Mikul Fikul

Notes on the cuffs

Jan Rluman
Publishing House

Preface

Let's say it plainly - this book is for no one. For those who already know C++ well, it is just a bunch of idiotic examples. On the other hand, for those who don't know it yet, this random set of pages is also a bunch of idiotic examples. I don't expect anyone to want to read it, but there is a small chance that after some rough party with your collegues in a dormitory, you might start exploring GitHub, having nothing better to do at 2AM, and stumble upon it accidentaly. So this is my explanation for you, my lost friend.

I mainly wrote it for myself as a collection of things that I learned - either because I found them interesting or I couldn't remember them. These sweet notes aren't mean to learn you C++; they don't explain the basics or cover super advanced topics. They are just compilation of well-known features but stored in one place. All the code and text were written by me or taken from the books listed in the bibliography. There is also a few real-life cases that I encountered in my daily work.

Anyway, if you decided to read this book I don't have much to say - good luck. By the way, in case when you notice a mistake, silly code example or something else (and I think you notice, there are plenty of them) I would be very grateful when you leave the comment. This is one of the reasons I made my notes public. For the rest of you, who have ended your journey with this sentence - good choice.

The LaTeX code for this document is in the repository. In the future I'm going to add chapters devoted to C++20 and C++23 and fill the missing text parts in *Concurrency* one.

Mikul

Contents

Chapter 1

C++11

1.1 Basics

1.1.1 Preprocessor

`#define` macro

When we need to define a very long directive it is possible to break the defined name with \.

```
#define TOTAL_SUM_OF_NUMBERS 12
#define TOTAL_AMOUNT_OF_NUMBERS 4

#define AVG \
(TOTAL_SUM_OF_NUMBERS/TOTAL_AM\
OUNT_OF_NUMBERS)
```

`#undef` macro

After `#undef name` compiler forgets about the `name`

```
#include <iostream>

#define VAR 1
void print()
{
    /* It is known here */
    std::cout << VAR << std::endl;
}
#undef VAR

int main()
{
    /* Prints "1" */
    print();
    /* Won't compile - VAR isn't known */
    // std::cout << VAR << std::endl;
}
```

`##` macro

It is an interesting operator which allows one to „glue” two names.

```
#define JOIN(var1,var2) var1##_##var2

int JOIN(variable,1) = 1;

int main()
{
    /* Prints "1" */
    std::cout << variable_1 << std::endl;

    return 0;
}
```

`#name` macro

To print the name of a parameter, we can use the `#name` operator. It can be handy during the debugging process

```
#include <iostream>

#define DEBUG(var) std::cout << #var << std::endl

int main()
{
    /* Observe that x hasn't been defined! */
    /* Prints "x" */
    DEBUG(x);
    return 0;
}
```

`#error` *text* macro

When a compiler meets that directive, it immediately will stop execution and print the *text*.

```
#define VAR 2

#if (VAR == 1)
    #define COMPILATION_PENDING 1
#endif

#ifdef COMPILATION_PENDING
    #define COMPILATION_DONE 1
#else
    /* This will appear in a terminal */
    #error "Compilation corrupted!"
#endif
```

`#include` macro

The only missing thing to explain is the difference between `#include<...>` and `#include "..."`. The former looks for suitable files in standard directory like */usr/lib*; the latter will focus on given path.

1.1.2 Four types of initialization

In C++11 we have four ways to initialize the variable

```
int value = 1;
int value (1);
int value {1};
int value = {1};
```

The last though is a bit different than the others and should be preferred. There are at least a few strong reasons why:

1. **Prevents Narrowing Conversions**
   Brace initialization enforces strict type safety by preventing implicit narrowing conversions (e.g., converting a `double` to an `int` or a `long` to a `float`).

   ```
   int a = 3.14;        // Allowed: Truncates to 3
   int b{3.14};         // Error: Narrowing conversion is not allowed
   ```

   This helps catch potential bugs at compile time, ensuring that only safe conversions are performed.

2. **Works for All Types (Consistency)**
   Brace initialization works for:

   - Primitive types
   - User-defined types (classes, structs)
   - Arrays and STL containers (e.g., `std::vector`, `std::initializer_list`)

   ```
   /* Primitive type */
   int x{42};

   /* Object */
   std::string s{"Hello, world"};
   std::vector<int> v{1, 2, 3};
   ```

   This uniformity makes it easier to use and read compared to mixing `=` and `()` initialization styles.

3. **Avoids the Most Vexing Parse**
   Brace initialization avoids the **most vexing parse**, a situation where the compiler interprets an initialization as a function declaration instead of an object definition.

   ```
   /* Creates a vector of 10 elements, each initialized to 20 */
   std::vector<int> v1(10, 20);

   /* Could declare a function (!) named v2 that returns vector<int> */
   std::vector<int> v2();

   /* All these problems are easily resolved by the braces */
   std::vector<int> v3{10, 20};
   std::vector<int> v4{};
   ```

4. **Supports Initialization Lists**
   With {}, we can initialize objects using an **initializer list**. This is especially useful for collections or objects with multiple fields.

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

/* Traditional */
Point p1 = {1, 2};

/* Uniform initialization */
Point p2{1, 2};
```

5. **Better Default Initialization**
   Brace initialization ensures that objects are **value-initialized** (initialized to their default values) instead of being left **uninitialized** or **undefined**.

   ```cpp
   /* Uninitialized - could contain garbage value */
   int x;

   /* Value initialized by 0 */
   int y{};
   ```

   For classes and structs, this behavior ensures that all fields are initialized correctly.

For these reasons, {} is often the best and most robust choice for initialization in C++.

1.1.3 Literal type

Literal types are the types of `constexpr` variables and they can be constructed, manipulated, and returned from `constexpr` functions. They need to fulfill a few requirements so they need to be one of

- possibly cv-qualified `void` (so that `constexpr` functions can return `void`) (since C++14),
- scalar type,
- reference type,
- an array of literal types,
- possibly cv-qualified class type that has all of the following properties:
  - has a trivial (until C++20) or `constexpr` (since C++20) destructor,
  - being one of
    - a closure type (since C++17),
    - an aggregate union type that:
      - has no variant members,
      - has at least one variant member of the non-`volatile` literal type.
    - a non-union aggregate type, and each of its anonymous union members:
      - has no variant members, or
      - has at least one variant member of non-`volatile` literal type.
    - a type with at least one `constexpr` (possibly template) constructor that is not a copy or the move constructor.

1.1.4 Number conversion algorithm

1. If either operand is type `long double`, the other operand is converted to `long double`.
2. Otherwise, if either operand is type `double`, the other operand is converted to `double`.
3. Otherwise, if either operand is type `float`, the other operand is converted to `float`.
4. Otherwise both operands are integrals. If both are signed or unsigned, and one is a lower rank than the other, it is converted to a higher rank.
5. Otherwise, one operand is signed and one is unsigned. If the unsigned operand is of higher rank the latter is converted to the type of the unsigned operand.
6. Otherwise, if the signed type can represent all values of the unsigned type, the unsigned operand is converted to the type of the signed type.
7. Otherwise, both operands are converted to the unsigned version of the signed type.

1.1.5 Simple definition of lvalue and rvalue

If an object has its name it's lvalue; otherwise, it's rvalue.

1.1.6 `register` and `volatile`

The **register** variable is similar to automatic variables and exists inside a particular function only. It is supposed to be faster than the local variables. If a program encounters a register variable, it stores the variable in the processor's register rather than memory if available.

The **volatile** variable warns a compiler that the object of that type can change its value without the compiler's knowledge so that the compiler shouldn't rely on *cache memory* (objects used often can be held in RAM), but it shall take its value directly from the memory where the variable is defined.

1.1.7 Comma operator

In C++11 comma operator **,** ensures an order of expression evaluation.

```
#include <iostream>
int main()
{
    int i = 10;
    int j = (i *= 2, i * 20);
    /* Prints "400" */
    std::cout << j << std::endl;

    return 0;
}
```

1.1.8 `constexpr` and `const`

`constexpr`	`const`
The `constexpr` value is computed during compilation;	The const value is computed during program execution.
The `constexpr` value can be used in switch/case statements;	Not any const value can be.
The `constexpr` value is safe for multithreading since the value is prepared before program execution;	The const value may not be fully constructed when another thread needs it.

1.1.9 Storage duration, scope and linkage

All information needed is included in the following tables.

Variable

Type	Declaration	Scope	Storage Duration	Linkage
Local variable	`int x`	Block	Automatic	None
Static local variable	`static int s_x`	Block	Static	None
Dynamic local variable	`int* x { new int{} }`	Block	Dynamic	None
Function parameter	`void foo(int x);`	Block	Automatic	None
External non-constant global variable	`int g_x`	Global	Static	External
Internal non-constant global variable	`static int g_x`	Global	Static	Internal
Internal constant global variable	`constexpr int g_x {1}`	Global	Static	Internal
External constant global variable	`extern const int g_x {1}`	Global	Static	External
Inline constant global variable (C++17)	`inline constexpr int g_x {1}`	Global	Static	External

Function

Forward declaration type	Example	Notes
Function forward declaration	`void foo(int x)`	Prototype only, no function body
Non-`const` variable forward declaration	`extern int g_x`	Must be uninitialized
const variable forward declaration	`extern const int g_x`	Must be uninitialized
The `constexpr` variable forward declaration	`extern constexpr int g_x`	Not allowed, `constexpr` cannot be forward declared

Thus if we define an object market `static` inside a function body, this object will keep its value until the next function call. Static objects are created in the same part of memory as global objects and are zero-initialized by default.

1.1.10 Enums

Usually, the base type of enumeration is `int` but there are no contradictions to not do it with any other integral type

```
enum class CharEnum : char
{
    Char1,
    Char2,
    Char3
};
```

Each enumeration has its **range**, and we can assign any integer value in the range, even if it is not an enumeration value, by using a type cast to the enumeration variable. For instance

```
enum Numbers { nfive = -5, two = 2, four = 4, nine = 9};
Numbers number { static_cast<Numbers>(6) };
```

Any enumeration variable doesn't represent 6 but belongs to the range so the assignment is valid.

To calculate the enumeration range we take the largest enumeration value and find the smallest power of 2 greater than this value and subtract it by one. In our case, we have $9 < 2^4 - 1$, so 15 is the upper end of the range.

Next, to find the lower limit, we find the smallest evaluation value. If it is non-negative, the lower limit of the range is 0; otherwise is negative and we do the same thing as for the upper end of the range but with a minus sign. In our case, we have $-(2^3 - 1) < -5$ so the lower of the range is -7.

1.1.11 Unions

Some basic information can be found in cppreference. One missing part is that unions can have `private` and `public` members, but not `protected` as `union` does not support inheritance.

1.2 Pointers and arrays

It is platform-dependent which integral type for pointers is used. For instance, one might have a platform for which type `int` is a 2-byte value and addresses are 4-byte values.

1.2.1 Array name and first element pointer

An array index starts from 0 since it is not a real index. It just provides information on how many object widths we need to move to get the required one. Sometimes there is a difference between an array name and the first element address.

```
#include <iostream>

template <typename T>
std::size_t size(const T* ptr)
{
    return sizeof(ptr);
}

int main()
{
    const char char_arr[3] { 'a', 'b', 'c' };

    /* Prints "Char array size: 3 */
    std::cout << "Char array size: " << sizeof(char_arr) << std::endl;

    /* Notice that this number is bigger since it is stored as int type! */
    /* Prints "Pointer size: 8" */
    std::cout << size(char_arr) << std::endl;

    const int int_arr[3] { 1, 2, 3 };

    /* Prints "Int array size: 12 */
    std::cout << "Int array size: " << sizeof(int_arr) << std::endl;
```

```
    /* Prints "Pointer size: 8" */
    std::cout << size(int_arr) << std::endl;

    return 0;
}
```

1.2.2 C-ctrings

The `C-string` is defined as a C-array with chars. It has some corner cases presented on the program below.

```
#include <iostream>

char arr1[80] {"str"};
char arr2[80] {'s', 't', 'r'};
char arr3[] {"str"};
char arr4[] {'s', 't', 'r'};
char arr5[] { '%' };

int main()
{
    /* Prints "str80" */
    std::cout << arr1 << ":" << sizeof(arr1) << std::endl;

    /* Prints "str80" */
    std::cout << arr2 << ":" << sizeof(arr2) << std::endl;

    /* Prints "str4" */
    std::cout << arr3 << ":" << sizeof(arr3) << std::endl;

    /* Maybe prints "str%:3 or something other */
    std::cout << arr4 << ":" << sizeof(arr4) << std::endl;
}
```

Let's discuss the code

- `arr1` is defined as the array of chars with 80 elements. We've used parenthesis initialization, so it's quite obvious that all missing elements have been zero-initialized.
- `arr2` is defined differently but also with the initialization list, so that is the same case as before.
- `arr3` hasn't used the explicit number of elements, thus the compiler calculated them itself. We've used `str` and there is a hidden `\0` element, that's why we've got 4 elements.
- `arr4` is the most important case. Here the compiler calculated just three elements, without ending `\0`, so when we print it, it won't stop until it finds a random `\0` in the memory. The `%` sign may appear on the screen (the content of `arr5`) if only the compiler puts these two arrays one after another.

1.2.3 Pointers to multidimensional arrays

Let's consider two-dimensional array `int array2D[4][16]`. This is a table of 4 elements, each of which is a table of 16 elements which means that `array2D` is the

pointer to the first table of 16 elements, `array2D + 1` is the pointer to the second table of 16 elements, etc. Let's assume we need to get the value of `array2D[1][10]`. To achieve that we need to move to the second element of `array2D` (array of 16 elements) and then move ten steps to the right in that element. In other words, the whole movement was $(1 * 16) + 10$. To generalize it - we saw that getting `array2D[1][10]` required (1 * 16) + 10 movements of `sizeof(int)` size so it can be deduced that the general equation to achieve the element `arr2D[i][j]` is

$$step = (i * 10) + j.$$

There is no information about the number of rows ([4]) which is redundant for the compiler. It is because a two-dimensional array in the memory is actually flat - the arrangement of elements in memory is not relevant, the way we move on that memory plays the main role.

The example above provides the reason why it is redundant to add an array's row number to the array pointer. Therefore these two ways of passing a pointer to a 2D array are equivalent

```
void read(int ptr[][16]);
/* 4 is redundant */
void read(int ptr[4][16]);
```

It works in the same way for any multidimensional array. It will be easy to remember that we just drop the first „dimension"

```
/* We drop [1] */
int (*ptr)[2][3][4] = new int [1][2][3][4];
delete[] ptr;
```

1.2.4 Pointer with `mutable` or `const`

There is one possible way that `mutable` and `const` can be used together with a pointer.

```
struct Data
{
    /* Won't compile */
    // mutable const int i;

    /* Won't compile */
    // mutable int* const ptr;

    /* Okay */
    mutable const int* ptr;
}
```

The last expression is possible in use since `mutable` is related to the pointer which can be modified (only the pointed value is `const`).

1.2.5 `auto` and reference

The `auto` keyword can introduce some problems in the code when used in the wrong way. The main issue is that `auto` „loses" information whether an object is `const` or `volatile` and information about referenceness of it.

```
const int i = 1;

/* j is the copy of i, so there is no information
 * whether j is const or volatile!
 */
auto j = i;

double x = 1.0;
double& x_ref = x;

/* y is a copy of x_ref, so its type is double, not double& */
auto y = x_ref;
```

But if we use `&` after `auto` keyword, compiler will add `const` or `volatile` to it. This shouldn't be surprise as changing `const` object would be a serious violation.

```
const float& f = 2.0;
auto& f_ref = f;

/* Compilation error, the compiler added const implicitly */
f_ref++;
```

1.2.6 `auto` and pointers

Let's start with the observation that `auto` recognizes whether an object is a pointer or not

```
int i = 0;
/* auto knows that &i is the address, so the type of ptr is int* */
auto ptr = &i;
```

To recall, we have three ways of `const` modifier usage with pointers

```
int i = 0;

/* ptr1 is the pointer of int type to const int */
const int* ptr1 = &i;

/* We cannot modify the value */
// *ptr1 = 1;

/* But we can move the pointer */
ptr1++;

/* ptr2 is the const pointer of int type to int (we cannot move it) */
int* const ptr2 = &i;

/* We can modify the value */
*ptr2 = 1;  // can modify value

/* But we cannot move the pointer */
// ptr2++;

/* ptr3 is the const pointer of type int to const int */
const int* const ptr3 = &i;

/* We neither can modify the value nor move the pointer */
```

```
// *ptr3 = 1;
// ptr3++;
```

Having the code above we can draw some conclusions

- The `const` qualifier is related to the **nearest type which stands on its right side** (so `const int* ptr1 = &i` means that `const` is related to `int` **but not `int*`!**)
- If there is no type on the right side, it is related to the **nearest type which stands on its left side** (so `int* const ptr2 = &i` means that `const` is related to `int*`)

That little difference has a huge impact on `auto`. Assume that

```
int i = 0;
```

and let's analyze the second example being simpler.

According to the rules observed above the expression

```
auto const ptr = &i;
```

means [*int**] `const ptr` where the type deduced is *int** thus we get **`auto =`** *int** **`const`** which is expected behavior.

Now let's have a look at the first case. According to the rules

```
const auto ptr = &i;
```

means `const` [*int**] `ptr` but knowing that the `const` is related to the **entire type** *int** we receive **`auto =`** *int** **`const`** which is valid expression but not expected by us.

The final example is the worst of all - it is a mix of these two - the below expression has the form of `const` [*int**] `const ptr`

```
const auto const ptr = &i;
```

so a compiler deduces that **`auto =`** *int** **`const`** but observe that there is one another `const` after that, so the code above is interpreted as

```
int* const const ptr = &i;
```

and this, of course, produces the compilation error.

The fix for this problem is quite simple - **we need to inform the compiler that the required type is the pointer type, so just add asterisk after `auto`, thus `auto*`.**

1.3 Functions

1.3.1 Inline functions

To make function `inline` we need to fulfill at least one of the conditions given below

- Declaration or definition (not both!) includes `inline` keyword.
- Function is short (one, two lines) and non-recursive.
- Is defined within a class body.

1.3.2 Default function's arguments

As we know, C++ allows the default values of arguments. Here we discuss some restrictions imposed on functions with default arguments.

- A compiler will raise an error if it meets the default argument defined twice, so the following implementation won't compile.

```
/* File "declaration.h */
#pragma once

void function(int value = 0);


/* File "main.cpp */
#include "declaration.h"

/* error: default argument given for parameter 1 of
 * 'void function(int)' [-fpermissive]
 */
void function(int value = 0) {}

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

- There is an opportunity to repeat the declaration with „floating" default argument

```
#include <iostream>

void print(int a, int b, int c, int d = 3);
void print(int a, int b, int c = 2, int d);
void print(int a, int b = 1, int c, int d);
void print(int a = 0, int b, int c, int d);

void print(int a, int b, int c, int d)
{
    std::cout << a << ":" << b << ":"
              << c << ":" << d << std::endl;
}

int main()
{
    /* Prints "0:1:2:3" */
    print();

    /* Prints "5:1:2:3" */
    print(5);

    /* Prints "5:6:2:3 */
    print(5, 6);

    /* Prints "5:6:7:3 */
    print(5, 6, 7);

    /* Prints "5:6:7:8 */
```

```
    print(5, 6, 7, 8);
}
```

Observe that we cannot swap any two print declarations, for instance

```
void print(int a, int b, int c = 2, int d);
void print(int a, int b, int c, int d = 3);
```

will produce an error, since `d` is supposed to have a default value if `c` has.

- Function declaration can appear even in the other function body. In such cases, it hides the main declaration.

```
#include <iostream>

void print(int a = 0, int b = 1)
{
    std::cout << a << ":" << b << std::endl;
}

int main()
{
    /* Internal declaration! */
    void print(int a, int b = 2);

    /* Won't work - the declaration above hides the main one */
    // print();

    /* Prints "0:2" */
    print(0);

    return 0;
}
```

- The default argument can be calculated by some other function

```
#include <iostream>

int get_arg(int i)
{
    return i % 2;
}

void print(int a = 0, int b = get_arg(1))
{
    std::cout << a << ":" << b << std::endl;
}

int main()
{
    /* prints "0:1" (/
    print();

    return 0;
}
```

1.3.3 General view on function overload resolution

Overload resolution includes thee main steps

1. Assemble a list of candidate functions. These are functions or template functions that have the same names as the called function.
2. From the candidate functions, assemble a list of viable functions. These are functions with the correct number of arguments and for which there is an implicit conversion sequence, which includes the case of an exact match for each type of actual argument to the type of corresponding formal argument. for example, a function call with a type `float` argument could have that value converted to a `double` formal parameter, and a template could generate an instantiation for a `float`.
3. Determine whether there is a best viable function. If so, you use that function; otherwise, the function call is an error. There is a ranking from the best to the worst matching
   - Exact match, with regular functions outranking templates.
   - Conversion by promotion (for instance from `char` to `int`).
   - Conversion by standard conversion (for instance `long` to `double`).
   - User-defined conversions, such as those defined in class declarations.

To understand it better, consider the following example

```
void function(int);                  // #1
float function(float, float = 3);    // #2
void function(char);                 // #3
char* function(const char*);         // #4
char function(const char&);          // #5

template <typename T>
void function(const T&);             // #6

template <typename T>
void function(T*);                   // #7

int main()
{
    function('A');
    return 0;
}
```

Let's do it step by step:

1. There is nothing to analyze in the first step. All functions above have proper names, so all of them are treated as candidates.
2. Notice at first, that functions `#4` and `#7` are not viable because an integral type cannot be implicitly converted to a pointer type.
3. Now we have five functions to consider. `#1` is better than `#2` because `char` to `int` is promotion and `char` to `float` is conversion. Functions `#3`, `#5`, and `#6` are better than `#1` because they are exact matches but `#3` and `#5` are better than #6 because they are not templates. So now we have best-two matching functions: `#3` and `#5` which is an error of course.

1.3.4 Function mangling and C library

C++ mangles a method by emitting the function name, followed by `___` , followed by encodings of any method qualifiers (such as `const`), followed by the mangling of the method's class, followed by the mangling of the parameters, in order. For example `Class::function(int, long) const` is mangled as `function__C3Classil`. This approach, however, leads to problems with importing library functions from the C language which does not support mangling.

Assume we want to import the function `int pow(int value, int exponent)` from some C library in the C++ file. During the compilation, C++ mangles it, and we get something like `pow__ii`, but there is no `int pow__ii(int value, int exponent)` function to import. Due to that C++ provides `extern "C"` command which disables mangling.

```
extern "C" pow(int value, int exponent);
```

In case of more than one function declaration, we can use the extended version of it

```
extern "C"
{
    int pow(int value, int exponent);
    double pow(double a, int exponent);
}
```

or even include the whole C header file

```
extern "C"
{
    #include "clib.h"
}
```

1.3.5 A `constexpr` function

A `constexpr` function can be called during the compilation phase. In C++11 they are subject to tough restrictions:

- a compiler must have the full definition when calling it (the same requirement applies to inline functions),
- each argument of `constexpr` function has to be `constexpr` (`const` value or reference to `constexpr` value),
- a result of a `constepxr` function has to be `constexpr`,
- a body of `constexpr` function can have just one statement - `return` (thus we cannot have variables defined inside),
- a body of `constexpr` function can't have any loops, conditional, or switch statements except the ternary operator,
- a body of `constexpr` function can have empty instructions, simple declarations, and `using` directives,
- a `constexpr` function can use global or static variables if only if it doesn't modify them.

In other words, a `constexpr` function can't have any side effects, so it is *pure functional*. The only possible changes can affect its internal (auxiliary) variables.

```
constexpr double miles_to_km(double miles)
{
    return 1.609344 * mile;
}

int main()
{
    /* Result "11.265408" is visible in the IDE */
    constexpr double value = miles_to_km(7);
}
```

Remember that `constexpr` function will be used by the compiler during the compilation phase if only if an argument will be

- explicit constant, for instance, 1, `str`, etc,
- or `constexpr` object,
- or reference to `constexpr` object.

If none of these conditions is fulfilled a compiler will call this function at the runtime.

