

// one shot c++ ----- LETS DO IT-----

/*

What is C++?#

C++ is a general-purpose, object-oriented programming language developed by Bjarne Stroustrup in 1979. It is an extension of C. Depending on our requirements, we can use C++ to either code in “C style” or “object-oriented style”.

C++ features#

The most important features of C++ are listed below.

C++ features

Platform dependent#

C++ is a platform-dependent language. This means that a program written and compiled on a particular operating system won’t run on any other operating system. For example, a C++ program developed and compiled in the Windows operating system will not run on macOS, Linux, or Android OS.

Intermediate-level language#

C++ supports the features of both high-level and low-level programming languages. That is why it is known as an intermediate-level programming language.

Object-oriented#

C++ is an object-oriented programming language (OOP). OOP makes development easier by breaking a complex problem into subproblems using objects.

Structured#

C++ is a structured programming language, which means that we can divide a program into different parts using functions.

Statically-typed#

C++ is a statically-typed language. In a statically-typed language, the variable types are explicitly declared and are determined at compile time.

Compiled#

C++ is a compiled programming language, which means that programs written on it cannot be executed without compilation.

Support-rich library#

C++ Standard Template Library (STL) provides a lot of inbuilt functions. STL makes development faster and easier.

Speed#

The compilation and execution time of C++ is much faster than other general-purpose programming languages.

Multi-paradigm#

C++ supports different styles of programming. Developers can choose a programming style according to their use case.

Pointer#

C++ supports the features of pointers. Pointers are used to interact with the memory.

History of C++

In 1979, Bjarne Stroustrup, while working on his Ph.D. thesis came across a simulation-based language called Simula, which also supported the object-oriented model. He found it very fascinating.

He thought of implementing the object-oriented model in software development but Simula was very slow for practical use.

He added the object-oriented model in C language without compromising its speed and lower-level functionality.

Cfront was the first “C with classes” compiler. It is derived from a C compiler called CPre. Cfront was used to translate the code of “C with classes” to ordinary C.

History of C++

In 1983, “C with classes” was renamed to C++. The “++” is an incremental operator that gives some insight into the fact of how C++ is an extension of the C language.

In 1985, C++ was released as a commercial product.

In 1990, “The Annotated C++ Reference Manual” was released.

In 1998, the C++ standards committee released the first international standard for C++, known as ISO C++ 98.

In 2003, some bugs in C++98 were fixed, and a new version of C called C++03 was introduced.

Later on, different versions called C++11, C++14, and C++17 were released.

Applications of C++#

C++ is widely used in today's embedded systems, browsers, graphical user interfaces, music players, video games, operating systems, compilers, system drivers, databases, and cloud computing.

svg viewer

Applications of C++

C++ supports object-oriented programming. Therefore, it lowers the development cost by giving a clear structure to the program and allowing the reusability of code.

Its syntax is close to C# and Java. Therefore, it is easy for the programmer to switch between these languages.

It is efficient and close to machine programming.

It is fun and simple to learn!

Who's using C++?#

Many popular systems use C++ for implementing the critical part of their system. These are:

Google

Google has written some of its parts in C++, which includes the Google Chromium-browser and Google file system.

Bloomberg

Bloomberg quickly provides market financial information to investors around the world. The basic development environment for Bloomberg is written in C++.

MySQL

MySQL is the most frequently used open-source relational database management system. It is also coded in C++.

Microsoft

Microsoft is an American based technology company. For decades, Microsoft Windows was the world's most used operating system. Most parts of Microsoft Windows are written in C++.

Adobe

Most Adobe applications are written in C++. Examples include Adobe Illustrator, Adobe Photoshop, and Adobe Premier.



Due to their power and ease of use, C and C++ were used in the programming of the special effects for Star Wars.

```
*/  
/*  
Hello, World!" program#
```

Below is the source code for your first C++ program. First, have a look at the code, then we will discuss it.

```
*/  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main()  
// {  
//   cout << "Hello, World!"  
// }  
// ======  
/*
```

Explanation#

The highlighted lines in the above program will appear in every C++ program. We will cover the functionality of these lines in the upcoming chapters. For now, just remember that we will always write our code inside the curly braces {}.

Note: If you want to dive into the details, you can visit this link [Anatomy of a “Hello World!” program](#).

The segment of the program we want to pay attention to right now is on Line No. 6. When this line executes, it will print Hello, World! on the console.

svg viewer

“Hello, World!”#

In C++, we write our content inside double-quotes. Anything written inside double quotes is known as a string. Here, Hello, World! is a string. Don't worry about the details of a string yet. We'll cover these details in an upcoming section of the course.

<<#

<< is called the insertion or output operator. It takes the content written on its right-hand side and inserts it into the cout. You will learn thoroughly about operators in C++ in an upcoming chapter.

cout#

cout knows that it should print everything that is sent via an insertion operator onto the console.

:#

A statement is a command that the programmer gives to the computer. Here, Line no. 6 is a statement. It instructs the machine to display Hello, World! on the console. Every statement in the C++ program ends with a semicolon, which indicates the end of the current statement and also that the next one is ready to execute.

*/

/*

Printing Styles:

C++ output#

In the previous lesson, you learned that cout is used with an insertion operator << to print anything on the console. We can use multiple cout statements in a program.

*/

// =====

// #include <iostream>

// using namespace std;

// int main()

// {

// cout << "Educative is an interactive platform ";

// cout << "for learning";

// }

// =====

/*

The code above uses two cout statements and prints Educative is an interactive platform for learning. Have you noticed something?

The second cout statement prints for learning right after Educative is an interactive platform. It means the cout statement does not add a new line at the end of the text. But what if we want to print our text in multiple lines?

endl in C++#

We can use endl with cout to add a new line after the text. Let's write a code to print the string in multiple lines.

*/

// =====

// #include <iostream>

// using namespace std;

```
// int main()
//{
//   cout << "Educative is an interactive platform" << endl;
//   cout << "for learning";
//}
//=====
/*
```

When we run the above code, it prints the text on multiple lines. endl in the above code adds a new line right after Educative is an interactive platform. Therefore, for learning is printed on a separate line.

C++ Comments:

Introduction to comments#

Comments are statements written inside the source code to make it easily understandable. The compiler ignores the comments, which is why they don't affect the logic of our program.

 Note: We write comments for the documentation of our program and to help other people understand our code.

Example#

Suppose Educative hires you for the maintenance of its website. The source code of a website consists of thousands of lines. To repair the code, you have to understand it first. It's quite an impractical approach to read every line of the code. Here, comments come to the rescue!.

Types#

In C++, we can add:

Single-line comments

Multi-line comments

Single-line comments#

We write a single-line comment after two backslashes //. The compiler ignores anything written after //. Let's add single-line comments in a C++ program.

```
/*
// =====
// #include <iostream>

// using namespace std;
// int main()
//{
//   // I am a single-line comment
//   // Compiler will ignore me
//   // cout << "Hey"
//   cout << "Hello World";
//}
// =====
/*
```

In the above code, we wrote single-line comments from Lines 5 to 7. Therefore, the compiler did not execute them.

Multi-line comments#

Multi-line comments start with `/*` and end with `*/`. The compiler will ignore anything written inside `/*...*/`. Let's add multi-line comments in a C++ program.

```
/*
=====
// #include <iostream>

// using namespace std;
// int main()
//{
//   /* I am a multi-line comment
//   Compiler will ignore me
//   cout << "Hey" */

//   cout << "Hello World";
//}
// =====
/*
```

In the code above, we wrote multi-line comments from Lines 5 to 7. Therefore, they are ignored by the compiler.

 Best coding practice: Add meaningful comments in source code to make it understandable.

White spaces#

C++ compiler simply ignores the white spaces in source code. White space may include spaces, tabs, and blank lines. Let's add white space in a program.

```
/*
=====
// #include <iostream>

// using namespace std;

// int main()
//{
//   //
//   //
//   //
//   //
//   cout << "Hello World";
//}
// =====
```

```
/*
```

When you run the above code, the compiler does not take the blank lines, tabs, and spaces into consideration, i.e., they do not impact the output of the program in any way.

 Best coding practice: Add spaces in a program to increase its readability.

```
*/
```

```
//
```

```
//
```

```
// -----
```

```
/*
```

Variables#

Suppose we have cabinets of different types. In each cabinet, we can only put one item. To store something in a cabinet, first, we'll decide the cabinet type. Then, we'll put a unique label on a cabinet to keep track of the item it contains.

Variables

A variable is just like a cabinet that can store data. To store something in a variable, we decide its data type (similar to a cabinet type) and give it a unique name (analogous to a label in the above diagram). Each variable can store exactly one item, but the data stored in a variable can be changed over time.

In terms of programming language, a variable is a location in the computer's memory where we can store data. The value of this data can be changed during the execution of a program. Each variable has a unique and meaningful name which is known as an identifier.

 Note: A big advantage of variables is that they allow us to store data so that we can use it later in the program. We can always change the value of a variable during the running program.

Variable declaration#

A variable declaration means that we want the compiler to reserve a space for a data with the given name and type.

The basic syntax for declaring a variable in C++ is:

```
variable_datatype variable_name;
```

 Note: Don't worry about the data types yet. We will cover these in detail in the next chapter. For this chapter, we will just work with int. int is used to store an integer value in a variable. A variable declared with an int data type cannot store floating-point values.

```
*/
```

```
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//   int number;
//}
//=====
/*
```

We can declare more than one variable in a single line.

```
int number1, number2, number3;
```

The above line declares three variables: number1, number2, and number3.

Variable initialization#

Variable initialization means to actually store value in the reserved space.

The basic syntax for initializing a variable in C++ is given below:

In C++, we will write the following lines for initializing the variable of integer type:

```
*/
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   int number;
//   number = 100;
//}
//=====
/*
```

 Do you know? C++ is a statically-typed language. In a statically-typed language, a variable is declared with its type before its first use.

Variable declaration and initialization in one step#

At this point, you're probably wondering if you can simply just declare a variable and assign it a value in one go? The answer is yes! We can do this in the following way:

```
/*
// =====
// #include<iostream>

// using namespace std;

// int main()
//{
//    int number = 100;
//}
// =====

/*
```

Example program#

We can use a variable to keep track of the current amount in our bank account. Suppose you have \$100 in your bank account. After some time, your friend transfers \$20 to it. Now, the current amount is \$120. Let's write a code in C++ that can keep track of your account balance.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {
//    // Declares a variable current_amount
//    int current_amount;
//    // Initialize a variable current_amount to 100
//    current_amount = 100;
//    // Prints the value of current_amount
//    cout << "Your current amount is: " << current_amount << endl;
//    // Updates the value of current_amount
//    current_amount = 120;
//    // Prints the updated value of current_amount
//    cout << "Your current amount is: " << current_amount << endl;
//}

// =====
```

Line No. 7: Declares a variable `current_amount` that will store the integer value.

Line No. 9: Initially, there is \$100 in a bank account. Therefore, we store 100 in variable `current_amount`.

Line No. 11: Displays the value of `current_amount`.

 To print the value of a variable, use cout followed by the insertion operator << and variable name.

Line No. 13: When your friend transfers \$20 to your account, the current_amount becomes \$120. Therefore, we update the value of the current_amount to 120, changing the value of a variable during the program execution.

Line No. 15: Displays the updated value of current_amount.

Identifiers in C++:

A variable in C++ is given a unique name that is known as an identifier.

 Best coding practice: Use descriptive and meaningful names for the variables to make the code self-explanatory.

Rules for naming a variable#

The general rules for naming a variable are:

An identifier can only contain uppercase alphabets (A to Z), lowercase alphabets (a to z), numbers (0 to 9), and underscore (_).

The first letter of an identifier can be an alphabet or an underscore.

 Best coding practice: It is not good practice to start an identifier with an underscore.

The first letter of an identifier cannot be a number.

C++ is a case-sensitive language. Therefore, an identifier written in the upper case will be different from one written in lower case.

 Note: numbers and Numbers are two different identifiers.

An identifier cannot contain white space.

An identifier cannot have special characters such as &, @, *, !, etc.

We cannot use keywords as identifiers.

 Note: Keywords are a collection of reserved words and predefined identifiers in a language used for specific purposes.

Example program with valid identifiers#

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   int number1;
//   int _number;
//   int number;
// }
// =====
```

```
/*
```

The code given above contains valid identifiers. Therefore, running the code does not generate an error.

Example program with invalid identifiers#

Let's write some invalid variable names.

 Error: The program given below will generate an error.
*/

```
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
//   int 1;  
//   int number 1;  
//   int return;  
// }  
// ======  
  
/*  
C++ Constants/Literals
```

Constants or literals#

Let's write a program in which we will overwrite the value of a variable.

*/
// ======

// #include <iostream>

// using namespace std;
// int main()
// {
// int number = 10;
// cout << "Number = " << number << endl;
// number = 20;
// cout << "Number = " << number << endl;
// number = 30;
// cout << "Number = " << number << endl;
// }
// ======

```
/*
```

In the above code, we have declared a variable number. We see that we can overwrite the value of the number during the execution of the program. Initially, the value of the number is 10, then 20, and finally 30. What if we want to declare a variable whose value remains fixed throughout the program execution? This is where constants come in.

```
*/
```

```
/*
```

Constants are similar to variables except that we can't change their value during the code execution.

Define constants using the const keyword#

In C++, we can use the const keyword to declare a constant. The basic syntax for creating a constant is:

```
const const_datatype const_variable_name = value; (compulsory definition )
```

 Note: Don't worry about the constant data types yet. We will cover these in detail in the next chapter. In this chapter, we just have to work with int. int is used to store an integer value in a constant. A constant declared with int data type cannot store floating-point values.

 Note: Don't worry about the constant data types yet. We will cover these in detail in the next chapter. In this chapter, we just have to work with int. int is used to store an integer value in a constant. A constant declared with int data type cannot store floating-point values.

Example program#

Let's write a program in which we define a constant and print its value.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//   const int number = 10;
//   cout << "Number = " << number << endl;
//   // Uncommenting the following line will result in a compiler error
//   // number = 20;
//}

// =====
/*
```

Line No. 6: Declares a constant number that can take an integer value. We store 10 in a number.

Line No. 7: Displays the value of the number.

 Common programming error: In C++, you have to initialize a constant at the time of its declaration. If you don't initialize a constant at the time of creating it, an error will occur.

```
/*
//
```

```
//
```

```
//
```

```
//
```

```
//
```

```
/*
```

Data Types:

Introduction to computer memory#

Consider an Excel sheet that consists of a large number of cells where each cell is used to store data. We can locate each cell in the Excel sheet using a row and column number.

Computer memory is just like an Excel sheet that contains cells of data arranged in a logical order.

However, in the computer's memory, cells are arranged linearly. Each cell in memory can store 1 byte of data. As we know, 1 byte = 8bit; therefore, each cell can store any value from 0–255.

Introduction to data types#

```
*/
```

```
// =====
// #include <iostream>
```

```
// using namespace std;
```

```
// int main()
//{
//    int number = 10;
//    cout << "Number = " << number << endl;
//}
```

```
// =====
```

```
/*
```

The code given above declares a variable number of type int, stores 10 in number, and then prints its value. How does the compiler know how much memory should be allocated to the particular variable? Here is where data types come in!

In the variable declaration, it is necessary to specify the data type before the identifier. This is why C++ is a statically-typed language. Data type reserved the space for the particular variable based on its type. Here, int is a data type, and it can only store an integer value.

Data type tells the compiler what type of data a particular variable can store. The compiler allocates the memory to the variable based on its data type.

Data types do the following two things:

Specify the type of value a particular variable can store, i.e., variable declared with int data type can store integer values only.

Reserve the number of bytes for a variable in memory, i.e., a variable with int type will reserve four consecutive bytes in memory. With 4 bytes, we can represent any value from -2147483648 to 2147483647. Therefore, the range of values that a variable can store depends upon its data type.

Data types in C++#

C++ supports the following datatypes:

Primitive or fundamental data types

Derived data types

User-defined data types

Data types in C++

Primitive or fundamental data types#

Primitive data types are predefined data types. These are:

Integer

Floating-point

Double

Void

Character

Boolean

Derived data types#

Data types that are derived from primitive data types are known as derived data types. These are:

Function

Arrays

Pointers

Reference

User-defined data types#

Data types that are defined by the user are known as user-defined data types. These are:

Structure

Union

Enum

Class

Typedef

The integer data type comprises all positive and negative whole numbers. We use the int keyword to define the integer data type. A variable of int type is allocated 4 bytes of memory. It can store any value from -2³¹ to 2³¹-1.

```
int integer_number = 100;
```

💡 Do you know? If you store 100.5 in a variable of integer type, it would be truncated to 100.

Floating-point data type#

The floating-point data type contains a number with a fractional part. We use the float keyword to define the floating-point data type. A variable of a float type is allocated 4 bytes of memory. It can store any value from -231 to 231-1.

```
float float_number = 10.7;
```

Double data type#

The double data type contains the number with the fractional part. We use the double keyword to define the double data type. A variable of double type is allocated 8 bytes of memory. It can store any value from -263 to 263-1.

```
double double_number = 10.65417;
```

Difference between float and double#

The precision of a floating-point number is the number of digits that can be stored after a decimal point. A float can store seven digits after a decimal point precisely. Whereas, double can store 15 digits after a decimal point precisely. It is recommended to use double for floating-point values.

 Note: We can store a scientific number in double or float data types. The number after e shows the power of 10.

Example program#

```
/*
=====
=====

// #include <iostream>

// using namespace std;

// int main()
//{
// Create variable of different types
//   int integer_number = 10;
//   float float_number = 10.5;
//   /* Stores scientific value. The number after "e"
//    represents the power of 10*/
//   float float_scientific = 9.007e4;
//   double double_number = 10.5;

//   // Prints value of variables
//   cout << "int = " << integer_number << endl;
//   cout << "float = " << float_number << endl;
//   cout << "float_scientific = " << float_scientific << endl;
//   cout << "double = " << double_number << endl;
//}

/*
=====
```

```
/*
```

Character data type#

The character data type contains a single character from the ASCII set. We use the `char` keyword to define the character data type. A variable of `char` type is allocated 1 byte of memory. It can store any Unicode value from -27 to 27-1.

