$ be the fraction that is parallel. Then, the speedup is given by:*

$$S(N) = \frac{1}{f_s + \frac{f_p}{N}} \tag{9.1}$$

If we define t_n as the time to run the program on N processors, and t_1 as the time to run the program on a single processor, then the speedup can be expressed as:

$$S(N) = \frac{t_1}{t_n} \tag{9.2}$$

So we get:

$$t_n = t_1 \left(f_s + \frac{f_p}{N} \right) \tag{9.3}$$

From this, we can see that it takes only a small fraction of the program to be sequential to limit the speedup. For example, let us take a look at the following graph:

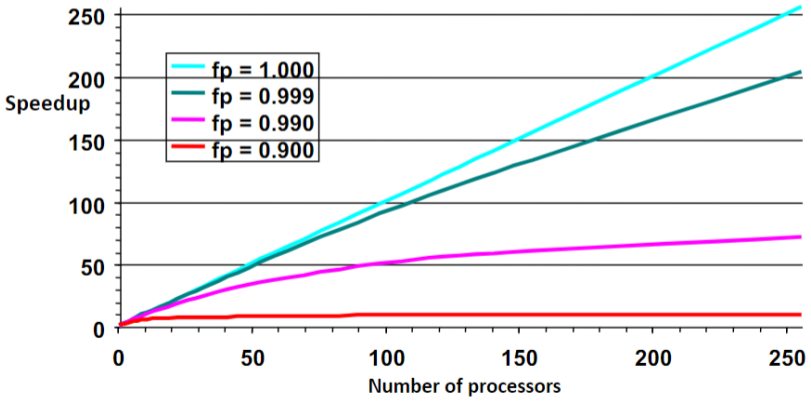

Figure 9.7: Amdahl's law

Amdahl's law vs Reality

Amdahl's law provides a theoretical upper limit to the speedup of a program, assuming that there is no parallelization overhead. However, in reality, overhead will result in a further degradation of the speedup. Let us look at the following graph:

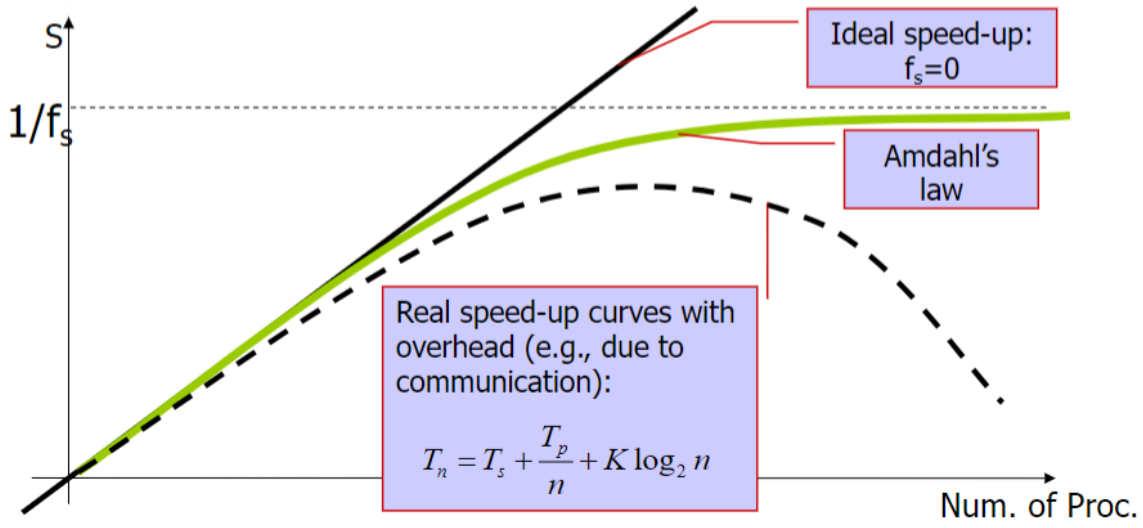

Figure 9.8: Amdahl's law vs Reality

9.3.2 Sources of parallel overhead

There are 3 main sources of overhead in parallel computing:

- **Interprocessor communication**: time to transfer data between the processors is usually the most significant source of parallel processing overhead.

- **Load imbalance**: in some parallel applications it is impossible to equally distribute the subtask workload to each processor. So, at some point, all but one processor might be done and waiting for the last one to complete.