1.3.6 The `noexcept` specifier and operator

The `noexcept` specifier informs a compiler that the function doesn't throw an exception.

```
/* No guarantee - it can either throw or not */
void function();

/* Doesn't throw */
void function2() noexcept;

/* Doesn't throw */
void function3() noexcept(true);

/* No guarantee - it can either throw or not */
void function4() noexcept(false);
```

We also have `noexcept` operator mostly used to check whether a function throws or not. It is important that **`noexcept` doesn't invoke a function so only declaration is needed**.

```
#include <iostream>

void function1(double) noexcept(true);
void function2(double);

int main()
{
    constexpr bool value1 = noexcept (function1(1.0));

    /* Prints "1" so true */
    std::cout << value1 << std::endl;
```

```
    constexpr bool value2 = noexcept (function2(1.0));

    /* Prints "0" so false */
    std::cout << value2 << std::endl;
}
```

1.3.7 The `main()` function

It is the only function that doesn't need to have a return clause - if it doesn't have it, a compiler will write it by itself in the most simple way

```
int main()
{
    /* compiler will add "return 0;" here */
}
```

1.3.8 A lambda expression

There are most important information about **lambdas** in C++ 11

- `mutable` allows to modify values passed in captures.
- In C++11 a lambda can't have the default parameter values.
- A lambda is a functor written outside of any class, so it cannot „see" any internal class members without passing `this`.
- To create recursive lambda we use lambda's capture list

```
#include <iostream>

int main()
{
    /* Pass itself in the capture list */
    const std::function<int(int)> fib = [&fib](int idx)
    {
       if (idx < 0)
       {
          std::runtime_error("Idx < 0!");
       }

       if (idx == 0)
       {
          return 0;
       }

       if (idx == 1)
       {
          return 1;
       }

       return fib(idx - 1) + fib(idx - 2);
    };

    /* Prints "55" */
    std::cout << fib(10) << std::endl;
```

```
    return 0;
}
```

1.4 Classes

1.4.1 Constructors and destructors

A few important notes about constructors and destructors:

- Constructors are not inherited.
- It is prohibited to throw an exception from a destructor.
- If a given class is the base class, the destructor must be virtual.
- If all created objects should have the same initial value of member, we can initialize it directly

```
struct Data
{
    const int value { 5 };
};
```

- To catch an exception thrown by the initialization list object, we use a special syntax. It is important to remember that an exception is **thrown again** no matter if caught or not.

```
#include <exception>
#include <iostream>

struct Throwing
{
    Throwing()
    {
       throw std::runtime_error("Exception!");
    }
};

class Initializing
{
public:
    Initializing()
       try : m_throwing() // try : catch (...) {}
       {
       }
       catch (std::runtime_error& e)
       {
          std::cout << "Constructor caught: " << e.what() << std::endl;
          /* Exception is thrown again here! */
       }

private:
    Throwing m_throwing;
};
```

```
int main()
{
    try
    {
        Initializing initializing {};
    }
    catch(std::runtime_error& e)
    {
        std::cout << "Exception handled: " << e.what() << std::endl;
    }
    return 0;
}
```

- Constructors can be delegated.
- Constructors can be reused from base class via `using` directive.

1.4.2 Members initialization

When we have a member initializer list that initializes more than one item, the items are initialized in the order in which they were declared, not in the order in which they appear in the initializer list.

```
#include <iostream>

struct Data1
{
    Data1() { std::cout << "Data1 "; }
};

struct Data2
{
    Data2() { std::cout << "Data2 "; }
};

struct Datas
{
    Datas(): data1 (), data2 ()
    {
    }

    Data2 data2;
    Data1 data1;
};

int main()
{
    /* Prints "Data2 Data1" not "Data1 Data2" */
    Datas datas {};

    return 0;
}
```

1.4.3 Special member functions

Since C++11 we have six special member functions

1. Default constructor,
2. Copy constructor,
3. Move constructor,
4. Copy assignment operator,
5. Move assignment operator,
6. Destructor.

The overview of when special member functions are automatically generated depending on which (other) constructors and special member functions are declared is presented in the picture

user declaration of ↓ / forces →	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
move constructor	undeclared	deleted	deleted	user declared	undeclared (fallback disabled)	defaulted
move assignment	defaulted	deleted	deleted	undeclared (fallback disabled)	user declared	defaulted
destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	user declared

The three main things worth mentioning in this table are that

- a default constructor is only declared automatically if no other constructor is explicitly declared.
- the custom copying member functions and the destructor disables move support, however, a request to move an object still works because of **copying as a fallback**.
- The custom moving member functions disable copying.

```
/* Declared copying disables moving but fallback is enabled */
struct CopyOnly
{
    CopyOnly() = default;

    /* Or just CopyOnly(const CopyOnly&) = default */
    CopyOnly(const CopyOnly&)
    {
        std::cout << "const CopyOnly& constructor" << std::endl;
```

```
    }

    /* Or just CopyOnly& operator=(const CopyOnly&) = default */
    CopyOnly& operator=(const CopyOnly&)
    {
        std::cout << "const CopyOnly& operator=" << std::endl;
        return *this;
    }

    /* No special moving functions declared */
};

/* Declared moving disables copying */
struct MoveOnly
{
    MoveOnly() = default;

    /* Or just MoveOnly(MoveOnly&&) = default */
    MoveOnly(MoveOnly&&)
    {
        std::cout << "MoveOnly&& constructor" << std::endl;
    }

    /* Or just MoveOnly& operator=(MoveOnly&&) = default */
    MoveOnly& operator=(MoveOnly&&)
    {
        std::cout << "Move&& operator= called" << std::endl;
        return *this;
    }

    /* No special copying functions declared */
};

int main()
{
    CopyOnly copy0;
    /* These make a copy due to the fallback */
    CopyOnly copy1 { std::move(copy0) };
    copy0 = std::move(copy1);

    MoveOnly move0;
    /* error: use of deleted function
     * 'constexpr MoveOnly::MoveOnly(const MoveOnly&)'
     */
    // MoveOnly move1 { move0 };

    MoveOnly move1;
    /* error: use of deleted function
     * 'MoveOnly& MoveOnly::operator=(const MoveOnly&)'
     */
    // move1 = move0;

    return 0;
}
```

1.4.4 The rule of zero, three and five

Rule of zero

If you can avoid defining any default operation, do it.

Rule of three

Either declare all three: a copy constructor, a copy assignment operator, and a destructor or none of them. This rule was applied before the introduction of the C++11 standard.

Rule of five

Either declare all five: a copy constructor, a move constructor, a copy assignment operator, a move assignment operator and a destructor or none of them. This rule has been applied since the introduction of the C++11 standard.

1.4.5 Conversions

Converting constructor is any constructor `Class::Class(T)` where `T` is some type. That way, we define the conversion `T` $\rightarrow$ `Class`. If the constructor is preceded by the `explicit` keyword, it will prevent implicit conversions.

Converting operator is any operator `Class::operator T()` where `T` is some type. That way, we define the conversion `Class` $\rightarrow$ `T`. If the operator is preceded by the `explicit` keyword, it will prevent implicit conversions.

There are some limitations related to `T1` $\rightarrow$ `T2` conversions:

- `T1` cannot be one of `T2` or `T2&`,
- `T1` cannot be the base class of `T2`,
- `T1` cannot be `void` type,
- `T1` cannot be a function type,
- `T1` cannot be a C-array type

As we see, there are two possible conversion options. We can take any of them, but if we take both, a compiler may have difficulty selecting which of them should be taken. Nevertheless, a constructor has some minorities:

- We cannot define a constructor converting to built-in type (`int`, `float` etc).
- We cannot add a constructor to the library class.
- A constructor must fit the declared type.
- A converting constructor (as any constructor) cannot be inherited.

1.4.6 Literal constants defined by the user

Defining a literal constant employs a special class operator "" defined by `T operator"" _<suffix>(arg)` where the suffix is defined by the user. Two examples below should be enough to present the possibilities of that operator.

```
#include <iostream>

constexpr std::size_t operator"" _TRAIN(unsigned long long number)
{
        return number;
}

std::string operator"" _PLATFORM(char platform)
{
    return std::string("PLATFORM ") + std::string(1, platform);
}

std::string operator"" _INFO(const char* txt, std::size_t size)
{
    return std::string(txt, size);
}

constexpr std::size_t operator"" _SEC(unsigned long long sec)
{
    return sec / 60;
}

int main()
{
    /* Prints "Train number: 20 */
    std::cout << "Train number: " << 20_TRAIN << std::endl;

    /* Prints "From: PLATFORM C" */
    std::cout << "From: " << 'C'_PLATFORM << std::endl;

    /* Prints "Delay: 60min" */
    std::cout << "Delay: " << " " << 3600_SEC << "min" << std::endl;

    return 0;
}
```

And now the mix of `constexpr` function and the mentioned operator - binary number literal program

```
constexpr std::size_t calculate_size(const char* arr,
                                     std::size_t size = 0)
{
    return arr[0] == '\0'
       ? size
       : arr[0] != '0' && arr[0] != '1'
          ? 0
          : calculate_size(arr + 1, size + 1);
}

constexpr std::size_t to_int(char c)
```

```
{
    return c - 48;
}

constexpr std::size_t bin_power(std::size_t N)
{
    return N == 0 ? 1 : 2 * bin_power(N - 1);
}

constexpr std::size_t to_dec_base(const char* arr, std::size_t size,
                                  std::size_t acc = 0)
{
    return size == 0
    ? acc
    : to_dec_base(arr + 1, size - 1,
                  acc + bin_power(size - 1) * to_int(arr[0]));
}

constexpr std::size_t operator"" _bin(const char* bits)
{
    return calculate_size(bits) == 0
    ? 0
    : to_dec_base(bits, calculate_size(bits));
}

/* IDE prints "9" */
constexpr std::size_t dec = 1001_bin;
```

1.4.7 Constexpr in classes

The `constexpr` constructor allows to create `constexpr` objects. Since `constexpr` functions cannot have more than one `return` statement and a constructor does not have any, the body of such constructor is empty. Therefore, the crucial point is the initialization list - if all of the members have a `constexpr` initialization, the whole object will be `constexpr` created during compilation time; otherwise, the object will be created at runtime.

To have a `constexpr` function as a class member it

- has to be a `constexpr` function,
- cannot be virtual.

In C++11 `constexpr` functions are `const` functions (it will change since C++14).

```
class Calculator
{
public:
    constexpr Calculator(int a, int b): m_a(a), m_b(b)
    {
    }

    constexpr int Add() const
    {
       return m_a + m_b;
```

```
    }

    constexpr int Subtract() const
    {
        return m_a - m_b;
    }

    constexpr int Multiply() const
    {
        return m_a * m_b;
    }

    constexpr int Divide() const
    {
        return m_a / m_b;
    }

private:
    /* Notice that they are not constexpr */
    int m_a;
    int m_b;
};

int main()
{
    /* constexpr is needed */
    constexpr Calculator calculator { 2, 1 };

    /* IDE prints "3" */
    constexpr auto res1 = calculator.Add();

    /* IDE prints "1" */
    constexpr auto res2 = calculator.Subtract();

    /* IDE prints "2" */
    constexpr auto res3 = calculator.Multiply();

    /* IDE prints "2" */
    constexpr auto res4 = calculator.Divide();
}
```

Remember that if a class contains more than one `constexpr` constructor we shouldn't define our destructor if it isn't **trivial** so

- it isn't defined by the user,
- each super classes have their trivial destructors,
- each non-static members have its trivial destructors.

1.4.8 Friend function defined in the class body

If a `friend` function is defined in the class body, then

- it is inline,
- it can use objects defined with `using`, `enum`, `struct` etc. inside the class.

1.4.9 Defining a class inside a function body

C++ supports the possibility of defining classes inside a function body. Such a class needs

- not to have a static member,
- not to have member initialization outside its body,
- not to use automatic variables created in a function body.

```
#include <iostream>

void function()
{
    /* Not available from the class */
    int value = 5;

    class Class
    {
        public:
            Class(int value): m_value(value) {}

            void print() const { std::cout << m_value; }
            void update() { m_value++; }

        private:
            int m_value;
    };

    Class c { value };
    c.update();
    c.print();
}

int main()
{
    /* Prints "6" */
    function();

    return 0;
}
```

1.4.10 Inheritance

The following table contains information about inheritance properties and member visibility in inherited classes.

Property	Public	Protected	Private
Public members become	Public members of the derived class	Protected members of the derived class	Private members of the derived class
Protected members become	Protected members of the derived class	Protected members of the derived class	Private members of the derived class
Private members become	Accessible only through the base-class interface	Accessible only through the base-class interface	Accessible only through the base-class interface
Implicit upcasting	Yes	Yes (but only the derived class) within	No

Remember that inheritance is sometimes misleading

```
class Class1 {};
class Class2 {};

/* Problem - Class2 is private! */
class Derived : public Class1, Class2 {};
```

If we use `using` directive for constructors, we will make all of them visible

```
class Str: public std::string
{
public:
    /* All std::string constructors available */
    using std::string::string;
};
```

1.4.11 RAII

RAII (*Resource Acquisition Is Initialization*) is a paradigm that mostly helps with exception handling, but is useful in many other cases (like automatic file closing). The concept revolves around setting up all needed resources in a constructor and cleaning them in a destructor.

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

struct Resources
{
    Resources(): m_data { 1, 2, 3, 4, 5 }
    {
        std::cout << "Resources created" << std::endl;
    }

    ~Resources()
    {
        std::cout << "Resources deleted" << std::endl;
    }

    std::vector<int> m_data;
};

struct RAII
{
    RAII(): m_resources(new Resources)
    {
    }

    ~RAII()
    {
        delete m_resources;
    }

    const std::vector<int>& Get() const
    {
        return m_resources->m_data;
    }
```

```
    Resources* m_resources;
};

void ResourcesTest()
{
    /* Prints "Resources created */
    RAII raii {};
    for(auto&& data : raii.Get())
    {
        std::cout << "data=" << data << std::endl;
    }
    /* Prints "Resources deleted */
}

int main()
{
    /* Prints
     * "Resources created
     * data=1
     * data=2
     * data=3
     * data=4
     * data=5
     * Resources deleted"
     */
    ResourcesTest();

    return 0;
}
```

1.4.12 Non-type template parameters in C++11

A non-type template parameter shall have one of the following (optionally cv-qualified) types

1. Integral or enumeration type.
2. Pointer to object or pointer to function.
3. Lvalue reference to an object or lvalue reference to a function.
4. Pointer to member.

1.4.13 Templates as parameters

A template can also have a parameter that is itself a template. In C++11 we need to use `class` instead of `typename` inside.

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

template <typename T>
struct Items
{
```

```
    Items(std::initializer_list<T> init): m_items { init }
    {
    }

    const T& operator[](int idx) const
    {
        return m_items[idx];
    }

    std::vector<T> m_items;
};

/* It can be written as
 * template <
 *     template typename <T>
 *     class Collection
 *  >
 * T cannot be used here - only Collection is visible, so we need to put
 * additional template parameter S.
 */
template <template <typename T> class Collection, typename S>
struct Initializer
{
    Initializer(std::initializer_list<S> init): m_collection { init }
    {
    }

    const S& operator[](int idx) const
    {
        return m_collection[idx];
    }

    Collection<S> m_collection;
};

int main()
{
    Initializer<Items, int> int_initializer { 1, 2, 3 };
    Initializer<Items, std::string> str_initializer { "a", "b", "c" };
    Initializer<Items, float> flt_initializer { 1.0, 2.0, 3.0 };

    /* Prints "1" */
    std::cout << int_initializer[0] << std::endl;

    /* Prints "b" */
    std::cout << str_initializer[1] << std::endl;

    /* Prints "3.0" */
    std::cout << flt_initializer[2] << std::endl;

    return 0;
}
```

1.4.14 Virtual functions

The usual way compilers handle virtual functions is to add a hidden member to each object. The hidden member holds a pointer to an array of function addresses. Such an array is usually termed a *virtual function table*. The virtual function table holds the addresses of the virtual functions declared for objects of that class. For example, an object of a base class contains a pointer to a table of addresses of all the virtual functions for that class. An object of a derived class contains a pointer to a separate table of addresses. If the derived class provides a new definition of a virtual function, the virtual table holds the address of the new function. If the derived class doesn't redefine the virtual function, the virtual table holds the address of the original version of the function; if the derived class defines a new function and makes it virtual, its address is added to the table. Note that whether you define one or ten virtual functions for a class, you add just one address member to an object - the table size varies.

It is well-known that virtual functions use *dynamic binding* which is slower than the static one. However, there are exceptions to this rule at times when virtual functions can use *static binding*:

1. Explicit usage of scope qualifier

   ```
   ptr -> Class::Function();
   ```

2. Calling virtual function inside a constructor (a compiler knows which object is creating, moreover constructors can't be marked as `virtual`).

3. Calling `inline` virtual function, which seems to be an oxymoron as late binding excludes „pasting" function's code where it is called:

   - If there is real dynamic binding, `inline` will be ignored.
   - Otherwise (i.e. one of the previous conditions is fulfilled) `inline` will be taken into account.

1.4.15 Diamond problem

The Diamond Problem occurs when a child class inherits from two parent classes who both share a common grandparent class. That can produce ambiguity.

```
class Base
{
public:
    void display() { std::cout << "Base" << std::endl; }
};

class Derived1 : public Base
{
public:
    void display1() { std::cout << "Derived1" << std::endl; }
};

class Derived2 : public Base
{
public:
    void display2() { std::cout << "Derived2" << std::endl; }
```

```
};

/* Two publics needed since
 *     class DiamondDerived : public Derived1, Derived2
 * means
 *     class DiamondDerived : public Derived1, private Derived2
 */
class DiamondDerived : public Derived1, public Derived2
{
public:
    void display3() { std::cout << "Derived3" << std::endl; }
};

int main()
{
    DiamondDerived diamond;

    /* Prints "Derived1" */
    diamond.display1();

    /* Prints "Derived2" */
    diamond.display2();

    /* Prints "Derived3" */
    diamond.display3(); // "Derived3"

    /* Ambiguity - which object should be called? */
    // diamond.display();

    return 0;
}
```

There is no difference whether we use `public`, `protected`, or `private` inheritance - all of them will produce the same error. The solution is to use `virtual` base classes. In such a case, there will be only one base-class object inside the derived class.

```
#include <iostream>

class Base
{
public:
    void display() { std::cout << "Base" << std::endl; }
};

class Derived1 : public virtual Base  // virtual base class
{
public:
    void display1() { std::cout << "Derived1" << std::endl; }
};

class Derived2 : public virtual Base  // virtual base class
{
public:
    void display2() { std::cout << "Derived2" << std::endl; }
};
```

```
/* Two publics needed since
 *     class DiamondDerived : public Derived1, Derived2
 * means
 *     class DiamondDerived : public Derived1, private Derived2
 */
class DiamondDerived : public Derived1, public Derived2  // virtual not needed here
{
public:
    void display3() { std::cout << "Derived3" << std::endl; }
};

int main()
{
    DiamondDerived diamond;

    diamond.display1(); // "Derived1"
    diamond.display2(); // "Derived2"
    diamond.display3(); // "Derived3"

    diamond.display(); // No ambiguity as we have only one base object

    return 0;
}
```

1.4.16 override and final

In C++, `override` and `final` are two important identifiers introduced in C++11 to enhance the clarity and safety of object-oriented programming, particularly with inheritance and polymorphism.

override

The `override` identifier is used to explicitly declare that a member function in a derived class overrides a virtual function from the base class.

- Ensures that the function actually overrides a function in the base class.
- Prevents subtle bugs due to mismatches in the function signature (e.g., a typo or wrong parameter types).

When we omit `override`, we can expect that the compiler will allow the code to compile even if the function does not properly override a base class function. For example, if the derived function's signature is incorrect (e.g., due to a typo or incorrect parameter type), the function will not override the base class's function. Instead, it will be treated as a new function, which might cause unexpected behavior.

```
struct Base
{
    virtual void display() const
    {
        std::cout << "Base" << std::endl;
    }
};

struct Derived : Base
```

```
{
public:
    /* The compiler is not able to detect the problem */
    void display(std::string str) const
    {
        std::cout << "Derived with " << str << std::endl;
    }
};
```

Remember that if we redeclare a function in a derived class with the same signature but without explicitly using `override`, the function in the base class **will be hidden instead of being overridden!** This means that name hiding occurs, not function overriding.

`final`

The `final` specifier is used to indicate that a class or a virtual function cannot be further overridden or derived. It

- prevents further modification of a virtual function or inheritance of a class,
- ensures the behavior defined by the class or function remains unchanged in derived classes.

```
struct Base
{
    virtual void print() const
    {
        std::cout << "Base" << std::endl;
    }
};

struct Derived1 : Base
{
    /* This cannot be overrides further */
    void print() const final
    {
        std::cout << "Derived1" << std::endl;
    }
};

/* Error: Cannot override a final function
 * struct Derived2 : Derived1
 * {
 *     void print() const override
 *     {
 *         std::cout << "Derived2" << std::endl;
 *     }
 * };
 */
```

It is even possible to mark an entire class with `final`

```
struct Final final
{
    void display() const
    {
```

```
        std::cout << "Final" << std::endl;
    }
};

/* Error: Cannot inherit from a final class
 * struct Derived : Final {};
 */
```

Of course, `override` and `final` can be combined.

```
struct Base
{
    virtual void display() const
    {
        std::cout << "Base" << std::endl;
    }
};

struct Derived : Base
{
    /* Override and prohibits further overriding */
    void display() const override final
    {
        std::cout << "Derived" << std::endl;
    }
};
```

1.4.17 Aggregate

To define *aggregate* we need to fulfill the following conditions

1. All members are public.
2. There is no constructor defined by a user.
3. There are no initializers for non-static members.
4. It is not a derived class.

A destructor or an assignment operator can be user-defined.

```
struct Aggregate
{
    int value;
    char sign;
};
```

1.5 Miscellaneous

1.5.1 Reading complex pointer declarations

We start with the name. Then, if possible, we move to the right (on the right, only operators `()` `[]` can be placed, so the strongest possible ones). Upon reaching a potential closing parenthesis ")", we move to the left until everything inside that parenthesis is read. Then we exit beyond that parenthesis and continue reading from the right. Thus, having

```
int (* (*ptr) (int, char*)) [2];
```

we read *ptr is the pointer to a function with two arguments* `(int, char*)`*, which returns a pointer to two elements array of int type.*

1.5.2 Stack unwinding

When an exception is thrown and control passes from a try block to a handler, the C++ run time calls destructors for all automatic objects constructed since the beginning of the try block. This process is called **stack unwinding**. The automatic objects are destroyed in reverse order of their construction. (Automatic objects are local objects that have been declared auto or register, or not declared static or extern. An automatic object x is deleted whenever the program exits the block in which x is declared.)

Suppose an exception is thrown during the construction of an object consisting of subobjects or array elements. In that case, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown. A destructor for a local static object will only be called if the object was successfully constructed.

If during stack unwinding a destructor throws an exception and that exception is not handled, the `std::terminate()` function is called. That's why **we should never throw an exception from any destructor!**

1.5.3 RTTI

RTTI (*Runtime Type Identification*) provides a standard way for a program to determine the type of object during runtime. Three basic functionalities are supporting RTTI:

- The `dynamic_cast` operator generates a pointer to a derived type from a pointer a to base type, if possible. Otherwise, the operator returns the null pointer.
- The `typeid` operator returns a value identifying the exact type of object.
- A `type_info` structure holds information about a particular type.

1.5.4 `alignas`

The `alignas` specifier creates a request for the compiler to put some value under the address which is the multiplication of 2, so $n = 2^k$ for some $k \in \mathbb{N}$.

```
/* k = 4 */
alignas(16) long value = 1;
```

It is rarely used.

1.5.5 Cast operators

In C++ we have four types of standard cast operators

1. The `static_cast` operator performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler. The `static_cast` can convert between related types, such as numeric types of a pointer in the same inheritance hierarchy.

2. The `dynamic_cast` operator performs **downcasting** so converting a pointer or reference of base class to a derived class (opposite way can be easily done by `static_cast`). It ensures type safety by performing a runtime check to verify the validity of the conversion. If the conversion is not possible, `dynamic_cast` returns a null pointer for pointer conversions or throws `bad_cast_exception` for reference conversions.

3. The `const_cast` operator performs temporary removal of the constancy of an object and allows modifications. It works with pointers and references.

4. The `reinterpret_cast` operator performs risk casting from a given pointer to any other type of pointer. It does not include any check on whether casting makes sense or not.