```
char character = 'b';
```

 Note: A `char` value is always written in single quotation marks.

Boolean data type#

The boolean data type stores a logical value. It can store true and false. We can also use 1 to represent true and 0 to represent false. We use the `bool` keyword to define the boolean data type. A variable of `bool` type is allocated 1 byte of memory.

```
bool boolean = false;
```

Example program#

```
*/
```

```
//
```

```
=====
```

```
==
```

```
// #include <iostream>
```

```
// using namespace std;
```

```
// int main()
```

```
//{
```

```
//   char character = 'A';
```

```
//   bool boolean = 7;
```

```
//   // Prints value of variables
```

```
//   cout << "char = " << character << endl;
```

```
//   cout << boolalpha;
```

```
//   cout << "bool = " << boolean << endl;
```

```
//}
```

```
//
```

```
=====
```

```
==
```

```
/*
```

In the code given above, we declare and initialize the variables of `char` and `bool` types. Then, we print their values on the console.

 Note: chars and bools are also stored as numbers.

```
*/
```

```
//
```

```
=====
```

```
=====
```

```
// #include <iostream>
```

```
// using namespace std;
```

```
// int main()
```

```
//{
```

```
// long a = false;
// int b = 'C';
// cout << "variable a = " << a << endl;
// cout << "variable b = " << b << endl;
// return 0;
//}
```

//

```
=====
=====
```

/*

Void data type#

The void data type represents an entity without a value. When the data type is void, no memory is allocated.

 Note: We will see the use of void data type in functions.

Data Type Modifiers:

The maximum value that can be stored in a variable of type int is 2147483647. What if we want to store a value greater than 2147483647.

```
*/
//
```

```
=====
=====
```

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

```
// int main()
//{
//    // Initialize variable
//    int number = 2147483649;
//    // Display variable value
//    cout << number;
//}
//
```

```
=====
=====
```

/*

If we run the code given above, it does not give us the expected output. The above code should print 2147483649, but it is printing -2147483647 in output. So how can we handle values greater than the range of a data type? Similarly, how can we decrease the amount of space allocated to a particular variable? Here, data type modifiers come to the rescue.

Data type modifiers are used with primitive data types to change the meaning of predefined data types according to the situation.

Data type modifiers in C++#

C++ supports the following data type modifiers:

long
short
unsigned
signed

We can use data type modifiers with int, double and char data types.

long#

long is used to increase the length of a data type to 4 more bytes. We can use long with int and double data types. Let's use a long modifier with built-in data types.

*/

//

=====

=====

```
// #include <iostream>
```

```
// using namespace std;
```

```
// int main()
```

```
// {
```

```
//   // Initialize variables
```

```
//   int integer = 2147483649;
```

```
//   long int long_integer = 2147483649;
```

```
//   // Display variables value
```

```
//   cout << "integer = " << integer << endl;
```

```
//   cout << "long_integer = " << long_integer << endl;
```

```
// }
```

//

=====

=====

/*

From the code above, we can see that we can precisely store values greater than the int range using a long modifier.

short#

short decreases the available length of a data type to 2 bytes. We can use short with an int data type.

*/

//

=====

=====

```
// #include <iostream>
```

```
// using namespace std;
```

```
// int main()
```

```
// {
```

```
//   // Initialize variables
```

```
// int integer = 32768;
// short int short_integer = 32768;
// // Display variables value
// cout << "integer = " << integer << endl;
// cout << "short_integer = " << short_integer << endl;
// }
```

```
//
```

```
=====
=====
```

```
/*
```

The program given above generates unexpected results because short int reserves less space in memory.

An int reserves 4 bytes in memory. However, using a short modifier with int reserves 2 bytes in memory. Therefore, the maximum value that can be represented with short int is 32767.

unsigned#

unsigned allows us to store positive values only. We can use unsigned with char and int data types. With unsigned int, we can store any value from 0 to 4294967295. With unsigned char, we can store any value from 0 to 255.

```
*/
```

```
//
```

```
=====
// #include <iostream>
```

```
// using namespace std;
```

```
// int main()
//{
// // Initialize variables
// int integer = -10;
// unsigned int unsigned_integer = -10;
```

```
// char character = 'A';
// unsigned char unsigned_character = 'B';
// // Display variables value
// cout << "integer = " << integer << endl;
// cout << "unsigned_integer = " << unsigned_integer << endl;
```

```
// cout << "character = " << character << endl;
// cout << "unsigned_character = " << unsigned_character << endl;
// }
```

```
//
```

```
=====
=====
```

```
/*
```

```
signed#
signed allows us to store both positive and negative values. We can use signed with char and int data
types. With signed int, we can store any value from -2,147,483,648 to 2,147,483,647. With signed char, we can
store any value from -128 to 127.

*/
// =====
// #include <iostream>
// using namespace std;

// int main() {
//   // Initialize variables
//   int integer = -90;
//   signed int signed_integer = -90;

//   char character = 'A';
//   signed char signed_character = 'A';
//   // Display variables value
//   cout << "integer = " << integer << endl;
//   cout << "signed_integer = " << signed_integer << endl;

//   cout << "character = " << character << endl;
//   cout << "signed_character = " << signed_character << endl;

// }
// =====
```

```
/*
```

Note: From the above table, it is obvious that signed is the default declaration for int and char (signed int is similar to int, and signed char is similar to char).

Type-Casting:

Suppose you have initialized a variable with a char data type. At some point in a program, you need its integer value. In such situations, type-casting comes in.

Type-casting is a way to convert the value of one data type to another data type.

Types of type-casting#

Type-casting has two types:

Implicit casting

Explicit casting

Implicit casting#

In implicit casting, the compiler automatically converts one data type to another.

For example, if you store a floating-point value into a variable of integer type, the compiler will convert the float value into int without any user intervention.

Example program#

```
*/
```

```
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   // Stores floating point value in variable of type int
//   int int_value = 13.9;
//   // Displays the value of variable
//   cout << int_value;

// }
```

```
/*
```

In the code above, we have stored a floating-point value in the variable of type int. The compiler automatically knows that it should truncate the value after the decimal point. Therefore, it stores 13 in the variable int_value.

We should always do type-casting from smaller to larger data types. Otherwise, you can lose your data. For example, in the above program, we are losing the information after the decimal point.

The arrows in the figure given below show the order in which we can do the conversion without any data loss. For example, we can convert short int into int without any loss of data or precision.

Explicit casting#

In explicit casting, the user manually converts one data type to another. The basic syntax for explicit type casting in C++ is:

```
(datatype)variable_name;
```

Suppose you want to know the ASCII value of a character stored in a variable. Let's write a program.

```
*/
```

```
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   // Initializes a variable of char data type
//   char character = 'A';
```

```
// // Declares a variable of int type
// int ASCII;
// // Converts char data type into int explicitly
// ASCII = (int) character;
// // Prints value of variable
// cout << "ASCII value = " << ASCII;

// }
```

```
//=====
```

```
/*
Strings and Escape Sequences:
```

String#

String is plain text that represents alphanumeric data. A string comprises one or more characters. A character can be a letter, number, or space.

Note: We consider string as text even if it contains a number. If that is so, then how can the string be distinguished from the actual code? To differentiate both, we always write string data inside double-quotes.

```
*/
// =====
// #include <iostream>

// using namespace std;

// int main()
// {
//   // Initialize a string variable
//   string text = "Hey12345";
//   // Displays value of string variable
//   cout << text;
// }
// =====
/*
```

In the code above, because of the double quotes, Hey12345 is recognized as a string and not as a number.

 Note: Strings are not allocated a fixed amount of memory during the time of declaration.

Escape sequence#

An escape sequence comprises two or more characters that are used to modify the format of the output. The first character in the escape sequence is the backslash \. The remaining characters determine what our escape sequence will actually do. Here is a list of the most commonly used escape sequences:

- \n : new line
- \t: horizontal tab
- \" : insert double quotation in the text.
- \r: moves the cursor to the beginning of the current line.
- \\": display backslash character in the text.
- \': inserts single quotes

```

\b: inserts a backspace

*/
=====

// #include <iostream>

// using namespace std;

// int main()
//{
//  // Initialize string variable with text and escape characters
//  string text = "Hello\n\tam\tJohn";
//  // Displays value of string variable
//  cout << text;
//}

=====
/*

```

hen we run the code above, it displays the text in a special format. First, it displays Hello. Then, it encounters the escape character \n, which causes it to move the cursor to the next line. In the next line it displays I, then it encounters an escape character \t and resultantly moves the cursor eight spaces towards the right. It prints am and then, once again, moves the cursor eight spaces towards the right because of an escape character, where it displays John.

C++ User Input

Introduction#

Until now, we have looked at output operation, where we take the data stored in memory and display it on the console. However, a program would be boring without any input operations. Imagine the Instagram app without any user interaction!

Input operation is the exact opposite of output operation. In input operation, we take data from the user and store it in the memory. In C++, an input device is a keyboard.

Syntax#

The basic syntax for taking input from the user is given below:

```
cin>>variabel_name;
```

We use a cin statement in combination with the extraction operator >> to take input from the user.

cin#

cin is connected to the keyboard. It takes anything coming from the keyboard and sends it to the extraction operator.

>>#

>> is called the extraction or input operator. It takes content from cin and stores it into the variable to its right.

variable_name#

In C++, we use `cin` to take user input from the keyboard. To use the input later, we must store it somewhere. We use variables to store input taken from the user.

Example program#

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//    // Declares variable
//    float number;
//    // Displays text
//    cout << "Please enter your number:" << endl;
//    // Waits for the user input
//    cin >> number;
//    // Displays entered number
//    cout << "You have entered: " << number;
//}

// =====

/*
Line No. 7: Declares a variable number of type float to store the user input.
```

Line No. 9: Displays Please enter your number: on the console and moves the cursor to the next line.

Line No. 11: `cin` is connected to the input device (the keyboard). It takes user input through the keyword. Then, the extraction operator `>>` is used to extract this input from `cin` and store it into a variable `number`.

Line No. 13: Prints the user input.

So far we have covered data types and user input from beginning to end.

Test your understanding by solving a simple challenge in the upcoming lesson.

Operators#

In computer language, an operator is a symbol that takes one or more values as input and outputs another value after performing a particular operation.

Operands are the data items on which an operation is being done.

Example#

Consider a group of four people sharing a pizza with eight slices. You want to give each person an equal share of pizza. How would you do that?

Operators come to the rescue in such situations. We will divide the number of pizza slices with the number of people present in a group.

 The operator operates upon the operands to do a specific task, just like a doctor operates on their patient to treat an injury.

Types of operators#

Based on the number of operands involved in an operation, we can divide the operators into three categories:

Unary operator

Binary operator

Ternary operator

Unary operator#

The unary operator operates on one operand.

Binary operator#

The binary operator operates on two operands.

Ternary operator#

The ternary operator operates on three operands.

Operators in C++#

Listed below are the types of built-in operators provided by C++. They are based on the type of operations that they perform.

Arithmetic operators

Assignment operators

Relational operators

Logical operators

Bitwise operators

Arithmetic Operators

Example program with int operands#

Consider two operands of type int. The value of operand1 is 50, and the value of operand2 is 26. Let's apply each arithmetic operator on them.

```
*/  
// ======  
// #include <iostream>  
// using namespace std;  
  
// int main() {  
//   // Initialize operand1 and operand2  
//   int operand1 = 50;  
//   int operand2 = 26;
```

```

// // Prints value of operand1 and operand2
// cout << "Values of operands are:" << endl;
// cout << "operand1 = " << operand1 << ", operand2 = " << operand2 << endl;
// // Adds operand1 and operand2; and print their result
// cout << "Addition = " << operand1 + operand2 << endl;
// // Subtracts operand1 and operand2; and print their result
// cout << "Subtraction = " << operand1 - operand2 << endl;
// // Multiplies operand1 and operand2; and print their result
// cout << "Multiplication = " << operand1 * operand2 << endl;
// // Divides operand1 and operand2; and print their result
// cout << "Division = " << operand1 / operand2 << endl;
// // Returns remainder of operand1 and operand2; and print it
// cout << "Modulus = " << operand1 % operand2 << endl;
// return 0;
// }

```

```
// =====
```

/*

Result of / operator#

All the operators in C++ show the same results that an ordinary calculator would show, except for the division operator. If you put 50/26 in a calculator, it returns 1.92307692308 in output. However, our C++ program is returning 1 in the output. So why does the C++ division operator show a result different than that of the calculator's division operator?

The reason is that the data type of our operands is int, which means that our output is also of type int. Therefore, C++ only gives you the whole number part of the quotient, excluding the remainder to keep the type consistent. If you want a quotient with a fractional part, use the float or double data type operands.

Example program with float operands#

Consider two operands of type float. The value of operand1 is 50.0, and the value of operand2 is 26.0.

```

*/
// =====
// #include <iostream>
// using namespace std;

// int main()
//{
//   // Initilaize operand1 and operand2
//   float operand1 = 50.0;
//   float operand2 = 26.0;
//   // Prints value of operand1 and operand2
//   cout << "Values of operands are:" << endl;
//   cout << "operand1 = " << operand1 << ", operand2 = " << operand2 << endl;
//   // Adds operand1 and operand2; and print their result
//   cout << "Addition = " << operand1 + operand2 << endl;
//   // Subtracts operand1 and operand2; and print their result
//   cout << "Subtraction = " << operand1 - operand2 << endl;
//   // Multiplies operand1 and operand2; and print their result
//   cout << "Multiplication = " << operand1 * operand2 << endl;

```

```

// // Divides operand1 and operand2; and print their result
// cout << "Division = " << operand1 / operand2 << endl;
// // Returns remainder of operand1 and operand2; and print it
// cout << "Modulus = " << operand1 % operand2 << endl;
// return 0;
//}

// =====

/*

```

Using % with float operands#

Uncomment line No 20 in the above code. An error will be generated.

This happens because using a mod operator with floating-point operands generates an error.

 We can also apply arithmetic operators to the char data type operands. In this case, operators operate upon the ASCII value of the characters.

Assignment and Compound Assignment Operator

The assignment operator takes the value on its right-hand side and assigns it to the operand on the left-hand side.

```
*/
```

```

// =====
// #include <iostream>
// using namespace std;

// int main()
//{
// // Assigns value to the operands
// int operand1 = 50;
// float operand2 = 26;
// double operand3 = 78;
// bool operand4 = true;
// char operand5 = 'A';
// string operand6 = "Welcome";

// // Prints value of the operands
// cout << "operand1 = " << operand1 << endl;
// cout << "operand2 = " << operand2 << endl;
// cout << "operand3 = " << operand3 << endl;
// cout << "operand4 = " << operand4 << endl;
// cout << "operand5 = " << operand5 << endl;
// cout << "operand6 = " << operand6 << endl;

```

```
// return 0;
// }

// =====
/*
```

The compound assignment operator is used to perform an operation and then assign the result to the operand on the left-hand side.

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main() {
//   // your code goes here
//   int operand1 = 50;
//   int operand2 = 26;
//   cout << "Before using compound assignment operator:" << endl;
//   cout << "operand1 = " << operand1 << endl;
//   operand1 += operand2;
//   cout << "After using compound assignment operator:" << endl;
//   cout << "operand1 += operand2 = " << operand1 << endl;

//   return 0;
// }
// =====
```

```
/*
```

Relational Operators:

A relational operator compares the value of two operands.

 The output of a relational operator is a bool data type.

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main()
// {

//   int operand1 = 50;
//   int operand2 = 26;
//   cout << " operand1 = " << operand1 << ", operand2 = " << operand2 << endl;
//   cout << " Is operand1 less than operand2? " << (operand1 < operand2) << endl;
//   cout << " Is operand1 less than or equal to operand2? " << (operand1 <= operand2) << endl;
//   cout << " Is operand1 greater than operand2? " << (operand1 > operand2) << endl;
//   cout << " Is operand1 greater than or equal to operand2? " << (operand1 >= operand2) << endl;
//   cout << " Is operand1 equal to operand2? " << (operand1 == operand2) << endl;
//   cout << " Is operand1 not equal to operand2? " << (operand1 != operand2) << endl;
```

```
// return 0;
//}
// =====
```

```
/*
```

☞ In C++, we can also compare the float, string, and char data types using relational operators.

☞ When we apply relational operators to the operands of type char, the compiler will compare the ASCII values of the character.

☞ Writing relational expressions without round brackets in the print statement will generate an error.

Example program with a string#

Consider two operands of a string data type. Let's apply a relational operator to these operands and see the results.

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main() {

// string operand1 = "Microsoft";
// string operand2 = "Samsung";
// cout << " Is operand1 greater than operand2? " << (operand1 > operand2) << endl;

// return 0;
//}
// =====
```

```
/*
```

In the code above, the compiler continuously compares the strings character by character while the ASCII value of characters in both strings is equal.

☞ If you try to write `<=`, `>`, `>=`, `==`, and `!=` with a space, a syntax error will occur.

Logical Operators `&&` `||` `!`

Difference between relational and logical operators#

Suppose there are 20 students in your class. The teacher just displayed the final result of the “Fundamentals of Computer Science” course with an announcement that the student with the highest marks will be given a reward. Sounds interesting! But how would you know if you are the student with the highest marks in class?

You will compare your marks with the rest of the students in the class. Relational operators can only compare the scores of two students and return the result.

Here, logical operators come to the rescue! Logical operators allow you to make as many comparisons as you want.

Example program with bool operands#

Consider two operands of type bool: The value of operand1 is false because 2 is not greater than 3, and the value of operand2 is true. The program given below demonstrates the working of logical operators.

```
/*
//=====
// #include <iostream>
// using namespace std;

// int main()
//{
//    bool operand1 = 2 > 3;
//    bool operand2 = true;
//    cout << "Values of operands are:";

//    cout << "operand1 = " << operand1 << ", operand2 = " << operand2 << endl;
//    cout << "operand1 && operand2 = " << (operand1 && operand2) << endl;
//    cout << "operand1 || operand2 = " << (operand1 || operand2) << endl;
//    cout << "!operand1 = " << (!operand1) << endl;
//    cout << "!operand2 = " << (!operand2) << endl;

//    return 0;
//}
//=====

/*
 Logical operators are generally used to control the flow of the program. They allow a program to decide the flow of execution based on certain conditions.
```

Bitwise Operators: & | ^ ~ << >>

Introduction#

A bitwise operator performs bit by bit processing on the operands.

Bitwise operators operate on binary numbers. They convert operands in decimal form into binary form, perform the particular bitwise operation, and then return the result after converting the number back into decimal form.