- **Extra computation**: sometimes the best sequential algorithm is not easily parallelizable and one is forced to use a parallel algorithm based on a poorer but easily parallelizable sequential algorithm. Sometimes, repetitive work is done on each of the N processors, which leads to extra computation.

Communication effect

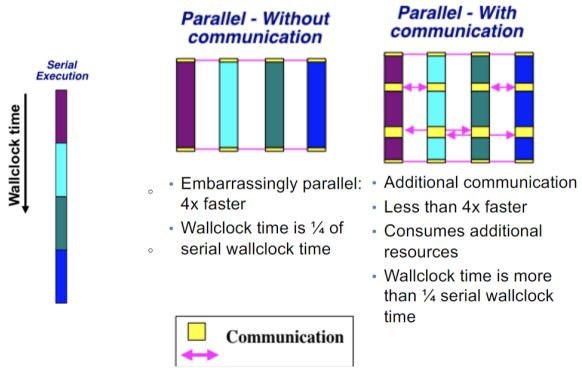

Figure 9.9: Communication effect

Load imbalance effect

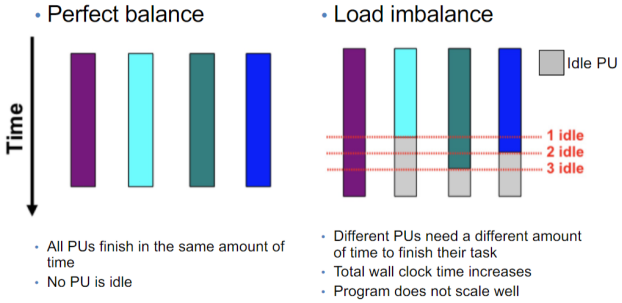

Figure 9.10: Load imbalance effect

Serial performance vs parallel performance

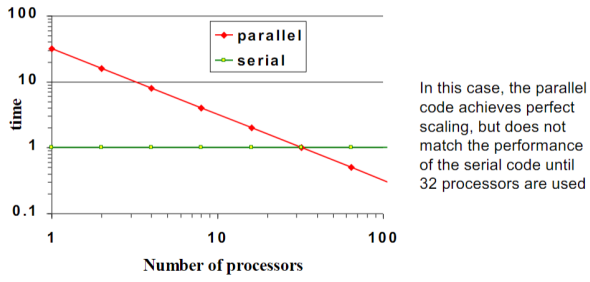

Figure 9.11: Serial performance vs parallel performance

9.3.3 Superlinear speedup

In some cases, we can get a superlinear speedup. This is when the speedup is greater than the number of processors. This can happen due to:

- **Caching effects**: the data fits in the cache, so the memory access is faster.
- **Better algorithms**: the parallel algorithm is better than the sequential one.

9.3.4 SLOW: Starvation, Latency, Overhead, Waiting

SLOW is a concept that summarizes the main issues in parallel computing:

- **Starvation**: there is not enough work to do due to insufficient parallelism or poor load balancing among distributed resources.
- **Latency**: waiting for access to memory or other parts of the system.
- **Overhead**: extra work that must be done to manage program concurrency and parallel resources, rather than the real work you want to perform.
- **Waiting for contention**: delays due to fighting to use shared resources. Network bandwidth is a major constraint.

We should notice that performance enhancing comes with a price: **complexity**. Before parallelizing a code, we should ask ourselves: is it worth our time to rewrite the code? Do the CPU requirements justify the parallelization? Will the code be used enough to justify the effort?

In practice, writing parallel applications is hard. It requires a lot of effort, and the results are not always as expected. We need to be careful when parallelizing a code, and we need to be sure that the parallelization is worth the effort.

Sometimes, performance characteristics of applications change, and become architecture dependent. And also, the debugging of parallel applications is harder than debugging sequential ones.

9.4 Examples of parallel programs

In this section, we will see some examples of parallel programs, and how they can be parallelized.

9.4.1 Single Program, Multiple Data (SPMD)

SPMD stands for Single Program, Multiple Data. This is the dominant parallel programming model. In this model, only a single source code is written. The code can have conditional execution based on which processor is executing the copy. All copies of the code are started simultaneously and communicate and synch with each other periodically.