The following examples explain the differences and ways of usage of the mentioned operators

static_cast

```cpp
#include <iostream>

struct Base
{
    virtual void print() const
    {
        std::cout << "Base class" << std::endl;
    }
};

struct Derived : Base
{
    void print() const override
    {
        std::cout << "Derived class" << std::endl;
    }
};

int main()
{
    /* Example 1: Basic type conversion */
    /* Prints 0 */
    std::cout << static_cast<double>(int{0}) << std::endl;

    /* Prints 1 */
    std::cout << static_cast<int>(double{1.5}) << std::endl;


    /* Example 2: Upcasting - this is safe */
    Derived derived{};

    /* Upcasting */
    Base* base_ptr = static_cast<Base*>(&derived);

    /* Prints "Derived class" */
    base_ptr->print();
```

```
    /* Example 3: Downcasting - this is unsafe! */
    Base base{};

    /* Downcasting */
    Derived* derived_ptr = static_cast<Derived*>(&base);

    /* Undefined behavior */
    derived_ptr->print();
    return 0;
}
```

dynamic_cast

```
#include <exception>
#include <iostream>

struct Base
{
    virtual void print() const
    {
        std::cout << "Base class" << std::endl;
    }
};

struct Derived : Base
{
    void print() const override
    {
        std::cout << "Derived class" << std::endl;
    }
};

int main() {
    /* Example 1: Downcasting - this will work */
    Base* base_ptr1 = new Derived;

    /* With pointer */
    Derived* derived_ptr1 = dynamic_cast<Derived*>(base_ptr1);
    if (derived_ptr1)
    {
        /* This case will be handled - prints "Derived class" */
        derived_ptr1->print();
    }
    else
    {
        std::cout << "Downcasting failed" << std::endl;
    }

    /* With reference */
    Derived& derived1 = dynamic_cast<Derived&>(*base_ptr1);

    /* Prints "Derived class" */
    derived1.print();
```

```
    delete base_ptr1;


    /* Example 2: Downcasting - this won't work */
    Base* base_ptr2 = new Base;

    /* With pointer */
    Derived* derived_ptr2 = dynamic_cast<Derived*>(base_ptr2);
    if (derived_ptr2)
    {
        derived_ptr2->print();
    }
    else
    {
        /* This case will be handled - program checked in
         * runtime that this conversion isn't safe */
        std::cout << "Downcasting failed" << std::endl;
    }

    /* With reference */
    /* This one raises exception std::bad_cast() */
    try
    {
        Derived& derived2 = dynamic_cast<Derived&>(*base_ptr2);
    }
    catch(std::bad_cast e)
    {
        std::cout << "Exception: " << e.what() << std::endl;
        delete base_ptr2;
    }

    return 0;
}
```

const_cast

```
#include <iostream>

int main()
{
    /* Example 1: const_cast and non-const object */
    int x = 0;

    /* With pointer */
    const int* const_x_ptr = &x;
    int* non_const_x_ptr = const_cast<int*>(const_x_ptr);

    /* This is safe since the original object is not const */
    *non_const_x_ptr = 1;
    /* Prints "x value modified and x = 2" */
    std::cout << "x value modified and x = " << x << std::endl;

    /* With reference */
    const int& const_x_ref = x;
    int& non_const_x_ref = const_cast<int&>(const_x_ref);

    /* This is safe since the original object is not const */
    non_const_x_ref = 2;
    /* Prints "x value modified and x = 2" */
    std::cout << "x value modified and x = " << x << std::endl;


    /* Example 2: const_cast and const object */
    const int y = 0;

    /* With pointer */
    const int* const_y_ptr = &y;
    int* non_const_y_ptr = const_cast<int*>(const_y_ptr);

    /* Behavior is undefined since the original object is const */
    *non_const_y_ptr = 1;
    /* Hard to say what will print but probably y  = 0*/
    std::cout << "y value modified and y = " << y << std::endl;

    /* With reference */
    const int& const_y_ref = y;
    int& non_const_y_ref = const_cast<int&>(const_y_ref);

    /* Behavior is undefined since the original object is const */
    non_const_y_ref = 2;
    /* Hard to say what will print but probably y  = 0*/
    std::cout << "y value modified and y = " << y << std::endl;

    return 0;
}
```

reinterpret_cast

```
#include <iostream>
```

```cpp
struct Base
{
    virtual void print() const { std::cout << "Base class" << std::endl; }
};

struct Derived : Base
{
    void print() const override { std::cout << "Derived class" << std::endl; }
};

int main() {
    /* Example 1: Downcasting - compare with dynamic_cast */
    Base* base_ptr = new Base;

    /* With pointer */
    Derived* derived_ptr = reinterpret_cast<Derived*>(base_ptr);
    if (derived_ptr)
    {
        /* This case will be handled - prints "Base class" */
        derived_ptr->print();
    }
    else
    {
        std::cout << "Downcasting failed" << std::endl;
    }

    /* With reference */
    /* This one doesn't raise exception std::bad_cast() */
    Derived& derived = reinterpret_cast<Derived&>(*base_ptr);

    /* Prints "Base class" */
    derived.print();

    delete base_ptr;

    /* Example2 - unrelated objects */
    int x = 50000;

    /* Interpret integer as pointer to char */
    const char* char_ptr = reinterpret_cast<const char*>(&x);

    /* Can print anything */
    std::cout << *char_ptr << std::endl;

    return 0;
}
```

1.5.6 Classes

Smart pointers

- The `std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `std::unique_ptr` goes out of scope. The object is disposed of, using the associated deleter when either of the following happens

- the managing `std::unique_ptr` object is destroyed, or
- the managing `std::unique_ptr` object is assigned another pointer via `operator=()` or `reset()`.

The object is disposed of, using a potentially user-supplied deleter. The default deleter uses the delete operator, which destroys the object and deallocates the memory.

```
#include <iostream>
#include <memory>

struct Data
{
    Data()
    {
        std::cout << "Data constructor" << std::endl;
    }

    ~Data()
    {
        std::cout << "Data destructor" << std::endl;
    }
};

struct Deleter
{
    Deleter() = default;

    template <typename T>
    void operator()(T* ptr)
    {
        std::cout << "Deleter calls: ";
        delete ptr;
    }
};

int main()
{
    /* Prints "Data constructor" */
    std::unique_ptr<Data, Deleter> ptr { new Data {}, Deleter {} };

    /* Prints "Deleter calls: Data destructor" */
    ptr.reset();

    return 0;
}
```

- The `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several `std::shared_ptr` objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens
  - the last remaining `std::shared_ptr` owning the object is destroyed, or
  - the last remaining `std::shared_ptr` owning the object is assigned another pointer via `operator=` or `reset()`.

The object is destroyed using delete-expression or a custom deleter that is supplied to `std::shared_ptr` during construction. This type works since it keeps internal counter of its copies. This counter is `std::atomic` so it is safe to use it in multithread environment. Apart from copies counter, it also keeps `std::weak_ptr` counter, also atomic to manage weak pointer resources.

```
#include <iostream>
#include <memory>

struct Data
{
    Data()
    {
        std::cout << "Data constructor" << std::endl;
    }

    ~Data()
    {
        std::cout << "Data destructor" << std::endl;
    }
};

struct Deleter
{
    Deleter() = default;

    template <typename T>
    void operator()(T* ptr)
    {
        std::cout << "Deleter calls: ";
        delete ptr;
    }
};

int main()
{
    /* Prints "Data constructor" */
    std::shared_ptr<Data> ptr { new Data {}, Deleter {} };

    /* Prints "Deleter calls: Data destructor" */
    ptr.reset();

    return 0;
}
```

The thing related with `std::shared_ptr` that we need to care about is **avoiding cyclic dependency**

```
#include <iostream>
#include <memory>

struct Struct2;

struct Struct1
{
    Struct1()
```

```
    {
        std::cout << "Struct1 constructor" << std::endl;
    }

    ~Struct1()
    {
        std::cout << "Struct1 destructor" << std::endl;
    }

    /* Struct1 has dependency to Struct2 */
    std::shared_ptr<Struct2> ptr;
};

struct Struct2
{
    Struct2()
    {
        std::cout << "Struct2 constructor" << std::endl;
    }

    ~Struct2()
    {
        std::cout << "Struct2 destructor" << std::endl;
    }

    /* Struct2 has dependency to Struct1 */
    std::shared_ptr<Struct1> ptr;
};

int main()
{
    {
        /* Printouts
         * "Struct1 constructor"
         * "Struct2 constructor"
         */
        auto struct1 = std::make_shared<Struct1>();
        auto struct2 = std::make_shared<Struct2>();

        /* Circullar dependency */
        struct1->ptr = struct2;
        struct2->ptr = struct1;
    }

    /* No more printouts! */

    return 0;
}
```

In the example above, neither `struct1` nor `struct1` were deleted. It is because `ptr` from `Struct1` has ownership of the `Struct2` pointer, and `ptr` from `Struct2` has ownership of the `Struct1` pointer. Because of that none of them will release resources. The solution is to use `std::weak_ptr` in one of these classes

```
#include <iostream>
#include <memory>
```

```cpp
struct Struct2;

struct Struct1
{
    Struct1()
    {
        std::cout << "Struct1 constructor" << std::endl;
    }

    ~Struct1()
    {
        std::cout << "Struct1 destructor" << std::endl;
    }

    /* Struct1 doesn't own Struct2 pointer */
    std::weak_ptr<Struct2> ptr;
};

struct Struct2
{
    Struct2()
    {
        std::cout << "Struct2 constructor" << std::endl;
    }

    ~Struct2()
    {
        std::cout << "Struct2 destructor" << std::endl;
    }

    /* Struct2 has dependency to Struct1 */
    std::shared_ptr<Struct1> ptr;
};

int main()
{
    {
        /* Printouts
         * "Struct1 constructor"
         * "Struct2 constructor"
         */
        auto struct1 = std::make_shared<Struct1>();
        auto struct2 = std::make_shared<Struct2>();

        /* Circullar dependency */
        struct1->ptr = struct2;
        struct2->ptr = struct1;
    }

    /* Printouts
     * "Struct2 destructor"
     * "Struct1 destructor"
     */

    return 0;
```

```
}
```

- The `std::weak_ptr` has been discussed already but it is worth to mention that it can also check the validity of the data by calling one of `expired()` or `lock()` and it is handy to resolve problems with a dangling pointer.

```
/* Dangling pointer problem */
int* ptr = new int { 5 };
int* ref_ptr = ptr;

/* Now ref_ptr points the undefined data */
delete ptr;
```

The `std::weak_ptr` can solve this issue.

```
#include <iostream>
#include <memory>

void check_ptr(const std::weak_ptr<int>& weak)
{
    auto value = weak.lock();
    if (value == nullptr)
    {
        std::cout << "weak has expired" << std::endl;
        return;
    }
    std::cout << "weak points value=" << *value << std::endl;
}

int main()
{
    std::shared_ptr<int> shared { new int { 5 } };
    std::weak_ptr<int> weak = shared;

    /* Prints "Weak points value=5 */
    check_ptr(weak);

    /* Destroys old data */
    shared.reset(new int { 6 });

    /* Prints "Weak has expired" */
    check_ptr(weak);  // prints "weak has expired"

    return 0;
}
```

std::bitset

The `std::bitset` class allows to keep the number in the binary form. C++ does not provide any `std::cout` manipulator to print a number with a binary form, but `std::bitset` has `operator«` overload to do it.

```
#include <iostream>
#include <bitset>

int main()
```

```
{
    /* Bitset needs unsigned integer value */
    unsigned long value = 100;

    /* The number will be represented by 8 bits */
    std::bitset<8> bitset0 { value };

    /* Prints "01100100:8" */
    std::cout << bitset0 << ":"
              << bitset0.size() << std::endl;

    /* The number will be cut - three last bits */
    std::bitset<3> bitset1 { value };

    /* Prints "100:3" */
    std::cout << bitset1 << ":"
              << bitset1.size() << std::endl;

    /* std::bitset can handle strings */
    std::bitset<1> bitset2 { "1" };

    /* Prints "1:1" */
    std::cout << bitset2 << ":"
              << bitset2.size() << std::endl;

    return 0;
}
```

Chapter 2

C++14

2.1 Basics

2.1.1 `auto` as a result of function

From now the compiler can deduce a returning type of the expression

```
#include <iostream>

template <typename T1, typename T2>
auto multiply(T1 t1, T2 t2)
{
    return t1 * t2;
}

int main()
{
    std::cout << multiply(1, 2.0) << std::endl;

    return 0;
}
```

If there is more than one `return` statement, the compiler will check their consistency - if they are not the same, compilation will fail (even if there is a conversion from one type to another)

```
#include <iostream>

template <typename T1, typename T2>
auto max(T1 t1, T2 t2)
{
    if (t1 < t2)
    {
        return t2;
    }
    return t1;
}

int main()
{
    /* error: inconsistent deduction for auto return type */
    // std::cout << max(1, 2.0) << std::endl;
```

```
    return 0;
}
```

In 4.3.6 subchapter we will discuss the way of using `decltype(auto)`, also available since C++14. The only thing we have to keep in mind is that `decltype` gives the declared type of the expression that is passed to it. The `auto` keyword does the same thing as template type deduction so, for example, if you have a function that returns a reference, `auto` will still be a value (you need `auto&` to get a reference), but `decltype` will be exactly the type of the return value.

2.1.2 The variable template

C++ introduces, in addition to class and function templates, the template of variables. Beyond the ability to avoid casting (by creating the variable of perfectly matching type), we can employ them to type traits. Since now, we don't need to create special structures with `value` member - we just can have a single variable.

```
#include <iostream>
#include <string>
#include <type_traits>

/* For any given type be false */
template <typename T>
/* Must be constexpr ! */
constexpr bool is_string_v = false;

/* For string be true */
template <>
constexpr bool is_string_v<std::string> = true;

template <typename T,
          std::enable_if_t<is_string_v<T>, bool> = true>
void info(const T&)
{
    std::cout << "std::string" << std::endl;
}

template <typename T,
          std::enable_if_t<!is_string_v<T>, bool> = true>
void info(const T&)
{
    std::cout << "not std::string" << std::endl;
}

int main()
{
    /* Prints "not std::string" */
    info(int {});

    /* Prints "std::string" */
    info(std::string {});

    return 0;
}
```

2.2 Functions

2.2.1 A `constexpr` function

Starting from C++14, many of the constraints of `constexpr` functions are relaxed. Since now we can

- use loops,
- use `switch` and `if` statements,
- use define own `constexpr` variables inside the body of `constexpr` function.

Let's get back to the example 1.4.6 and rewrite it with our new opportunities

```
constexpr std::size_t to_int(char c)
{
    return c - 48;
}

constexpr std::size_t bin_power(std::size_t N)
{
    if (N == 0)
    {
        return 1;
    }

    std::size_t value { 1 };
    for (int i = 0; i < N; ++i)
    {
        value *= 2;
    }
    return value;
}

constexpr std::size_t to_dec_base(const char* arr, std::size_t size)
{
    std::size_t acc { 0 };
    for (std::size_t i = 0; i < size; i++)
    {
        acc += bin_power(size - 1 - i) * to_int(arr[i]);
    }
    return acc;
}

constexpr std::size_t operator"" _bin(const char* bits)
{
    if (bits[0] == '\0')
    {
        return 0;
    }

    std::size_t size { 0 };
    while (bits[size] != '\0')
    {
        size++;
    }
```

```
    return to_dec_base(bits, size);
}

/* IDE prints "9" */
constexpr std::size_t dec = 1001_bin;
```

2.2.2 Lambdas

The C++14 standard extends lambda capabilities to have **universal type of paramenter**, for instance

```
/* Lambda can have universal parameter type */
[](auto u, auto v) { return u + v; };
```

The mechanism behind the universal lambda is that the compiler converts it into the functor with the template `operator()` function, thus in our case, it could be

```
struct Lambda
{
    template <typename T1, typename T2>
    auto operator(T1 t1, T2 t2) { return t1 + t2; }
};
```

Apart from universal parameters, C++14 gives us one more convenient tool which is initialization in the capture list

```
#include <iostream>

int main()
{
    int value = 1;

    /* Named variable in the capture list */
    auto lambda = [shift = value * 3](auto u, auto v)
    {
        return u + v + shift;
    };

    /* Prints "6" */
    std::cout << lambda(1, 2) << std::endl;

    return 0;
}
```

That is done by adding a member to the functor

```
#include <iostream>

template <typename Shift>
struct Lambda
{
    explicit Lambda(Shift shift):
        m_shift(shift)
    {
    }

    template <typename T1, typename T2>
```

```
    auto operator()(T1 t1, T2 t2)
    {
        return t1 + t2 + m_shift;
    }

    Shift m_shift;
};

int main()
{
    int value = 1;

    /* Prints "6" */
    std::cout << Lambda<decltype(value)> { value * 3 } (1, 2)
              << std::endl;
    return 0;
}
```

What is more, the lambda expression supports move semantics

```
#include <iostream>

struct Data
{
    Data(): m_value(0)
    {
        std::cout << "Data" << std::endl;
    }

    Data(const Data& data): m_value(data.m_value)
    {
        std::cout << "const Data&" << std::endl;
    }

    Data(Data&& data): m_value(std::move(data.m_value))
    {
        data.m_value = 0;
        std::cout << "Data&&" << std::endl;
    }

    ~Data()
    {
        std::cout << "~Data" << std::endl;
    }

    void set(int value)
    {
        m_value = value;
    }

    int m_value;
};

int main()
{
    /* Prints "Data" */
    Data data;
```

```
    /* Data is moved even if the lambda is not called! */
    /* Prints "Data&&" */
    auto lambda = [moved_data = std::move(data)] () mutable
    {
        moved_data.set(1);
        return moved_data;
    };

    /* Prints "const Data&" */
    auto&& result = lambda();

    /* Prints "0" */
    std::cout << data.m_value << std::endl;

    /* Prints "1" */
    std::cout << result.m_value << std::endl;

    return 0;
}
```

2.3 Miscellaneous

C++14 provided a bit more useful elements

1. New attributes like `[[noreturn]]`, `[[deprecated]]` etc.
2. A separators in numbers `int value = 500‘000‘000`.
3. `new` and `delete` operators overloading.

Chapter 3

C++17

3.1 Basics

3.1.1 `if` and `switch` with initialization

Since C++17 both `if` and `switch` statements can be equipped with **initialization**.

```
#include <iostream>
#include <random>
#include <string>
#include <utility>

const int EXPECTED = 2;

int get_random()
{
    std::random_device device;
    std::mt19937 engine { device() };
    std::uniform_int_distribution<std::mt19937::result_type> dstr(0, 4);

    return dstr(engine);
}

inline bool more_than(int value, int expected)
{
    return value > expected;
}

int main()
{
    /* Initialization in if statement */
    if (int value = get_random(); more_than(value, EXPECTED))
    {
        std::cout << value << " > " << EXPECTED << std::endl;
    }
    else
    {
        std::cout << value << " < " << EXPECTED << std::endl;
    }

    /* Initialization in switch statement */
    switch (int value = get_random(); more_than(value, EXPECTED))
```

```
    {
        case true:
            std::cout << value << " > " << EXPECTED << std::endl;
            break;
        default:
            std::cout << value << " < " << EXPECTED << std::endl;
    }

    return 0;
}
```

3.1.2 constexpr if

The compile-time `if` opens the door to use conditional statements during compilation. The big difference between standard `if` is that we usually need to use `else` complementary part explicitly.

```
#include <string>

template <int value>
auto function()
{
    if constexpr (value > 0)
    {
        return value;
    }
    return std::string { "0" };
}

int main()
{
    function<0>();

    /* Won't compile */
    // function<1>();

    return 0;
}
```

Observe that the condition `value < 0` discards `if constexpr` so `std::string` is the only one deduced type. The situation is drastically different when we use `value > 0` - in that case, `if constexpr` is fulfilled, and then two different types i.e. `int` and `std::string` are deduced as return value. This is the reason for the failure. To correct it, we need to add the `else` clause

```
#include <string>

template <int value>
auto function()
{
    if constexpr (value > 0)
    {
        return value;
    }
    else
    {
```

```
        return std::string { "0" };
    }
}

int main()
{
    std::string str = function<0>();
    int i = function<1>();

    return 0;
}
```

When `value > 0` the condition `if constexpr` is fulfilled but `else` discarded, thus there is no conflict for the return value.

Besides making `type_traits` much more convenient to write, it allows to construct functions with different return type

```
template <int value>
auto function()
{
    if constexpr (value > 0)
    {
        return value;
    }
}
```

This function returns either `int` or `void` depending on whether `value` is positive or not. As a for regular `if`, its compile-time counterpart supports the initialization statement.

At the end, it is, of course, possible to use `if else` clause

```
#include <iostream>
#include <type_traits>

template <typename T>
void check_type()
{
    if constexpr (std::is_same_v<T, int>)
    {
        std::cout << "int" << std::endl;
    }
    else if constexpr (std::is_same_v<T, float>)
    {
        std::cout << "float" << std::endl;
    }
    else if constexpr (std::is_same_v<T, double>)
    {
        std::cout << "double" << std::endl;
    }
    /* Required ! */
    else
    {
        std::cout << "Unknown type" << std::endl;
    }
}
```

```
int main()
{
    /* Prints "int" */
    check_type<int>();

    /* Prints "float" */
    check_type<float>();

    /* Prints "double" */
    check_type<double>();

    /* Prints "Unknown type" */
    check_type<char>();

    return 0;
}
```

3.2 Functions

From now lambdas achieve a few more possibilities

- they are `constexpr` by default and if possible we can use them as usual `consexpr` functions.

```
#include <iostream>
#include <type_traits>

auto is_negative = [](auto value) { return value < 0; };

template <int value,
          std::enable_if_t<is_negative(value), bool> = true>
int absolute_value()
{
    return -value;
}

template <int value,
          std::enable_if_t<!is_negative(value), bool> = true>
int absolute_value()
{
    return value;
}

int main()
{
    /* Prints "1" */
    std::cout << absolute_value<-1>() << std::endl;

    /* Prints "0" */
    std::cout << absolute_value<0>() << std::endl;

    /* Prints "1" */
    std::cout << absolute_value<1>() << std::endl;
```

```
    return 0;
}
```

- In C++11 and C++14 passing [&], [=] or [this] always allows to modify class, because the copy of the `this` pointer. C++17 fills this gap by introducing `[*this]` expression

```
#include <iostream>

struct Data
{
    int value { 0 };

    void modify()
    {
        /* Instant call of lambda */
        [copy=*this] () mutable { copy.value = 1; } ();
    }
};

int main()
{
    Data data;
    data.modify();

    /* Remains untouched so prints "0" */
    std::cout << data.value << std::endl;

    return 0;
}
```

Remember about it when you're working with multithreading.