```
*/
```

```
// =====
// #include <iostream>
```

```

// using namespace std;

// int main() {

// int operand1 = 3;
// int operand2 = 2;
// cout << "operand1 = " << operand1 << ", operand2 = " << operand2 << endl;
// cout << "operand1 & operand2 = " << (operand1 & operand2) << endl;
// cout << "operand1 | operand2 = " << (operand1 | operand2) << endl;
// cout << "operand1 ^ operand2 = " << (operand1 ^ operand2) << endl;

// return 0;
//}

// =====

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

// int main() {
// // your code goes here
// int operand1 = 2;
// int operand2 = 1;

// cout << "operand1 >> operand2 = " << (operand1 >> operand2) << endl;
// cout << "operand1 << operand2 = " << (operand1 << operand2) << endl;

// return 0;
//}
// =====
/*
~ operator#
The tilde operator is a bitwise complement operator that inverts all the bits in a number.

```

Let's consider an example in which we have to find 2's complement of a number. We can do that by inverting all bits in a number and then adding 1 to it. Here's a program that demonstrates that the 2's complement of 5 is -5.

```

*/
// =====
// #include <iostream>
// using namespace std;

// int main()
//{
// int operand = 5;
// cout << "2's complement of " << operand << " = " << ~operand + 1 << endl;
// return 0;
//}
// =====

```

```
/* 2's complement notation is used by computers to represent negative numbers. Moreover, the ~ operator is also useful for image processing applications where we have to invert bits of an image while applying any mask etc.
```

```
*/
```

```
/*
```

Precedence and Associativity:

Precedence#

In case there is more than one operator in an expression, precedence determines the order in which the operators should be evaluated. The operator with higher precedence will be evaluated first in an expression. For example, multiplication * has higher precedence than addition +. Therefore, we first evaluate multiplication in an expression.

Note: In case of parenthesis (), we first evaluate the expression inside the parenthesis.

Example program#

Let's write a program that will evaluate an expression according to its operator precedence.

```
*/
```

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

// int main()
//{
//   cout << 4 - (8 + 10) * 3;
//}
// =====
```

```
/*
```

Associativity#

Associativity determines the order in which the operators with the same precedence should be evaluated.

In left associativity, we evaluate the expression from left to right if two or more operators have the same precedence. For example, addition + and subtraction - have the same precedence. Thus, if they both appear in an expression, we evaluate them from left to right.

In right associativity, we evaluate the expression from right to left if two or more operators have the same precedence.

Example program#

Let's write a program that will evaluate an expression according to its operator associativity.

```
*/
```

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

// int main()
//{

```

```
// cout << 4 - 8 + 10 + 3;  
//  
//=====
```

```
/*
```

Operators are listed below in the order of their precedence from highest to lowest. The operators that are listed in the same row have the same precedence.

The ternary operator ?: is a conditional operator and will be covered later in the course.

```
*/  
//  
//  
//  
//  
//  
//  
//  
/*
```

Introduction to Conditional Statements

Conditional statements#

In our daily life, we make decisions based on certain conditions. For example:

If it rains, I will take an umbrella to the office, otherwise I won't.

If mom gives me \$20, I will buy a watch, otherwise I won't.

Similarly, a computer program can decide whether to execute a particular block of code or not based on the evaluation of certain conditions. The statements which are used in combination with these conditions and helping a program make a logical decision, are called conditional statements or decision-making statements.

Why use conditional statements?#

Generally, the C++ compiler executes the statements sequentially, starting from the first statement and running them in the order in which they are written. Whenever we run our program, it will give us the same results on execution. But hey, we live in the modern era and we like to have options!

Therefore, we use conditional statements that evaluate the specified conditions. Depending upon the result of the evaluation, these statements may cause a change in the flow of the execution of a computer program.

Types of conditional statements#

C++ supports the following conditional statements:

- If statement
- If-else statement
- Nested else-if statement
- Switch statement

If Statement

Introduction#

Suppose you can buy a watch if you get at least \$20 in an allowance. Otherwise, you cannot. In C++, how can we make a decision based on a condition?

We can use an if statement to demonstrate this kind of behavior.

The if statement instructs a compiler to execute a particular block of code when the condition evaluates to true.

Syntax#

The general syntax of an if statement consists of the if keyword followed by the round brackets (). These round brackets hold a condition specified by the programmer. Following the if condition is a block of code encapsulated in the curly brackets. This block of code is called the body of the if statement.

 We can use relational and logical operators for comparison in the condition inside the round brackets.

 In C++, a zero or null value is considered false, and non-zero values are considered true.

Example program when the condition is true#

Let's convert the above example into a C++ program.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   // Initialize money to 21
//   int money = 21;
//   // If condition
//   if (money >= 20) {
//     // If body
//     cout << "You can buy a watch";
//   }
//   // Exit
//   return 0;
// }
// =====
/*
```

Line No. 7: Sets the value of money to 21.

Line No. 9: Checks if the value of money is greater than or equal to 20. If yes, then the condition returns 1, and the code inside the curly brackets will be executed. The value of money is greater than 20; therefore, the condition returns 1.

Line No. 11: It prints You can buy a watch in the output since the condition in Line No. 9 is true

 Writing the if keyword in the upper case will generate a syntax error.

Example program when the condition is false#

Let's see what happens if the condition evaluates to false.

```
/*
// =====
// #include <iostream>
```

```
// using namespace std;

// int main() {
// Initialize variable money
// int money = 9;
// // if condition
// if (money >= 20) {
// // if body
// cout << "You can buy a watch";
// }
// // exit
// return 0;
//}
//=====
/*
Line No. 7: Sets the value of money to 9.
```

Line No. 9: The value of money is less than 20; therefore, the condition returns 0.

Line No. 11: The condition in Line No. 9 is false; therefore, the code inside the body of the if statement does not execute.

If-else Statement

Introduction#

Suppose you can buy a watch for your friend if you have at least \$20. Otherwise, you can gift them a pen.

In C++, we can demonstrate this kind of behavior using the if-else statement.

In the if-else statement, when the condition in an if statement evaluates to false, the compiler executes the code inside the else block.

```
*/
//=====
// #include <iostream>

// using namespace std;

// int main() {
// // Initialize variable money
// int money = 10;
// // if condition
// if (money >= 20) {
// // if block
// cout << "You can gift a watch" << endl;
// } else {
// // else block
// cout << "You can gift a pen " << endl;
```

```
// }  
// return 0;  
//}  
  
// ======  
  
/*
```

Line No. 7: Sets the value of money to 10.

Line No. 9: Checks if the value of money is greater than or equal to 20. Since money is less than 20, the condition in the if statement returns 0.

Line No. 11: Prints You can gift a watch in the output. The condition in the if statement is false; therefore, this line does not execute.

Line No. 14: The condition in the if statement returns 0; therefore, the code inside the else block executes, and it prints You can gift a pen to the console.

```
//  
//
```

Introduction#

Suppose you want to buy a present for your friend's birthday. Below is a list of things, along with their costs, that you can buy for them:

Wrist-watch: \$20

Comic book: \$10

Chocolate: \$5

Pen: \$0

Based on available money, we have multiple choices to buy a present. How can we translate this example into a C++ program?

In C++, we can use the else-if statement to check multiple conditions in a program.

```
*/  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
//   // Initialize variable money  
//   int money = 6;  
//   // if block  
//   if (money >= 20) {  
//     cout << "You can gift a watch" << endl;  
//   }  
//   // else-if block  
//   else if (money >= 10) {
```

```
// cout << "You can gift a comic book " << endl;
// }
// else if (money >= 5) {
//   cout << "You can gift a chocolate " << endl;
// }
// // else block
// else {
//   cout << "You can gift a pen " << endl;
// }
// return 0;
//}
//=====
```

```
/*
switch case:
```

We can use the else-if statement here, but choice is extensive. Therefore, else-if makes our code slow and complicated. Here, the switch statement comes in. Whenever we have to check the value of a single variable against an extensive number of choices, it is better to use the switch statement.

The switch statement evaluates the given expression and then compares its value with each case label. If the value of a case label equals the value of the expression, the statement(s) specific to that case is executed.

- ☒ Switch expression and case label only accept variables of int or char data types.
- ☒ If we don't add a break statement to a case, the code specific to all the proceeding cases is also executed.
- ☒ The default case is optional in the switch statement. It can be used anywhere in the switch statement.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   // Initialize variable grade
//   char grade = 'C';
//   // switch statement
//   switch (grade) {
//     // first case
//     case 'A':
//       cout << "Exceptional performance!";
//       break;
//     // second case
//     case 'B':
//       cout << "Well done!";
//       break;
//     // third case
//     case 'C':
//       cout << "Good!";
```

```

//    break;
// // fourth case
// case 'D':
//   cout << "You need to do more hardwork!";
//   break;
// // fifth case
// case 'F':
//   cout << "Fail";
//   break;
// // default case
// default:
//   cout << "Invalid input";
// }

// return 0;
//}
//=====
/*

```

Example program for ranges of values#

Consider the example given in the previous lesson. We can use the switch statement to test the range of values, but it is not a good way to do so.

```

*/
//=====
// #include <iostream>

// using namespace std;

// int main()
//{
// // Initialize variable money
// int money = 6;
// switch (money)
//{
// // first case
// case 20 ... 100:
//   cout << "You can gift a watch" << endl;
//   break;
// // compares value of case label from 10 to 19 with the value of money
// case 10 ... 19:
//   cout << "You can gift a comic book " << endl;
//   break;
// // compares value of case label from 9 to 5 with the value of money
// case 5 ... 9:
//   cout << "You can gift a chocolate " << endl;
//   break;
// // default case
// default:
//   cout << "You can gift a pen " << endl;
// }
// return 0;

```

```
//}  
// ======  
/*
```

Explanation#

In the above code, it seems that the switch statement is working in the same way as the else-if statement. However, there is a difference. Try to run the above code for money = 101.

With the switch statement, the output is "You can gift a pen". Whereas, with else-if, the output is "You can gift a watch".

In a switch statement, you have to define both the upper and lower ranges of values. The upper range of money is unknown; therefore, the switch statement is not a good option for testing ranges of values. If you want to test ranges, use the else-if statement.

 range in switch case is not the feature standard C++ but it is the extension of the gcc compiler.

Conditional Operator

Introduction#

Let's have a look at another form of the if-else statement.

The conditional operator evaluates the given condition and returns the result accordingly.

Example program#

Consider the same example we discussed in the if-else lesson. You can buy a watch for your friend if you have at least \$20, else you can gift them a pen. Let's convert this example into a C++ program.

```
*/  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
//   // Initialize variable money  
//   int money = 10;  
//   // Declare variable result  
//   string result;  
//   // Ternary operator  
//   result = (money >= 20) ? "You can gift a watch" : "You can gift a pen " ;  
//   // prints result  
//   cout << result;  
//   return 0;  
// }  
  
// ======  
  
/*
```

Introduction to Loops

Introduction#

Suppose we have 1000 boxes in a warehouse. We want a robot to move all boxes from one point to another. The robot is pretty dumb, so we must teach it how to move one box in great detail. We will provide the robot with the following instructions:

Go to point A.

Lift the box and load it.

Move to point B.

Unload the box here.

Then, we will ask it to repeat the same steps for the rest of the boxes until there are no more boxes at point A. Loops work in a similar fashion. We teach something to the computer, and then we instruct it to repeat the same procedure until the specified condition is fulfilled.

In computer language, loops allow you to repeat a particular block of code until the specified condition is met.

Example#

Suppose we have to write a program to print the first five whole numbers. It's pretty easy!

*/

```
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   int number = 0;
//   cout << number++ << endl;
//   return 0;
//}
// =====
```

/*

Advantages of using loops#

We use loops to:

Execute a particular piece of code multiple times

Avoid duplication in our code

Make the code more readable

Save our time

Create an efficient and manageable program

Make programming fun!

Types of loops#

In C++, we have:

while loop

do-while loop

for loop

Nested loops

Infinite loops

while Loop in C++

Introduction#

Suppose you have \$20, and the price of an ice-cream is \$5. You want to keep buying the ice-cream until you have no money left. This task is repetitive, and you don't know in advance how many ice-creams you can buy.

In the era of programming, we can use the while loop to implement repetitive tasks.

The while loop keeps executing a particular code block until the given condition is true. It does not know in advance how many times the loop body should be executed.

The condition in the while loop is evaluated before executing the statements inside its body. Therefore, the while loop is called an entry-controlled loop.

The while loop first evaluates the given condition.

If the condition evaluates to true, the code inside the body of the while loop is executed.

After that, the while loop again evaluates the condition. This process continues until the given condition remains true.

```
*/  
//=====  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
//   // Initialize the variable money  
//   int money = 20;  
//   // Initialize the variable icecream_price  
//   int icecream_price = 5;  
//   // Prints value of variables  
//   cout << "Intial money = " << money << endl;  
//   cout << "Ice-cream price = " << icecream_price << endl;  
//   // Start of the while loop  
//   while (money >= icecream_price){  
//     // Body of the while loop  
//     cout << "Buy an ice-cream" << endl;  
//     money = money - icecream_price;  
//     cout << "Remaining money = " << money << endl;
```

```
// }  
// // End of the while loop  
// cout << "You can't buy an ice-cream" << endl;  
  
// return 0;  
//}  
  
// =====
```

/*

Explanation#

Line No. 7: Initializes the value of money.

Line No. 9: Initializes the value of icecream_price.

Line No. 11: Prints the value of money to the console.

Line No. 12: Prints the value of icecream_price to the console.

Line No. 14: Checks if the value of money is greater than or equal to icecream_price. If true, then execute Lines No. 16 to 19. If false, then it executes Line No. 21.

Line No. 16: Prints Buy an ice-cream to the console.

Line No. 17: Subtracts icecream_price from the money.

Line No. 18: Prints the new value of money.

Line No. 19: Jumps to Line No. 14.

Line No. 21: Prints You can't buy an ice-cream to the console.

*/

/*

do-while Loop:

Introduction#

Suppose we want to execute the body of a loop at least once even if the condition evaluates to false. How can we accomplish this task in C++?

In the era of programming, we can use the do-while loop to implement such tasks.

The do-while loop is similar to the while loop, with the exception that it executes the block of code and then checks the given condition. Because of this, it is called an exit-controlled loop.

The general syntax of the do-while loop consists of a do keyword followed by curly brackets { }, which contain statements to be executed. It is followed by the while keyword and the condition to be checked.

 Like the while loop, the do-while loop does not know in advance how many times the loop body should be executed.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//    // Initialize the variable money
//    int money = 0;
//    // Initialize the variable icecream_price
//    int icecream_price = 5;
//    // Prints value of variables
//    cout << "Intial money = " << money << endl;
//    cout << "Ice-cream price = " << icecream_price << endl;
//    // Start of the do-while loop
//    do
//    {
//        // Body of the do-while loop
//        cout << "Buy an ice-cream" << endl;
//        money = money - icecream_price;
//        cout << "Remaining money = " << money << endl;
//    } while (money >= icecream_price);
//    // End of the do-while loop
//    cout << "You can't buy an ice-cream" << endl;

//    return 0;
//}
//=====
```

/*

Explanation#

Line No. 7: Initializes the value of money.

Line No. 9: Initializes the value of icecream_price.

Line No. 11: Prints the value of money to the console.

Line No. 12: Prints the value of icecream_price to the console.

Line No. 14: Executes Lines No. 16 to 19.

Line No. 16: Prints Buy an ice-cream to the console

Line No. 17 Subtracts an icecream_price from the money.

Line No. 18: Prints the new value of money to the console.

Line No. 19: Checks if the value of money is greater than icecream_price. If yes, it jumps to Line No. 14. If no, it executes Line No. 21.

Line No. 21: Prints You can't buy an ice-cream to the console.

```
*/  
/*  
for Loop:
```

Introduction#

Suppose you have a coupon to buy five ice-creams free of cost. You know in advance how many free ice-creams you can buy.

In the era of programming, we can use the for loop for such situations.

The for loop keeps executing a particular code block as long as the given condition is true. It knows in advance the number of times the loop body should be executed.

 The for loop is a count controlled loop since the program knows in advance the number of times the loop body should be executed.

```
/*  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main()  
// {  
//     // Initialize variable icecream  
//     int icecream;  
//     // for loop start  
//     for (icecream = 5; icecream > 0; icecream--)  
//     {  
//         // loop body  
//         cout << "Number of free icecream = " << icecream << endl;  
//         cout << "Buy an icecream" << endl;  
//     }  
//     // Exit loop  
//     return 0;  
// }  
  
// ======
```

/*

Explanation#

Line No. 7: Declares a variable icecream.

Line No. 9:

icecream = 5: The initial value of icecream is set to 5. Here, icecream is a loop control variable.

icecream > 0: It is the loop continuation condition. It ensures the repetitive execution of the body of for loop until it evaluates to true.

In the code above, loop statements are repeated until the value of the icecream is greater than 0. When the loop condition evaluates to true, it executes the statements from Lines No. 11 to 13. After executing the loop block, it jumps back to Line No. 9. At this point, it updates the value of the icecream and again evaluates the condition.

icecream--: This statement decrements the value of the icecream by 1.

Line No. 11: Prints the value of icecream to the console.

Line No. 12: Prints Buy an icecream to the console.

Infinite Loop

Introduction#

Sometimes, erroneously, we end up writing a piece of code in which a loop condition never evaluates to false and the loop block keeps executing repeatedly. Such types of loops are known as infinite loops.

The infinite loop keeps executing repeatedly and never terminates.

*/

```
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//   for (;;)
//   {
//     cout << "Hey, I am infinite loop" << endl;
//   }

//   return 0;
//}
// =====

/*
```

We have not specified the termination condition. Therefore, the loop runs repeatedly and prints Hey, I am infinite loop to the console.

On our platform, the compiler stops the execution after 30 seconds and gives you an error if something like that happens. Therefore, you cannot see the output.

However, if you run the same program on your computer, it keeps printing the output to the console and never stops.

 To stop the infinite loop on your computer, press CTRL+C.

Example program#

We can also generate an infinite loop by setting the conditions in such a way that the program never returns false.