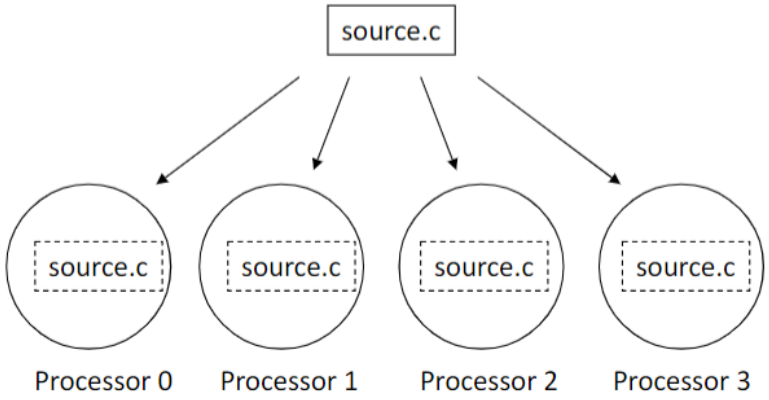

Figure 9.12: SPMD model

Note: a running program is called a **process**, and it is managed by the operating system.

9.4.2 Basics of data parallel programming

Let us suppose we have a simple program that runs on 2 CPUs. This program has an array of data to be operated by both. So, to do this, the array can be split into 2 parts, and each CPU can operate on its part. This is the basic idea of data parallel programming:

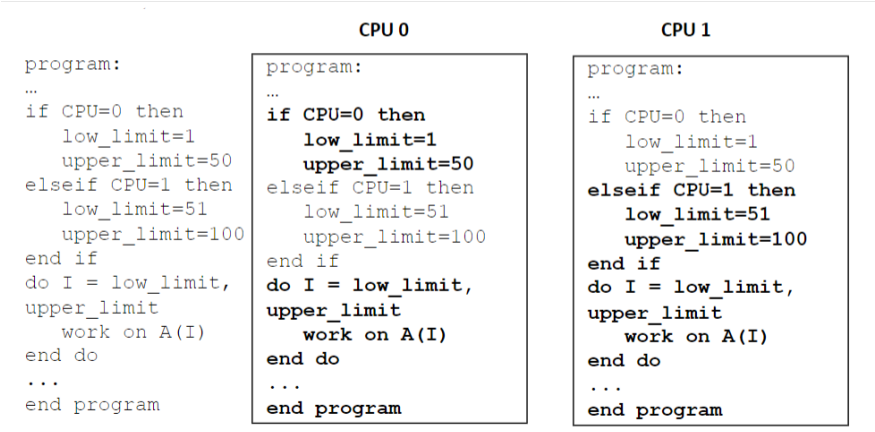

Figure 9.13: Data parallel programming

Accessing shared variables

If multiple processors want to write to a shared variable at the same time, there may be conflicts. For example, let us suppose we have 2 processors that want to write to the same variable at the same time. This can lead to the following behavior:

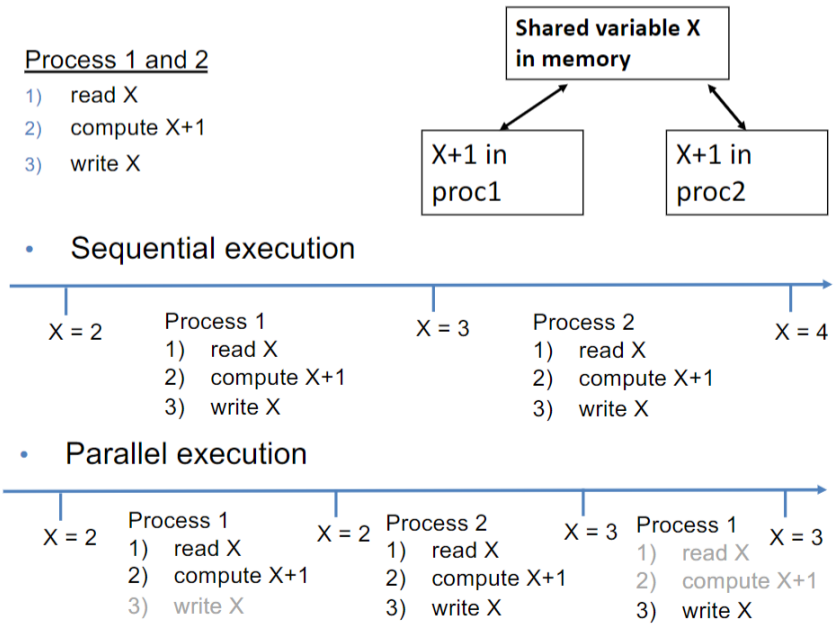

Figure 9.14: Accessing shared variables

This is known as a **race condition**. This happens when the application behavior depends on the sequence or timing of processes which should operate properly. Critical race conditions result un an invalid execution and bugs, as the example above.