3.3 Classes

3.3.1 Inline variables

Before C++17 there was no way to initialize **non-const** [1] static members inside a class if this class is in the header file

```
struct Struct
{
    static int counter {};
};

/* Compilation gives error
 * "error: ISO C++ forbids in-class initialization of
 * non-const static member 'Struct::counter'"
 */
int main()
{
    return 0;
}
```

We could move the definition outside of the class and that will work

[1] For `const` case we can initialize member inside.

header.h

```
#pragma once

struct Struct
{
    static int counter;
};

int Struct::counter = 0;
```

main.cpp

```
#include "header.h"

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

but what if we include the same header in another *.cpp* file ?

includer.cpp

```
#include "header.h"
```

In such a case the compilation will end with error *multiple definition of 'Struct::counter'*.

Since C++17 we have a solution in the form of **inline variable**

```
#include <iostream>

struct Struct
{
    Struct()
    {
        std::cout << "Struct " << ++counter << " created\n";
    }

    ~Struct()
    {
        std::cout << "Struct " << counter-- << " destroyed\n";
    }

    /* static inline variable */
    static inline int counter { 0 };
};

int main()
{
    /* Prints "Struct 1 created" */
    Struct s1;
    {
        /* Prints "Struct 2 created" */
        Struct s2;
        {
```

```
            /* Prints "Struct 3 created" */
            Struct s3;
        }
        /* Prints "Struct 3 destroyed" */
    }
    /* Prints "Struct 2 destroyed" */

    return 0;
    /* Prints "Struct 1 destroyed" */
}
```

That could be handy with `thread_local` keyword - threads won't share the same object anymore. Moreover, `constexpr` implies `inline`, thus

```
static inline constexpr int value { 0 };
```

means the same as

```
static constexpr int value { 0 };
```

3.3.2 Aggregate

Since C++17, aggregates can have base classes, so that for such structures being derived from other classes or structures list initialization is allowed. From now the definition of aggregate changed and now the *aggregate* means

- either an array,
- or a class type (class, struct, or union) with:
  - no user-declared or explicit constructor
  - no constructor inherited by a `using` declaration
  - no private or protected non-static data members
  - no virtual functions
  - no virtual, private, or protected base classes

```
#include <string>

struct AggregateBase
{
    int i;
    float f;
};

struct AggregateDerived : AggregateBase
{
    std::string string;
};

int main()
{
    /* Base class can have separate parenthesis */
    AggregateDerived derived0 { { 0, 0.0 }, "str0" };

    /* But we don't need to use them */
```

```
    AggregateDerived derived1 { 1, 1.1, "str1" };

    /* We can skip initial values - missing ones
     * will be zero-initialized
     */
    AggregateDerived derived3;
    AggregateDerived derived4 { 4, 4.4 };
    AggregateDerived derived5 { { 5 }, "str5" };

    return 0;
}
```

Aggregates extend over derived classes from non-aggregates. This isn't a surprise as a derived class "holds" the base class object inside

```
#include <string>

struct AggregateString : std::string
{
    int i;
};

int main()
{
    AggregateString aggregate0 { { "str0" }, 0 };
    AggregateString aggregate1 { {  }, 1 };

    return 0;
}
```

or even multiple classes

```
#include <string>
#include <vector>

struct AggregateMultipleBase : std::string, std::vector<int>
{
    int i;
};

int main()
{
    AggregateMultipleBase derived0 { { "str0" }, { 1, 2, 3 }, 0 };
    AggregateMultipleBase derived1 { "str1", {},  1 };

    return 0;
}
```

3.3.3 Structured bindings

Structured bindings can be used for structures with public data members, raw C-style arrays, and *tuple-like objects:*

- If in structures and classes, all non-static data members are public, you can bind each non-static data member to exactly one name.

- For raw arrays, you can bind a name to each element.
- For any type you can use a tuple-like API to bind names to whatever the API defines as "elements." The API roughly consists of the following elements for a type type:
  - `std::tuple_size<type>::value` has to return the number of elements.
  - `std::tuple_element<idx,type>::type` has to return the type of the idx-th element.
  - A global or member `get<idx>()` has to yield the value if the idx-th element.

  The standard library types `std::pair`, `std::tuple`, and `std::array` already provide this API.

To understand structured bindings, be aware that there is a new anonymous variable involved. The new names introduced as structure bindings refer to members or elements of this anonymous object. The exact behavior of an initialization

```
struct Data
{
    type1 i;
    type2 s;
} data;

auto [u,v] = data;
```

is as we'd initialize a new entity `entity` with `data` and let the structured bindings `u` and `v` become alias names for the members of this new object, similar to defining

```
auto entity = data;
aliasname u = entity.i;
aliasname v = entity.s;
```

Note that `u` and `v` are not references to `entity.i` and `entity.s`, respectively. They are just other names for the members. Thus, `decltype(u)` is the type of the member `i` and `decltype(v)` is the type of the member `s`.

Qualifiers

We can use qualifiers, such as `const` or `volatile` and references. Again, these qualifiers apply to the anonymous entity as a whole

```
const auto& [u,v] = data;
```

Qualifiers don't necessarily apply to the structured bindings. Both `u` and `v` are declared as references and this only specifies that the anonymous entity is a reference - `u` and `v` have the type of the members of `data`.

Structured bindings **do not *decay*** [2] although `auto` is used. Look at the following example

[2] The *decay* is the type same as when arguments are passed by value, which means that raw arrays convert to pointer and top-level qualifiers, such as `const` and references, are ignored.

```
struct Struct
{
    const char x[6];
    const char y[3];
};

Struct s{};
auto [u, v] = s;
```

the type of u and v is const char[6] and const char[3] respectively since **auto doesn't decay in this case** [3].

```
#include <iostream>
#include <type_traits>

struct Struct
{
    const char x[6];
    const char y[3];
};

int main()
{
    Struct s{};
    auto [u, v] = s;

    /* Prints "1" */
    std::cout << std::is_same_v<decltype(u), const char[6]> << std::endl;

    /* Prints "1" */
    std::cout << std::is_same_v<decltype(v), const char[3]> << std::endl;
    return 0;
}
```

Structured binding with classes

Note that there is only limited usage of inheritance possible - all non-static data members must be members of the same class definition. Therefore, they have to be direct members of the type or the same unambiguous public base class

```
struct Base
{
    int a {1};
    int b {2};
};

struct Derived : Base {};

int main()
{
    /* Copy of values - everything works */
    auto [u, v] = Derived{};
    return 0;
}
```

[3] Compare it with 1.2.6 section

The code above compiles without any problem, but

```
struct Base
{
    int a {1};
    int b {2};
};

struct Derived : Base
{
    int c = {3};
};

int main()
{
    /* Won't compile */
    // auto [u, v, z] = Derived {};
}
```

do not compile - the error says *cannot decompose class type 'Derived': both it and its base class 'Base' have non-static data members.*

The C++ language provides a way how to manually implement structured bindings for user-defined types.

```
#include <iostream>
#include <string>
#include <utility>

class Class
{
public:
    Class(std::string str1, std::string str2, long val):
        m_str1 ( std::move(str1) ),
        m_str2 ( std::move(str2) ),
        m_val ( val )
{
}

const std::string& get_str1() const
{
    return m_str1;
}

std::string& get_str1()
{
    return m_str1;
}

const std::string& get_str2() const
{
    return m_str2;
}

std::string& get_str2()
{
```

```
    return m_str2;
}

const long& get_val() const
{
    return m_val;
}

long& get_val()
{
    return m_val;
}

private:
    std::string m_str1;
    std::string m_str2;
    long m_val;
};

/* Most important part - specialization for user-defined class */
template <>
struct std::tuple_size<Class>
{
    static constexpr std::size_t value = 3;
};

template <std::size_t Idx>
struct std::tuple_element<Idx, Class>
{
    using type = std::string;
};

template <>
struct std::tuple_element<2, Class>
{
    using type  = long;
};

/* The function name must be get!
 * decltype(auto) is crucial. Firstly, auto is needed
 * since we return different types from this function.
 * Secondly, decltype(auto) do not decay - we return
 * references here.
 */
template <std::size_t N>
decltype(auto) get(Class& c)
{
    static_assert(N < 3, "Class has only three elements!");
    if constexpr (N == 2)
    {
        return c.get_val();
    }
    else if constexpr (N == 1)
    {
        return c.get_str2();
    }
```

```
    else
    {
        return c.get_str1();
    }
}

template <std::size_t N>
decltype(auto) get(const Class& c)
{
    static_assert(N < 3, "Class has only three elements!");
    if constexpr (N == 2)
    {
        return c.get_val();
    }
    else if constexpr (N == 1)
    {
        return c.get_str2();
    }
    else
    {
        return c.get_str1();
    }
}

template <std::size_t N>
decltype(auto) get(Class&& c)
{
    static_assert(N < 3, "Class has only three elements!");
    if constexpr (N == 2)
    {
        return std::move(c.get_val());
    }
    else if constexpr (N == 1)
    {
        return std::move(c.get_str2());
    }
    else
    {
        return std::move(c.get_str1());
    }
}

int main()
{
    Class c { "str1", "str2", 0 };

    const auto& [cstr1, cstr2, cval] = c;

    /* Prints "str1" */
    std::cout << cstr1 << std::endl;

    /* Prints "str2" */
    std::cout << cstr2 << std::endl;

    /* Prints "0" */
    std::cout << cval << std::endl;
```

```
    auto& [str1, str2, val] = c;
    str1 = "mod str1";
    str2 = "mod str2";
    val = 1;

    auto&& [mstr1, mstr2, mval] = c;

    /* Prints "mod str1" */
    std::cout << mstr1 << std::endl;

    /* Prints "mod str2" */
    std::cout << mstr2 << std::endl;

    /* Prints "mod str1" */
    std::cout << mval << std::endl;

    return 0;
}
```

3.3.4 Template argument deduction

Before C++17 template argument deduction was allowed only for functions. In C++17 this constraint was relaxed and now a compiler can deduce class template parameters as well.

```
std::vector vector {1, 2};
```

This will not work if we use different types, since `std::vector<T>` expects a homogenous type

```
std::vector vector {1, 2.5, "str"};
```

but will work for a variadic-template types

```
std::tuple { 1, 2.5, "str"};
```

Not all differences between functions and classes are resolved in C++17 - it still doesn't support partial template argument deduction but there is a question if it should, since in the following example

```
std::tuple<int> tuple { 0, 1 };
```

we don't know whether it was intentional or accidental to pass two arguments but define just one type.

Type deduction prefers copy initialization if some expression could be interpreted that way

```
std::vector vector1 {1, 2};
/* vector2 = std::vector<int> { 1, 2 } so copy */
std::vector vector2 { vector1 };
```

When the expression cannot be interpreted as a copy, C++17 uses type deduction

```
std::vector vector1 {1, 2};
/* vector2 = std::vector<std::vector<int>> { vector1, vector1 }; */
std::vector vector2 { vector1, vector1 };
```

There is one corner case. Let's look at the example given below

```
#include <vector>

template <typename... Args>
auto make_vector(const Args&... args)
{
    return std::vector { args... };
}

int main()
{
    std::vector vector1 { 1, 2 };
    /* Is it the copy or the vector of vectors? */
    std::vector vector2 = make_vector(vector1);

    return 0;
}
```

In such cases, a compiler producer can decide what to do, as the standard does not define the exact rules to follow.

Deduction guides

Due to the automatic deduction in classes, helper functions like `std::make_pair`, `std::make_unique`, which were designed for type deduction, seem to be redundant. But they still do something more than type deduction - they **decay**.

```
/* [auto] = std::pair<const char*, const char*> */
auto pair = std::make_pair("str1", "str2");
```

Let's have a look at the naive implementation of std::pair

```
template <typename T1, typename T2>
struct Pair
{
    Pair(const T1& t1, const T2& t2): m_first(t1), m_second(t2)
    {
    }

    T1 m_first;
    T2 m_second;
};

int main()
{
    /* Won't compile */
    // Pair p {"str1", "str2"};

    return 0;
}
```

The program above reveals the nature of `auto` - **it decays if only it is working on values but not references**. Because of that, the code presented before is internally working like that

```
/* Passing parameters phase */
const char arg1[5] {"str1"};
const char arg2[5] {"str2"};

/* Initialization - can't be done that way so we got error */
char[5] m_first { arg1 };
char[5] m_second { arg2 };
```

There won't be an error if we pass by the value which decays.

On the other hand, notice that `std::pair` is somehow working with references. It is possible because of **deduction guides**.

```
template <typename T1, typename T2>
struct Pair
{
    Pair(const T1& t1, const T2& t2): m_first(t1), m_second(t2)
    {
    }

    T1 m_first;
    T2 m_second;
};

/* Deduction guide */
template <typename T1, typename T2>
/* No references, so argument will decay */
Pair(T1, T2) -> Pair<T1, T2>;

int main()
{
    Pair p {"str1", "str2"};

    return 0;
}
```

Deduction guides compete with the constructors of a class. Class template argument deduction uses the constructor guide that has the highest priority according to overload resolution. **If a constructor and a deduction guide match equally well, the deduction guide is preferred**. Otherwise, for instance, when no conversion is needed, the constructor is taken.

```
#include <string>

template <typename T>
struct Struct
{
    T value;
};

Struct(const char*) -> Struct<std::string>;

int main()
{
    /* It is std::string inside now */
    Struct { "value" };
```

```
    return 0;
}
```

3.4 Fold expressions

From now on we have two basic ways of using **fold expressions**. As they are counter-intuitive, they must be memorized

1. **left fold**: `...` **op** `args` which is

$$(((arg1 \ \mathbf{op} \ arg2) \ \mathbf{op} \ arg3) \ op \ ...)$$

2. **right fold**: `args` **op** `...` which is

$$(arg1 \ \mathbf{op} \ (arg2 \ \mathbf{op} \ (arg3 \ \mathbf{op} \ (...))))$$

The following program can make it clear

```
#include <iostream>

template <typename... Args>
auto left_sum (const Args&... args)
{
    return (... + args); // ( ( ( arg1 + arg2 ) + arg3 ) + ... )
}

template <typename... Args>
auto right_sum (const Args&... args)
{
    return (args + ...); // ( arg1 + ( arg2 + ( arg3 + (...) ) ) )
}

int main()
{
    /* Prints "0" */
    std::cout << left_sum(1, 2.0, -3) << std::endl;

    /* Prints "0" */
    std::cout << right_sum(1, 2.0, -3) << std::endl;

    return 0;
}
```

Sometimes left or right fold matters. Assume we've invoked `left_sum` function like

```
left_sum("str1", "str2", std::string {"str3"});
```

the code won't compile - the expression is unfolded to

```
((const char* + const char*) + std::string)
```

which doesn't work since `const char* + const char*` is not defined. On the other hand

```
right_sum("str1", "str2", std::string {"str3"});
```

compiles, because the expression unfolded is

```
const char* + (const char* + std::string)
```

and this equals

```
(const char* + std::string) = std::string.
```

We should remember that comma operator **not always must be used!**. Let's have a look at this example

```
template <std::size_t N>
using thread_pool = std::array<std::thread, N>;

template <typename... Functions>
auto create_threads(Functions&&... functions)
{
    return thread_pool<sizeof...(Functions)>
    {
        std::thread{ std::forward<Functions>(functions) }...
    };
}
```

If we used

```
return thread_pool<sizeof...(Functions)>
{
    /* Comma operator used */
    std::thread{ std::forward<Functions>(functions) }, ...
};
```

we would get only **only last thread**. This is because of the comma operator behavior - as we already know, the comma operator `((expr1, expr2, ...))` evaluates all its operands in left-to-right order and returns the result of the last operand. In this case, we're constructing `std::thread` objects, but we're not actually returning them or storing them in the array properly. The result of this expression is just the last `std::thread` object, not a sequence of threads.

3.4.1 Deal with empty parameters pack

Neither `left_sum` or `right_sum` work with empty parameters pack. That can be fixed with some trick

```
template <typename... Args>
auto left_sum (const Args&... args)
{
    /*  ( ( ( arg1 + arg2 ) + arg3 ) + ... + 0 ) */
    return (0 + ... + args);
}

template <typename... Args>
auto right_sum (const Args&... args)
{
    /* ( 0 + ( arg1 + ( arg2 + ( arg3 + (...) ) ) ) ) */
    return (args + ... 0);
}
```

We've added explicit value which will be returned if an empty parameter pack is passed.

Both compile-time `if` and fold expressions shall be used with `type_traits` to reduce the amount of the code.

3.5 Miscellaneous

C++17 provided some more elements than discussed above, but they are „one-liners" which can be easily found in the preference. They are here just to point out their existence

1. New attributes like `[[fallthrough]]`, `[[nodiscard]]` etc.
2. Defined expression evaluation order.
3. Nested namespaces.
4. New classes `std::optional`, `std::variant`, `std::any`, `std::byte`, `std::string_view`, etc.
5. New algorithms `std::size()`, `std::empty()`, `std::clamp()`, etc.
6. `std::vector`, `std::list`, `std::forward_list` support incomplete types.

Chapter 4

Move semantics

4.1 Basics

4.1.1 *RVO*

RVO - Return Value Optimization is a compiler technique to avoid copying an object that a function returns as its value, including avoiding the creation of a temporary object. This optimization permits a function to efficiently return large objects while also simplifying the function's interface and eliminating scope for issues such as resource leaks.

```
#include <iostream>

struct Data
{
    Data()
    {
        std::cout << "Default constructor" << std::endl;
    }

    Data(const Data&)
    {
        std::cout << "const Data& constructor" << std::endl;
    }

    Data(Data&&)
    {
        std::cout <<"Data&& constructor" << std::endl;
    }

    Data& operator=(const Data&)
    {
        std::cout << "const Data& operator" << std::endl;
        return *this;
    }

    Data& operator=(Data&&)
    {
        std::cout << "Data& operator" << std::endl;
        return *this;
    }

    int value { 0 };
```

```
};

Data get()
{
    /* Unnamed object returned */
    return Data {};
}

int main()
{
    /* Prints "Default constructor" */
    Data data = get();
    return 0;
}
```

In the example above, neither copy constructor (copy operator) nor move constructor (move operator) is called when `get()` returns its value. It would work even if all constructors (except the default one) and all assign operators were deleted. The temporary object is „stolen" by `data` variable without doing a copy.

We may apply *RVO* if the object returned from a function is a **prvalue** expression. This expression must have the same type as the type of the function's return value by signature, without `cv`-qualifiers. Besides, with C++17, *RVO* is no longer an optimization, but a rule that compilers must follow. This rule is applied even if a move/copy constructor has side effects.

4.1.2 *NRVO*

NRVO - Named Return Value Optimization which is *RVO* where the temporary object to be returned has its name.

```
#include <iostream>

struct Data
{
    Data()
    {
        std::cout << "Default constructor" << std::endl;
    }

    Data(const Data&)
    {
        std::cout << "const Data& constructor" << std::endl;
    }

    Data(Data&&)
    {
        std::cout <<"Data&& constructor" << std::endl;
    }

    Data& operator=(const Data&)
    {
        std::cout << "const Data& operator" << std::endl;
        return *this;
    }
```

```
    Data& operator=(Data&&)
    {
        std::cout << "Data& operator" << std::endl;
        return *this;
    }

    int value { 0 };
};

Data get()
{
    /* Named object returned */
    Data data;
    data.value = 1;
    return data;
}

int main()
{
    /* Prints "Default constructor" */
    Data data = get();
    return 0;
}
```

Effectively, *NRVO* transforms the code by using `new` and `delete` operators which may look like this

```
Data* get()
{
    Data* data = new Data {};
    data->value = 1;
    return data;
}

int main()
{
    Data* data = get();
    delete data;
    return 0;
}
```

Remember that we can apply *NRVO* only when the type of the returned object and the type of the object returned according to the function signature coincide completely!

A common mistake disabling *NRVO* is to use `std::move()` - it is *anti-pattern* and has to be avoided being **pessimization** (opposite to **optimization**)

```
Data get()
{
    Data data;
    data.value = 1;

    /* It disables NRVO! */
    return std::move(data);
}
```

Here we tell the compiler that the returned object must be cast to the rvalue object, so it will use the move constructor or the move assign operator which can even be deleted for *NRVO*. Moreover, the returned type is `Data&&` now instead of `Data` so *NRVO* cannot be applied.

4.1.3 *SSO*

SSO - Small String Optimization is an optimization that the code uses an internal `char` buffer to store contents of small strings (i.e. their size is smaller than some value) to eliminate dynamic allocation of the memory.

4.1.4 Move semantics rules

We can follow a few simple rules, to have a clean way of using move semantics

- **Avoid objects with names**

```
/* BAD */
std::string str { "string" };
print(str);
// str is not used later


/* GOOD */
print(std::string { "string" };
```

- **Use `std::move()` for objects with names**

```
std::string str { "string" };
print(std::move(str));
```

- **Use `std::move()` to initialize members of a class when possible**. Before move semantics the code would look like that one

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

struct Data
{
    Data(const std::string& str, const std::vector<int>& vec):
        m_str ( str ),
        m_vec ( vec )
    {
    }

    std::string m_str;
    std::vector<int> m_vec;
};

int main()
{
    /* "string" is const char* object, so will be copied to str,
     * which is be copied to m_str;
     * {1, 2, 3} is an initializer list which creates
     * a new vector vec, which is be copied to m_vec;
```

```
     */
    Data data { "string", {1, 2, 3} };

    return 0;
}
```

Having that possibility, we can move the arguments to be members

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

struct Data
{
    Data(std::string str, std::vector<int> vec):
        m_str ( std::move(str) ),
        m_vec ( std::move(vec) )
    {
    }

    std::string m_str;
    std::vector<int> m_vec;
};

int main()
{
    /* "string" is const char* object, so will be copied to str,
     * and then moved m_str;
     * {1, 2, 3} is an initializer list which creates
     * a new vector vec but moved to m_vec;
     */
    Data data { "string", {1, 2, 3} };

    return 0;
}
```

To summarize it - if there is no important reason, **we shouldn't use the constructor with simple `const &` parameter type**. One of the special cases may be `std::array` which hasn't cheap move semantics implemented or if we already have a value and we want just to modify it within a class.

- **Avoid unnecessary `std::move()`**
  - **Never return local objects with `std::move()`**

```
/* BAD */
std::string get()
{
    std::string str { "string" };
    /* ... some actions on str ... */
    return std::move(str);
}

/* GOOD */
std::string get()
{
    std::string str { "string" };
```

```
    /* ... some actions on str ... */
    return str; // NRVO active!
}
```

– **Never use `std::move()` if we already have a temporary object**

```
/* BAD */
std::string str { std::move(get()) };

/* GOOD */
std::string str { get() };
```

4.1.5 Noexcept and move semantics

The C++ standard introduced a rule called **the strong exception handling guarantee**. Let's say we want to add something to the vector by calling `push_back()` function and then an exception arises. **If we used just moving semantics, there is no option to *rollback*** - by moving we destroyed the original object, so it cannot be restored; but if we used copying, we just remove the last object which caused the exception, and we can continue the program execution.

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

struct Info
{
public:
    Info(const char* msg): m_msg ( msg )
    {
        std::cout << "Default constructor for: "
                  << m_msg << std::endl;
    }

    Info(const Info& info): m_msg ( info.m_msg )
    {
        std::cout << "const Info& constructor for: "
                  << m_msg << std::endl;
    }

    Info(Info&& info): m_msg ( std::move(info.m_msg) )
    {
        std::cout << "Info&& constructor for: "
                  << m_msg << std::endl;
    }

    std::string get_msg() const
    {
        return m_msg;
    }

private:
    std::string m_msg;
};

int main()
{
```

```
    std::vector<Info> infos
    {
        "Everything ok",
        "Exception thrown!",
        "Operation aborted"
    };

    /* Prints 3 */
    std::cout << "Capacity: " << infos.capacity() << std::endl;

    /* Printouts:
     * "Info&& constructor for: Additional info!"
     * "const Info& constructor for: Everything ok"
     * "const Info& constructor for: Exception thrown!"
     * "const Info& constructor for: Operation aborted"
     */
    infos.push_back("Additional info!");

    /* Prints "6" */
    std::cout << "Capacity: " << infos.capacity() << std::endl;

    return 0;
}
```

As expected, only the copy constructor has been used. We can do something with it - **we can *give the guarantee* for the compiler that the moving constructor doesn't throw.**

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

struct Info
{
public:
    Info(const char* msg): m_msg ( msg )
    {
        std::cout << "Default constructor for: "
                  << m_msg << std::endl;
    }

    Info(const Info& info): m_msg ( info.m_msg )
    {
        std::cout << "const Info& constructor for: "
                  << m_msg << std::endl;
    }

    Info(Info&& info) noexcept:
        m_msg ( std::move(info.m_msg) )
    {
        std::cout << "Info&& constructor for: "
                  << m_msg << std::endl;
    }

    std::string get_msg() const
    {
```

```
        return m_msg;
    }

private:
    std::string m_msg;
};

int main()
{
    std::vector<Info> infos
    {
        "Everything ok",
        "Exception thrown!",
        "Operation aborted"
    };

    /* Prints 3 */
    std::cout << "Capacity: " << infos.capacity() << std::endl;

    /* Printouts:
     * "Info&& constructor for: Additional info!"
     * "Info&& constructor for: Everything ok"
     * "Info&& constructor for: Exception thrown!"
     * "Info&& constructor for: Operation aborted"
     */
    infos.push_back("Additional info!");

    /* Prints "6" */
    std::cout << "CAPACITY: " << infos.capacity() << std::endl;

    return 0;
}
```

The code above works well, but the question arises - is it okay to mark move constructor with `noexcept` keyword? If we throw it there, despite the guarantee of no exceptions, the program will call `std::terminate()` immediately, so it would be better to create a **conditional noexcept declaration**

```
Info(Info&& info) noexcept(
    std::is_nothrow_move_constructible_v<std::string>
    && noexcept(std::cout << m_msg)):
    m_msg ( std::move(info.m_msg) )
{
    std::cout << "Info&& constructor for: "
              << m_msg << std::endl;
}
```

If we add it, the compiler will get back to copying instead of moving, since `std::cout` can throw. But there is a positive aspect. If we just type

```
Info (Info&&) = default;
```

the compiler will detect `noexcept` guarantees by itself, and then **no copying will be enabled**. Therefore, if we implement a move constructor explicitly, **we should declare whether and when it throws**; otherwise **we shouldn't specify anything at all**.

4.1.6 Moved-from objects

Let's start with the fact that **`std::move()` doesn't move anything**. It semantically means *I no longer need this value here*. The `std::move()` only marks the object to be movable but doesn't move anything. It allows the implementation of the call to benefit from this mark by performing some optimizations. **Whether the value is moved is something the caller doesn't know** and because of that, after `std::move()` we should treat the object as valid but in an unspecified state

```
#include <iostream>
#include <string>

int main()
{
    std::string str1 {"Move only"};

    /* Still prints "Move only" probably */
    std::move(str1);
    std::cout << str1 << std::endl;


    std::string str2 {"Move and assign"};
    auto from_string = std::move(str2);

    /* Prints nothing probably */
    std::cout << str2 << std::endl;

    return 0;
}
```

In both scenarios above, it is guaranteed that `str` is still a valid `std::string` object on which we can perform any action of string, unfortunately, we don't know the initial value - the standard says nothing if the string keeps old value or was cleared. The only guarantee we have is that the **moved-from** object is not (even partially) destroyed for which at least the destructor will be called.

The reason for keeping moved-from objects undestroyed is their reusable benefits, for instance quite cheap implementation of `swap`

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

/* Helpers */
struct Timer
{
    Timer() : m_begin(std::chrono::high_resolution_clock::now())
    {
    }

    ~Timer()
    {
        using namespace std::chrono;
        using ms = std::chrono::milliseconds;

        auto end = high_resolution_clock::now();
```

```
        auto value = duration_cast<ms>(end - m_begin);
        std::cout << "Time: " << value.count() << " ms" << std::endl;
    }

    std::chrono::time_point<std::chrono::high_resolution_clock> m_begin;
};

void fill(std::vector<int>& a, std::vector<int>& b)
{
    std::size_t max = 10000000;
    for (std::size_t i = 0; i < max; i++)
    {
        a.push_back(i);
        b.push_back(max-i);
    }
}

/* Main functions */
template <typename T>
void lswap(T& a, T& b)
{
    Timer timer {};

    T tmp { a };
    a = b;
    b = tmp;
}

template <typename T>
void rswap(T& a, T& b)
{
    Timer timer {};

    T tmp { std::move(a) };
    a = std::move(b);
    b = std::move(tmp);
}

int main()
{
    std::vector<int> a;
    std::vector<int> b;
    fill(a, b);

    lswap(a, b);

    /* Usually much faster than lswap */
    rswap(a, b);

    return 0;
}
```

It is worth to mention a special case of a moved-from object when we move the object to itself

```
auto value = std::move(value);
```

As was said, we know that the `value` is still valid, but in this case, we can't say anything about its state.