The program given below will generate an infinite loop!

```
/*
// =====
// #include <iostream>
// #include <iostream>
// using namespace std;

// int main()
//{
//    int number = 1;
//    while (number > 0)
//    {
//        cout << number << endl;
//    }
//    return 0;
//}
// =====

/*
Nested Loop
```

Introduction#

Suppose you want to print the times tables of 6, 7, and 8 in a program. First, we need to choose the number whose table we want to print. Then, we print the table for that number. How can we do this task?

In C++, we can use nested loops to accomplish such tasks.

A loop inside the body of another loop is called a nested loop.

Types#

In C++, we have the following types of nested loops:

Nested while loop

Nested do-while loop

Nested for loop

```
*/
```

```
// =====
```

```
// #include <iostream>
```

```
// using namespace std;
```

```
// int main() {
```

```
// // Declares variable inner and outer
```

```
// int inner, outer;
```

```
// // Outer for loop
```

```
// for (outer = 6; outer <= 8; outer++) {
```

```
// // Outer for loop body
```

```
// cout << "Table of " << outer << " is:" << endl;
```

```
// // Inner for loop
// for (inner = 1; inner <= 5; inner++) {
//   // Inner for loop body
//   cout << outer << " * " << inner << " = " << (outer * inner) << endl;
// }
// // Exit inner for loop
// }
// // Exit outer for loop
// return 0;
//}
```

```
// =====
```

```
/*
```

```
Explanation#
```

In the nested for loop, for the single value of the outer variable, the inner loop iterates over all its values. For example, for outer = 6, the inner loop runs from inner = 1 to inner = 5. After this is done, outer is incremented to 7, and the inner loop iterates over all its values again. This process continues until the value of the outer is less than or equal to 8.

Line No. 7: Declares inner and outer variables.

Line No. 9: Defines an outer for loop that takes the values from 6 to 8.

outer = 6: The initial value of the outer is set to 6.

outer <= 8: If the loop condition evaluates to true, it executes the statements from Lines No. 10 to 18.

outer++: After executing the loop block, it will jump back to line No. 9. At this point, it will increment the value of the outer by 1 and again evaluate the condition.

Line No. 11: Prints the value of outer to the console.

Line No. 13: Defines an inner for loop that takes the values from 1 to 5.

inner = 1: The initial value of the inner is set to 1.

inner <= 5: If the loop condition evaluates to true, it executes the statements from lines No. 14 to 16.

inner++: After executing the loop block, it jumps back to Line No. 13. At this point, it increments the value of the inner by 1 and evaluates the condition again.

Line No 15: Multiplies the value of outer by inner and display it on the screen.

```
// -----
```

break Statement

```
Introduction#
```

Suppose you have a coupon to buy five ice-creams free of cost, but the ice-cream man only has three ice-creams. In this case, while you can have some free ice-creams, the ice-cream eventually man runs out of ice-creams before you have utilized all your coupons.

In programming, we can use the break statement for such situations. The break statement can be used to jump out of the loop immediately when a particular condition evaluates to true.

The break statement terminates the loop and transfers control to the very next statement after the loop body.

Use case#

Let's go over a use case of the break statement. It is very simple to use. You just have to write a break after the line that you want to terminate the loop after!

The basic syntax of a break statement consists of an if keyword followed by a condition in round brackets. The curly brackets contain a break keyword that terminates the loop when the condition evaluates to true.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {
//   // Initialize variable icecream
//   int icecream;
//   // for loop start
//   for (icecream = 5; icecream > 0; icecream--) {
//     // loop body
//     cout << "Number of free ice-creams = " << icecream << endl;
//     // break statement
//     if (icecream == 2) {
//       break;
//     }
//     cout << "Buy an icecream" << endl;
//   }
//   // Exit loop
//   cout << "Sorry! We ran out of ice-cream" << endl;
//   return 0;
// }
// =====
/*
```

Explanation#

In the code above, we have a for loop that iterates from 5 to 1. However, since we have a break statement that is executed when the value of the loop variable is 2, the loop terminates, and it transfers control to the very next statement after the loop body.

Line No. 7: Declares a variable icecream.

Line No. 9:

icecream = 5: The initial value of icecream is set to 5.

icecream > 0: When the loop condition evaluates to true, it executes the statements from Lines No. 11 to 17.

icecream--: After executing the loop block, it jumps back to Line No. 9 where it decrements the value of icecream by 1 and evaluates the condition again.

Line No. 11: Prints the value of icecream to the console.

Line No. 13: Checks if the value of icecream is 2. If yes, then executes Line No. 14 to Line No. 15. If no, then jumps to Line No. 16.

Line No. 14: Breaks the loop. When the break statement is executed, the program will exit the loop body and jump to Line No. 19.

Line No. 16: Prints Buy an icecream to the console

Line No. 19: Prints Sorry! We ran out of ice-cream to the console.

*/

/*

continue Statement:

Introduction#

Suppose you have a coupon to get five ice-creams free of cost, but the ice-cream man has only three ice-creams. So when you ask for the fourth one, he tells you that he ran out of ice-cream and one of your coupons is wasted. However, after some time, the ice-creams are restocked, and you are able to get your free ice-cream.

In programming, we can use the continue statement for such situations.

The continue statement makes the compiler skip the current iteration and move to the next one.

Use case#

Let's go over the syntax of the continue statement. It is very simple to use: you just need to write continue before the statements you want to skip in a certain loop iteration!

The basic syntax of a continue statement consists of an if keyword followed by a condition in round brackets. The curly brackets contain a continue keyword that skips the current iteration when the condition evaluates to true.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
// {
//     // Initialize variable icecream
//     int icecream;
//     // for loop start
//     for (icecream = 5; icecream > 0; icecream--)
```

```

// {
//   // loop body
//   cout << "Number of free ice-creams = " << icecream << endl;
//   // continue statement
//   if (icecream == 2)
//   {
//     cout << "Sorry! We ran out of ice-cream" << endl;
//     continue;
//   }
//   cout << "Buy an icecream" << endl;
// }
// =====
/*

```

Explanation#

In the code above, we have a for loop iterating from 5 to 1. However, since we have a continue statement that is executed when the value of the loop variable is 2, the loop skips this iteration, and transfers control to the loop condition.

Line No. 7: Declares a variable icecream. Line No. 9:

icecream = 5: The initial value of the icecream is set to 5.

icecream > 0: When the loop condition evaluates to true, it executes the statements from Line No. 11 to 18.

icecream--: After executing the loop block, it jumps back to Line No. 9 where it decrements the value of the icecream by 1 and evaluates the condition again.

Line No. 11: Prints the value of ice-cream to the console.

Line No. 13: Checks if the value of ice-cream is 2. If true, it executes Line No. 14 to Line No. 16.

Line No. 14: Prints Sorry! We ran out of ice-cream to the console

Line No. 15: Exits the loop body and jumps to Line No.9

Line No. 17: Prints Buy an icecream to the console

*/

```

// =====
// #include <iostream>

// using namespace std;

// int main() {
//   // Initialize variables
//   int decimal = 10, binary = 0;
//   int remainder, product = 1;
//   // Prints value of decimal

```

```
// cout << "Decimal Number = " << decimal << endl;
// // while block
// /*Checks if the value of `decimal` is not equal to '0'.
// If yes, then execute line No. 17 to 21.
// If no, then execute line No. 23.
// */
// while (decimal != 0) {
//     remainder = decimal % 2;
//     binary = binary + (remainder * product);
//     decimal = decimal / 2;
//     product *= 10;
// }
// // while exit
// cout << "Binary Number = " << binary;
// return 0;
//}
//=====================================================================
//
//
//
//
//
/*

```

Functions in C++

Introduction#

Suppose you want to make juice for yourself. You will follow the following steps:

Put fruits and water in a blender.

Turn on the juicer.

Enjoy the juice after 1 minute.

A function is like a blender that performs a specific action on some ingredients and returns a modified product. We can use the same blender to extract the juice of different fruits. Similarly, functions are reusable, and we can use the same function anytime with different inputs.

In computer language, a function is a block of code that performs a particular task. It is also given a name.

Functions in programming are just like mathematical functions. They take something as input, perform some operation on it, and return an output.

Types of functions#

In C++, we have two types of functions:

Library functions

User-defined functions

Library functions#

These functions are also known as built-in functions. They are already defined in the C++ header files such as `\<iostream>`, `\<string>`, and `\<cmath>` etc. We can use these functions by including the relevant header file and then call them later in a program.

User-defined functions#

These functions are defined by users according to their needs. We can call them anywhere in the current program.

Why use functions?#

We use functions in a program to:

Make our code reusable

Divide our code into small modules

Make the debugging of the program easier

Make our code neat

Avoid code repetition

 The purpose of a function is to define the code block once and then use it many times.

Declaring a Function:

Function creation#

In C++, function creation consists of the following two steps:

Function declaration

Function definition

Function declaration#

Function declaration informs the compiler about:

The return type of function

The function name

The number of parameters and their data types.

The basic syntax for declaring a function in C++ is:

```
return_type function_name(function_parameters);
```

return_type#

Return type specifies what type of data a function returns in output to the calling point after performing its task.

 It is possible for a function to return nothing in output. Such functions have a void return type.

function_name#

Whenever we declare a function, we give it a unique name. We then use the same name to call it throughout the program.

function_parameters#

When we call a function, we pass values to the function parameters. These values are known as arguments, or actual parameters.

 Passing parameters in a function is optional.

*/

```
// =====
// #include <iostream>

// using namespace std;
// // Function declaration
// int make_juice(int water_glass, int fruit);
// // int make_juice(int , int);

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

/*

 It is not necessary to give parameter names in the function declaration. You may only declare their data type. You can check this out by uncommenting Line No. 6 in the above code.

Defining a Function:

Function definition#

A function's definition tells what a function will do when it is called. The basic syntax for defining a function in C++ is:

```
return_type function_name(parameters){
    function body;
}
```

function_body #

A function body consists of a group of statements that do a particular task. We write our function code inside the curly braces. Everything written inside the curly braces is what the function does when it is called.

Main function#

In the code below, you see the highlighted lines in every C++ program. If you look closely at these lines, you see that the main() is the function here. It is the point from where every C++ program starts its execution. Whenever the C++ program is executed, the operating system gives control to the main function.

 Every program in C++ must have a main function.

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main()
//{
//    // your code goes here

//    return 0;
//}
```

```
// =====  
/*
```

Anatomy of the main function#
int specifies that the main function returns an integer value in the output.

{ indicates the beginning of the main function.

return 0 returns 0 to the calling point on the successful execution of the program.

 Note: Adding a return 0 statement in a program is not mandatory.

} indicates the end of the main function.

```
*/
```

```
// =====
```

```
// #include <iostream>
```

```
// using namespace std;  
/// Function declaration  
// int make_juice(int water, int fruit);
```

```
// int main()  
// {
```

```
//   return 0;  
// }
```

```
/// Function definition  
// int make_juice(int water, int fruit)  
// {  
//   // Define new variable juice of int type  
//   int juice;  
//   // Adds water in apple and save output in juice  
//   juice = water + fruit;  
//   // Prints text on the screen  
//   cout << "Your juice is ready" << endl;  
//   // Returns juice value in output  
//   return juice;  
// }
```

```
// =====
```

```
/*
```

Introduction#

The functions created in a program are not executed until we call them. When we call the function, control is given to the very first statement inside the called function. The basic syntax for calling a function is given below:

To call a function in a program, we have to write a function name, followed by values of arguments in the round brackets and the semicolon.

 We can call a function from any other function in a program.

```
/*
// =====
// #include <iostream>

// using namespace std;
// // Function declaration
// int make_juice(int water, int fruit);
//
// int main() {
//   // Initialize variables apple and water
//   int apples = 5;
//   int water_glass = 3;
//   // Declares a variable juice_glass
//   int juice_glass;
//   // Calls function make_juice and save its output in juice_glass
//   juice_glass = make_juice(water_glass, apples);
//   // Prints value of juice_glass
//   cout << "Number of juice glass = " << juice_glass;

//   return 0;
// }

// // Function definition
// int make_juice(int water, int fruit) {
//   // Define new variable juice of int type
//   int juice;
//   // Adds water in apple and save output in juice
//   juice = water + fruit;
//   // Prints text on the screen
//   cout << "Your juice is ready" << endl;
//   // Returns juice value in output
//   return juice;
// }
// =====

/*
Explanation#
Line No. 9: Initialize apples to 5.

Line No. 10: Initialize water_glass to 3.

Line No. 12: Declares a variable juice_glass.

Line No. 14: Calls the function make_juice We call a function by writing its name and follow it by round brackets. It returns an integer value in the output, which is stored in juice_glass. When we call a function make_juice in the main(), the program control is given to the first statement in the function's body.

Line No. 16: Prints value of juice_glass.
```

Line No. 10: Initialize water_glass to 3.

Line No. 12: Declares a variable juice_glass.

Line No. 14: Calls the function make_juice We call a function by writing its name and follow it by round brackets. It returns an integer value in the output, which is stored in juice_glass. When we call a function make_juice in the main(), the program control is given to the first statement in the function's body.

Line No. 16: Prints value of juice_glass.

s it necessary to declare a function?

In the above code, we declare a function before the main function. Then, we define it after the main function.

In C++, statements are executed from top to bottom. If we don't declare the function before main(), our program will be unaware of it and we will get a compilation error.

 We cannot declare the function after the main function or we will get an error.

You are probably wondering if it's possible to define a function before main() and then call it later in a program.

Yes, it is possible. If you are defining your function before the main function, then function declaration is not necessary.

In the above code, we have removed the function declaration and defined our function before the main function. This gives us the same output.

Calling a function multiple times#

We can call the function as many times as we want with different inputs.

```
/*
// =====
// #include <iostream>
// using namespace std;

/// Function definition
// int make_juice ( int water , int fruit){
/// Define new variable juice of int type
// int juice ;
/// Adds water in apple and saves the output in juice
// juice = water + fruit;
/// Prints text on the screen
// cout << "Your juice is ready" << endl ;
// // Returns juice value in output
// return juice;

//}

// int main() {
// // Declares a variable juice_glass
// int juice_glass;

// // Calls function make_juice and save its output in juice_glass
// juice_glass = make_juice ( 2 , 5);
// // Prints value of juice_glass
// cout << "Number of juice glass = " << juice_glass << endl;
// juice_glass = make_juice ( 6 , 11);
// // Prints value of juice_glass
// cout << "Number of juice glass = " << juice_glass << endl;

// return 0;
```

```
//}  
// =====
```

```
/*
```

In the above code, we call the make_juice function twice in a program.

Line No. 23: Calls the make_juice function and then stores the returned value in juice_glass. You can notice that we are passing values directly as arguments to the function.

```
make_juice (2 , 5)
```

 We can initialize a variable and then pass the identifier to the function parameter, or we can pass the value directly to the function parameters.

Line No. 26: Calls the make_juice function and then stores the returned value in juice_glass. Now, we are calling the function with different values.

```
make_juice (6 , 11)
```

C++ Function Parameters

Get acquainted with actual parameters, formal parameters, and the default values of the parameters.

Function parameters#

We can declare the variables inside the function definition as parameters. We specify the list of parameters separated by a comma inside the round brackets. In C++, we have:

Formal parameters

Actual parameters

Formal parameters#

Formal parameters are the variables defined in the function definition. These variables receive values from the calling function. Formal parameters are commonly known as parameters.

Actual parameters#

Actual parameters are the variables or values passed to the function when it is called. These variables supply value to the called function. Actual parameters are commonly known as arguments.

```
*/
```

```
// =====  
// #include <iostream>  
// using namespace std;  
  
/// Function definition  
// int make_juice ( int water , int fruit){  
/// Define new variable juice of int type  
// int juice ;  
/// Adds water in apple and saves the output in juice  
// juice = water + fruit;
```

```

/// Prints text on the screen
// cout << "Your juice is ready" << endl ;
// // Returns juice value in output
// return juice;

//}

// int main() {
// // Declares a variable juice_glass
// int juice_glass;
// // Calls function make_juice and save its output in juice_glass
// juice_glass = make_juice ( 2 , 5);
// // Prints value of juice_glass
// cout << "Number of juice glass = " << juice_glass << endl;
// return 0;
//}

// =====

/*

```

Line No. 5: We defined the function make_juice. In the make_juice definition, we declare the variables water and fruit that take integer values. These are the formal parameters.

Line No. 22: In the main function, we call the function make_juice. make_juice takes 2 and 5 inside the round brackets. Here, 2 and 5 are the actual parameters.

Default parameter values#

If we provide fewer or no arguments to the calling function, the default values of the parameters are used. We specify the default values in the function declaration using an equal sign =.

```

*/
// =====
// #include <iostream>
// using namespace std;

// // Function definition
// int make_juice(int water = 1, int fruit = 3)
//{
// // Define new variable juice of int type
// int juice;
// // Adds water in apple and saves the output in juice
// juice = water + fruit;
// // Prints text on the screen
// cout << "Your juice is ready" << endl;
// // Returns juice value in output
// return juice;
//}

// int main()
//{

```

```

// // Declares a variable juice_glass
// int juice_glass;

// // Calls function make_juice without any actual parameters
// juice_glass = make_juice();
// cout << "Number of juice glass = " << juice_glass << endl;
// // Calls function make_juice with only one actual parameter
// juice_glass = make_juice(5);
// cout << "Number of juice glass = " << juice_glass << endl;
// // Calls function make_juice and save its output in juice_glass
// juice_glass = make_juice(2, 5);
// cout << "Number of juice glass = " << juice_glass << endl;

// return 0;
//}
// =====

```

/*

Explanation#

In the code above:

Line No. 23: If we call the function without specifying the actual values of the water and fruit, the compiler uses the default values of the parameters.

Line No. 26: If we call the function with one actual parameter, the compiler uses the actual value for water and the default value for fruit.

Line No. 29: If we specify the actual values for both water and fruit, the compiler uses their actual values.

 If we specify the default value of the parameters, the parameters following it must have a default value. Otherwise, you get an error. However, it is not necessary to assign the default values to the parameters preceding it.