To avoid this, we can use **locks**. A lock ensures mutual exclusive access to a shared resource. When a processor wants to access the shared resource, it must first acquire the lock. If the lock is already acquired, the processor must wait until the lock is released. As we can see, this is another source of overhead in parallel computing.

Note: beware of deadlocks! Deadlocks occur when two or more processors are waiting for each other to release a lock, resulting in neither of them being able to proceed. This can be avoided by using a timeout mechanism, for example.

Chapter 10

MPI: Message Passing Interface

10.1 Parallel programming with MPI

10.1.1 What is MPI?

MPI (Message Passing Interface) is a standard for parallel programming that defines a set of functions that allow the exchange of messages between processes. It is an **interface specification**, basically, a document stating the functionality that vendors should provide and users can rely upon.

The goal of MPI is to provide a portable, efficient, and flexible standard for message-passing programming. This standard defines both a Fortran and a C specification, and some modern languages like Python. There are many alternative implementations of MPI, but our focus will be on the **OpenMPI**.

10.1.2 Programming model

In MPI, all parallelism is **explicit**. Identifying what should be parallelized and how is the responsibility of the programmer. The programmer must explicitly create and manage processes, and explicitly send and receive messages.

MPI was designed for **dostributed memory architectures**, even if the implementations currently support any common parallel architecture. Hence, MPI virtually allows running your code on any parallel system.

In message-passing programs, a program running on one core is usually called a **process**. Two processes can communicate by calling functions:

- One process calls a **send** function.
- The other process calls a **receive** function.

MPI also supports **global** communication functions that can involve more than two processes. These functions are called **collective** functions.

10.2 Preeliminaries: passing parameters to C++ programs

It is possible to pass some values from the command line to `C/C++` programs when they are executed: **command line arguments**.

This is important when you want to control your program from outside instead of hardcoding those values inside the code. The command line arguments are handled using the `main()` function arguments:

- `argc` refers to the number of arguments passed to the program.
- `argv[]` is an array of pointers, which point to each argument passed to the program. Each argument is representes as a C `char[]`, hence `argv[]` type is `char*`.

Let's consider a simple example which checks if there is a single argument from the command line and takes action accordingly:

```
#include <iostream>

int main(int argc, char* argv[]) {
    if (argc == 2) {
        std::cout << "The argument is: " << argv[1] << std::endl;
    } else if (argc > 2) {
        std::cout << "Too many arguments" << std::endl;
    } else {
        std::cout << "One argument expected" << std::endl;
    }
}
```

Then, we compile the program and run it with different arguments:

```
$ g++ -o program program.cpp
$ ./program
One argument expected

$ ./program 10
The argument is: 10

$ ./program 10 20
Too many arguments
```

It should be noted that:

- The first argument is always the name of the program, corresponding to `argv[0]`.
- The arguments are always passed as strings, so you need to convert them to the desired type if needed.
- `*argv[argc - 1]` is the last argument passed to the program.

10.3 MPI basics

10.3.1 Hello, World!

The first program that is usually written in any programming language is the **Hello, World!** program. In MPI, the **Hello, World!** program is a bit more complex than in other languages, but it is still simple.

The following is the **Hello, World!** program in MPI:

```cpp
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::cout << "Hello, World! I am process " << rank << " of " << size << std::
        endl;

    MPI_Finalize();
}
```

Note: In parallel programming, it is common to identify each process with a non-negative integer, called the **rank**. The rank of a process is unique within a communicator, which is a group of processes that can communicate with each other. If there are **size** processes in a communicator, the ranks go from 0 to **size - 1**.

Then, we compile and run the program:

```
$ mpicxx -o hello hello.cpp

$ mpiexec -np 2 ./hello
Hello, World! I am process 0 of 2
Hello, World! I am process 1 of 2

$ mpiexec -np 4 ./hello
Hello, World! I am process 0 of 4
Hello, World! I am process 2 of 4
Hello, World! I am process 1 of 4
Hello, World! I am process 3 of 4
```

Notice that the output is not deterministic, as the order of the processes is not guaranteed. The only thing that is guaranteed is that the rank of each process is unique. How many processes are created is determined by the `-np` flag in the `mpiexec`.