To recap, a moved-from object has to fulfill a few conditions

1. it needs to be destructible,
2. it needs to support the assignment of a new value to it,
3. it needs to be copyable, movable, and assignable to other objects.

Especially the first condition is important. For instance `std::future` is equipped with a special function `valid()` that returns `false` if the object is moved-from object. Now let's have a look at a badly-designed object

```
#include <array>
#include <exception>
#include <chrono>
#include <iostream>
#include <thread>

class Tasks
{
public:
    Tasks() = default;

    /* Copying disabled */
    Tasks(Tasks&&) = default;
    Tasks& operator=(Tasks&&) = default;

    template <typename T>
    void start(T operation)
    {
        m_threads[m_numThreads] = std::thread { std::move(operation) };
        ++m_numThreads;
    }

    ~Tasks()
    {
        for (int i = 0; i < m_numThreads; i++)
        {
            m_threads[i].join();
        }
    }

private:
    std::array<std::thread, 10> m_threads;
    int m_numThreads { 0 };
};

int main()
{
    try
    {
        Tasks tasks;
        tasks.start([]
```

```
        {
            std::this_thread::sleep_for(std::chrono::seconds { 2 });
            std::cout << "Action1 done" << std::endl;
        });

        tasks.start([]
        {
            std::cout << "Action2 done" << std::endl;
        });

        /* PROBLEM IS HERE!!! */
        Tasks other { std::move(tasks) };
    }
    catch (const std::exception& e)
    {
        std::cout << "EXCEPTION CAUGHT: " << e.what() << std::endl;
    }

    return 0;
}
```

The code part

```
Tasks other { std::move(tasks) };
```

moves the container with threads, so the `Tasks` object doesn't contain them anymore. But when the object is about to end its life, destructor is trying to perform joining threads, which fails as there are no threads to be joined.

We have two ways to fix it keeping the move semantics

1. We can check if threads can be joined by calling `joinable()` function in the destructor.
2. We can replace the default implementation of the move constructor and the move operator with a customized one.

4.1.7 Copying as fallback

When a function provides special implementation taking non-`const` rvalue reference, a compiler can optimize the copying of a value by „stealing" the value from the source. However, if there is no optimized version of a function for the move semantics, then the usual copying is used as a fallback

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

struct Data
{
    Data() {}

    void insert(const std::string& str)
    {
        m_data.push_back(str);
    }
```

```
    std::vector<std::string> m_data;
};

int main()
{
    Data data;
    std::string str { "string" };

    /* It uses copy semantics */
    data.insert(std::move(str));

    /* str value hasn't changed probably */
    std::cout << str << std::endl;

    return 0;
}
```

For the generic code, it is important that we can always mark an object with `std::move()` if we no longer need its value. The corresponding code compiles even if there is no move semantics support. For the same reason, we can even mark objects of fundamental data type such as `int` or raw pointer with `std::move()`.

One more important thing is that objects declared with `const` cannot be moved because any optimizing implementation requires that the passed argument can be modified.

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

struct Data
{
    Data() = default;

    void insert(const std::string& str)
    {
        std::cout << "const std::string&" << std::endl;
        m_data.push_back(str);
    }

    void insert(std::string&& str)
    {
        std::cout << "std::string&&" << std::endl;
        /* std::move() needed - str is lvalue here! */
        m_data.push_back(std::move(str));
    }

    std::vector<std::string> m_data;
};

int main()
{
    Data data;
    std::string str { "string" };
    const std::string cstr { "cstring" };
```

```
    /* Prints "std::string&&" */
    data.insert(std::move(str));

    /* Prints "const std::string&" */
    data.insert(std::move(cstr));

    /* Empty probably */
    std::cout << str << std::endl;

    /* Keeps its value so prints "cstring" */
    std::cout << cstr << std::endl;

    return 0;
}
```

Notice that an object marked with `std::move()` can still be passed to a function that takes an ordinary lvalue reference and keeps its value. It will not work with non-`const` lvalue reference and the code won't compile.

4.1.8 Function overloading - short summarization

1. **`function(T value)`** can handle **any of type `T` but it will copy unless we use move semantics explicitly**
2. **`function(T& value)`** can handle **modifiable named objects** only.
3. **`function(const T& value)`** can handle
   - **modifiable named objects**,
   - **`const` named objects**,
   - **temporary objects that don't have a name**,
   - **object marked by `std::move()`**.
4. **`function(T&& value)`** can handle
   - **temporary objects that don't have a name**,
   - **non-`const` objects market with `std::move()`**.
5. **`function(const T&& value)`** can handle
   - **temporary objects that don't have a name**,
   - **`const` or non-`const` objects market with `std::move()`**.

   However, there is no useful semantic meaning in this case - from its definition, an rvalue reference steals the value, but being `const` at the same time, it can't do it. Nonetheless, it will work with `const` objects

```
#include <iostream>
#include <string>

void print(const std::string& str)
{
    std::cout << "const std::string&:" << str << std::endl;
}
```

```
void print(const std::string&& str)
{
    std::cout << "const std::string&&:" << str << std::endl;
}

int main()
{
    const std::string str { "string" };

    /* Prints "const std::string&:string" */
    print(str);

    /* Prints "const std::string&&:string" */
    print(std::move(str));

    return 0;
}
```

This behavior is usually covered by `const` reference, but there exist a small number of examples that employ it, like `std::optional`.

4.2 Classes

Move semantics in classes can be disabled by mistake. **If any of the following special member functions are declared by the user (*explicitly declared*, even with `default` keyword), class objects would be not movable**

- Copy constructor.
- Copy assignment operator.
- Another move operation.
- Destructor.

Assuming that, the following declaration

```
struct Data
{
        ~Data() = default;
};
```

will disable move semantics. As a consequence, **a polymorphic base class has move semantics disabled**

```
struct BaseClass
{
    virtual ~BaseClass() {}
};
```

The code is working anyway, because of copying as fallback. Anyway, due to the fact mentioned a moment ago remember to **not implement your own destructor unless it is necessary!**

4.2.1 How to implement moving semantics in a class

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

struct Data
{
    Data():
        m_str (),
        m_vec ()
    {
        std::cout << "Default constructor" << std::endl;
    }

    Data(const Data& data):
        m_str ( data.m_str ),
        m_vec ( data.m_vec )
    {
        std::cout << "const Data& constructor" << std::endl;
    }

    Data& operator=(const Data& data)
    {
        if (this == &data)
        {
            return *this;
        }

        m_str = data.m_str;
        m_vec = data.m_vec;
        std::cout << "const Data& operator" << std::endl;
        return *this;
    }

    Data(Data&& data):
        m_str ( std::move(data.m_str) ),
        m_vec ( std::move(data.m_vec) )
    {
        std::cout << "Data&& constructor" << std::endl;
    }

    Data& operator=(Data&& data)
    {
        if (this == &data)
        {
            return *this;
        }

        m_str = std::move(data.m_str);
        m_vec = std::move(data.m_vec);
        std::cout << "Data&& operator" << std::endl;
        return *this;
    }

    std::string m_str;
```

```
    std::vector<int> m_vec;
};

int main()
{
    /* Prints "Default constructor" */
    Data data0;

    /* Prints "const Data& constructor" */
    Data data1 { data0 };

    /* Prints "const Data& operator" */
    data0 = data1;

    /* Prints "Data&& constructor" */
    Data data2 { std::move(data0) };

    /* Prints "Data&& operator" */
    data2 = std::move(data1);

    return 0;
}
```

Keep in mind that move semantics is not passed through. When we initialize the members in the move constructor, we have to mark them with `std::move()`; otherwise, we would just copy them. Analogous comments apply to the move assignment operator.

4.2.2 Reference qualifiers

Before we start, let's have a short look at range-`for` loop implementation in the pseudocode

```
for (range_declaration : range_expression)
{
  auto&& __range = range_expression;
  begin_expr = std::begin(__range);
  end_expr = std::end(__range);
  for (auto __begin = begin_expr,
            __end = end_expr;
            __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

We will discuss the `auto&&` statement later, for now, it is enough to observe, that `range_expression` initializes `__range` variable. If it is a pure value, we copy it but if it is a reference, we initialize `__range` object with the given type of reference

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

struct Data
{
    Data():
        m_str ( "abcdefghijklmnoprstuwvz" )
```

```
    {
    }

    std::string value_get() const
    {
        return m_str;
    }

    const std::string& cref_get() const
    {
        return m_str;
    }

    std::string m_str;
};

Data make_data()
{
    return Data {};
}

int main()
{
    /* value_get() returns a value so we have a copy inside
     * of a for loop. Then, Data dies but copy persists.
     */
    /* Prints "abcdefghijklmnoprstuwvz" */
    for (char c : make_data().value_get())
    {
        std::cout << c;
    }
    std::cout << std::endl;

    /* cref_get() returns the reference, so we have a reference
     * inside of a for loop. Then, Data dies thus everything
     * can happen.
    */
    /* Prints whatever it needs */
    for (char c : make_data().cref_get())
    {
        std::cout << c;
    }
    std::cout << std::endl;

    return 0;
}
```

To solve the problem, it would be nice to have a function that automatically detects if it works on a temporary object. That is the reason why C++ standard provided **reference qualifiers**.

The code above can be corrected by implementing `get_cref()` function as `get_rref() &&`

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

```
struct Data
{
    Data():
        m_str ( "abcdefghijklmnoprstuwvz" )
    {
    }

    std::string value_get() const
    {
        return m_str;
    }

    /* Object is no longer needed! */
    std::string rref_get() &&
    {
        return m_str;
    }

    std::string m_str;
};

Data make_data()
{
    return Data {};
}

int main()
{
    /* Prints "abcdefghijklmnoprstuwvz" */
    for (char c : make_data().value_get())
    {
        std::cout << c;
    }
    std::cout << std::endl;

    /* Prints "abcdefghijklmnoprstuwvz" */
    for (char c : make_data().rref_get())
    {
        std::cout << c;
    }
    std::cout << std::endl;

    return 0;
}
```

We can add two special qualifiers to each member function of class

- **&** for lvalue objects.
- **&&** for rvalue objects when the object (member) is no longer needed.

The qualifiers after the parameters list in parenthesis allow us to qualify one more object that is not passed as a parameter - the object we call the method. From now on we can have functions dedicated to temporary objects and other ones.

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

```
struct Data
{
    Data() {};

    void get() &
    {
        std::cout << "get() &" << std::endl;
    }

    void get() &&
    {
        std::cout << "get() &&" << std::endl;
    }

    void get() const &
    {
        std::cout << "get() const&" << std::endl;
    }

    void get() const &&
    {
        std::cout << "get() const&&" << std::endl;
    }
};

int main()
{
    Data data;

    /* Prints "get() &" */
    data.get();

    /* Prints "get() &&" */
    std::move(data).get();

    /* Prints "get() &&" */
    Data{}.get();


    const Data cdata;

    /* Prints "get() const&" */
    cdata.get();

    /* Prints "get() const&&" */
    std::move(cdata).get();

    return 0;
}
```

Usually, we have only two or three of these overloads (`&&` and `const &` for getters). Note that **overloading for both reference and non-reference qualifiers is not allowed, so a compilation of the following code**

```
struct Data
{
```

```
    Data() = default;

    void get() && {}
    void get() const {}
};
```

gets the error *error: 'void Data::get() const' cannot be overloaded with 'void Data::get() &&'*.

When to use qualifiers

Generally, rvalue reference qualifiers should be used in any place where an object is modified - there is no need to perform some actions on the object which are about to die. **As standard suggests it could be better to declare the assignment operator with reference qualifiers wherever we can.**

```
struct Data
{
    Data () = default;
    Data& operator=(const Data&) = default;
};

Data Get()
{
    return Data {};
}

int main()
{
    /* It works but makes no sense */
    Get() = Data {};

    return 0;
}
```

To avoid such bizarre activities, we can add a reference qualifier

```
struct Data
{
    Data () = default;

    /* The qualifier & added */
    Data& operator=(const Data&) & = default;
};

Data Get()
{
    return Data {};
}

int main()
{
    /* error: passing 'Data' as 'this' argument
     * discards qualifiers [-fpermissive]
     */
    // Get() = Data {};
```

```
    return 0;
}
```

4.3 Perfect forwarding

Let's start with looking at the motivating example where we don't use the perfect forwarding

```
#include <iostream>
#include <string>

namespace detail
{
    void print(const std::string&)
    {
        std::cout << "const std::string&" << std::endl;
    }

    void print(const std::string&&)
    {
        std::cout << "const std::string&&" << std::endl;
    }

    void print(std::string&)
    {
        std::cout << "std::string&" << std::endl;
    }

    void print(std::string&&)
    {
        std::cout << "std::string&&" << std::endl;
    }
}  // namespace detail

void print(const std::string& str)
{
    detail::print(str);
}

void print(const std::string&& str)
{
    detail::print(str);
}

void print(std::string& str)
{
    detail::print(str);
}

void print(std::string&& str)
{
    detail::print(str);
}
```

```
int main()
{
    const std::string cstr { "cstring" };

    /* Prints "const std::string&" */
    print(cstr);

    /* Prints "const std::string&" - parameter not forwarded! */
    print(std::move(cstr));

    std::string str { "string" };

    /* Prints "const std::string&" */
    print(str);

    /* Prints "const std::string&" - parameter not forwarded */
    print(std::move(str));

    return 0;
}
```

As we see, the function `print(std::move(cstr))` doesn't call the function `detail::print(const std::string&&)` which is similar to the case of `print(std::move(str))` that doesn't invoke `detail::print(std::string&&)`. A quick fix would be

```
void print(const std::string&& str)
{
    detail::print(std::move(str));
}

void print(std::string&& str)
{
    detail::print(std::move(str));
}
```

but what if we had three arguments instead of one? In such a case, we would implement 81 overloads, definitely too much. That's why C++ has a special formula called **perfect forwarding**.

Perfect forwarding uses the same syntax as rvalue reference which is misleading at first glance. [1] Because of that, we need to have a clear view of what the perfect forwarding is. To enable perfect forwarding

1. The type of parameter has to be a template parameter of the function.
2. Call parameter must be taken as a **pure reference**, so without cv-qualifiers.
3. Parameter should be forwarded by calling `std::forward` function.

The correction of the motivating example enriched with perfect forwarding looks as this

[1] There was a proposition to use `&&&`, finally refused which seems not to be a good decision.

```cpp
#include <iostream>
#include <string>

namespace detail
{
    void print(const std::string&)
    {
        std::cout << "const std::string&" << std::endl;
    }

    void print(const std::string&&)
    {
        std::cout << "const std::string&&" << std::endl;
    }

    void print(std::string&)
    {
        std::cout << "std::string&" << std::endl;
    }

    void print(std::string&&)
    {
        std::cout << "std::string&&" << std::endl;
    }
}  // namespace detail

/* Perfect forwarding */
template <typename T>
void print(T&& arg)
{
    detail::print(std::forward<T>(arg));
}

int main()
{
    const std::string cstr { "cstring" };

    /* Prints "const std::string&" */
    print(cstr);

    /* Prints "const std::string&&" */
    print(std::move(cstr));

    std::string str { "string" };

    /* Prints "std::string&" */
    print(str);

    /* Prints "std::string&&" */
    print(std::move(str));

    return 0;
}
```

We pointed out that `&&` might give the impression that the usual rules for rvalue references apply. However not in this case. An rvalue reference, not qualified with `const` and

`volatile`, of function template parameter doesn't follow the rules of ordinary rvalue references. This type of reference is called a **universal reference** or **forwarding reference** and

- it can universally bind to objects of all types (`const`, non-`const`) and value categories,
- it is usually used to forward arguments.

The most important rule to remember is that **the universal reference doesn't have any cv-qualifiers, so adding one of `const` or `volatile` before the `&&` qualifier creates usual rvalue reference!**

As a summarization, if we have a function

```
template <typename T>
return_type function(T&& arg);
```

where `return_type` is some type, the type `T&&` is

- lvalue reference, if we refer to lvalue;
- rvalue reference, if we refer to rvalue.

The fact that ordinary rvalue references and universal references share the same syntax creates multiple problems. The most important is that we cannot declare a universal reference of a specific type. However, since C++17 we can use the following workaround (in C++20 we can have `requires` instead)

```
#include <iostream>
#include <type_traits>

using namespace std;

template <typename T,
          typename = enable_if_t<is_convertible_v<T, string>>>
void print(T&& arg)
{
    cout << "Argument: " << arg << endl;
}

int main()
{
    print("string");
    /* Won't compile since int cannot be converted to std::string */
    // print(1);

    return 0;
}
```

One more example of misleading rvalue references and universal references can be pointed out - **an rvalue reference to template parameter of class is not a universal reference!**

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

```cpp
template <typename T>
struct Data
{
    Data() = default;

    void insert(T&& value)
    {
        m_items.push_back(std::forward<T>(value));
    }

    std::vector<T> m_items;
};

int main()
{
    Data<int> data;
    const int value = 0;

    /* Won't work since insert() expects rvalue! */
    // data.insert(value);
    return 0;
}
```

but of course we can fix this using `<type_traits>` by declaring the function insert as template one

```cpp
template <typename U = T,
          typename = std::enable_if_t<std::is_convertible_v<U, T>>>
void insert(U&& value)
{
    m_items.push_back(std::forward<U>(value));
}
```

To finish this part, it also shouldn't be a surprise that partial specialization also doesn't work like universal references.

4.3.1 Overload resolution with universal references

The following table presents the precedence of overload resolution of the function. The lower the value of the number the higher the priority of resolution we have (so 1 is the higher one). The *no* value means that there is no possible call.

Call	`f(X&)`	`f(const X&)`	`f(X&&)`	`f(const X&&)`	`template <typename T> f(T&&)`
`f(v)`	1	3	no	no	2
`f(c)`	no	1	no	no	2
`f(X{})`	no	4	1	3	2
`f(std::move(v))`	no	4	1	3	2
`f(std::move(c))`	no	3	no	1	2

As we see, **the universal reference is always the second-best option**. A perfect match is always better, but the need to convert the type (even making `const` or converting rvalue to lvalue) is a worse match than just instantiating the function template for an exact match.

The fact that a universal reference binds better than a type conversion in overload resolution has a very nasty side effect - if we have a constructor that takes a single universal reference, this is a better match than:

- the copy constructor if passing a non-`const` object,
- the move constructor if passing a `const` object.

```
#include <iostream>
#include <string>

struct Data
{
    Data() = default;

    Data(const Data&)
    {
        std::cout << "const Data& constructor" << std::endl;
    }

    Data(Data&&)
    {
        std::cout << "Data&& constructor" << std::endl;
    }

    template <typename T>
    Data(T&&)
    {
        std::cout << "Universal constructor" << std::endl;
    }
};

int main()
{
    const Data cdata;
    Data data;

    /* Prints "const Data& constructor" */
    Data copy0 { cdata };

    /* Prints "Universal constructor" */
    Data copy1 { data };

    /* Prints "Universal constructor" because of const qualifier */
    Data move0 { std::move(cdata) };

    /* Prints "Data&& constructor" */
    Data move1 { std::move(data) };

    return 0;
}
```

4.3.2 Universal reference collapsing rule

When universal reference is used, the C++ standard applies the following rule of reference resolution

- **(T&)&** is **T&**,
- **(T&&)&** is **T&&**,
- **(T&)&&** is **T&**,
- **(T&&)&&** is **T&&**.

4.3.3 std::move and std::forward

Let's start with possible implementations of these functions. The `std::forward` function can look like that one

```
template <typename T>
inline T&& forward(typename std::remove_reference<T>::type& t) noexcept
{
    return static_cast<T&&>(t);
}
```

when `std::move()` would be implemented like this

```
template <typename T>
typename remove_reference<T>::type&& move(T&& arg)
{
    return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

Since `std::forward` is a template function we can directly cast one type to another using collapsing rules

- `std::forward<T>(t)` = `static_cast<T&&>(t)`
- `std::forward<T&>(t)` = `static_cast<(T&)&&>(t)`
  = `static_cast<T&>(t)`
- `std::forward<const T&>(t)` = `static_cast<(const T&)&&>(t)`
  = `static_cast<const T&>(t)`
- `std::forward<T&&>(t)` = `static_cast<(T&&)&&>(t)`
  = `static_cast<T&&>(t)`

The important question is when and where we should use these functions. A quick guide might be helpful

- This is **rvalue reference**

  ```
  void function(std::string&& value);
  ```

  Here we use `std::move()` so with any **non-template** type (like `int`, `std::string` etc.)

- This is **universal/forwarding reference**

  ```
  template <typename T>
  void function(T&& value);
  ```

  Here we use `std::forward` so with any template type.