Passing actual parameters to the function#

We can pass the actual parameters to the function in the following two ways:

Pass by value

Pass by reference

If you specify the default value of the parameters, then the parameters following it must have a default value. Otherwise, you will get an error.

```

int number_sum (int num1 = 30 , int num2 ){
return num1 + num2;
}

```

```

int main() {
    int sum = number_sum (20) ;
    cout << sum ;
    return 0;
}

```

```
}
```

```
*/
```

```
/*
```

Pass by Value in Functions

Introduction#

Suppose you have sent an email with an attached file to your friend. Your friend has downloaded the file and then made some changes to it. The original document can not be changed by any of the changes made by your friend because your friend has a copy of the original file.

Pass by value is just like sending a copy of a file to another person.

In pass by value, when we call a function, we pass the copy of the actual parameters to the formal parameters in the function.

In pass by value, the actual and formal parameters are stored in different memory locations. Any changes made in the formal parameters inside the function will not affect the values of actual parameters in the main function. In C++, by default, actual parameters are passed by value to the function.

```
/*
// =====
// #include <iostream>

// using namespace std;
// // function definition
// void passValue(int number)
// {
//   // Multiply the number by 10
//   number = number * 10;
//   cout << "Value of number inside the function = " << number << endl;
// }

// int main()
//{
//   // Initialize variable
//   int number = 10;
//   cout << "Value of number before function call = " << number << endl;
//   // Call function
//   passValue(number);
//   cout << "Value of number after function call = " << number << endl;

//   return 0;
//}
// =====
/*
```

Explanation#

In the code above, we have two functions:

passValue function

main function

passValue function #

Line No. 5: The passValue function takes a number of type int. It will perform its task and then returns nothing in output.

Line No. 7: Multiplies the number by 10 and stores the result in the number.

Line No. 8: Prints the updated value of the number.

main function #

Line No. 13: Initializes a variable number.

Line No. 14: Prints the value of the number before the function call.

Line No. 16: Calls a function passValue. The program execution control is transferred to Line No. 5.

Line No. 17: Prints the value of the number after the function call.

```
*/  
/*
```

Pass by Reference in Functions

Introduction#

Suppose you have sent an email to your friend with a link to a file present on Google Drive. Your friend made some changes to the document. Since you and your friend are sharing the same file, you will both see the changes made by either of you in the document.

In pass by reference, when we call a function, we pass the address of the actual parameters to the formal parameters in the function.

In pass by reference, the actual and formal parameters refer to the same memory location. Any changes made in the formal parameters inside the function affect the values of actual parameters in the main function.

```
*/  
  
// ======  
// #include <iostream>  
  
// using namespace std;  
/// // function definition  
// void passReference(int &number) {  
//   // Multiply the number by 10  
//   number = number * 10;  
//   cout << "Value of number inside the function = " << number << endl;  
// }  
//  
// int main() {  
//   // Initialize variable  
//   int number = 10;  
//   cout << "Value of number before function call = " << number << endl;  
//   // Call function  
//   passReference(number);  
//   cout << "Value of number after function call = " << number << endl;
```

```
// return 0;  
//}  
// ======  
/*
```

Explanation#

In the code above, we have two functions:

passReference function
main function

passReference function#

Line No. 5: The passReference function takes a value of type int by reference. It performs its task and then returns nothing in output.

Line No. 7: Multiplies the number by 10 and stores the result in the number.

Line No. 8: Prints the updated value of the number.

main function#

Line No. 13: Initializes a variable number.

Line No. 14: Prints the value of the number before the function call.

Line No. 16: Calls a function passReference. The execution control is transferred to Line No. 5.

Line No. 17: Prints the value of the number after the function call.

Scope of Variable:

Introduction#

The scope of a variable defines which part of the program that particular variable is accessible in. In C++, the variable can be either of these two:

Local variable
Global variable

Local variable#

Suppose you are staying at a hotel. The hotel manager gives you a key to room No. 5. You can only access room No. 5 in the hotel.

The local variable is just like a hotel room-specific key. It is only accessible within the block in which it is declared.

 Block is a section of code enclosed inside the curly braces.

The local variable can only be accessed within the block in which it is declared.

A block can be a function, loop, or conditional statement. These variables are created when the compiler executes that particular block and destroyed when the compiler exits that block.

```
/*
// =====
// #include <iostream>
// using namespace std;

// void function () {
//   int function_local = 10;
//   cout << main_local;
// }

// int main() {
//   int main_local = 20;
//   cout << function_local;
//   return 0;
// }
// =====
```

*

Explanation#

In the code above, variable function_local is only accessible within the body of the function(). We cannot access it in the main().

Similarly, main_local is only accessible within the body of the main(). We cannot access it in the function().

The program is generating an error because we are trying to access the variable main_local in the body of the function() and function_local in the body of the main().

Global variable#

Again, consider the example of a hotel. The hotel manager has the master key. Unlike us, the hotel manager can access each and every room in the hotel.

Similar to the master key, global variables are accessible in the whole program.

Global variables can be accessed from the point they are declared to the end of the program. They are declared at the very start of the program before defining any function.

```
/*
// =====
// #include <iostream>
// using namespace std;
// int global = 3;

// void function()
//{
//   int function_local = 10;
//   cout << "global = " << global << endl;
//}

// int main()
```

```
// {
//   int main_local = 20;
//   cout << "global = " << global << endl;
//   function();
//   return 0;
//}
// =====
/*

```

Explanation#

In the above program, the value of global is accessible in both main() and function().

 If two variables with the same name are declared twice within the same scope, the compiler will generate an error.

Overwriting the Value of the Global Variable:

Introduction#

We know that we can overwrite the value of local variables any number of times in a program. Similarly, we can also overwrite the value of global variables in a function.

```
/*
// =====
// #include <iostream>

// using namespace std;
// int global = 3;

// void function()
//{
//   int function_local = 10;
//   cout << "global = " << global << endl;
//   global = 9;
//}
// int main()
//{
//   int main_local = 20;
//   global = 5;
//   cout << "global = " << global << endl;
//   global = 7;

//   function();
//   cout << "global = " << global << endl;

//   return 0;
//}
```

```
// =====
/*
```

Explanation#

```
function( )#
```

Line No. 8: The value of global is printed.

Line No. 9: Updates the value of global to 9.

```
main( )#
```

Line No. 4: Initializes a variable global to 3.

Line No. 13: In the main function, we overwrite the value of global to 5.

Line No. 14: Prints the updated value of the global.

Line No. 15: We again overwrite the value of global to 7.

Line No. 17: Calls the function in a program.

Line No. 18: Prints the value of global.

```
*/
```

```
/*
```

Function Overloading:

Definition#

Function overloading is the concept of affecting a function's behavior based on the number of parameters or their types.

This way, functions with different parameters can coexist with the same name. Function overloading works with different parameters. The function prototype can change and the return type changes according to parameters being returned.

```
*/
```

```
// =====
```

```
// #include <iostream>
```

```
// #include <string.h>
```

```
// const char* min(const char* s, const char* t){
```

```
//   return (strcmp(s,t) < 0) ? s : t;
```

```
// }
```

```
// float min(float x, float y){
```

```
//   return (x < y) ? x : y;
```

```
// }
```

```
// int main() {
```

```
//   const char* s = min("abc", "xyz");
```

```
//   float f = min(4.45F, 1.23f);
```

```
//   int f2 = min(2011, 2014);
```

```
// // float f3 = min("abc", 1.23f);  
// std::cout << s << std::endl;  
// std::cout << f << std::endl;  
// std::cout << f2 << std::endl;  
// }  
// =====
```

```
/*
```

The `min` function behaves differently based on the types of arguments provided. We have defined it to handle `const char*` and `float` arguments.

In line 14, the `int` arguments are implicitly converted to `float` and an error is not thrown. The result is again converted to an `int` to match the type of `f2`.

Line 15 would not work since the function prototype is not written to handle different types of arguments (i.e., the combination of strings, floats, and integers)

Increase the number of arguments#:

```
*/  
// =====  
// #include <iostream>  
// #include <string.h>  
  
// float min(float x, float y)  
// {  
//     return (x < y) ? x : y;  
// }  
  
// float min(float x, float y, float z)  
// {  
//     return x < y ? (x < z ? x : z) : (y < z ? y : z);  
// }  
  
// int main()  
// {  
//     float f = min(4.45F, 1.23f);  
//     float f2 = min(4.45f, 1.23f, 0.19f);  
  
//     std::cout << f << std::endl;  
//     std::cout << f2 << std::endl;  
// }  
// =====  
/*
```

Now, the function can handle three `float` arguments as well. We can extend this to as many as we want.

Rules for choosing the right function#

We need to search for a function with the exact type.

We need to apply type promotion to the arguments.

We need to convert arguments.

Another thing to keep in mind is that the compiler ignores references when overloading functions. For example, `min(int x, int y)` is the same as `min(int &x, int &y)`.

Furthermore, the `const` and `volatile` qualifiers are also ignored. `min(int x, int y)` is the same as `min(volatile int x, volatile int y)`. However, `min(const int& x, const int& y)` is a different overloading function as the `const` keyword applies to `int` but not to the reference property.

Why do we overload?#

In the end, we can always create functions to perform different operations. But what would happen if we were running a large scale application that needs to support a variety of similar functions?

It would be impractical to remember the names of all unique functions. By overloading, we can introduce a great deal of simplicity and readability in our code.

We only need to provide the appropriate arguments and the compiler will handle the rest of the functionality.
*/

/*

Lambda Functions:

A lambda function, or lambda, is a function without a name.

A lambda can be written in-place and doesn't require complete implementation outside the scope of the main program.

A cool feature of lambdas is that they can be treated as data. Hence, they can be stored or copied in variables.

syntax:

`[]() -> {...}`

`[]`: Captures the used variables.

`()`: Necessary for parameters.

`->`: Necessary for complex lambda functions.

`{}`: Function body, per default `const`.

`[]() mutable -> {...}` has a non-constant function body.

What exactly do we mean by capture?

Function vs. function object#

The first thing we need to know is that lambdas are just function objects automatically created by the compiler.

A function object is an instance of a class for which the call operator, operator (), is overloaded. This means that a function object is an object that behaves like a function. The main difference between a function and a function object is that a function object is an object and can, therefore, have a state.

```
/*
// =====
// int addFunc(int a, int b){ return a + b; }

// int main(){

//   auto addObj = [] (int a, int b){ return a + b; };

//   addObj(3, 4) == addFunc(3, 4);

//}
// =====
/*
```

That's all! If the lambda expression captures its environment and therefore has a state, the corresponding struct, AddObj, gets a constructor for initializing its members. If the lambda expression captures its argument by reference, so does the constructor. The same holds for capturing by value.

Closure#

Lambda functions can bind their invocation context. This is perhaps the best feature of C++ lambdas.

Binding allows any variables passed in the surrounding scope(invocation context) to be passed to the lambda. This is what the [] in the beginning is for. Within these square brackets, we can specify which variables we want the lambda to capture.

The empty brackets we've used so far indicate that no variables should be bound.

```
*/
```

```
/*
```

Generic lambda functions#

With C++14, we have generic lambdas, which means that lambdas can deduce their argument types. Therefore, we can define a lambda expression such as [](auto a, auto b){ return a + b; };. What does that mean for the call operator of AddObj?

The call operator becomes a template. I want to emphasize it explicitly: a generic lambda is a function template.

```
/*
// =====
// #include <iostream>
// #include <vector>
// #include <numeric>
// using namespace std::string_literals;

// int main()
//{
//   auto add11 = [] (int i, int i2)
//   { return i + i2; };
//   auto add14 = [] (auto i, auto i2)
```

```

// { return i + i2; }
// std::vector<int> myVec{1, 2, 3, 4, 5};
// auto res11 = std::accumulate(myVec.begin(), myVec.end(), 0, add11);
// auto res14 = std::accumulate(myVec.begin(), myVec.end(), 0, add14);

// std::cout << res11 << std::endl;
// std::cout << res14 << std::endl;

// std::vector<std::string> myVecStr{"Hello"s, " World"s};
// auto st = std::accumulate(myVecStr.begin(), myVecStr.end(), ""s, add14);
// std::cout << st << std::endl; // Hello World
//}
// =====

```

/*

Capturing local variables#

The difference between the usage of functions and lambda functions boils down to two points:

We cannot overload lambdas.

A lambda function can capture local variables.

Here is a contrived example of the second point.

```

*/
// =====
// #include <functional>

// std::function<int(int)> makeLambda(int a)
//{
//   return [a](int b)
//   { return a + b; };
//}

// int main()
//{
//   auto add5 = makeLambda(5);

//   auto add10 = makeLambda(10);

//   add5(10) == add10(5);
//}

// =====

```

/*

What is Recursion?:

Introduction to recursion#

Suppose you are standing in a line to buy a movie ticket. You want to know the price of the ticket, but only the first person in line knows the price.

You can approach the first person and ask him the price directly. However, if you leave the line, someone else will take your place. Therefore, you will have to use the following approach.

You will ask for the ticket price from the person in front of you.

That person does not want to leave the line either. Therefore, they will ask the same question from the person in front of them. This process will continue until the price is asked from the first person in the line.

The first person in line will tell the ticket price. After that, each person in line will know the ticket price through the person standing in front of them.

How can we translate this problem into code? Here, recursion comes in.

The first person asking for the price of a ticket from the person in front of them is like calling some function inside another function body. The person asking the same question from a person in front of them is like calling the same function in its own body but with different arguments.

A function that calls itself repeatedly until some condition is met is known as a recursive function. The process whereby a function repeatedly calls itself until a condition is met is known as recursion.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int ticket_price(int person)
// {
//     int price;
//     if (person == 1)
//     {
//         price = 100;
//         return price;
//     }
//     else
//     {
//         person--;
//         return ticket_price(person);
//     }
// }

// int main()
// {
//     int price;
//     price = ticket_price(5);
//     cout << "Ticket price = " << price << endl;
// }
// =====
/*
```

Structure of a Recursive Program:

```
return_type recursive_func_name(parameters){
if(base_condition){
    return type;
```

```

}
else{
    return recursive_function(input_parameters);
}
}

```

The recursive function consists of the following two parts:

if: represents the base case when the function should stop calling itself. It simply returns the result to the calling point.

else: represents the recursive case when the function calls itself repetitively until it reaches a base case.

Illustration#

The recursive function keeps calling itself in a nested manner until it encounters a base condition. When the base condition is reached, it continues returning the value to the calling function until the original calling function is reached.

 The purpose of the main function is to serve as a starting point for a program. Therefore, the main function is not called recursively in a program.

Recursive case#

Whenever we write a recursive function, we break a complex problem into a smaller subproblem and a simpler version of itself.

```

*/
// =====
// #include <iostream>
// using namespace std;

// int ticket_price(int person)
// {
//     int price;
//     person--;
//     return ticket_price(person);
// }

// int main()
// {
//     int price;
//     price = ticket_price(10);
//     cout << "Ticket price = " << price << endl;
// }
// =====
/*

```

Explanation#

In the code above, we write a function `ticket_price` that takes a value of type `int` in its input parameters (which is similar to the number of a person in a line who is asking the `ticket_price`) and returns a value of type `int` (which similar to returning ticket price) in the output.

Lines No. 5 to 9: If person != 1, it means we have not reached the first person yet. Therefore, we decrement the person value and again call the function ticket_price but with a different value of a person (which is similar to asking the price from the different person). Here, we are calling ticket_price in the body of the function ticket_price. Therefore, ticket_price is known as a recursive function.

Why is the code given above generating an error?#

The code gives an error because we are continuously calling the function ticket_price in its body. We need to stop the recursive calls when we meet our condition. Otherwise, the program runs out of memory and gives us an error.

Base case#

When should we stop calling the function recursively in a program?#

Since we have divided the larger subproblem into a series of smaller subproblems of itself, after some time the problem will become so simple that you can solve it directly without dividing it any further. This is the base case.

The condition where the function stops calling itself in its body is known as the base case.

The recursive function only knows how to solve the simplest case known as a base case. When we call the recursive function with a base case, it simply returns us the result. There are no recursive calls in the function.

 Every recursive function must have a base case or an error is generated because of memory overflow.

Why use recursion?#

You might encounter a problem that is too scary. The easiest way to solve such problems is to use the divide and conquer rule.

Recursion is a very powerful tool when we can define the problem in terms of itself.

Recursion helps to write shorter code.

We can convert loops into a recursive function.

Recursion vs Iteration:

Recursive solution#

We can implement a recursive solution iteratively. Let's write a program to calculate the factorial of a number using a loop. In the iterative solution, we are not calling the same problem with a simpler version; instead, we are using the counter variable and we keep incrementing it until the given condition is true.

*/

```
// =====
// #include <iostream>

// using namespace std;

// // Iterative factorial function
// int factorial(int n)
// {
//     int fact = 1;
```

```

// if (n == 0)
// {
//     fact = 1;
// }

// for (int counter = 1; counter <= n; counter++)
// {
//     fact = fact * counter;
// }
// return fact;
//}

// main function
// int main()
//{
//     int n = 5;
//     int result;
//     // Call factorial function in main and store the returned value in result
//     result = factorial(n);
//     // Prints value of result
//     cout << "Factorial of " << n << " = " << result;
//     return 0;
//}
//=====================================================================
/*

```

Differences#

The following are the differences between recursion and iteration:

In the computer language, iteration allows you to repeat a particular set of instructions until the specified condition is met. The recursive function allows you to keep calling itself in the function body until some condition is met.

The sole purpose of iteration and recursion is to achieve repetition. Loops achieve repetition through the repetitive structure, whereas recursion achieves repetition through repetitive function calls.

Iteration terminates when loop condition fails. On the other hand, recursion terminates when the base condition evaluates to true.

Iteration happens inside the same function, which is why it takes less memory. In the recursive function, there is the overhead of function calls that makes our program slow and consumes more memory since each function call calls another copy of the function.

In iteration, our code size is very large. Meanwhile, recursion helps to write shorter code.

Iterative code is faster than recursive code.

Infinite loops will stop further execution of the program but do not lead to system crash. Infinite recursive calls, on the other hand, will result in a CPU crash because of memory overflow.

Why use recursion?#

Using depends upon the requirements of your problem. The problems that can be defined in terms of itself are the best candidates for recursion.

Use recursion when a problem can be divided into simpler versions of itself. Consider the example of a file searching system. In a file searching system, you start with the main folder and then go through each subfolder to find a particular file.

For such a type of problem, use recursion. The recursive solution helps you write shorter and more understandable code that makes debugging easier.

However, if performance is your main concern, write the iterative solution since it takes less time and memory for its execution. Recursive calls take more time and extra memory.

That said, whichever you want to use is totally your choice.

What is an array?

In the lesson on variables, we saw that a variable is just like a cabinet that can store one item only. To store the item in the cabinet, we must decide its type (analogous to data type) and put a unique label on it (analogous to variable name).

If we have to store a lot of items of the same type, putting a label on each cabinet is quite a tedious task. Instead, we can just store the items of the same type under the same label.

This is where arrays come in.

i An array is a sequential collection of values of the same data type under the same name. It is a derived data type.

In the above figure, we have stored items of the same type under a single label Cash.

Basic terms

Let's get introduced to the basic terms associated with an array.

Element

The array element is a value stored in an array. Elements in an array are stored at neighboring memory locations.

Index

An array index identifies the position of an element in an array. It starts from 0 and increments by one for each element added in an array.

Size

The size of an array is the total number of elements stored in an array.

In the above figure, you can see an array with 6 elements. The name of an array is Number, and its size is 6.

The first element, 10, is stored at index 0.

The second element, 20, is stored at index 1.

The third element, 30, is stored at index 2.

The fourth element, 40, is stored at index 3.

The fifth element, 50, is stored at index 4.

The sixth element, 60, is stored at index 5.

Why use arrays?#

The limitation of fundamental data types such as int, long, char, etc., is that they can store one value at a time. When we have large volumes of data, we need a data type that can store and access different amounts of data under a single name.

Example#

Suppose there are 100 students in a class, and you want to store their roll numbers. Declaring 100 variables and then storing the roll number of each student is quite an impractical approach. Here, arrays will come in handy!

Creating an Array

Introduction#

An array is a collection of elements of the same data type a the single name. Let's see how we can declare and initialize an array in C++.

Array declaration#

The general syntax for declaring an array is given below:

```
data_type array_size[array_size];  
/*  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
  
//   int Roll_Number[5];  
  
// }  
// ======  
/*
```

We declare an array Roll_Number that can store 5 integer values. The compile reserves space for 5 elements of type int consecutively in memory. Since the data type of an element is int, it reserves 4 bytes for each element, and in total, it reserves $5 \times 4 = 20$ bytes with the name Roll_Number. Since an array can store 5 elements, the size of an array is 5.

Array initialization#

Approach 1

We can assign a value to an array element by accessing its index.