On this example, we can see new functions that are used in MPI. We will explain them in the following sections.

10.3.2 `MPI_Init` and `MPI_Finalize`

The function `MPI_Init` initializes the MPI environment. It must be called before any other MPI function. It is responsible for allocate storage for message buffers and decide which process gets which rank.

The arguments for `MPI_Init` are as follows:

```
int MPI_Init(int* argc_p, char*** argv_p);
```

- `argc_p` is a pointer to the number of arguments passed to the program.
- `arg_p` is a pointer to the array of arguments passed to the program.

If you are not interested in the arguments, you can pass `nullptr` to `MPI_Init`. Like most MPI functions, `MPI_Init` returns an integer error code, and in most cases, we will ignore it.

The function `MPI_Finalize` is the opposite of `MPI_Init`. It is the last MPI function that should be called. It is responsible for cleaning up the MPI environment and releasing any resources that were allocated.

In general, no MPI function should be called after `MPI_Finalize`. The function `MPI_Finalize` has the following signature:

```
int MPI_Finalize();
```

10.4 Point-to-point communication

MPI processes can be addressed via **communicators**. A communicator is a collection of processes that send messages to each other.

The standard provides mechanisms for defining your own, but it general, you will use one of the predefined communicators. The most common communicator is `MPI_COMM_WORLD`, that colects all processes that were started with the same `mpiexec` command.

The function `MPI_Comm_rank` is used to get the rank of the calling process in the communicator. It has the following signature:

```
int MPI_Comm_rank(MPI_Comm comm, int* rank);
```

- `comm` is the communicator in which the rank is to be determined.
- `rank` is a pointer to the integer where the rank will be stored.

The function `MPI_Comm_size` is used to get the number of processes in the communicator. It has the following signature:

```
int MPI_Comm_size(MPI_Comm comm, int* size);
```

- `comm` is the communicator in which the size is to be determined.
- `size` is a pointer to the integer where the size will be stored.

Point-to-point communication means that we explicitly state which among the communicator's processes you want to communicate with. The most basic point-to-point communication functions are `MPI_Send` and `MPI_Recv`.

The function `MPI_Send` is used to send a message to another process. It has the following signature:

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm);
```

- `buf` is a pointer to the data to be sent.
- `count` is the number of elements to be sent.
- `datatype` is the type of the elements to be sent.
- `dest` is the rank of the destination process.
- `tag` is an integer that can be used to identify the message.
- `comm` is the communicator in which the message will be sent.

The function `MPI_Recv` is used to receive a message from another process. It has the following signature:

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status* status);
```

- `buf` is a pointer to the memory where the received data will be stored.
- `count` is the number of elements to be received.
- `datatype` is the type of the elements to be received.
- `source` is the rank of the source process.
- `tag` is an integer that can be used to identify the message.
- `comm` is the communicator in which the message will be received.
- `status` is a pointer to a `MPI_Status` object that will contain information about the received message.

Let us see an example of point-to-point communication: a sorted `Hello, World!` program.

```cpp
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    constexpr unsigned MAX_STRING = 18;
    std::ostringstream builder;
    builder << "Hello, World! I am process " << rank << " of " << size;
    std::string message = builder.str();

    if (rank > 0) {
        MPI_Send(&message[0], MAX_STRING, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
    else {
        std::cout << message << std::endl;
        for (int r = 1; r < size; ++i) {

            MPI_Recv(&message[0], MAX_STRING, MPI_CHAR, r, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
            std::cout << message << std::endl;
        }
    }

    MPI_Finalize();
}
```

In this example, the process with rank 0 prints its message first, and then waits for the other processes to send their messages. The other processes send their messages to the process with rank 0, which receives them and prints them.

This will print the following output:

```
$ mpiexec -np 4 ./hello
Hello, World! I am process 0 of 4
Hello, World! I am process 1 of 4
Hello, World! I am process 2 of 4
Hello, World! I am process 3 of 4
```

Notice that now, we get the messages in order, as the process with rank 0 is the one printing them.