4.3.4 Problems with the universal reference

The common problem with the universal reference is when a compiler needs to deduce the type of the function having the second parameter of the same type as universal reference

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

template <typename T>
void insert(std::vector<T>& vector, T&& value)
{
    vector.push_back(value);
}

int main()
{
    std::vector<int> vector;
    int value = 0;

    /* Won't compile due to conflicting type deduction:
     *  - std::vector<T> deduces T as int
     * - T&& being universal reference deduces T as const int&
    */
    // insert(vector, value);

    return 0;
}
```

Thankfully, we have at least two options to fix it. First one is named as `insert_1()` and employs `type_traits`, the second one (`insert_2()`) just uses the second template parameter.

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

template <typename T>
void insert_1(std::vector<std::remove_reference_t<T>>& vector,
              T&& value)
{
    vector.push_back(value);
}

template <typename T1, typename T2>
void insert_2(std::vector<T1>& vector, T2&& value)
{
    vector.push_back(value);
}

int main()
{
    std::vector<int> vector;
    int value = 0;

    insert_1(vector, value);
    insert_2(vector, value);
```

```
    return 0;
}
```

4.3.5 Universal reference and `auto`

To save a calculated value and then forward it perfectly to the function, we have to use `auto&&` expression. Let's have a short look at the motivating example

```
#include <iostream>
#include <string>

const std::string& get(const std::string& str)
{
    return str;
}

std::string& get(std::string& str)
{
    return str;
}

std::string&& get(std::string&& str)
{
    return std::move(str);
}

const std::string&& get(const std::string&& str)
{
    return std::move(str);
}

void print(const std::string&)
{
    std::cout << "const std::string&" << std::endl;
}

void print(std::string&)
{
    std::cout << "std::string&" << std::endl;
}

void print(const std::string&&)
{
    std::cout << "const std::string&&" << std::endl;
}

int main()
{
    const std::string cstr { "cstring" };
    std::string str { "string" };

    /* auto = std::string */
    auto value1 { get(cstr) };

    /* auto = std::string */
```

```
    auto value2 { get(str) };

    /* auto = std::string */
    auto value3 { get(std::move(cstr)) };


    /* auto = std::string */
    auto value4 { get(std::move(str)) };

    /* All these functions print "std::string&" since
     * each value is non-const std::string
     */
    print(value1);
    print(value2);
    print(value3);
    print(value4);

    return 0;
}
```

In this example, all of `auto`s don't store any information about the referenceness and constness of the type, [2] thus we need some mechanism to correct it. As was mentioned below 4.2.2, such mechanism is `auto&&` being just another type of forwarding or universal reference.

```
#include <iostream>
#include <string>

const std::string& get(const std::string& str)
{
    return str;
}

std::string& get(std::string& str)
{
    return str;
}

std::string&& get(std::string&& str)
{
    return std::move(str);
}

const std::string&& get(const std::string&& str)
{
    return std::move(str);
}

void print(const std::string&)
{
    std::cout << "const std::string&" << std::endl;
}

void print(std::string&)
```

[2] `auto` decays

```
{
    std::cout << "std::string&" << std::endl;
}

void print(const std::string&&)
{
    std::cout << "const std::string&&" << std::endl;
}

void print(std::string&&)
{
    std::cout << "std::string&&" << std::endl;
}

int main()
{
    const std::string cstr { "cstring" };
    std::string str { "string" };

    /* auto&& = const std::string& */
    auto&& value1 { get(cstr) };

    /* auto&& = std::string& */
    auto&& value2 { get(str) };

    /* auto&& = const std::string&& */
    auto&& value3 { get(std::move(cstr)) };

    /* auto&& = std::string&& */
    auto&& value4 { get(std::move(str)) };

    /* Notice how to use std::forward here! */
    /* Prints "const std::string& " */
    print(std::forward<decltype(value1)>(value1));

    /* Prints "std::string&" */
    print(std::forward<decltype(value2)>(value2));

    /* Prints "const std::string&&" */
    print(std::forward<decltype(value3)>(value3));

    /* Prints "std::string&&" */
    print(std::forward<decltype(value4)>(value4));

    return 0;
}
```

It is super important to use `std::forward<decltype<T>(t)` to perfectly forward the value with the exact type.. If we didn't use it we would get `print(std::string)` for all cases (we would decay).

As was said **if we declare something with `auto&&`, we also declare a universal reference**. We define a reference that binds to all value categories where the type of this reference preserves the type and value category. By the rule, the type of reference `auto&&` is

- an lvalue reference, if we refer to lvalue;
- an rvalue reference, if we refer to rvalue.

Recall 4.2.2 again to explain how the range-`for` loop is implemented. There was a snippet

```
auto && __range = range_expression ;
begin_expr = std::begin(__range);
end_expr = std::end(__range);
```

The `__range` expression is the universal reference. The reason for having it is obvious - we need to bind every type of data without redundant copies done by `std::begin(__range)` and `std::end(__range)` and without losing the information about reference type and cv-qualifiers. Remember that **there is no other way to do it!**

In the generic code, when we can't assume anything about the container value type, we need to use universal reference as well

```
#include <functional>
#include <iostream>
#include <string>
#include <vector>

template <typename T>
void iterate(T&& container)
{
    /* Binds to any type of a value */
    for (auto&& item: container)
    {
        std::cout << item << " ";
    }
    std::cout << std::endl;
}

int main()
{
    std::string str {"str"};
    iterate(str);

    int a = 1;
    int b = 2;
    int c = 3;

    std::vector<std::reference_wrapper<int>> vector { a, b, c };
    iterate(vector);

    /* Special case - std::vector<bool> is not the vector of copies!
     * Check cppreference for more details
     */
    iterate(std::vector<bool> {false, true, false});

    return 0;
}
```

Universal reference and lambda

Since C++14 we can use `auto` keyword in lambdas, which implies the kind of universal reference we're discussing in this section

```
#include <functional>
#include <iostream>
#include <string>

void print(const std::string&)
{
    std::cout << "const std::string&" << std::endl;
}

void print(std::string&)
{
    std::cout << "std::string&" << std::endl;
}

void print(const std::string&&)
{
    std::cout << "const std::string&&" << std::endl;
}

void print(std::string&&)
{
    std::cout << "std::string&&" << std::endl;
}

int main()
{
    const auto lambda = [](auto&& arg)
    {
        /* Notice decltype(arg)! */
        print(std::forward<decltype(arg)>(arg));
    };

    const std::string cstr { "cstring" };
    std::string str { "string" };

    /* Prints "const std::string& */
    lambda(cstr);

    /* Prints "std::string&" */
    lambda(str);

    /* Prints "const std::string&&" */
    lambda(std::move(cstr));

    /* Prints "std::string&&" */
    lambda(std::move(str));

    return 0;
}
```

As for the template-type universal reference, adding any of cv-qualifier will replace the universal reference with a regular rvalue reference.

4.3.6 Perfect returning

Here is the motivating example of why `auto` is not the best option for all cases. We've discussed it briefly before, so here is just a reminder

```
#include <iostream>
#include <string>
#include <type_traits>

namespace detail
{
    const std::string& get(const std::string& str)
    {
        return str;
    }

    std::string& get(std::string& str)
    {
        return str;
    }

    std::string&& get(std::string&& str)
    {
        return std::move(str);
    }

    const std::string&& get(const std::string&& str)
    {
        return std::move(str);
    }
}

/* auto keyword used */
template <typename T>
auto get(T&& str)
{
    return detail::get(std::forward<T>(str));
}

template <typename T>
constexpr void check_type(T&& t)
{
    using type = decltype(t);
    if constexpr (std::is_same_v<type, std::string>)
    {
        std::cout << "std::string" << std::endl;
    }
    else if constexpr (std::is_same_v<type, const std::string&>)
    {
        std::cout << "const std::string&" << std::endl;
    }
    else if constexpr (std::is_same_v<type, std::string&>)
    {
        std::cout << "std::string&" << std::endl;
    }
    else if constexpr (std::is_same_v<type, const std::string&&>)
    {
```

```
        std::cout << "const std::string&&" << std::endl;
    }
    else if constexpr (std::is_same_v<type, std::string&&>)
    {
        std::cout << "std::string&&" << std::endl;
    }
    else
    {
        std::cout << typeid(T).name() << std::endl;
    }
}

int main()
{
    const std::string cstr { "cstring" };
    std::string str { "string" };

    /* All these functions print "std::string" */
    check_type(get(cstr));
    check_type(get(str));
    check_type(get(std::move(cstr)));
    check_type(get(std::move(str)));

    return 0;
}
```

We know that `auto` decays, so drops reference and `const` or `volatile`. What is worse, in such cases we might create an unnecessary copy when the object is returned. **The solution is to add `decltype` keyword** so instead

```
template <typename T>
auto get(T&& str)
{
    return detail::get(std::forward<T>(str));
}
```

we should have

```
template <typename T>
decltype(auto) get(T&& str)
{
    return detail::get(std::forward<T>(str));
}
```

and the result is

```
int main()
{
    const std::string cstr { "cstring" };
    std::string str { "string" };

    /* Prints "const std::string&" */
    check_type(get_wrapper(cstr));

    /* Prints "std:string&" */
    check_type(get_wrapper(str));
```

```
    /* Prints "const std::string&&" */
    check_type(get_wrapper(std::move(cstr)));

    /* Prints "std::string&&" */
    check_type(get_wrapper(std::move(str)));

    return 0;
}
```

The `decltype(auto)` expression is a placeholder type that tells the compiler to deduce type at initialization time. The rules which the compiler uses to find the proper type are

1. if we initialize it with or return a plain name, the return type is the type of the object with that name,
2. if we initialize it with or return an expression, the return type is the type and value category of the evaluated expression and
   - for a prvalue it yields its value, so the type is `T`,
   - for a lvalue, it yields its type as an lvalue reference, so the type is `T&`,
   - for a xvalue, it yields its type as an rvalue reference, so the type is `T&&`.

Deferred perfect returning

If we need to perfectly return the value computed before, **we need to declare this local object with `decltype(auto)`**

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

template <typename T>
decltype(auto) get(T&& arg)
{
    /* Notice the importance of std::forward! */
    decltype(auto) to_return { std::forward<T>(arg) };

    /* Perfectly forward the result */
    return std::forward<decltype(to_return)>(to_return);
}
```

and printouts are

```
int main()
{
    const std::string cstr { "cstring" };
    std::string str { "string" };

    /* Prints "const std::string&" */
    check_type(get(cstr));

    /* Prints "std:string&" */
    check_type(get(str));

    /* Prints "const std::string&&" */
    check_type(get(std::move(cstr)));
```

```
    /* Prints "std::string&&" */
    check_type(get(std::move(str)));

    return 0;
}
```

Perfect returning with lambdas

As always, we're going to start with the motivating example

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

int main()
{
    const auto identity = [](auto&& container)
    {
        return container;
    };

    std::vector<int> vector { 1, 2, 3 };
    /* Won't compile! */
    //  std::vector<int>& copy = identity(vector);
    // for (auto&& value : copy)
    // {
    //      std::court << value << " ";
    // }

    return 0;
}
```

The compilation breaks because of the line

```
std::vector<int>& copy = identity(vector)
```

The lambda `identity` is implemented implicitly as follows

```
const auto identity = [](auto&& container) -> auto
{
    return container;
}
```

which is the same as

```
template <typename T>
auto identity(T&& container)
{
    return container;
}
```

This might create a copy, but for sure doesn't return a reference. The solution is to add `decltype(auto)` explicitly

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

```cpp
int main()
{
    /* decltype(auto) instead of "raw" auto */
    const auto identity = [](auto&& container) -> decltype(auto)
    {
        return container;
    };

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

    std::vector<int>& copy = identity(vector);

    /* Prints "1 2 3" */
    for (auto&& value : copy)
    {
        std::cout << value << " ";
    }

    return 0;
}
```

Chapter 5

Concurrency

5.1 `std::thread`

The most fundamental type in C++ concurrency is `std::thread`. Threads begin execution of passed function immediately upon the construction of the associated thread object. We must remember to join or detach it, otherwise, the program will be immediately terminated when `std::thread` is destructing.

```
#include <iostream>
#include <thread>

int main()
{
    std::thread t1 { [] { std::cout << "Thread 1" << std::endl; } };
    std::thread t2 { [] { std::cout << "Thread 2" << std::endl; } };

    /* Don't wait for the thread till the end */
    t1.detach();

    /* Wait for the thread till the end */
    t2.join();

    return 0;
}
```

It is important to remember that after a detachment of the thread, it cannot be joined again (thus, it cannot be taken „from the background").

If case of exception, we must join or detach `std::thread` inside a `catch` block. Otherwise, the whole program will end with `std::terminate()` immediately.

```
#include <iostream>
#include <thread>

int main()
{
    std::thread thread { [] { std::cout << "Thread" << std::endl; }};

    try
    {
       throw std::runtime_error("Exception!");
```

```
    }
    catch(...)
    {
        std::cout << "Exception thrown!";
        thread.join();
        throw;
    }

    /* Will print "Exception thrown!" */
    thread.join();

    return 0;
}
```

As it is not very convenient to repeat such an action, standard library provides classes `std::lock_guard` and `std::unique_lock`.

5.2 std::mutex

5.2.1 Avoiding deadlocks

Two or more mutexes can provoke a **deadlock problem** so the problem is when one thread is waiting for the second and the second is waiting for the first one.

```
#include <mutex>
#include <thread>

int main()
{
    std::mutex mutex1;
    std::mutex mutex2;

    static const auto lock_both = [](std::mutex& m1, std::mutex& m2)
    {
        m1.lock();
        m2.lock();
    };

    std::thread thread1{[&] { lock_both(mutex1, mutex2); }};
    std::thread thread2{[&] { lock_both(mutex2, mutex1); }};

    thread1.join();
    thread2.join();

    /* Never reach this line */
    return 0;
}
```

The common advice for avoiding deadlocks is to always lock them in the same order. The standard thread library introduces `std::lock()` function allowing to lock two or more mutexes. In C++17 there is an even better option `std::scoped_lock`, a variadic template RAII class which is working similarly as `std::lock_guard`, except that it may take more than one object.

Another guideline to avoid deadlocks is not using a **nested locks**. The idea is simple - don't acquire a lock if we already hold one. An effective method of enforcing the proper order of mutex locking is to use a `hierarchical_mutex`, which isn't a part of the standard library yet so must be written by the hand.

```
#include <climits>
#include <chrono>
#include <exception>
#include <iostream>
#include <mutex>
#include <thread>

class hierarchical_mutex
{
public:
    explicit hierarchical_mutex(unsigned long hierarchy_value):
        m_hierarchy_value { hierarchy_value },
        m_previous_hierarchy_value { 0 }
    {
    }

    void lock()
    {
        check_for_hierarchy_violation();
        m_mutex.lock();
        update_hierarchy_value();
    }

    void unlock()
    {
        if (this_thread_hierarchy_value != m_hierarchy_value)
        {
            throw std::logic_error { "Mutex hierarchy violated" };
        }

        /* Like a popping from the stack - next hierachy_value taken */
        this_thread_hierarchy_value = m_previous_hierarchy_value;

        m_mutex.unlock();
    }

    bool try_lock()
    {
        check_for_hierarchy_violation();
        if (!m_mutex.try_lock())
        {
            return false;
        }
        update_hierarchy_value();
        return true;
    }

private :
    void check_for_hierarchy_violation()
    {
        if (this_thread_hierarchy_value <= m_hierarchy_value)
```

```
        {
            throw std::logic_error { "Mutex hierarchy violated!" };
        }
    }

    void update_hierarchy_value()
    {
        m_previous_hierarchy_value = this_thread_hierarchy_value;
        this_thread_hierarchy_value = m_hierarchy_value;
    }

    static thread_local unsigned long this_thread_hierarchy_value;

    std::mutex m_mutex;
    const unsigned long m_hierarchy_value;
    unsigned long m_previous_hierarchy_value;
};

thread_local unsigned long
    hierarchical_mutex::this_thread_hierarchy_value { ULONG_MAX };

using namespace std::chrono_literals;

hierarchical_mutex high_level_mutex { 100 };
hierarchical_mutex medium_level_mutex { 50 };
hierarchical_mutex low_level_mutex { 10 } ;

void low_level_function()
{
    /* hierarchical_mutex has std::mutex api */
    std::lock_guard<hierarchical_mutex> lock { low_level_mutex };
    std::cout << "Low level function is working" << std::endl;
}

void high_level_function()
{
    std::lock_guard<hierarchical_mutex> lock { high_level_mutex };
    std::cout << "High level calls low level function" << std::endl;

    /* Everything ok - lower priority function can be executed */
    low_level_function();
    std::this_thread::sleep_for(1s);
}

void medium_level_function()
{
    /* Try to lock medium_level_mutex and then high_level_mutex first */
    std::lock_guard<hierarchical_mutex> lock { medium_level_mutex };

    std::cout << "Medium level function is working" << std::endl;
    high_level_function();
}

int main()
{
    std::thread thread1 { [] { high_level_function(); } };
```

```
    std::thread thread2 { [] { medium_level_function(); }};

    /* Printouts:
    * "High level calls low level function"
    * "Low level function is working"
    * "Medium level function is working"
    * and then the exception is thrown.
    */
    thread1.join();
    thread2.join();

    return 0;
}
```

This kind of mutexes will effectively exclude the possibility of locking them in the wrong order.

As a recap, there are basic functionalities related to `std::mutex`

- `std::mutex` is a basic mechanism of data protection in a multithreaded environment. We saw the basic usage of it before.
- `std::shared_mutex` (since C++17, before we had `boost::shared_mutex`) implements **read-write mutex** especially used for rarely updated data. When data has to be read we use `std::shared_lock` (since C++14, before we have to use `boost::shared_lock`), but when we need to update the data `std::lock_guard` or `std::unique_lock` shall be placed there instead. The idea is that `std::shared_lock` is non-blocking for reading, but blocking for writing and `std::lock_guard` (`std::unique_lock`) blocks everything.
  The code below presents basic usage of `std::shared_mutex` and may be useful for cases when **the resource is rarely updated**.

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

using Data = std::vector<int>;

struct User
{
    User(): m_mutex()
    {
    }

    void read(const Data& data, std::size_t idx) const
    {
        std::shared_lock<std::shared_mutex> { m_mutex };
        std::cout << "Read: " << data[idx] << std::endl;

        /* Side note: Direct way of chrono::<time-unit>
```

```
         * initialization
         */
        std::this_thread::sleep_for(
            std::chrono::milliseconds(200));
    }

    void write(Data& data, std::size_t idx, int value)
    {
        std::lock_guard<std::shared_mutex> { m_mutex };
        std::cout << "Changing value from " << data[idx]
                  << " to " << value << std::endl;
        data[idx] = value;

        /* Side note: chrono::<time-unit> initialization
         * with chrono_literals
         */
        std::this_thread::sleep_for(200ms);

    }

    /* Read-write mutex */
    mutable std::shared_mutex m_mutex;
};

int main()
{
    User user {};
    Data data { 1, 2 };

    std::thread thread1
    {
        &User::read, &user, std::ref(data), 0
    };

    std::thread thread2
    {
        &User::write, &user, std::ref(data), 0, 0
    };

    std::thread thread3
    {
        &User::read, &user, std::cref(data), 0
    };

    std::thread thread4
    {
        &User::read, &user, std::cref(data), 1
    };

    std::thread thread5
    {
        &User::write, &user, std::ref(data), 1, 1
    };

    std::thread thread6
    {
```

```
        &User::read, &user, std::cref(data), 1
    };

    thread1.join();
    thread2.join();
    thread3.join();
    thread4.join();
    thread5.join();
    thread6.join();

    return 0;
}
```

- `std::recursive_mutex` can be locked multiple times **within the same thread**. Remember that the number of unlocks must be equal to the number of locks. This kind of mutexes is usually a signal of bad code design and should be avoided.
- `std::lock_guard` implements RAII model with `std::mutex` lock during creation and `std::mutex` release during destruction. It can take one more parameter `std::adopt_lock` which informs the guard that the mutex is already locked.
- `std::unique_lock` is more flexible version of `std::lock_guard` since it is equipped with `lock()`, `unlock()` and `try_lock()` functions to dynamic `std::mutex` managing and also it can take `std::defer_lock` keeping internal mutex unlocked. As the name suggests, it cannot be copied but can be moved.
- `std::lock` is a function that allows to locking of one or more mutex objects at the same time.
- `std::scoped_lock` is a variadic equivalent of `std::lock_guard` introduced in C++17. It is preferred over „naked" `std::lock`.
- `std::once_flag` and `std::call_once` prevents locking mutex more than one time. This is useful for **lazy initialization** i.e. when data must initialized just one time, it is counter-effective to use one of `std::lock_guard` and `std::unique_lock`, each time we need to obtain a lock, to check whether the data was initialized or not. In these cases, the better option, in terms of performance, relies on `std::call_once`.

```
#include <iostream>
#include <mutex>
#include <numeric>
#include <thread>
#include <vector>

struct Data
{
    Data() = default;

    void init()
    {
        std::cout << "Initialization begins" << std::endl;
        std::iota(m_resources.begin(), m_resources.end(), 0);
        std::cout << "Initialization ends" << std::endl;
    }
```

```
    std::vector<int> m_resources;
};

struct Initializer
{
    Initializer():
        m_data(),
        is_initialized()
    {
    }

    void init_once()
    {
        std::call_once(is_initialized, &Data::init, &m_data);
    }

    Data m_data;
    std::once_flag is_initialized;
};

int main()
{
    Initializer initializer {};

    /* Only one of these threads will initialize data */
    std::thread thread1 { [&] { initializer.init_once(); } };
    std::thread thread2 { [&] { initializer.init_once(); } };
    std::thread thread3 { [&] { initializer.init_once(); } };
    std::thread thread4 { [&] { initializer.init_once(); } };

    /* "Initialization begins"
     * "Initialization ends"
     * will appear just one time
     */
    thread1.join();
    thread2.join();
    thread3.join();
    thread4.join();

    return 0;
}
```

5.3 Managing threads

To get a rich description of the thread library see (Williams. Appendix D). Here we present only a few most important functionalities and tools.

5.3.1 `std::condition_variable`

The `std::condition_variable` class allows a thread to wait for a condition to become true. **Instances of `std::condition_variable` are neither copyable nor movable!**

```
#include <condition_variable>
#include <iostream>
```

```cpp
#include <mutex>
#include <thread>
#include <vector>

struct Collection
{
    Collection():
        m_resources(),
        m_condition_variable(),
        m_mutex()
    {
    }

    void insert(int idx)
    {
        std::lock_guard<std::mutex> { m_mutex };

        m_resources.push_back(idx);
        std::cout << "Element " << idx << " inserted\n";

        / * Only one waiting thread will be woken up */
        m_condition_variable.notify_one();
    }

    void pop()
    {
        /* Here we are waiting until the vector is not empty.
         * We have to pass unique_lock since m_condition_variable
         * unlocks mutex when a condition is not fulfilled.
         * That's why we cannot use std::lock_guad.
         */
        std::unique_lock<std::mutex> lock { m_mutex };

        auto predicate = [this]{ return !m_resources.empty(); }
        m_condition_variable.wait(lock, predicate);

        std::cout << "Popping " << m_resources.back() << std::endl;
        m_resources.erase(m_resources.end() - 1);
    }

    std::vector<int> m_resources;
    std::condition_variable m_condition_variable;
    std::mutex m_mutex;
};

int main()
{
    Collection collection {};

    std::thread thread1 {&Collection::pop, &collection};
    std::thread thread2 {&Collection::pop, &collection};
    std::thread thread3 {&Collection::insert, &collection, 1};
    std::thread thread4 {&Collection::insert, &collection, 2};

    thread1.join();
    thread2.join();
```

```
    thread3.join();
    thread4.join();

    return 0;
}
```

5.3.2 std::future

The `std::future` provides a facility for handling asynchronous results that may be performed on another thread. The `std::future` an object is quite similar to `std::unique_ptr` so that it is a unique representation of the given future. We also have the `std::shared_future` (similar to `std::shared_ptr`), which allows to share the same future by multiple `std::shared_future` objects. In case when the future is ready, all waiting threads get it at the same time. A specialization for `void` means that we are just waiting for a function to be finished.

For the given future we need to call one of `get()` or `wait()` functions. The difference between them is that a `get()` function returns a value when `wait()` only waits for the future readiness, to be obtained by `get()` further with immediate effect. We can call them only once - that is quite understandable, the future can be ready only one time.

The most basic way to use futures is to combine them with `std::async()` function which can work asynchronously or synchronously depending on the argument:

- `std::launch::asynch` - run the function asynchronously (in the moment of call).
- `std::launch::deferred` - wait until `get()` or `wait()` will be called.
- `std::launch::asynch | std::launch::deferred` - let the compiler choose the best option.

Let's have a look on short example[1].

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <future>

using Iter = decltype(std::declval<std::vector<int>>().begin());
using Size = decltype(std::declval<Iter>() - std::declval<Iter>());

Size sort(Iter begin, Iter end)
{
    const auto elements { end - begin };
    std::stable_sort(begin, end);
    return elements;
}

/* std::future is movable but not copyable */
std::future<Size> invoke(Iter begin, Iter end,
    std::launch launch = std::launch::async | std::launch::deferred)
{
    return std::async(launch, &sort, begin, end);
```

[1] This example is silly and has to be corrected.

```
}

int main()
{
    std::vector<int> data {10, 1, -5, 4, 7, -9};

    /* Run in separate thread */
    auto size1 = invoke(data.begin(), data.begin() + 2,
        std::launch::async);

    /* Run when wait() or get() invoked */
    auto size2 = invoke(data.begin() + 2, data.begin() + 4,
        std::launch::deferred);

    /* Default setup */
    auto size3 = invoke(data.begin() + 4, data.end());

    /* Prints "Sorted 6 elements */
    std::cout << "Sorted " << size1.get() + size2.get() + size3.get()
              << " elements" << std::endl;

    return 0;
}
```

5.3.3 std::promise

The `std::promise` class template provides a means of setting an asynchronous result, which may be retrieved from another thread through an instance of `std::future`. The `std::promise` object doesn't set the variable in any mysterious, asynchronously way by itself, it is only a wrapper returning `std::future`.

```
#include <iostream>
#include <future>

int main()
{
    std::promise<int> promise;

    /* No value set yet */
    std::future<int> future = promise.get_future();

    /* Set the value */
    promise.set_value(1);

    /* Prints "1" */
    std::cout << future.get() << std::endl;

    return 0;
}
```

We set the value by invoking `set_value()` and get it by `get_future()`. When we need to store an exception we can either destroy the `std::promise` object (`broken promise` will be thrown) or use `set_exception()` member function. Let's have a look at some multithreaded example

```
#include <exception>
#include <future>
```

```cpp
#include <iostream>
#include <numeric>
#include <thread>
#include <vector>

std::vector<int> prepare(bool throws = false)
{
    if (throws)
    {
        throw std::runtime_error("Cannot prepare!");
    }

    std::vector<int> values(100000);
    std::iota(values.begin(), values.end(), 0);
    return values;
}

int main()
{
    std::promise<std::vector<int>> values;
    std::future<std::vector<int>> future = values.get_future();

    /* Async run */
    std::thread thread { [&]
    {
        try
        {
            values.set_value(std::move(prepare()));
        }
        catch (...)
        {
            /* Save last thrown exception */
            values.set_exception(std::current_exception());
        }
    }};
    thread.detach();

    /* Do something in the meantime */
    std::promise<std::vector<int>> error;
    std::future<std::vector<int>> error_future = error.get_future();

    /* Async run */
    std::thread error_thread { [&]
    {
        try
        {
            error.set_value(std::move(prepare(true)));
        }
        catch (...)
        {
            error.set_exception(std::current_exception());
        }
    }};
    error_thread.detach();

    /* Check error first */
```

```
    try
    {
        error_future.get();
    }
    catch (std::runtime_error err)
    {
        std::cout << err.what() << std::endl;
    }

    /* Back to initial task */
    std::cout << "Last item=" << future.get().back;;

    return 0;
}
```

Like above, we can perform some actions in the meantime and then get the result of a future.