```
array_name[index_number] = value;  
*/  
  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
  
//   int Roll_Number[5];  
  
//   Roll_Number[0] = 100;  
//   Roll_Number[1] = 101;  
//   Roll_Number[2] = 102;  
//   Roll_Number[3] = 103;  
//   Roll_Number[4] = 104;  
  
// }  
// ======  
/*
```

The code above initializes an array Roll_Number that stores:

100 at index 0

101 at index 1

102 at index 2

103 at index 3

104 at index 4

You must be wondering if we can just declare and initialize all elements in an array in one go. The answer is yes.

We can assign a value to the array elements in the declaration step.

```
data_type array_name[array_size(optional)]={value1, value2, value3,.....valueN};  
*/  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main() {  
  
//   int Roll_Number[ ] = { 100, 101, 102, 103, 104 };  
  
// }
```

```
// =====
/*
[i] If we are initializing an array in the declaration step, we don't need to specify the size of the array. The compiler automatically determines its size.
```

Initializing an array with fewer elements than its total size#

If we initialize an array with elements fewer than its total size, the compiler automatically initializes the remaining elements with their default values.

```
*/
// =====
// #include <iostream>
// using namespace std;
// int main()
//{
//    int Roll_Number[5] = {100, 101};
//    cout << Roll_Number[4];
//}
// =====
/*
```

In the code above, even though we have not initialized the values from index 2 to 4, the compiler automatically initializes them to their default values.

 If we specify the size of an array in the declaration step and then initialize more values than the specified size, the compiler will generate an error.

Accessing an Array

Array traversal#

We can access the elements stored in an array by writing the array name, which is followed by its index in the square brackets.

The first element of an array is stored at index 0, and the last element is stored at index size-1. Array[0] refers to the first element in an array. Array[1] refers to the second element, and so on.

```
/*
// =====
// #include <iostream>
// using namespace std;
// int main(){
//    int Roll_Number[5] = {100, 101, 102, 103, 104};
```

```
// cout << Roll_Number[4];
// }
// =====
/*
The code given above accesses the element at an index of four, which is the 5^{th}
5
th
```

element of an array.

 If you try to access an index that is greater than the size of an array, the compiler won't generate an error, but may give you an unexpected output.

Print all elements of an array using for loop#

Accessing each and every element in an array and then printing its value is a repetitive task. Let's write a code that prints all the elements of an array using the for loop.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
// {

//     int size = 5;
//     // Initialize array
//     int Roll_Number[] = {100, 101, 102, 103, 104};

//     // Print Array
//     for (int i = 0; i < size; i++)
//     {
//         // Access element at index i
//         cout << Roll_Number[i] << " ";
//     }
//     cout << endl;
// }

// =====
/*
Array updation#
We can change the value of the elements in an array by accessing its index and assigning a new value to that array index.
*/
// =====
// #include <iostream>

// using namespace std;

// int main()
```

```

//{

// int size = 5;
// // Initialize array
// int Roll_Number[size] = {100, 101, 102, 103, 104};

// cout << "Values of array before updation: " << endl;
// // Print values of array
// for (int i = 0; i < size; i++)
// {
//     // Accesss elements of array at index i
//     cout << Roll_Number[i] << " ";
// }
// cout << endl;
// // Update values of array element at index 3 and 4
// Roll_Number[3] = 22222;
// Roll_Number[4] = 33333;
// cout << "Values of array after updation: " << endl;
// // Print updated values of array
// for (int i = 0; i < size; i++)
// {
//     // Access elements of array at index i
//     cout << Roll_Number[i] << " ";
// }
// cout << endl;
//}
//=====================================================================
/*

```

Arrays and Functions

Passing an array to a function#

To pass an array to a function, we just have to specify the array type, followed by an array name and square brackets in the function parameters.

```

/*
// #include <iostream>

// using namespace std;

// // print_array function will print the values of an array
// void print_array(int number[], int size)
//{
//    for (int i = 0; i < size; i++)
//    {
//        cout << number[i] << " ";
//    }
//    cout << endl;
//}

// // modify_array function

```

```

// void modify_array(int number[], int size)
//{
//    // Traverse array
//    for (int i = 0; i < size; i++)
//    {
//        // If value less than 50 set it to -1
//        if (number[i] < 50)
//            number[i] = -1;
//    }
//    cout << "Values of array inside the function:" << endl;
//    // Call print_array function
//    print_array(number, size);
//}

//// main function
// int main()
//{
//    // Initialize size of an array
//    int size = 8;
//    // Initialize values of array
//    int number[size] = {67, 89, 56, 43, 29, 15, 90, 67};

//    cout << "Values of array before function call:" << endl;
//    // Call print_array function
//    print_array(number, size);
//    // Call modify_array function
//    modify_array(number, size);
//    cout << "Values of array after function call:" << endl;
//    // Call print_array function
//    print_array(number, size);
//}
// =====
/*

```

Arrays are passed by reference#

In the code above, did you notice that any change made in the elements of an array inside the `modify_array` function is reflected in the `main` function?

This was not the case with variables because, by default, variables are passed by value.

When we pass an array to the function, we don't need to specify the size of an array in square brackets. This is because we need the size of an array when we are creating a new array. However, when we pass an array in the function, we are just passing an original array to the function. This means if we made any changes inside the function, we would see those changes outside the function. That is why we can say that by default, arrays are passed by reference.

Creating a Two-Dimensional Array

Two-dimensional arrays#

A two-dimensional array is an array of arrays.

Two-dimensional arrays represent a matrix. We can access the element in a two-dimensional array by the row and column index. Both the row and column index start at 0.

Declaration#

The general syntax for declaring a two-dimensional array is:

```
array_data_type array_name[row_size][col_size];
```

In the 2D array declaration, we specify the data type of an array followed by an array name, which is further followed by the row index and column index in square brackets.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {

//   int Student[10][5];

// }

// =====
/*
```

We have declared a two-dimensional array `Student[10][5]` that can hold 10 arrays of `Student[5]`. Each `Student[5]` array can store 5 integer values.

The code given above reserves space for $10 \times 5 = 50$ elements of type `int` consecutively in memory. Since the element is of type `int`, the compiler reserves 4 bytes for each element, and in total, it reserves $50 \times 4 = 200$ bytes with the name `Student`.

Array initialization#

We can assign a value to the array elements in a 2D array by accessing its row and column index.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main() {

//   int Student[2][2];

//   Student[0][0] = 100;
//   Student[0][1] = 134;

//   Student[1][0] = 34;
```

```

// Student[1][1] = 189;

// }
// =====
/*
Array initialization in the declaration step#
*/
// =====
// #include <iostream>

// using namespace std;

// int main() {

// int Student[][3] = {{100, 134, 234}, {34, 189, 221}, {109, 139, 56}};

// }
// =====
/*
[i] If we initialize an array with elements fewer than its total size, it automatically initializes the remaining elements with their default values.

[i] When initializing a 2-D array, specifying the first dimension is optional. The compiler will infer the number of rows from the statement. In the above program, changing Student[3][3] to Student[][3] is fine, but either Student[][] or Student[3][] isn't valid.

[i] If we aren't initializing a 2-D array, all of its dimensions must be specified.

```

That is all about creating a two-dimensional array in C++. In the next lesson, we learn how to access and update elements stored in two-dimensional arrays.

```

/*
*/
Accessing Two-Dimensional Arrays:

```

Array traversal#

To access elements in a two-dimensional array, we have to specify a row and column index.

Print all elements of a 2D array using for loop#

Accessing each and every element in an array and then printing its value is a repetitive task. Instead, let's write a code that prints all the elements of the 2D array using the for loop. We will need two nested for loops, one to iterate through the rows of the 2D array and the other to iterate through the columns in each row.

```

// =====
// #include <iostream>

// using namespace std;

// int main()
//{
// // Initialize row and column index

```

```
// int row = 3, column = 3;
// // Initialize static 2D array
// int Student[row][column] = {{100, 134, 234}, {34, 189, 221}, {109, 139, 56};

// // Print static 2D Array
// for (int i = 0; i < row; i++)
// {
//     for (int j = 0; j < column; j++)
//     {
//         // Access element at row index i and column index j
//         cout << Student[i][j] << " ";
//     }
//     cout << endl;
// }
// =====

/*
Computer Memory
```

Introduction to memory#

Suppose we have a fictional society named Memory Society. Memory Society has a lot of consecutive storage houses in a straight line. If a person wants to store his item in a Memory Society, they will be allocated a storage house with their name. If someone wants to access the items in the storage house, there must be some unique address to locate it. The address of the first storage house is 1000, the address of the second storage house is 1001, the third storage house is 1002, and so on.

Memory Society is just like a computer's memory, and the storage house is similar to the memory cell. Each memory cell can hold 1 byte of data. If someone wants to store 4 bytes of data, they will be allocated 4 consecutive cells in a computer's memory.

A computer's memory can be thought of as an array of bytes.

i Bit is short for binary digit. It is the smallest possible unit of information that can be stored on a computer. Its value is either 0 or 1. We bundled together the bits into 8 bits collections, known as bytes. There are 8 bits in 1 byte. With 1 byte, values from 0-255 can be represented.

The address of the storage house is like a memory address where the particular value is stored.

The number that uniquely identifies the location in the memory is known as the memory address.

Variables and Memory

Variables#

To understand pointers, you must know how variables are stored in memory.

We have already seen that a variable is a named location in a computer's memory where we can store our data.

```
*/
```

```
=====
```

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

// int main() {
//   // Declares a variable John of type int
//   int John;
// }
// =====
/*
```

How are variables stored in memory?

Consider the analogy given in the previous lesson. The statement on Line No. 7 says reserve the consecutive 4 houses for John in the Memory Society. We don't decide where John will actually live. This is automatically done by the society owner named Operating System. Operating System will allocate a space in the Memory Society that will have a unique address to locate it. Suppose it is 1003.

So in the computer program, the statement on Line No. 7 will reserve 4 bytes for the variable John at some location in the memory.

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main()
//{
//   // Declares a variable John of type int
//   int John;
//   // Stores 10 in variable John
//   John = 10;
//}
// =====
/*
```

After some time, John stores something in his storage house, and he uses the address of the storage house to reach it and not his name.

So when the program executes, we think that the data is accessed and modified using the identifier. But this is not how things work!

During the compilation time, the compiler maps each variable name to the unique memory address in the memory. Our machine accesses and modifies all the data by their addresses in the computer's memory. But why use an identifier when we can access our data using the memory address?

Identifiers have to be used because we have to declare multiple variables in a single program, so it must be difficult for the human to remember all the addresses. Therefore, we use identifiers to keep things simple.

Address-of Operator

Address-of operator#

We now know that when we declare a variable, the compiler allocates space someplace in the memory location. What if we want to know the memory address where the variable has been allocated memory?

For this, we will use the address-of operator & before the identifier to access the address of the variable.

The address-of operator (&) is a unary operator. It is used to extract the memory address of the variable.

```
/*
// =====
// =====
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//   // Declare a variable John
//   int John = 10;
//   // Prints the memory address in which value of John is stored
//   cout << "John Address = " << &John << endl;
//   // Prints the value of John
//   cout << "John Value = " << John << endl;
//   return 0;
//}
// =====
/*
```

Explanation#

Line No. 7: Stores 10 in John

Line No. 9: Shows us the address of John

 Ox at the start of the memory address shows that the address is in hexadecimal format.

Line No. 11: Displays the value stored in John

 The memory address depends upon your machine. Therefore, if you run the same program on a different machine, you will get a different memory address.

We can access the variable address using the address-of (&) operator. Here, we have noticed that memory cells can store numbers, and addresses are numbers, too. So can we declare a variable and store the address of another variable in it?

We cannot store the address of some variable in the value of another NORMAL variable.

What is a Pointer?

Introduction#

Suppose your friend asks you for a good source to study data structures from. You will find some good sources and send a hyperlink to them.

Since downloading all the content from the web pages and then sending them in an email requires a lot of memory, you would just send a link to the source. Whenever your friend wants to read the content, they can visit the link and they will be good to go.

Pointers are similar to the hyperlink that stores the location of some other data.

In C++, a pointer is a variable that stores the address of another variable.

Pointer declaration#

To declare a variable as a pointer, its identifier must be preceded by an asterisk *. When we use * before the identifier, it indicates that the variable being declared is a pointer.

```
data_type *variable;  
/*  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main()  
// {  
//   // Declares a pointer variable John  
//   int *John = nullptr;  
//   // cout << *John << endl; //never deference an null pointer  
//   cout << John << endl;  
//   cout << "hello " << endl;  
//   return 0;  
// }  
  
// ======
```

The statement in Line No. 7 declares a pointer John, and its sole purpose is to store the address of some other variable. Here, John only points to the value whose data type is int. Therefore, we can say that John is a pointer to int.

 It's considered a best practice to set a pointer to nullptr when it is declared, unless it is initialized to some valid address, as we shall see later in this lesson.

 It's a good practice to use ptr in a pointer's variable name. It indicates that a variable is a pointer, and must be handled differently.

 If we declare multiple pointers in the same line, we must use an asterisk * before each identifier.

Pointer initialization#

To initialize a pointer, we must store the address in it. The basic syntax for storing an address of another variable in the pointer variable is given below:

```
int *ptr = &variable;  
or  
int *ptr;  
ptr = &variable;  
  
/*  
  
// ======  
// #include <iostream>  
  
// using namespace std;  
  
// int main()  
// {  
//   // Declares a variable Alice  
//   int Alice = 5;  
//   // Declares a pointer variable John that can point to int value  
//   int *John;  
//   // Stores the address of Alice in John  
//   John = &Alice;  
//   // Prints value of Alice  
//   cout << "Value of Alice = " << Alice << endl;  
//   cout << "Value of John = " << *John << endl;  
//   // Prints value (address of Alice) of John  
//   cout << "Value of Alice = " << &Alice << endl;  
//   cout << "Value of John = " << John << endl;  
  
//   return 0;  
// }  
// ======  
/*
```

Explanation#

Line No. 11: John is a pointer, and &Alice gives us the memory address of Alice. Therefore, we are storing the address of Alice in John. We can say John is pointing to Alice.

Using ampersand & is like getting the address of Alice instead of seeing what Alice has stored in its storage house.

 An error occurs when the pointer points to the variable of a different data type.

Null pointer#

If the pointer is pointing to nothing, then it should be initialized to nullptr. It is known as a null pointer. The value of the null pointer is 0.

 If we don't initialize a pointer, it is automatically initialized to 0.

Dereferencing Operator:

Indirection/dereferencing operator#

Consider the example given in the previous lesson. John's storage house is pointing to Alice's storage house, so John is a pointer here. What if John wanted to know what value is stored in Alice's house?

For this, we will use the dereference operator * before the pointer name to access the value of the variable to which the pointer is pointing.

The dereference operator * is a unary operator. It gives the value of the variable to which the pointer is pointing. This process is known as dereferencing a pointer.

```
/*
// =====
// #include <iostream>

// using namespace std;

// int main()
//{
//     // Declares a variable Alice
//     int Alice = 5;
//     // Declares a pointer variable John that can point to int value
//     int *John = nullptr;
//     // Stores the address of Alice in John
//     John = &Alice;
//     // Prints value of Alice
//     cout << "Value of Alice = " << Alice << endl;
//     // Prints value (address of Alice) of John
//     cout << "Value of John = " << John << endl;
//     // Prints value of Alice
//     cout << "Value of Alice = " << *John << endl;

//     return 0;
//}
// =====
/*
```

Explanation#

Line No. 17: John accesses the value stored in Alice and prints it to the console. Using asterisk * is like going to Alice's house and seeing what she has stored in her house.

✗ Trying to dereference an uninitialized or null pointer generates an error.

```
*/
```

```
/*
```

References:

A reference is an alias for an existing variable. It can be created using the & operator.

Once created, the reference can be used instead of the actual variable. Altering the value of the reference is equivalent to altering the referenced variable.

Note that we don't have to worry about the * operator when de-referencing a variable reference. For example, lines 7, 11 and 14 in the following program.