10.4.1 MPI datatypes

MPI provides a set of predefined datatypes that can be used with the `MPI_Send` and `MPI_Recv` functions. The most common datatypes are:

MPI_CHAR	MPI_UNSIGNED_CHAR	MPI_FLOAT
MPI_SHORT	MPI_UNSIGNED_SHORT	MPI_DOUBLE
MPI_INT	MPI_UNSIGNED	MPI_LONG_DOUBLE
MPI_LONG	MPI_UNSIGNED_LONG	MPI_BYTE

Figure 10.1: MPI datatypes

10.4.2 Message matching

Imagine that we have a process **q** that sends a message, and a process **r** that receives it. The message can only be received by **r** if the following conditions are met:

- The source of the message is **q**.
- The tag of the message is the same as the tag of the receive function.
- The communicator of the message is the same as the communicator of the receive function.
- The datatype of the message is the same as the datatype of the receive function.
- The count of the message is less than or equal to the count of the receive function.

10.4.3 Non-overtaking messages

If a process **q** sends two messages to a process **r**, the messages will be received in the same order they were sent. This is called the **non-overtaking** property.

There is no restriction on the arrival of messages sent from different processes, meaning that messages sent from different processes can arrive in any order, even if they were sent in a specific order.

10.4.4 Deadlocks in MPI

A **deadlock** occur when processes block for communication, but their requests remain unmatched or otherwise unprocessed. Deadlocks can occur in MPI when two or more processes are waiting for each other to send or receive messages.

For example:

```
1 if (rank == 0) {
2     MPI_Send(&message[0], MAX_STRING, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
3     MPI_Recv(&message[0], MAX_STRING, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
4 }
5 else if (rank == 1) {
```

```
6       MPI_Send(&message[0], MAX_STRING, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
7       MPI_Recv(&message[0], MAX_STRING, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
8   }
```

There are two processes, 0 and 1. Process 0 sends a message to process 1, and then waits for a message from process 1. Process 1 sends a message to process 0, and then waits for a message from process 0. This will cause a deadlock, as both processes are waiting for each other to send a message.

There are 2 approaches to avoid this:

- Either you smartly rearrange the order of the sends and receives.
- Or use non-blocking communication functions (advanced topic).

10.4.5 Process Hang

If a process tries to receive a message and there's **no matching send**, the the process will **block forever**. When you design your MPI programs, you must ensure that there is a matching send for every receive.

Be careful that there are no inadvertent mistakes in calls to `MPI_Send` and `MPI_Recv`. If the tags don't match, or if the rank of the destination process is the same as the rank of the source process, the receive won't match the send. This will make that, either the program hangs, or the receive is matched with a different send.

10.4.6 MPI input and output

MPI does not specify which processes have acces to which I/O devices. Virtually all implementations allow all processes in `MPI_COMM_WORLD` to access the standard output and standard error streams, but notice that the output from different processes can be interleaved.

To have "sorted" outpur, the common practice is to have the process with rank 0 print the output, sent by the other processes. This is what we did in the sorted **Hello, World!** example.

Unlike output, most MPI implementations do not allow all processes to access the standard input stream. Only the process with rank 0 can access the standard input stream.

The common practice is to have the process with rank 0 read the input, and then it broadcasts, or scatters, the input to the other processes. For this, we need to use the **collective communication** functions.

10.5 Collective communication

The collective routines are used to communicate data between all processes in a communicator. They involve every process in the communicator, are only blocking routines, and transmit only predefined MPI datatypes. Also, we cannot use tags to identify messages.

Note: Be sure that every process in the communicator calls the **same** collective function, with the **same** arguments, to avoid deadlocks.

The most common collective communication functions are:

- `MPI_Bcast`: Broadcasts a message from the process with rank 0 to all other processes in the communicator.
- `MPI_Scatter`: Splits an array into equal-sized chunks and sends each chunk to a different process.
- `MPI_Gather`: Gathers data from all processes in the communicator and stores it in an array in the process with rank 0.
- `MPI_Reduce`: Reduces data from all processes in the communicator to a single value in the process with rank 0.

10.5.1 Broadcast

The function `MPI_Bcast` broadcasts a message from the process with rank 0 to all other processes in the communicator. It has the following signature:

```
int MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm
    );
```

- `buf` is a pointer to the data to be broadcast.
- `count` is the number of elements to be broadcast.
- `datatype` is the type of the elements to be broadcast.
- `root` is the rank of the process that broadcasts the message.
- `comm` is the communicator in which the message will be broadcast.