5.3.4 std::packaged_task

The `std::packaged_task` template class packages a function or other callable object so that when the function is invoked through `std::packaged_task`, the result is stored as an asynchronous result for retrieval through an instance of `std::future`. It can be considered as an equivalent of `std::promise` for functions i.e. `std::promise` is related to future values and `std::packaged_task` is related to future function calls.

As for `std::promise`, `std::packaged_task` doesn't invoke the function asynchronously by itself, it is only a wrapper returning `std::future`.

```
#include <future>
#include <iostream>
#include <string>

std::string get() { return "Get"; }

int main()
{
    std::packaged_task<std::string()> task { &get };
    std::future<std::string> future = task.get_future();

    std::cout << "Calling get() now: ";
    /* It must be called, otherwise .get() will wait forever! */
    task();
    std::cout << "value=" << future.get() << std::endl;

    return 0;
}
```

Some more realistic example of `std::packaged_task` usage is presented below

```
#include <functional>
#include <future>
#include <iostream>
#include <thread>
#include <vector>
```

```
using Data = std::vector<std::size_t>;
using Function = std::function<std::size_t(const Data&)>;
using PackagedTask = std::packaged_task<std::size_t(const Data&)>;
using Future = std::future<std::size_t>;

struct ThreadPool
{
    ThreadPool(): m_tasks()
    {
    }

    template <typename Function>
    void append(Function&& function)
    {
        auto task {PackagedTask{std::forward<Function>(function)}};
        m_tasks.push_back(std::move(task));
    }

    std::vector<Future> get()
    {
        std::vector<Future> futures;
        for (auto& task : m_tasks)
        {
            auto future = task.get_future();
            futures.push_back(std::move(future));
        }
        return futures;
    }

    void invoke(const std::vector<Data>& datas)
    {
        std::vector<std::thread> threads;
        for (std::size_t i = 0; i < m_tasks.size(); i++)
        {
            /* Important - tasks must be moved ! */
            std::thread thread {std::move(m_tasks[i]),
                std::cref(datas[i])};
            threads.push_back(std::move(thread));
        }

        for (auto& thread : threads)
        {
            thread.join();
        }
    }

    std::vector<PackagedTask> m_tasks;
};

std::size_t GetFirst(const Data& data)
{
    return data[0];
}

std::size_t GetSecond(const Data& data)
{
```

```
    return data[1];
}

std::size_t GetThird(const Data& data)
{
    return data[2];
}

int main()
{
    std::vector<Data> Collection
    {
        { 1, 2, 3 },
        { 10, 20, 30 },
        { 100, 200, 300 }
    };

    ThreadPool threadPool;

    threadPool.append(std::bind(&GetFirst,  std::placeholders::_1));
    threadPool.append(std::bind(&GetSecond, std::placeholders::_1));
    threadPool.append(std::bind(&GetThird,  std::placeholders::_1));

    /* No mulithtreading yet */
    auto futures { std::move(threadPool.get()) };

    /* Async invoke */
    threadPool.invoke(Collection);

    for (auto& future : futures)
    {
        std::cout << future.get() << std::endl;
    }

    return 0;
}
```

5.3.5 std::barrier

The `std::barrier` class is a synchronization primitive introduced in C++20 as part of the `<barrier>` header. It is used to coordinate multiple threads, making them wait until a predefined number of threads (called **the arrival count**) have reached a certain point in their execution (the synchronization point).

Once all threads reach the synchronization point, the barrier can optionally execute a callback function (if provided) and then reset itself, allowing the threads to continue execution and reuse the barrier.

```
#include <array>
#include <barrier>
#include <iostream>
#include <numeric>
#include <thread>
#include <future>
```

```
std::array<int, 5> data{};

template <typename Completion>
auto function(std::barrier<Completion>& barrier, int idx, int value)
{
    data[idx] = value;
    /* Without this line it is possible to get 0s only */
    barrier.arrive_and_wait();

    return std::accumulate(data.begin() + idx, data.end(),
        int{1}, std::multiplies{});
}

int main()
{
    std::ptrdiff_t threads_num = 5;
    auto completion_function = []()
    {
        std::cout << "All threads have arrived!" << std::endl;
    };

    std::barrier barrier{threads_num, completion_function};

    auto result1 = std::async(std::launch::async,
        [&barrier]() { return function(barrier, 0, 2); });
    auto result2 = std::async(std::launch::async,
        [&barrier]() { return function(barrier, 1, 3); });
    auto result3 = std::async(std::launch::async,
        [&barrier]() { return function(barrier, 2, 4); });
    auto result4 = std::async(std::launch::async,
        [&barrier]() { return function(barrier, 3, 5); });
    auto result5 = std::async(std::launch::async,
        [&barrier]() { return function(barrier, 4, 6); });

    /* The prinout contains numbers 720, 360, 120, 30, 6 */
    std::cout << result1.get() << ","
              << result2.get() << ","
              << result3.get() << ","
              << result4.get() << ","
              << result5.get() << std::endl;

    return 0;
}
```

5.3.6 `std::latch`

The `std::latch` is a synchronization primitive introduced in C++20. It is used to block threads until a specific number of threads have reached a certain point in their execution, at which point the latch "unlocks" and allows the threads to proceed. It is commonly used for situations where a set of threads needs to wait for a certain condition or event to occur before continuing.

Unlike `std::barrier`, which can be reused for multiple synchronization phases, `std::latch` is one-time use. Once the latch has been "counted down" to zero, it cannot be reused.

```cpp
#include <array>
#include <future>
#include <latch>
#include <iostream>
#include <numeric>
#include <thread>

std::array<int, 3> data{};

auto function(std::latch& latch, int idx, int value)
{
    if (!latch.try_wait())
    {
        data[idx] = value;

        /* May be called as one function
         * latch.arrive_and_wait();
         */
        latch.count_down();
        latch.wait();
        return std::accumulate(data.begin() + idx, data.end(),
            int{1}, std::multiplies{});
    }
    return int{};
}

int main()
{
    std::latch latch {3};

    auto result1 = std::async(std::launch::async,
        [&latch]() { return function(latch, 0, 2); });
    auto result2 = std::async(std::launch::async,
        [&latch]() { return function(latch, 1, 3); });
    auto result3 = std::async(std::launch::async,
        [&latch]() { return function(latch, 2, 4); });
    auto result4 = std::async(std::launch::async,
        [&latch]() { return function(latch, 0, 5); });
    auto result5 = std::async(std::launch::async,
        [&latch]() { return function(latch, 1, 6); });

    /* The prinout contains numbers 24, 12, 4, 0, 0.
     * Two zeros will appear at the end since
     * the latch counter will drop to 0 after 3 threads
     * will have done their job.
     */
    std::cout << result1.get() << ","
              << result2.get() << ","
              << result3.get() << ","
              << result4.get() << ","
              << result5.get() << std::endl;

    return 0;
}
```

5.3.7 std::semaphore

The `std::semaphore` type has been introduced along with `std::barrier` and `std::latch`. It is a synchronization primitive that controls access to a shared resource by maintaining an internal counter. Threads can acquire or release permits, with the counter representing the number of available resources. A thread attempting to acquire a permit will be blocked if none are available until another thread releases one.

`std::semaphore` comes in two variants: `std::counting_semaphore`, which allows the counter to have an arbitrary maximum value, and `std::binary_semaphore`, which behaves like a simple lock with a maximum count of one.

```
#include <array>
#include <atomic>
#include <chrono>
#include <future>
#include <iostream>
#include <numeric>
#include <semaphore>
#include <thread>

std::array<int, 3> data{};

template <std::size_t N>
auto function(std::counting_semaphore<N>& semaphore, int idx, int value)
{
    /* Wait until we have three threads working */
    if (semaphore.try_acquire())
    {
        data[idx] = value;

        /* Should be enough for this example */
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    else
    {
        return int{};
    }

    int result = std::accumulate(data.begin(), data.end(),
        int{0}, std::plus{});

    /* Decrease semaphore count */
    semaphore.release();

    return result;
}

int main()
{
    /* LeastMaxValue is maximal value of counts,
     * permits is number of locks allowed. In this example
     * LeastMaxValue is defined by "max", permitted number
     * of locks by "permits".
     */

    static constexpr std::ptrdiff_t max = 3;
```

```
    for (std::ptrdiff_t permits{}; permits <= max; ++permits)
    {
        std::cout << "Permits=" << permits << std::endl;

        std::counting_semaphore<max> semaphore{permits};

        auto result1 = std::async(std::launch::async,
            [&semaphore]() { return function<max>(semaphore, 0, 2); });
        auto result2 = std::async(std::launch::async,
            [&semaphore]() { return function<max>(semaphore, 1, 3); });
        auto result3 = std::async(std::launch::async,
            [&semaphore]() { return function<max>(semaphore, 2, 4); });
        auto result4 = std::async(std::launch::async,
            [&semaphore]() { return function<max>(semaphore, 0, 5); });
        auto result5 = std::async(std::launch::async,
            [&semaphore]() { return function<max>(semaphore, 1, 6); });

        /* Observe decreasing number of 0s. Possible realisation of
         * the entire loop
         *
         * Permits=0
         * 0,0,0,0,0
         *
         * Permits=1
         * 2,0,0,0,0
         *
         * Permits=2
         * 5,5,0,0,0
         *
         * Permits=3
         * 9,9,9,0,0
         */
        std::cout << result1.get() << ","
                  << result2.get() << ","
                  << result3.get() << ","
                  << result4.get() << ","
                  << result5.get() << std::endl;
        std::cout << std::endl;
    }

    return 0;
}
```

As an extra, final example a very useful function (similar to Erlang's `spawn`) to start asynchronous process dynamically

```
#include <future>
#include <iostream>
#include <thread>
#include <vector>

template <typename Function, typename... Args>
std::future <typename std::result_of_t<Function(Args&&...)>>
spawn_task(Function&& function, Args&&... args)
{
    using return_t = std::result_of_t<Function(Args&&...)>;
```

```
    std::packaged_task<return_t(Args&&...)> task {std::move(function)};
    std::future<return_t> future { task.get_future() };

    std::thread thread {std::move(task), std::forward<Args>(args)...};
    thread.detach();

    return future;
}

template <std::size_t N>
std::size_t put(std::vector<int>& vec, int value)
{
    vec[N] = value;
    return N * value;
}

int main()
{
    std::vector<int> vec (3);
    auto future0 = spawn_task(&put<0>, std::ref(vec), 1);
    auto future1 = spawn_task(&put<1>, std::ref(vec), 10);
    auto future2 = spawn_task(&put<2>, std::ref(vec), 100);

    std::cout << future0.get() << std::endl;
    std::cout << future1.get() << std::endl;
    std::cout << future2.get() << std::endl;
}
```

5.4 Atomic operations

An **atomic operation** is indivisible operation. We can't observe such an operation half-done from any thread in the system; it's either done or not done. Some non-atomic operation might be seen as half-done by another thread.

This section must be updated, for now it contains only examples

5.4.1 Check lock free types

```
#include <iostream>
#include <atomic>

int main()
{
    std::cout << "Is bool always lock free: "
              << std::atomic<bool>::is_always_lock_free
              << std::endl;
    std::cout << "Is bool* always lock free: "
              << std::atomic<bool*>::is_always_lock_free
              << std::endl;

    std::cout << "Is int always lock free: "
              << std::atomic<int>::is_always_lock_free
              << std::endl;
```

```
    std::cout << "Is int* always lock free: "
              << std::atomic<int*>::is_always_lock_free
              << std::endl;

    std::cout << "Is float always lock free: "
              << std::atomic<float>::is_always_lock_free
              << std::endl;
    std::cout << "Is float* always lock free: "
              << std::atomic<float*>::is_always_lock_free
              << std::endl;

    std::cout << "Is double always lock free: "
              << std::atomic<double>::is_always_lock_free
              << std::endl;
    std::cout << "Is double* always lock free: "
              << std::atomic<double*>::is_always_lock_free
              << std::endl;

    return 0;
}
```

5.4.2 std::atomic_flag

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

struct spinlock_mutex
{
    spinlock_mutex()
     : flag(ATOMIC_FLAG_INIT)
    {}

    void lock()
    {
       while (flag.test_and_set());
    }

    void unlock() { flag.clear(); }

    std::atomic_flag flag;
};

void write(int value, std::vector<int>& data, spinlock_mutex& mutex)
{
    std::lock_guard<spinlock_mutex>{ mutex };
    data.push_back(value);
}

void read(const std::vector<int>& data, spinlock_mutex& mutex)
{
    std::lock_guard<spinlock_mutex>{ mutex };
    for (auto&& item : data)
    {
       std::cout << item << " ";
```

```
    }
    std::cout << std::endl;
}

int main()
{
    spinlock_mutex mutex{};
    std::vector<int> data;

    std::thread thread1{ [&]() { write(1, data, mutex); } };
    std::thread thread2{ [&]() { write(2, data, mutex); } };

    /* May print "" or "1" or "2" or "1 2" or "2 1" */
    std::thread thread3{ [&]() { read(data, mutex); } };
    std::thread thread4{ [&]() { write(3, data, mutex); } };

    /* May print "" or any possible result from "1 2 3" */
    std::thread thread5{ [&]() { read(data, mutex); } };

    thread1.join();
    thread2.join();
    thread3.join();
    thread4.join();
    thread5.join();

    return 0;
}
```

5.4.3 `std::atomic` template

Basic functionalities

```
#include <atomic>
#include <iostream>

int main()
{
    std::atomic<int> atomic;

    /* We can get only the copy of value stored inside,
     * no reference!
     */
    // auto& init_value = atomic_bool.load();

    auto init_value = atomic.load();

    /* Prints "0" */
    std::cout << init_value << std::endl;

    /* It is also possible to read the value directly */
    int automatic_cast_value = atomic;
    std::cout << automatic_cast_value << std::endl;

    /* Set the new value */
    atomic.store(1);
```

```
    /* Prints "1" */
    std::cout << atomic.load() << std::endl;

    /* fetch_add() and returns old value and performs
     * atomic incremetation. It has its ++ and +=
     * wrappers but there is no point to present them
     * all.
     */
    /* Prints "1" */
    std::cout << atomic.fetch_add(2) << std::endl;

    /* Prints "3" */
    std::cout << atomic << std::endl;

    /* As equivalent of fetch_add() is fetch_sub() */
    /* Prints "3" */
    std::cout << atomic.fetch_sub(3) << std::endl;

    /* Prints "0" */
    std::cout << atomic << std::endl;

    /* There are many more equivalents of standard operations for integers
     * like fetch_and() (&), fetch_or (|), etc. See the documentation
     * to learn more.
     */

    return 0;
}
```

Compare-exchange weak - normally used in the loop as may spontaneously fail

```
#include <atomic>
#include <iostream>

int main()
{
    std::atomic<int> atomic;

    /* Expected argument has to be passed by l-value reference! */
    // std::cout << atomic.compare_exchange_weak(0, 1) << std::endl;
    int expected = 0;

    /* Prints "1" which means "true" and indicates success */
    std::cout << atomic.compare_exchange_weak(expected, 1) << std::endl;

    /* Prints "0" - value untouched */
    std::cout << expected << std::endl;

    /* Prints "1" */
    std::cout << atomic.load() << std::endl;

    /* Try to do the same - expected is still 0 */
    /* Prints "0" which means "false" and indicates loss */
    std::cout << atomic.compare_exchange_weak(expected, 1) << std::endl;

    /* Prints "1" - the expected value has been updated with the value
     * already stored in the atomic variable!
```

```
     */
    std::cout << expected << std::endl;

    /* Prints "1" */
    std::cout << atomic.load() << std::endl;

    return 0;
}
```

Compare-exchange strong - won't fail but may impact on performance (internal loop)

```
#include <atomic>
#include <iostream>

int main()
{
    std::atomic<int> atomic;

    /* Expected argument has to be passed by l-value reference! */
    // std::cout << atomic.compare_exchange_strong(0, 1) << std::endl;
    int expected = 0;

    /* Prints "1" which means "true" and indicates success */
    std::cout << atomic.compare_exchange_strong(expected, 1) << std::endl;

    /* Prints "0" - value untouched */
    std::cout << expected << std::endl;

    /* Prints "1" */
    std::cout << atomic.load() << std::endl;

    /* Try to do the same - expected is still 0 */
    /* Prints "0" which means "false" and indicates loss */
    std::cout << atomic.compare_exchange_strong(expected, 1) << std::endl;

    /* Prints "1" - the expected value has been updated with the value
     * already stored in the atomic variable!
     */
    std::cout << expected << std::endl;

    /* Prints "1" */
    std::cout << atomic.load() << std::endl;

    return 0;
}
```

5.4.4 Memory model

`std::memory_order_seq_cst`

Standard model - all threads see the same operations order

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
```

```
#include <vector>

std::vector<int> data;
std::atomic<bool> data_ready{ false };

void read()
{
    /* (1) */
    while (!data_ready.load())
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
    /* (2) */
    std::cout << "value=" << data[0] << std::endl;
}

void write()
{
    /* (3) */
    data.push_back(0);

    /* (4) */
    data_ready = true;
}

int main()
{
    std::thread thread1{ [&] { read(); } };
    std::thread thread2{ [&] { write(); } };

    thread1.join();
    thread2.join();

    return 0;
}
```

This is the same behavior as if the most strict `memory_order_seq_cst` model used explicitly

```
#include <atomic>
#include <cassert>
#include <iostream>
#include <thread>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x()
{
    x.store(true, std::memory_order_seq_cst);
}

void write_y()
{
    y.store(true, std::memory_order_seq_cst);
}
```

```
void read_x_then_y()
{
    while (!x.load(std::memory_order_seq_cst));
    if (y.load(std::memory_order_seq_cst))
    {
        ++z;
    }
}

void read_y_then_x()
{
    while (!y.load(std::memory_order_seq_cst));
    if (x.load(std::memory_order_seq_cst))
    {
        ++z;
    }
}

int main()
{
    std::thread thread1{ write_x };
    std::thread thread2{ write_y };
    std::thread thread3{ read_x_then_y };
    std::thread thread4{ read_y_then_x };

    thread1.join();
    thread2.join();
    thread3.join();
    thread4.join();

    /* This will never fail since all threads "saw" the same,
     * global order of events. In other words threads HAVE TO
     * agree on the order of events.
     */
    assert(z.load() != 0);

    return 0;
}
```

std::memory_order_relaxed

Only update is synchronized

```
#include <atomic>
#include <cassert>
#include <iostream>
#include <thread>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x_then_y()
{
    /* In this thread x is modified before y */
```

```
    x.store(true, std::memory_order_relaxed);
    y.store(true, std::memory_order_relaxed);
}

void read_y_then_x()
{
    /* However, because relaxed option is used
     * there is no synchronization between threads
     * so we have no guarantee that if y was set
     * this implies (as order in write_x_then_y suggests)
     * that x had been set before.
     */
    while (!y.load(std::memory_order_relaxed));
    if (x.load(std::memory_order_relaxed))
    {
        ++z;
    }
}

int main()
{
    std::thread thread1{ write_x_then_y };
    std::thread thread2{ read_y_then_x };

    thread1.join();
    thread2.join();

    /* This might fail since all threads "didn't see" the same,
     * global order of events. In other words threads DON'T
     * HAVE TO agree on the order of events.
     */
    assert(z.load() != 0);

    return 0;
}
```

memory_order_release and memory_order_acquire

```
#include <iostream>
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x()
{
    /* std::memory_order_release is devoted to save values */
    x.store(true, std::memory_order_release);
}

void write_y()
{
    y.store(true, std::memory_order_release);
```

```
}

void read_x_then_y()
{
    /* std::memory_order_acquire is devoted to load values */
    while (!x.load(std::memory_order_acquire));
    if (y.load(std::memory_order_acquire))
    {
        ++z;
    }
}

void read_y_then_x()
{
    while (!y.load(std::memory_order_acquire));
    if (x.load(std::memory_order_acquire))
    {
        ++z;
    }
}

int main()
{
    std::thread thread1{ write_x };
    std::thread thread2{ write_y };
    std::thread thread3{ read_x_then_y };
    std::thread thread4{ read_y_then_x };

    thread1.join();
    thread2.join();
    thread3.join();
    thread4.join();

    /* This might fail since x and y are not synchronized.
     * However, we have kind of control here, because
     * even though we haven't synchronized x and y
     * we've created a synchronization for x and y write-read
     * operations by std::memory_order_release (for write) and
     * std::memory_order_acquire (for read).
     */
    assert(z.load() != 0);

    return 0;
}
```

Managing memory with `std::memory_order_relaxed` and `std::memory_order_seq_cst`

```
#include <atomic>
#include <cstring>
#include <iostream>
#include <thread>

struct Values
{
    int x;
    int y;
```

```
    int z;
};

constexpr std::size_t ITERATIONS = 10;

std::atomic<int> x;
std::atomic<int> y;
std::atomic<int> z;
std::atomic<bool> start;

Values values1[ITERATIONS];
Values values2[ITERATIONS];
Values values3[ITERATIONS];
Values values4[ITERATIONS];
Values values5[ITERATIONS];

void reset()
{
    x = 0;
    y = 0;
    z = 0;
    start = false;

    std::memset(values1, 0, sizeof(values1));
    std::memset(values2, 0, sizeof(values2));
    std::memset(values3, 0, sizeof(values3));
    std::memset(values4, 0, sizeof(values4));
    std::memset(values5, 0, sizeof(values5));
}

void increment(std::atomic<int>* value,
               Values* values,
               std::memory_order order)
{
    /* Run all threads in the same time */
    while (!start)
    {
        std::this_thread::yield();
    }

    for (std::size_t i = 0; i < ITERATIONS; ++i)
    {
        values[i].x = x.load(order);
        values[i].y = y.load(order);
        values[i].z = z.load(order);

        value->store(i+1, order);
        std::this_thread::yield();
    }
}

void read(Values* values, std::memory_order order)
{
    while (!start)
    {
        std::this_thread::yield();
```

```
    }

    for (std::size_t i = 0; i < ITERATIONS; ++i)
    {
        values[i].x = x.load(order);
        values[i].y = y.load(order);
        values[i].z = z.load(order);
        std::this_thread::yield();
    }

}

void print(Values* values)
{
    while (!start)
    {
        std::this_thread::yield();
    }

    for (std::size_t i = 0; i < ITERATIONS; ++i)
    {
        if (i)
        {
            std::cout << ", ";
        }
        std::cout << "[" << values[i].x
                  << "," << values[i].y
                  << "," << values[i].z
                  << "]";
    }
    std::cout << std::endl;
}

int main()
{
    std::memory_order order = std::memory_order_relaxed;

    std::thread thread1{ [&] { increment(&x, values1, order); } };
    std::thread thread2{ [&] { increment(&y, values2, order); } };
    std::thread thread3{ [&] { increment(&z, values3, order); } };
    std::thread thread4{ [&] { read(values4, order); } };
    std::thread thread5{ [&] { read(values5, order); } };

    start = true;

    thread5.join();
    thread4.join();
    thread3.join();
    thread2.join();
    thread1.join();

    /* First row: first value increases from 0 to 9,
     * Second row: second value increases from 0 to 9,
     * Third row: third value increases from 0 to 9
     */
    print(values1);
```

```
    print(values2);
    print(values3);
    print(values4);
    print(values5);

    return 0;
}
```

5.5 STL algorithms

Since C++17 we can use STL algorithms with concurrency support. We also have multiple execution policies, all of the needed information can be found on cppreference page. Apart from that, `std::atomic<T>::is_always_lock_free` become `constexpr` member, so can be called in the compilation phase.