Note that we are using std::cout every time we want to display something to the screen, instead of cout since we haven't used the using namespace std statement. Similarly, we are using std::endl instead of endl. Both options work the same, but avoiding the using namespace statement has its advantages.

```
/*
// =====
// #include <iostream>

// int main()
//{
//    int i = 20;
//    int &iRef = i;
//    std::cout << "i: " << i << std::endl;
//    std::cout << "iRef: " << iRef << std::endl;

//    std::cout << std::endl;

//    iRef = 30; // Altering the reference

//    std::cout << "i: " << i << std::endl;
//    std::cout << "iRef: " << iRef << std::endl;
//}
// =====
*/
```

References vs. pointers#

There is a lot of overlap between pointers and references but the two have some stark differences as well.

A reference is never nullptr. Therefore, it must always be initialized by having an existing variable assigned to it. The following lines would not work:

References behave like constant pointers. A reference always refers to its initial variable. The value of the variable can change but the reference cannot be assigned to another variable.

Like pointers, a reference can only be initialized by a variable of the same type.

References as parameters#

References allow functions to modify the value of a variable. When a normal variable is passed to a function, a copy of its value is made and the variable itself remains untouched. However, if a reference is passed, the actual value of the variable is used and can therefore be modified.

```
/*
// =====
// #include <iostream>

// void xchg(int &x, int &y)
// { // Reference parameters
//     int t = x;
//     x = y;
//     y = t;
```

```

//}

// int main()
//{
//    int a = 10;
//    int b = 20;
//    std::cout << "a: " << a << std::endl;
//    std::cout << "b: " << b << std::endl;

//    xchg(a, b);
//    std::cout << std::endl;

//    std::cout << "a: " << a << std::endl;
//    std::cout << "b: " << b << std::endl;
//}
//=====
/*

```

Function and Pointers

Passing pointers to the function#

In a previous lesson, we discussed the following two ways of passing actual parameters to the formal parameters in the function:

Pass by value

Pass by reference

However, there is another way to pass arguments to the function that is passed by reference with a pointer parameter.

The function pointer parameter receives the address of the parameter. Then, it uses the dereference operator to access the value of the variable.

When we want to pass the pointer, we will declare a function parameter as a pointer. To declare a function parameter as a pointer, use an asterisk * before the function parameter. Then, pass the address of the variable in the actual parameter using the address-of & operator.

```

*/
//=====

// #include <iostream>

// using namespace std;
// // function definition
// void passPointer(int *number)
//{
//    // Multiply the number by 10
//    *number = *number * 10;
//    cout << "Value of number inside the function = " << *number << endl;
//}

// int main()
//{
//    // Initialize variable

```

```
// int num = 10;
// cout << "Value of number before function call = " << num << endl;
// // Call function
// passPointer(&num);
// cout << "Value of number after function call = " << num << endl;

// return 0;
//}
// =====
/*
```

Explanation#

In the code above, we have two functions:

passPointer function

main function

passPointer function#

Line No. 5: The passPointer function receives an address of the int value and stores it in the number. Since the function is of type void, no value is returned.

Line No. 7: Multiplies the value that the pointer number is pointing to by 10 and stores the result in the location pointed by the number.

Line No. 8: Prints the value pointed by the number.

main function#

Line No. 13: Initializes a variable num.

Line No. 14: Prints the value of the num before the function call.

Line No. 16: Calls a function passPointer and passes the address of the num to function. The execution control is transferred to Line No. 5.

Line No. 17: Prints the value of the num after the function call.

i By default, pointers are passed by value. When we call the function, the value of the address is copied to the pointer variable. So if we change the value of the pointer inside the function, we cannot see that change outside the function body.

```
void passPointer(int *number) {
    int value = 13;
    number = &value;
    *number = *number + 14;
}
```

The value of num will remain the same. In the passPointer body, we declared the new variable value and then updated the address stored in number. Now the number is pointing to value. passPointer is no more pointing to num.

Types of Allocation:

Introduction#

In C++, we can allocate memory in two ways:

Static allocation

Dynamic allocation

Static allocation#

In static allocation, a fixed amount of memory is allocated to the variables or arrays before the execution of the program (during compile-time), and we cannot request more memory while the program runs.

In static allocation:

We must know the size of an array or variable during the compile time.

Memory is allocated and deallocated to the variables by the compiler.

Dynamic allocation#

Sometimes you will encounter a situation where you don't know in advance how much memory is needed to store the data. Thus, dynamic allocation is needed.

If you initialize an array with fewer characters than the size of an input sentence, then you may get an error.

If you initialize an array with more characters than the actual size of an input sentence, then the unused memory is wasted.

Here, dynamic allocation comes in.

Dynamic allocation#

In dynamic allocation:

We can get as much memory as we want during the program execution;

Memory is allocated and deallocated by the programmer during the run-time.

Allocation of Dynamic Memory:

Introduction#

For dynamic allocation, we have to do the following two steps:

First, allocate the dynamic space.

Then, store the starting address of the dynamic space in the pointer.

`new operator:#`

The unary operator `new` allocates memory in bytes during the run time from the free store.

If memory is available on the free store, the `new` operator will reserve that memory and return its starting address.

Syntax#

The basic syntax for getting memory during the run-time is given below:

```

new datatype;

*/
// =====
// #include <iostream>
// using namespace std;

// int main() {
//   // Reserve 4 bytes in free store
//   new int;
//   return 0;
// }
// =====
/*

```

In the above program, we are requesting 4 bytes of memory from the free store. The new operator reserves the 4 bytes of memory in the free store and returns the starting address of the allocated space. To access the dynamically allocated space, we must store the returned address somewhere. Here, pointers come in handy!

Pointers#

```

datatype *ptr;
ptr=new int;

```

or

```
datatype *ptr = new int;
```

```

*/
// =====
// #include <iostream>
// using namespace std;

// int main()
// {
//   // Declare pointer ptr
//   int *ptr = nullptr;
//   // Store the starting address of dynamically reserved 4 bytes in ptr
//   ptr = new int;
//   return 0;
// }
// =====
/*
```

Accessing dynamic space#

```

*/
// =====
// #include <iostream>
// using namespace std;

// int main()
// {
```

```
// // Declare pointer ptr
// int *ptr = nullptr;
// // Store the starting address of dynamically reserved 4 bytes in ptr
// ptr = new int;
// // Store 100 in dynamic space
// *ptr = 100;
// // Print value pointed by ptr
// cout << *ptr;
// return 0;
//}
//=====================================================================
/*
```

Deallocation of Dynamic Memory

Introduction#

The compiler automatically deallocates the static space when it is not used anymore. Since dynamically allocated memory is managed by a programmer, when dynamically allocated space is not required anymore, we must free it.

delete operator#

The delete operator allows us to free the dynamically allocated space.

Syntax#

The basic syntax for releasing the memory that the pointer is pointing to is given below:

```
delete pointer;
```

```
/*
//=====================================================================
// #include <iostream>
// using namespace std;

// int main()
//{
// // Declare pointer ptr
// int *ptr = nullptr;
// // Store the starting address of dynamically reserved 4 bytes in ptr
// ptr = new int;
// // Store 100 in dynamic space
// *ptr = 100;
// // Print value pointed by ptr
// cout << *ptr;
// // Free the space pointed by pointer ptr
// delete ptr;
// return 0;
//}
//=====================================================================
=====
// #include <iostream>
// using namespace std;
```

```

// int main()
//{
// // Declare pointer ptr
// int *ptr = nullptr;
// // Store the starting address of dynamically reserved 4 bytes in ptr
// ptr = new int;
// // Store 100 in dynamic space
// *ptr = 100;
// // Print value pointed by ptr
// cout << *ptr << endl;
// // Free the space pointed by pointer ptr
// delete ptr;
// // Initialize a variable a
// int a = 70;
// // Store the address of a in ptr
// ptr = &a;
// // Prints the value pointed by the ptr
// cout << *ptr;
// return 0;
//}
//=====================================================================
/*

```

In the above code, initially, pointer ptr points to the int value in the free store. We free the space pointed by the pointer ptr. After the deallocation, we store the address of variable a in pointer ptr. Now, pointer ptr points to the value of a.

 It's good practice to set the pointer to nullptr after deallocation, unless you are pointing to some other valid target.

Dynamic Arrays

Dynamic arrays#

In the arrays lesson, we discussed static arrays. In a static array, a fixed amount of memory is allocated to the array during the compile time. Therefore, we cannot allocate more memory to the arrays during program execution.

Suppose we have declared an array that can store five integer values.

What if we want to store more than five values in an array? Here is dynamic arrays come in!

Dynamic arrays can grow or shrink during the program execution.

Declaration#

The general syntax for declaring dynamic arrays is given below:

```
data_type *array_name = new datatype[size];
```

Initialization#

We can initialize a dynamic array just like a static array.

```
array[index]=value;
```

Deallocating array#

The basic syntax for deallocating a dynamic array is given below:

```
delete[]arrayname;
```

Print all elements of a dynamic array#

The code will initialize the dynamic array and then print all its elements using the for loop.

```
/*
// =====

// #include <iostream>

// using namespace std;
```

```
// int main()
// {
//   int size = 5;
//   // Declare dynamic array
//   int *Array = new int[size];
//   // Initialize dynamic array
//   for (int i = 0; i < size; i++)
//   {
//     Array[i] = i;
//   }
//   // Prints dynamic array
//   for (int i = 0; i < size; i++)
//   {
//     cout << Array[i] << " ";
//   }
//   // Deletes a memory allocated to dynamic array
//   delete[] Array;
// }
```

```
/*
// =====
```

Line No. 8: Reserves space for 5 integers and stores the starting address of allocated space in a pointer Array.

Line No. 10: Traverses an array and initializes its elements.

Line No. 14: Traverses an array and prints its value.

Line No. 18: Frees the space pointed by an Array.

Resizing dynamic array#

Let's write a program in which we will increase the size of an array and store some new elements.

```
/*
// =====
```

```

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

//// printArray function
// void printArray(int arr[], int size)
//{
// for (int i = 0; i < size; i++)
// {
// cout << arr[i] << " ";
// }
// cout << endl;
//}

//// main function
// int main()
//{
// // Initialize variable size
// int size = 5;
// // Create Array
// int *Arr = new int[size];
// // Fill elements of an array
// for (int i = 0; i < size; i++)
// {
// Arr[i] = i;
// }
// // Call printArray function
// printArray(Arr, size);
// // Create new array
// int *ResizeArray = new int[size + 2];
// // Copy elements in new arary
// for (int i = 0; i < size; i++)
// {
// ResizeArray[i] = Arr[i];
// }
// // Delete old array
// delete[] Arr;
// // Pointer Array will point to ResizeArray
// Arr = ResizeArray;
// // Store new values
// Arr[size] = 90;
// Arr[size + 1] = 100;
// // Call printArray function
// printArray(Arr, size + 2);
//}
// =====
/*
Explanation#
Suppose we want to resize the already declared array Arr in a program. We will follow the following steps:
```

Line No. 26: Creates a new array ResizeArray with the new size.

Line No. 28: Copies the elements of the old array Arr into the new array ResizeArray.

Line No. 32: We don't need the memory pointed by the Arr anymore. So, we delete it.

Line No. 34: We still want our array to be called Arr. Therefore we change the pointer.

Line No. 36: Stores the new values in an array Arr.

Tada! We have just resized the original array using pointers.

Introduction to Structures:

What is structure?#

Consider a blueprint used to construct a building. A blueprint is a guide that tells us what the basic architecture of the building is. For example, it will detail the number of floors, rooms, windows, etc. in the building.

We can use the same blueprint to construct multiple buildings, but each building will be different from others in properties. For example, if one building is of a white color, the other will be a red color.

Structure is just like a blueprint from which we can create a variable of our own data type.

The Structure is a user-defined data type that is used to store variables or arrays of different data types under a single name.

Example#

Suppose there are 100 students in a class, and you want to store their names, roll numbers, and marks.

To store data of each student, we can create 3 variables for each student. In total, we have to create 300 variables, which does not make sense.

Arrays can be used to store data of a similar kind. Here, the student's name will be string, and their roll number will be int. So, we cannot use arrays here!

Structures let us store data of different types.

We will define a Student structure that will act as a blueprint in our program. The Student structure will have 3 members: name, roll_number, and marks to store the information of Student.

We will then declare a variable whose type will be a Student for each student in the class that is known as the structure variable.

If the structure is like a blueprint on the page, the structure variable is like a building that has an actual physical existence and where we can live.

Defining Structure in C++ :

Introduction#

Structure is a user-defined data type. Therefore, before using structure in a program, we must tell the compiler what our structure will look like.

Defining structure#

The basic syntax for defining a structure in C++ is given below:

```
/*
// =====
// #include <iostream>

// using namespace std;
// // Student structure
// struct Student
// {
//   string name;
//   int roll_number;
//   int marks;
//   struct wow
//   {
//     int text;
//   };
// };
// // main function
// int main()
// {

//   return 0;
// }
// =====
/*
```

To define a structure in a program, use the `struct` keyword followed by a structure name, which is followed by curly braces and a semicolon at the end. Inside the curly braces, we declare the data members of the structure.

 Forgetting a semicolon after the structure definition generates an error.

Explanation#

In the above program, we have defined the structure `Student` from Lines No 5 to 9 . `name`, `roll_number`, and `marks` are the data members of the `Student`.

Have you noticed anything?

Here, we declare the variables of different data types under the same name.

 We will use `struct_name` later in a program to create a structure variable.

 A structure cannot contain a member of its own type.

Declaring Structure Variables in C++

Introduction#

Until now, we have seen how to create a structure in a program. As discussed earlier, the structure is like a blueprint of the building drawn on the page. When a structure is created, the computer does not allocate any memory to it.

The structure variable is like the building construct from the blueprint. The building has an actual physical existence. Therefore, to allocate memory to the structure, we must declare the structure variable in a program.

Basic syntax#

```
struct_name variable_name;
```

To declare a structure variable in a program, we write the name of the structure followed by the name of a structure variable, which is further followed by a semicolon ;

```
/*
// =====
// #include <iostream>

// using namespace std;
// // Student structure
// struct Student {
//   string name;
//   int roll_number;
//   int marks;
// };
// // main function
// int main() {
//   Student s1, s2, s3;
//   return 0;
// }
// =====
/*
```

Explanation#

Line No. 12 declares three structure variables s1, s2, and s3 in a program. The data type of these variables is Student.

Declaring a structure variable in the structure definition#

The structure variables can also be declared after the structure definition in a program.

To declare a structure variable in a structure definition, we write the struct keyword followed by the name of the structure, which is further followed by structure variable names and a semicolon.

```
/*
// =====
// #include <iostream>
```

```
// using namespace std;
// // Student structure
// struct Student {
//   string name;
//   int roll_number;
//   int marks;
// } s1, s2, s3;
// // main function
// int main() {
//   return 0;
// }
// =====
/*
```

Initializing and Accessing Members of a Structure Variable in C++ :

Introduction#

We have seen how to define a structure and declare a structure variable in a program. Let's see how we can store data in the member variables of the structure.

```
structure_var.structure_var = value;
```

To access the member of the structure variable, we write the name of the structure variable, followed by a dot operator, which is further followed by the name of the member. To assign a value to the member variable, we use the equal operator followed by the value and the statement terminator (i.e., semicolon).

```
*/
```

```
// =====
// #include <iostream>

// using namespace std;
// // Student structure
// struct Student
// {
//   string name;
//   int roll_number;
//   int marks;
// };
// // main function
// int main()
// {
//   Student s1, s2, s3;
//   // Assign value to members of s1
//   s1.name = "John";
//   s1.roll_number = 1;
//   s1.marks = 50;
//   cout << "s1 Information:" << endl;
//   // Print members of s1
//   cout << "Name = " << s1.name << endl;
//   cout << "Roll Number = " << s1.roll_number << endl;
//   cout << "Marks = " << s1.marks << endl;
//   // Assign value to members of s2
```

```

// s2.name = "Alice";
// s2.roll_number = 2;
// s2.marks = 43;
// // Print members of s2
// cout << "s2 Information:" << endl;
// cout << "Name = " << s2.name << endl;
// cout << "Roll Number = " << s2.roll_number << endl;
// cout << "Marks = " << s2.marks << endl;

// return 0;
//}
//=====
/*
Explanation#

```

Line No. 14: We are accessing the member name of s1 using the dot operator and then we set it to John.

Similarly, we access the member's roll_number and marks and set their values.

We repeat the same procedure to set the values for the rest of the structure variables.

Initializing members in one line#

You are probably thinking, setting each member of the structure variable is a tedious task. So, is there a way to set all the members of structure variables in one line?

Yes, there is. We can initialize structure variables in one line using the initializer list.

```

struct_var = { member1_val, member2_val, .... memberN_val};
*/
// =====
// #include <iostream>

// using namespace std;

// struct Student {
//   string name;
//   int roll_number;
//   int marks;
// };

// int main() {
//   struct Student s1, s2, s3;

//   s1 = {"John", 1, 50};

//   cout << "s1 Information:" << endl;
//   cout << "Name = " << s1.name << endl;
//   cout << "Roll Number = " << s1.roll_number << endl;
//   cout << "Marks = " << s1.marks << endl;

//   s2 = {"Alice", 2, 43};

```

```
// cout << "s2 Information:" << endl;
// cout << "Name = " << s2.name << endl;
// cout << "Roll Number = " << s2.roll_number << endl;
// cout << "Marks = " << s2.marks << endl;

// return 0;
//}
//=====
/*

```

Array of Structures:

Introduction#

We have 100 students in a class, and we have to store the name, age, and roll_number of each student, which means we need 300 variables. We have found a way to store all these variables under a single name.

However, to store data for each student in the class, we still have to declare 100 structure variables. Declaring 100 structure variables and then keeping track of them is quite difficult.

Here, an array of structures comes in handy!