Let us see an example of the `MPI_Bcast` function:

```
std::string message;
if (rank == 0) {
    message = "Hello, World!";
}

MPI_Bcast(&message[0], message.size(), MPI_CHAR, 0, MPI_COMM_WORLD);

std::cout << "Process " << rank << " received the message: " << message << std::
    endl;
```

This program will print the following output:

```
$ mpiexec -np 4 ./hello
Process 0 received the message: Hello, World!
Process 1 received the message: Hello, World!
Process 2 received the message: Hello, World!
Process 3 received the message: Hello, World!
```

Notice that, when we are on a receiving process (a process with rank > 0), we don't need to initialize the message variable, as it will be overwritten by the `MPI_Bcast` function.

10.5.2 Reduce

The function `MPI_Reduce` reduces data from all processes in the communicator to a single value in the process with rank 0. It has the following signature:

```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, int dest, MPI_Comm comm);
```

- `sendbuf` is a pointer to the data to be sent.
- `recvbuf` is a pointer to the memory where the reduced data will be stored.
- `count` is the number of elements to be reduced.
- `datatype` is the type of the elements to be reduced.
- `op` is the operation to be performed.
- `dest` is the rank of the process that will receive the reduced data.
- `comm` is the communicator in which the data will be reduced.

This is specially useful when you want to compute the sum of all elements in an array, or the maximum value of an array, for example. Instead of having the root process compute the result after receiving all the data, you can use `MPI_Reduce` to evenly distribute the work among all processes.

Let us see an example of the `MPI_Reduce` function:

```
int number = rank + 1;
int sum;

MPI_Reduce(&number, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
    std::cout << "The sum of the numbers is: " << sum << std::endl;
}
```

This program will print the following output:

```
$ mpiexec -np 4 ./hello
The sum of the numbers is: 10
```

Notice that this program computes the sum of the numbers from 1 to 4, as each process contributes with its rank + 1. The result is stored in the `sum` variable of the process with rank 0. The following diagram explains the distribution of the work:

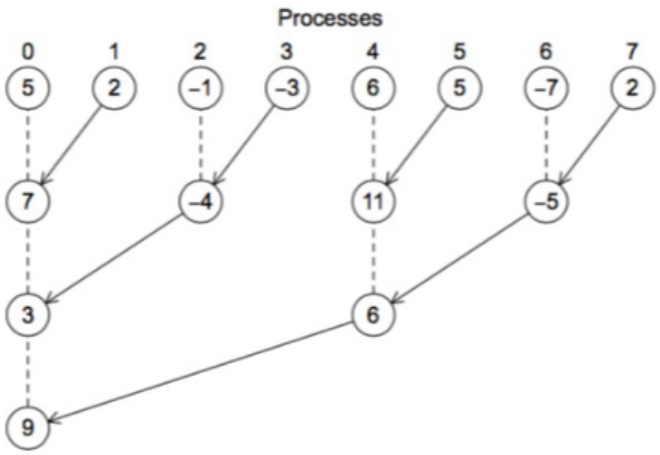

Figure 10.2: MPI reduce for summing numbers

The operations that can be performed with `MPI_Reduce` are the following:

MPI_MAX	MPI_LAND	MPI_LXOR
MPI_MIN	MPI_BAND	MPI_BXOR
MPI_SUM	MPI_LOR	MPI_MAXLOC
MPI_PROD	MPI_BOR	MPI_MINLOC

We are interested in only on these ones

Figure 10.3: MPI operations

Note that, by using a count argument greater than 1, we can **reduce arrays** instead of single values. The following code could thus be used to add a collection of N-dimensional vectors:

```
std::vector<double> local_x (N), sum (N);
// Fill local_x with data (partial computations)
MPI_Reduce(local_x.data(), sum.data(), N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Also, in some cases we would want for all the processes to receive the result of the reduction. This can be done by using the `MPI_Allreduce` function, which is similar to `MPI_Reduce`, but the result is broadcasted to all processes in the communicator. The signature of `MPI_Allreduce` is the same as `MPI_Reduce`, but the `dest` argument is not needed:

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm);
```

Collective comm "IN-PLACE"

In some cases, you may want to use the same buffer for both the send and receive buffers. This is called **in-place** communication.