5.6 Appendix

5.6.1 Data race

A **data race** occurs when two instructions from different threads access the same memory location, at least one of these accesses is a write and no synchronization is mandating any particular order among these accesses.

5.6.2 Race condition

A **race condition** can arise in software when a computer program has multiple code paths that are executing at the same time. If the multiple code paths take a different amount of time than expected, they can finish in a different order than expected, which can cause software bugs due to unanticipated behavior.

5.6.3 Deadlock

A **deadlock** is a state in which each member of a group of actions, is waiting for some other member to release a lock.

5.6.4 Livelock

A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change about one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Chapter 6

Miscellaneous

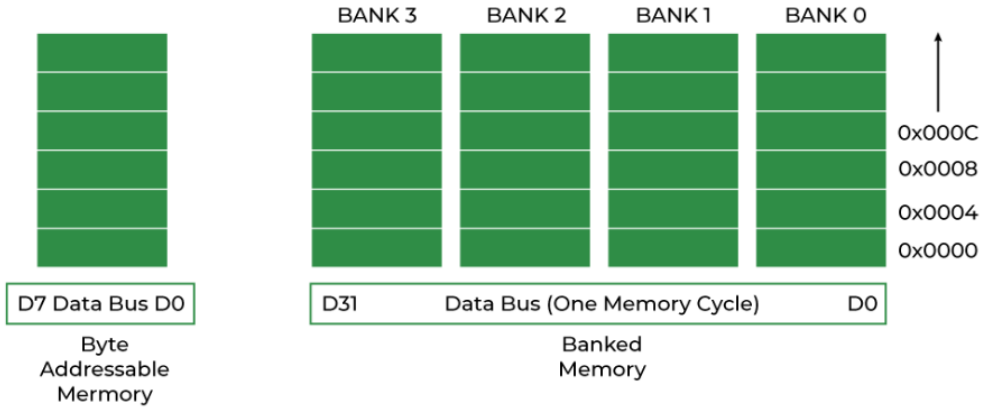
6.1 C++

6.1.1 Structure Member Alignment and Padding

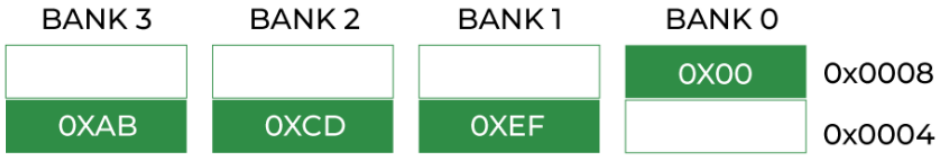
Data alignment in memory

Every data type in C will have alignment requirements (in fact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32-bit machine, the processing word size will be 4 bytes.

Historically, memory is byte-addressable and arranged sequentially. If the memory is arranged as a single bank of one-byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of an integer in one memory cycle. To take such advantage, the memory will be arranged as a group of 4 banks as shown below.



The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at $(X + 1)$, $(X + 2)$, and $(X + 3)$ addresses. If an integer of 4 bytes is allocated on X address (X is a multiple of 4), the processor needs only one memory cycle to read the entire integer. Whereas, if the integer is allocated at an address other than a multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycles to fetch the data.



Layout of misaligned data (0X01ABCDEF)

A variable's data alignment deals with the way the data is stored in these banks. For example, the natural alignment of `int` on a 32-bit machine is 4 bytes. When a data type is naturally aligned, the CPU fetches it in minimum read cycles. Similarly, the natural alignment of a short int is 2 bytes. It means a short `int` can be stored in bank 0 – bank 1 pair or bank 2 – bank 3 pair. A `double` requires 8 bytes and occupies two rows in the memory banks. Any misalignment of double will force more than two read cycles to fetch double data.

Note that a `double` variable will be allocated on an 8-byte boundary on a 32-bit machine and requires two memory read cycles. On a 64-bit machine, based on a number of banks, a `double` variable will be allocated on the 8-byte boundary and requires only one memory read cycle.

Structure padding

Structure padding is the addition of some empty bytes of memory in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure. Let's start with calculating the size of a few types

```
#include <iostream>

int main()
{
    /* Prints "1" */
    std::cout << sizeof(char) << std::endl;

    /* Prints "2" */
    std::cout << sizeof(short int) << std::endl;

    /* Prints "4" */
    std::cout << sizeof(int) << std::endl;

    /* Prints "8" */
    std::cout << sizeof(double) << std::endl;

    return 0;
}
```

Okay, we know how wide these types are so now, let's define a few structures

```
#include <iostream>

struct Struct1
{
```

```
    char c;
    short int s;
};

struct Struct2
{
    short int s;
    char c;
    int i;
};

struct Struct3
{
    char c;
    double d;
    int s;
};

struct Struct4
{
    double d;
    int s;
    char c;
};

int main()
{
    Struct1 struct1 {};
    Struct2 struct2 {};
    Struct3 struct3 {};
    Struct4 struct4 {};

    std::cout << sizeof(struct1) << std::endl;
    std::cout << sizeof(struct2) << std::endl;
    std::cout << sizeof(struct3) << std::endl;
    std::cout << sizeof(struct4) << std::endl;

    return 0;
}
```

We may expect that

- `sizeof(struct1)` $= 1 + 2 = 3$,
- `sizeof(struct2)` $= 2 + 1 + 4 = 7$,
- `sizeof(struct3)` $= 1 + 8 + 4 = 13$,
- `sizeof(struct4)` $= 8 + 4 + 1 = 13$,

but the reality is

- `sizeof(struct1)` $= 4$,
- `sizeof(struct2)` $= 8$,
- `sizeof(struct3)` $= 24$,

- `sizeof(struct4)` = 16.

The results are different than expected **because of the alignment requirements of various data types - every member of the structure should be naturally aligned so the members of the structure are allocated sequentially in increasing order.** Let us analyze each struct

- **`Struct1`** - the first element is `char` which is 1 byte aligned, followed by `short int` which is 2 bytes aligned. If the short int element is immediately allocated after the char element, it will start at an odd address boundary. The compiler will insert a ***padding byte*** after the `char` to ensure `short int` will have an address multiple of 2 (i.e. 2 byte aligned). The total size of `Struct1` will be

  `sizeof(char)` + ***padding byte*** + `sizeof(short int)` = 4.

- **`Struct2`** - the first element is `short` 2 byte aligned. Since `char` can be on any byte boundary, no padding is required between these types. But the next one is `int` which is 4 byte aligned, thus it cannot start at an odd byte boundary, so one padding byte is needed. Therefore,

  `sizeof(short int)` + `char` + ***padding byte*** + `int` = 8.

- **`Struct3`** - Observe that after adding padding byte after `char` we are not on the byte that is the multiplication of `sizeof(double)` = 8, so we need to add seven padding bytes to it, thus

  `char` + 7 * ***padding byte*** + `double` + `int` = 20.

  However, the size of the structure is 24. What happened? This is because structure-type variables also have natural alignment, for instance

  ```
  Struct3 structs[3];
  ```

  Assume that the base address of `structs[0]` is 0. The whole `Struct3` has a size equal to 20, so the second element `structs[1]` starts with 20 address, and therefore, the `double` member must start from 28 which is not the dividend of 8 (size of `double`). To avoid that, **the compiler introduces alignment requirements to every structure**, so in this case it needs to add **four padding bytes** to make the structure size multiple of its alignment. Eventually

  `char` + 7 * ***padding byte*** + `double` + `int` + 4 * ***padding byte*** = 24.

- **`Struct4`** - we have

  `sizeof(double)` + `sizeof(int)` + `sizeof(char)` = 13

  but because of `double` size, we need to add 3 padding bytes, then we have 16.

Structure padding is not avoidable, but can be easily optimized - it is enough to **declare the structure members in their increasing or decreasing order of size**. It is visible in the examples of `Struct3` and `Struct4`.

6.1.2 Value categories

C++11

Since C++11 we have the following **core categories of values**

- ***lvalue***, for instance
  - an expression that is just the name of a variable, function, or member.
  - an expression that is just a string literal.
  - the result of the built-in unary `*` operator (i.e., dereferencing operator).
  - the result of a function returned by **lvalue** reference.
- ***prvalue*** - *pure rvalue*, for instance
  - expressions that consist of a literal that **is not a string literal** or a user-defined literal, where the return type of the associated literal operator defines the category.
  - the result of the built-in unary `&` operator (i.e., taking an address of an expression).
  - the result of built-in arithmetic operators.
  - the result of a function returned by value.
  - a lambda expression.
- ***xvalue*** - *expiring value*, for instance
  - the result of a function returned by **rvalue** reference, especially returned by `std::move()`,
  - a cast to an **rvalue** reference to an object type,

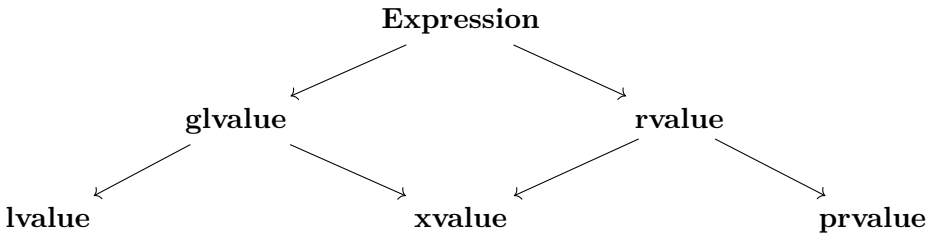
From the above, we can get a brief conclusion

- All names used as expressions are **lvalues**.
- All string literals used as expressions are **lvalues**.
- All other literals (`4.2`, `true`, `nullptr`, etc) are **prvalues**.
- All temporaries, especially objects returned by value, are **prvalues**.
- The result of `std::move()` is an **xvalue**.

The composite values are

- ***glvalue*** (*generalized value* which is the union of **lvalue** and **xvalue**),
- ***rvalue*** (the union of **xvalue** and **prvalue**).

All of these types create the following structure

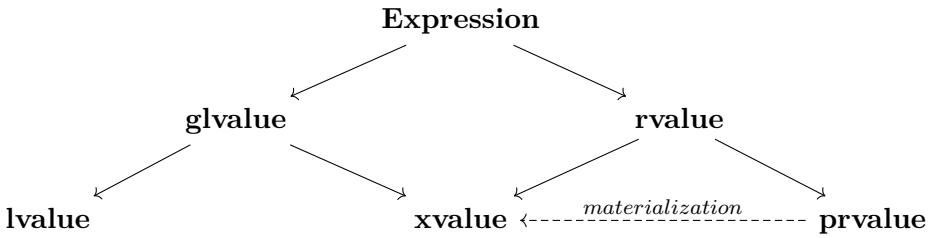


It needs to be emphasized, that **glvalues**, **prvalues** and **xvalues** are **terms for expressions and not for values**, so these names are a bit misleading. For instance, a variable itself isn't an **lvalue**, only the expression denoting the variable is **lvalue**

```
int i = 3;  // i is a variable, not an lvalue
int j = i;  // i is lvalue in this expression
```

C++17

C++17 didn't change these value categories, so it stays as it was, but clarifies their semantic meaning



To explain value categories now is that in general, we have two kinds of expressions

- **glvalues** - expressions for **locations of object or functions**,
- **prvalues** - expressions for **initializations**.

An **xvalue** is then considered a special location, representing an object whose resources can be reused, usually because it is near the end of its lifetime. The new thing in C++17 is the term **materialization**, for the moment a **prvalue** becomes a temporary object. Thus, a temporary materialization conversion is a **prvalue** $\rightarrow$ **xvalue** conversion.

6.2 General

6.2.1 Bit and Byte

Bit is the smallest unit of storage. Its value is either 0 or 1.
Byte is a collection of 8 bits.

6.2.2 Big Endian and Little Endian

Endianness refers to the order in which bytes are stored in memory for multi-byte data types (such as integers and floating-point numbers). The two main types of endianness are **big endian** and **little endian**. For example consider a 4-byte integer

$$\text{num} = 0x12345678 \tag{6.1}$$

The hexadecimal number `0x12345678` consists of four bytes

`0x12`, `0x34`, `0x56`, `0x78`

The way these bytes are stored in memory depends on the system's endianness.

Big Endian

In a big-endian system, the **most significant byte (MBS)** is stored at the lowest memory address (first). The bytes are stored in the same order as they appear in the number

Address	Byte #1	Byte #2	Byte #3	Byte #4
1000	0x12	0x34	0x56	0x78

This is the convention used by some architectures like Motorola 68k and SPARC.

Little Endian

In a little-endian system, the **least significant byte (LSB)** is stored at the lowest memory address (first). The bytes are stored in reverse order regarding the previous case

Address	Byte #1	Byte #2	Byte #3	Byte #4
1000	0x78	0x56	0x34	0x12

This is the convention used by x86 and ARM processors in little-endian mode.

6.2.3 Compiler and Linker

Compiler

A **compiler** is a program that translates high-level source code (written in languages like C, C++, Java, etc.) into a lower-level language that a computer can understand, typically machine code or assembly language. Its major responsibilities are

1. **Lexical Analysis:** Breaks the code into tokens (like keywords, variables, and symbols).
2. **Syntax Analysis:** Checks if the structure of the code follows the rules of the programming language.
3. **Semantic Analysis:** Ensures the code makes logical sense (e.g., type-checking and variable declarations).
4. **Intermediate Code Generation:** Creates an abstract, simplified version of the code.
5. **Optimization:** Improves the code for better performance (e.g., reducing unnecessary instructions or eliminating redundant computations).
6. **Code Generation:** Converts the optimized intermediate code into machine code or assembly language.

The output of the compiler is usually an **object file** (e.g., `.o` or `.obj`), which contains machine code but is not yet executable. Some compilers also generate debugging information, which can assist in troubleshooting during the later stages of development.

Linker

A **linker** is a program that combines the object files produced by the compiler into a single executable file that can run on a computer. Its responsibilities are

1. **Combining Object Files:** Links together multiple object files (e.g., from different source files or libraries) into a single program.
2. **Resolving Symbols:** Matches function calls and variable references to their definitions. For example, if we call a function defined in another file, the linker ensures the correct function is used.
3. **Library Handling:** Links in additional libraries (such as the standard library or third-party libraries) that our program depends on.
4. **Address Assignment:** Determines the memory locations for functions, variables, and other components of the program.
5. **Final Output:** Produces an executable file (e.g., `.exe` on Windows or an ELF file on Linux) that the operating system can load and execute.

The linker also ensures that all external references (e.g., function calls, global variables) are resolved and all necessary components are included. If any required symbols or files are missing, the linker will generate errors, preventing the creation of the executable.

6.2.4 Computational and memory complexity

Computational and memory complexity are concepts used in computer science and algorithm analysis to evaluate the efficiency of algorithms in terms of their performance and resource requirements.

Computational Complexity

Computational complexity measures **how many computational operations an algorithm needs to perform** to solve a task for input data of a given size. It describes how the runtime of the algorithm increases with the size of the input data. We have three main types of Computational Complexity:

- **Worst case** – analyzes the runtime of the algorithm in the worst possible scenario.
- **Average case** – analyzes the runtime of the algorithm in an "average" scenario.
- **Best case** – analyzes the runtime of the algorithm in the most optimistic scenario.

Asymptotic notation (e.g., Big-O notation) is used to describe how the runtime of the algorithm changes for large input sizes, for example

- $\mathcal{O}(1)$ – Constant complexity: the runtime of the algorithm does not depend on the size of the input.
- $\mathcal{O}(\log n)$ – Logarithmic complexity: the algorithm becomes faster for large inputs (e.g., binary search).
- $\mathcal{O}(n)$ – Linear complexity: the runtime grows proportionally to the number of input elements.

- $\mathcal{O}(n^2)$ – Quadratic complexity: the algorithm performs operations proportional to the square of the input size (e.g., bubble sort).
- $\mathcal{O}(2^n)$ – Exponential complexity: the algorithm becomes very slow even for small inputs (e.g., brute-force solution to the traveling salesman problem).

Memory Complexity

Memory complexity measures **how much memory an algorithm needs** to process input data of a given size. It includes memory used by:

1. Input data,
2. Auxiliary variables,
3. Data structures used in the algorithm (e.g., stacks, queues),
4. Recursive function calls.

There are two main categories

- **Fixed memory:** Occupied by variables whose size does not depend on the input size.
- **Dynamic memory:** Occupied by data structures that grow with the input size.

Some examples should clarify the explanation above

- $\mathcal{O}(1)$: The algorithm requires a constant amount of memory, regardless of the input size.
  - Example: Calculating the average of an array (summing elements).
- $\mathcal{O}(n)$: The algorithm requires memory proportional to the input size.
  - Example: Copying an array.
- $\mathcal{O}(n^2)$: The algorithm requires memory proportional to the square of the input size.
  - Example: A two-dimensional array for matrix computations.

Comparison of Computational and Memory Complexity

Category	Computational Complexity	Memory Complexity
What it measures?	Number of computational operations	Amount of memory required
Unit of measurement	Operations	Memory units (e.g., bytes)
Notation	$\mathcal{O}(n), \mathcal{O}(\log n), \mathcal{O}(n^2), \ldots$	$\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(n^2), \ldots$

6.2.5 Computational and Memory Complexity for STL containers

Container	Operation	Time	Space
std::vector	Insert (end)	$O(1)$ amortized	$O(n)$
	Insert (middle)	$O(n)$	$O(n)$
	Delete (end)	$O(1)$	$O(n)$
	Delete (middle)	$O(n)$	$O(n)$
	Search (by index)	$O(1)$	$O(1)$
	Search (value)	$O(n)$	$O(n)$
std::deque	Insert (end)	$O(1)$ amortized	$O(n)$
	Insert (beginning)	$O(1)$ amortized	$O(n)$
	Insert (middle)	$O(n)$	$O(n)$
	Delete (end)	$O(1)$	$O(n)$
	Delete (beginning)	$O(1)$	$O(n)$
	Delete (middle)	$O(n)$	$O(n)$
	Search (value)	$O(n)$	$O(n)$
std::list	Insert (anywhere)	$O(1)$ with iterator	$O(n)$
(Doubly Linked List)	Delete (anywhere)	$O(1)$ with iterator	$O(n)$
	Search (value)	$O(n)$	$O(n)$
std::forward_list	Insert (anywhere)	$O(1)$ with iterator	$O(n)$
(Singly Linked List)	Delete (anywhere)	$O(1)$ with iterator	$O(n)$
	Search (value)	$O(n)$	$O(n)$
std::set	Insert (unique)	$O(\log n)$	$O(n)$
(Balanced Binary Search Tree)	Delete (by value)	$O(\log n)$	$O(n)$
	Search (value)	$O(\log n)$	$O(n)$
std::multiset	Insert (duplicates allowed)	$O(\log n)$	$O(n)$
(Balanced Binary Search Tree)	Delete (all instances)	$O(\log n)$	$O(n)$
	Search (value)	$O(\log n)$	$O(n)$
std::unordered_set	Insert (unique)	$O(1)$ average, $O(n)$ worst	$O(n)$
(Hash table)	Delete (by value)	$O(1)$ average, $O(n)$ worst	$O(n)$
	Search (value)	$O(1)$ average, $O(n)$ worst	$O(n)$
std::map	Insert (key-value)	$O(\log n)$	$O(n)$
(Balanced Binary Search Tree)	Delete (by key)	$O(\log n)$	$O(n)$
	Search (by key)	$O(\log n)$	$O(n)$
std::multimap	Insert (duplicates allowed)	$O(\log n)$	$O(n)$
(Balanced Binary Search Tree)	Delete (all instances)	$O(\log n)$	$O(n)$
	Search (by key)	$O(\log n)$	$O(n)$
std::unordered_map	Insert (key-value)	$O(1)$ average, $O(n)$ worst	$O(n)$
(Hash Table)	Delete (by key)	$O(1)$ average, $O(n)$ worst	$O(n)$
	Search (by key)	$O(1)$ average, $O(n)$ worst	$O(n)$
std::stack	Push	$O(1)$	$O(n)$
(Uses std::deque)	Pop	$O(1)$	$O(n)$

Container	Operation	Time	Space
	Top	$O(1)$	$O(1)$
std::queue (Uses std::deque)	Enqueue (push)	$O(1)$	$O(n)$
	Dequeue (pop)	$O(1)$	$O(n)$
	Front	$O(1)$	$O(1)$
std::priority_queue (Heap based)	Insert (push)	$O(\log n)$	$O(n)$
	Delete (pop top)	$O(\log n)$	$O(n)$
	Top	$O(1)$	$O(1)$

6.2.6 CPU cache and preferching

A **CPU cache** is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have a hierarchy of multiple cache levels (L1, L2, often L3, and rarely even L4), with different instruction-specific and data-specific caches at level 1. The cache memory is typically implemented with a static random-access memory (SRAM), in modern CPUs by far the largest part of them by chip area, but SRAM is not always used for all levels (of I- or D-cache), or even any level, sometimes some latter or all levels are implemented with eDRAM.

When trying to read from or write to a location in the main memory, the processor checks whether the data from that location is already in the cache. If so, the processor will read from or write to the cache instead of the much slower main memory.

Cache prefetching is a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in slower memory to a faster local memory before it is actually needed (hence the term 'prefetch'). Most modern computer processors have fast and local cache memory in which prefetched data is held until it is required. The source for the prefetch operation is usually the main memory. Because of their design, accessing cache memories is typically much faster than accessing main memory, so prefetching data and then accessing it from caches is usually many orders of magnitude faster than accessing it directly from main memory. Prefetching can be done with non-blocking cache control instructions.

Cache prefetching can be accomplished either by hardware or by software.

1. **Hardware-based prefetching** is typically accomplished by having a dedicated hardware mechanism in the processor that watches the stream of instructions or data being requested by the executing program, recognizes the next few elements that the program might need based on this stream and prefetches into the processor's cache.

2. **Software-based prefetching** is typically accomplished by having the compiler analyze the code and insert additional prefetch instructions in the program during compilation itself.

Bibliography

[1] Gręboszz J., *Misja w nadprzestrzeń C++14/17*, Edition I, Helion, Gliwice, 2020

[2] Gręboszz J., *Opus Magnum C++11*, Edition II, Helion, Gliwice, 2020

[3] Josuttis N.M., *C++17 The complete guide*, Version 2019/02/16, Lean Publishing, 2019

[4] Josuttis N.M., *C++ move semantics The complete guide*, Version 2020-12-19, Lean Publishing, 2020

[5] Prata S., *C++ Primer Plus*, Edition IV, Addison-Wesley, Indiana, 2012

[6] Williams A., *C++ Concurrency in action*, Edition II, Manning Publications Co, 2012