MPI provides the special placeholder `MPI_IN_PLACE` to enable the use of a single buffer for both the send and receive buffers. For example:

```
int number = rank + 1;
MPI_Reduce(MPI_IN_PLACE, &number, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

10.5.3 Scatter

The function `MPI_Scatter` splits an array into equal-sized chunks and sends each chunk to a different process. It has the following signature:

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
    recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- `sendbuf` is a pointer to the data to be sent.
- `sendcount` is the number of elements to be sent to each process.
- `sendtype` is the type of the elements to be sent.
- `recvbuf` is a pointer to the memory where the received data will be stored.
- `recvcount` is the number of elements to be received.
- `recvtype` is the type of the elements to be received.
- `root` is the rank of the process that scatters the data.
- `comm` is the communicator in which the data will be scattered.

The `MPI_Scatter` function implements a form of data distribution called **block partition**. The data is divided into blocks of equal size, and each block is sent to a different process.

We can also have a **cyclic partition**, where the data is divided into blocks of equal size, but the blocks are sent to the processes in a cyclic order. This implementation is pretty straightforward, as we can see in the following example:

```
for (size_t i=rank; i < message.size(); i += size) {
    // Do something with message[i]
}
```

Process	Cyclic			
0	0	3	6	9
1	1	4	7	10
2	2	5	8	11

Process	Block			
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Figure 10.4: Block vs cyclic partition

For example, we can implement a function that reads a vector of integers from the standard input, and send block partitions of the vector to different processes:

```
std::vector<int> read_vector(
    unsigned n, std::string const &name, MPI_Comm const &comm
)
{
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    const unsigned local_n = n / size;
    std::vector<int> result(local_n);

    if (rank == 0) {
        std::vector<int> input(n);
        std::cout << "Enter " << name << "\n";
        for (int &e : input) {
            std::cin >> e;
        }
        MPI_Scatter(input.data(), local_n, MPI_INT, result.data(),
                    local_n, MPI_INT, 0 comm);
    }
    else {
        MPI_Scatter(nullptr, local_n, MPI_INT, result.data(),
                    local_n, MPI_INT, 0, comm);
    }
    return result;
}
```

We can use this function to implement parallel sum of vectors, for example.

10.5.4 Gather

The `MPI_Gather` function joins portions of data from all the processes, storing them all in a buffer. It has the following signature:

```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
    recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- `sendbuf` is a pointer to the data to be sent.
- `sendcount` is the number of elements to be sent by each process.
- `sendtype` is the type of the elements to be sent.
- `recvbuf` is a pointer to the memory where the received data will be stored.
- `recvcount` is the number of elements to be received by the root process.
- `recvtype` is the type of the elements to be received.
- `root` is the rank of the process that gathers the data.
- `comm` is the communicator in which the data will be gathered.

We can use the `MPI_Gather` function to gather and print the results of the parallel sum of vectors that we implemented with the `MPI_Scatter` function:

```
void print_vector(
    vector<int> const &local_v, unsigned n,
    string const &title, MPI_Comm const &comm
)
{
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    const unsigned local_n = local_v.size();

    if (rank > 0) {
        MPI_Gather(local_v.data(), local_n, MPI_INT,
                   nullptr, local_n, MPI_INT, 0, comm);
    }
    else {
        vector<int> global_v(n);
        MPI_Gather(local_v.data(), local_n, MPI_INT,
                   global_v.data(), local_n, MPI_INT, 0, comm);
        cout << title << "\n";
        for (int e : global_v) {
            cout << e << " ";
        }
        cout << endl;
    }
}
```

Notice that sometimes, we will need to distribut the gather result on all processes. This can automatically be done with the `MPI_Allgather` function, which is similar to `MPI_Gather`, but the result is broadcasted to all processes in the communicator. The signature of `MPI_Allgather` is the same as `MPI_Gather`, but the `root` argument is not needed:

```
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
    recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```

10.5.5 Final remarks: collective communication

We have some final remarks about collective communication:

- All the processes in the communicator must call the same collective function. For example, if a program attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process, it is erroneous and probably will hang or crash.
- The arguments passed by each process to an MPI collective communication must be "compatible". For example, if one process passes in 0 as the destination process and another passes in 1, the the outcome of a call to, e.g., `MPI_Reduce` is erroneous and the program will probably hang or crash.
- The `recvbuf` argument is only used on `dest` process. However, all of the processes still need to pass in an actual argument corresponding to `recvbuf`, even if it is just `nullptr`.
- Point-to-point communications are matched based on tags and communicators. Collective communications don't use tags, so they're mathced solely based on communicator and the order in which they're called.