In C++, each element of a structure array represents a structure variable.

```
*/
```

```
// =====
// #include <iostream>

// using namespace std;
// // structure Student
// struct Student
//{
//     string name;
//     int roll_number;
//     int marks;
// };
// // main function
// int main()
//{
//     struct Student s[100];

//     s[0] = {"John", 1, 50};

//     cout << "s1 Information:" << endl;
//     cout << "Name = " << s[0].name << endl;
//     cout << "Roll Number = " << s[0].roll_number << endl;
//     cout << "Marks = " << s[0].marks << endl;

//     s[1] = {"Alice", 2, 43};

//     cout << "s2 Information:" << endl;
//     cout << "Name = " << s[1].name << endl;
//     cout << "Roll Number = " << s[1].roll_number << endl;
```

```
// cout << "Marks = " << s[1].marks << endl;  
// return 0;  
// }  
// ======  
/*  
Explanation#
```

We declare an array named s with a capacity to store 100 structure variables of Student.

s[0] stores the information for the first student, s[1] for the second, s[2] for the third, and so on. We pass the initializer list to the first structure variable in an array.

We repeat the same process for setting values for the rest of the structure variables.

Pass a structure as a function argument#

In the previous lesson, we saw that printing the members of each structure variable is a repetitive task. Can we define a function in which we just pass the structure variable, and it prints the values for us? Yes, we can.

```
*/  
// ======
```

```
// #include <iostream>  
  
// using namespace std;  
  
// // Student structure  
// struct Student  
// {  
//   string name;  
//   int roll_number;  
//   int marks;  
// };  
// // printStudent function  
// void printStudent(Student s)  
// {  
//   cout << "Student information" << endl;  
//   cout << "Name = " << s.name << endl;  
//   cout << "Roll Number = " << s.roll_number << endl;  
//   cout << "Marks = " << s.marks << endl;  
// }  
// // main function  
// int main()  
// {  
//   struct Student s[100];  
  
//   s[0] = {"John", 1, 50};  
//   printStudent(s[0]);  
  
//   s[1] = {"Alice", 2, 43};  
//   printStudent(s[1]);
```

```
// return 0;
//}
// =====
/*
Explanation#
```

In the above program, we define a function printStudent on Line No. 12 that takes a structure variable s in its arguments and performs an operation on it.

Return a structure from a function#

So far, we have not seen a way to return multiple variables of different data types from a function. By returning a structure from a function, we can return multiple variables of different data types.

```
*/
```

```
// =====
```

```
// #include <iostream>

// using namespace std;
// // Student structure
// struct Student {
//   string name;
//   int roll_number;
//   int marks;
// };
// // function fillStudent
// Student fillStudent(string name, int roll_number, int marks) {
//   Student s;
//   s.name = name;
//   s.roll_number = roll_number;
//   s.marks = marks;
//   return s;
// }
// // printStudent function prints the members of structure variable
// void printStudent(struct Student s) {

//   cout << "Student information" << endl;
//   cout << "Name = " << s.name << endl;
//   cout << "Roll Number = " << s.roll_number << endl;
//   cout << "Marks = " << s.marks << endl;

// }

// int main() {
//   struct Student s[100];

//   s[0] = fillStudent("John", 1, 50);
//   printStudent(s[0]);

//   s[1] = fillStudent("Alice", 2, 43);
//   printStudent(s[1]);
```

```
// return 0;
//}
// =====
/*
Structure and Pointers
```

We have already learned that there are pointers to built-in data types such as int, char, double, etc.

Like we have a pointer to int, we can also have pointers to the user-defined data types such as structure.

The pointer that stores the address of the structure variable is known as a structure pointer.

Declaring structure pointer#

The basic syntax for declaring a structure pointer is given below:

```
struct structure_name *ptr;
```

Explanation#

Line No. 15: Declares a variable s1 of type Student

Line No. 17: Declares a pointer variable ptrs1 of type Student

Line No. 19: Stores the address of s1 in ptrs1

Accessing structure members through a structure pointer#

We can access members of structure through a structure pointer in two ways:

Indirection and the dot operator

Arrow operator

Indirection and dot operator#

the basic syntax for accessing the structure members through the structure pointer is given below:

```
*(structure_ptr).struct_name = value;
```

To access the members of the structure variable to which the structure pointer is pointing, we will first use the dereference operator with a structure pointer, which is followed by the dot operator and the member whose value you want to access.

```
*/
// =====
// #include <iostream>

// using namespace std;

/// / Student structure
// struct Student
// {
//   string name;
```

```

// int roll_number;
// int marks;
// };

/// main function
// int main()
//{
// // Declare structure variable
// struct Student s1;
// // Declare structure pointer
// struct Student *ptrs1;
// // Store address of structure variable in structure pointer
// ptrs1 = &s1;

// // Set value of name
// (*ptrs1).name = "John";
// // Set value of roll_number
// (*ptrs1).roll_number = 1;
// // Set value of marks
// (*ptrs1).marks = 50;

// // Print value of structure member
// cout << "s1 Information:" << endl;
// cout << "Name = " << (*ptrs1).name << endl;
// cout << "Roll Number = " << (*ptrs1).roll_number << endl;
// cout << "Marks = " << (*ptrs1).marks << endl;

// return 0;
//}

```

// =====

/*

Arrow operator#

You must be thinking that the above method of accessing structure members using a pointer is quite confusing! Can't we just access them using one simple operator?

Yes, we can. Here is where the arrow operator comes in!

structre_pointer -> var =value;

To access the structure members using the arrow operator, we have to write the name of the structure pointer followed by an arrow operator ->, which is further followed by a structure member and semicolon.
*/

```

// =====
// #include <iostream>

// using namespace std;

/// Student structure
// struct Student

```

```
// {
// string name;
// int roll_number;
// int marks;
// };

// // main function
// int main()
//{
// // Declare structure variable
// struct Student s1;
// // Declare structure pointer
// struct Student *ptrs1;
// // Store address of structure variable in structure pointer
// ptrs1 = &s1;

// // Set value of name
// ptrs1->name = "John";
// // Set value of roll_number
// ptrs1->roll_number = 1;
// // Set value of marks
// ptrs1->marks = 50;

// // Print value of structure member
// cout << "s1 Information:" << endl;
// cout << "Name = " << ptrs1->name << endl;
// cout << "Roll Number = " << ptrs1->roll_number << endl;
// cout << "Marks = " << ptrs1->marks << endl;

// return 0;
//}
// =====
```

/*

What is OOP?

Historical Background#

It was in the 60's and early 70's when the idea of object-oriented programming started occupying the minds of programmers. Keeping in view the benefits acquired through object orientation in SIMULA, the scientists encouraged languages using the same approach for programming. Object-oriented programming was a complete paradigm shift from the popular structural programming.

Object: A Fundamental Entity#

Object-oriented programming is based on the idea of an object. An object is an entity with some data and operations. Data is also referred to as properties of the object whereas operations include accessing and modifying those properties along with other functions that depict the behavior of the object.

Example#

A fork is an object with properties including a number of prongs, its size, and material (made of plastic or metal), etc. Behavior and functions of a fork include shredding, squashing, making design or may be simply eating.

Explanation#

Programmers realized that when we represent entities in the program as objects, having their behaviors and properties, it becomes easy to deal with the increasing code complexity as well as the code becomes more reusable. So, a fork object can be part of a dinner set, and a similar object may also be sold separately. Once, we know its properties and behavior, we only need to reuse the same piece of code whenever a fork is needed.

C++ as Object Oriented Language:

Origin of OOP in C++#

C is a procedural language and it was used and trusted by programmers at large. When the object-oriented paradigm was being introduced, Bjarne Stroustrup incorporated the object orientation features in C and hence introduced C++. It was initially introduced as C with classes and later renamed to C++.

Origin of OOP in C++#

C is a procedural language and it was used and trusted by programmers at large. When the object-oriented paradigm was being introduced, Bjarne Stroustrup incorporated the object orientation features in C and hence introduced C++. It was initially introduced as C with classes and later renamed to C++.

Concepts in OOP#

Soon C++ became widely acceptable object-oriented language. To introduce an object-oriented approach in C Stroustrup introduced concepts of Classes, objects, inheritance, dynamic binding, and polymorphism. We will be discussing each one of them later in our chapters in detail.

Preference over other Languages#

C++ was successfully used in many applications including embedded systems, system programming, as well as for higher level programming tasks. Some people though argue that C++ is not a pure object-oriented Language. Reason being the use of main() function in C++ that can exist without any Class. Since making a Class is not a primary requirement in C++ hence it is considered a partial object-oriented language. However, the choice and use of language depend on the programmer and his nature of the problem. To-date many

programmers still use C++ as object-oriented language regardless of other Pure object-oriented languages available.

What is a Class?:

Custom Objects#

In C++, we have several different data types like int, string, bool etc. An object can be created out of any of those types. An object is an instance of a class. Well, object-oriented programming wouldn't make sense if we couldn't make our own custom objects. This is where classes come into play.

Classes are used to create user-defined data types. The predefined data types in C++ are classes themselves. We can use these basic data types to create our own class. The cool part is that our class can contain multiple variables, pointers, and functions which would be available to us whenever a class object is created.

Data Members#

These are also known as the member variables of a class. This is because they contain the information relevant to the object of the class. A car object would have a top speed, the number of seats it has, and so many other pieces of data that we could store in variables.

Member Functions#

This category of attributes enables the class object to perform operations using the member variables. In the case of the car class, the Refuel() function would fill up the FuelTank property of the object.

Benefits of Using Classes#

The concept of classes allows us to create complex objects and applications in C++. This is why classes are the basic building blocks behind all of the OOP's principles.

Classes are also very useful in compartmentalizing the code of an application. Different components could become separate classes which would interact through interfaces. These ready-made components will also be available for use in future applications.

The use of classes makes it easier to maintain the different parts of an application since it is easier to make changes in classes .

Class Definition

Similar to functions, classes need to be created outside the main(). The written code of a class and its attributes are known as the definition or implementation of the class.

Definition Template#

*/

```
// =====
// class className {

// /* All member variables
// and member functions*/

// }; //The semi-colon operator ends the class
```

```
// =====  
/*  
The class command tells the compiler that we are creating our own custom class. All the members of the class  
will be defined within the class scope.
```

Creating a Class Object#

The className will be used to create an instance of the class in our main program. We can create an object just like we create objects of other data types:

```
/*  
// =====  
// class className  
// {  
//   // ...  
// };  
  
// int main()  
// {  
//   int i;    // integer object  
//   className c; // className object  
// }  
// =====  
/*
```

Access Modifiers

In C++, we can impose access restrictions on different data members and member functions. The restrictions are specified through access modifiers. Access modifiers are tags we can associate with each member to define which parts of the program can access it directly.

There are three types of access modifiers. Let's take a look at them one by one.

Private#

A private member cannot be accessed directly from outside the class. The aim is to keep it hidden from the users and other classes. It is a popular practice to keep the data members private since we do not want anyone manipulating our data directly. By default, all declared members are private in C++. However, we can also make members private using the private: heading.

```
/*  
// =====  
// class Class1  
// {  
//   int num; // This is, by default, a private data member  
//   // ...  
// };  
  
// class Class2  
// {  
// private: // We have explicitly defined that the variable is private  
//   int num;  
//   // ...  
// };
```

```
// =====
/*
Public#
This tag indicates that the members can be directly accessed by anything which is in the same scope as the class object.
```

Member functions are usually public as they provide the interface through which the application can communicate with our private members.

Public members can be declared using the public: heading.

```
/*
// class myClass
//{
// int num; // Private variable

// public: // Attributes in this list are public
// void setNum()
// {
//   // The private variable is directly accessible over here!
// }
//};

/*

```

Public members of a class can be accessed by a class object using the . operator. For example, if we have an object c of type myClass, we could access setNum() like this:

```
myClass c; // Object created
c.setNum(); // Can manipulate the value of num
c.num = 20; // This would cause an error since num is private
```

Protected#

The protected category is unique. The access level to the protected members lies somewhere between private and public. The primary use of the protected tag is to implement inheritance, which is the process of creating classes out of classes. Since this is a whole other topic for the future, we'll refrain from going into details right now.

Class Data Members:

We've learned that the data members contain all the information we store in a class. All the data members have to be defined at compile time.

It is a very safe practice to make our member variables private. Making them public could possibly crash the application because any external force could manipulate them in any way.

Data Types of Member Variables#

C++ gives us a lot of freedom in selecting the data type of a data member. We can choose any of the in-built types such as int, double etc. Arrays, vectors, and pointers can also be used. The object of our custom class could have a number of arrays and variables! pretty cool right?

```
*/
```

```

// =====
// #include<iostream>
// using namespace std;

// class myClass {
//   // All private members
//   string name;
//   int age;
//   string *address;
//   char grades [10]; // A student can have a maximum of 10 grades
// };
// =====
/*

```

We should always be careful with arrays inside classes. We should have appropriate checks in place to make sure our program never goes out of bounds due to an array. A good practice is to store the size of the array in a variable. This way, we'll always remember the maximum capacity.

Objects of Other Classes as Data Members#

This is another feature which adds to the flexibility of classes. We can use an object from our own class as a data member in our other classes. This is a slightly more advanced concept which we will study later on in the course.

All the data members we have seen till now are simply declared variables. They don't have any meaningful values. Be patient, we will learn how to assign values to our private data members through member functions.

```

*/
/*
Class Data Member Initialization

```

Class data members can be initialized in the class declaration. The following example shows how this can be done:

```

/*
// #include <iostream>
// #define PI 3.1416
// using namespace std;

// class Circle
// {
// private:
//   string mod = "debug";
//   // int arr[] = {1, 2, 3, 4, 5};
//   int radius = 1;

// public:
//   void setRadius(int r)
//   {
//     if (r >= 0)
//     {

```

```

//     radius = r;
// }
// }

// int getRadius()
// {
//     return radius;
// }

// double getArea()
// {
//     return PI * radius * radius;
// }

// int main()
//{
// Circle c;
// cout << "The area of a circle of radius " << c.getRadius() << " is " << c.getArea() << endl;
// return 0;
//}
//=====
/*

```

We define a class to represent a circle. Our intention is to have instances that are unit circles, unless otherwise specified. To achieve this, we have set radius to 1 on line 9.

Also note that we can initialize members of other data types like string as shown on line number 7. However, array members can't be initialized in classes. Similarly, if we uncomment line 8, we get a compilation error. This is because if we declare an array without providing the size, it results in a one-size array, arr[0] in C++ classes.

Another way to initialize class data members is using constructors, as we shall soon see.

Class Member Functions

The Purpose of Member Functions#

Member functions act as the interface between a program and the data members of a class in the program.

These functions can either alter the content of the data variables or use their values to perform a certain computation. All the useful member functions should be public, although, some functions which do not need to be accessed from the outside could be kept private.

Declaration and Definition#

Like all functions, member functions can either be defined straightaway, or they could be declared first and defined later.

Here's an example of a function being defined in the class:

1234567891011121314

```

*/
// =====
// class Rectangle {
//   int length;
//   int width;

// public:
// void setLength(int l){ // This function changes the value of length
//   length = l;
// }

// int area(){
//   return length * width; // Only the values of the data members are
//                         // accessed and used to calculate the area
// }
// };
// =====
/*

```

Scope Resolution Operator#

The scope resolution operator (::) allows us to simply declare the member functions in the class and define them elsewhere in the code. To use the scope resolution operator, we follow a certain syntax:

```

returnType className::function()
*/

// =====
// class Rectangle {
//   int length;
//   int width;

// public:

// // We only write the declaration here
// void setLength(int l);
// int area();
// };

// // Somewhere else in the code
// void Rectangle::setLength(int l){ // Using the scope resolution operator
//   length = l;
// }

// int Rectangle::area(){
//   return length * width;
// }
// =====
/*

```

The :: operator tells the compiler that the particular function belongs to the class preceding it.

Why is this useful? We'll find out the answer in the data hiding section.

Get and Set Functions#

These two types of functions are very popular in OOP. A get function retrieves the value of a particular data member, whereas a set function sets its value.

It is a common convention to write the name of the corresponding member variable with the get or set command. We have already seen an example of a set function in the code above. `setLength(int l)` is a perfect example.

Let's write get and set functions for the length and width in our Rectangle class:

```
/*
// =====

// class Rectangle {
// int length;
// int width;

// public:

// // get and set for length
// void setLength(int l);
// int getLength();

// // get and set for width
// void setWidth(int w);
// int getwidth();

// int area();
// };

// void Rectangle::setLength(int l){
// length = l;
// }

// int Rectangle::getLength(){
// return length;
// }

// void Rectangle::setWidth(int w){
// width = w;
// }

// int Rectangle::getwidth(){
// return width;
// }

// int Rectangle::area(){
// return length * width;
// }

// =====
/*
Overloading#
```

Member functions can be overloaded just like any other function. This means that multiple member functions can exist with the same name on the same scope, but must have different arguments.

What is a Constructor?

As the name suggests, the constructor is used to construct the object of a class. It is a special member function that outlines the steps that are performed when an instance of a class is created in the program.

A constructor's name must be exactly the same as the name of its class.

The constructor is a special function because it does not have a return type. We do not even need to write void as the return type. It is a good practice to declare/define it as the first member function.

So, let's study the different types of constructors and use them to create class objects.

Default Constructor

Default Constructor

The default constructor is the most basic form of a constructor. Think of it this way:

In a default constructor, we define the default values for the data members of the class. Hence, the constructor creates an object in which the data members are initialized to their default values.

This will make sense when we look at the code below. Here, we have a Date class, with its default constructor, and we'll create an object out of it in main():

*/

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

// class Date
//{
// int day;
// int month;
// int year;

// public:
// // Default constructor
// Date()
// {
// // We must define the default values for day, month, and year
// day = 0;
// month = 0;
// year = 0;
// }

// // A simple print function
// void printDate()
// {
// cout << "Date: " << day << "/" << month << "/" << year << endl;
```

```
// }  
// };  
  
// int main()  
//{  
// // Call the Date constructor to create its object;  
  
// Date d; // Object created with default values!  
// d.printDate();  
//}  
  
// ======  
/*
```

Notice that when we created a Date object in line 28, we don't treat the constructor as a function and write this:

d.Date()

We create the object just like we create an integer or string object. It's that easy!

The default constructor does not need to be explicitly defined. Even if we don't create it, the C++ compiler will call a default constructor and set data members to some junk values.

Parameterized Constructor#

The default constructor isn't all that impressive. Sure, we could use set functions to set the values for day, month and year ourselves, but this step can be avoided using a parameterized constructor.

In a parameterized constructor, we pass arguments to the constructor and set them as the values of our data members.

We are basically overriding the default constructor to accommodate our preferred values for the data members.

Let's try it out:

```
*/  
// ======  
  
// #include <iostream>  
// #include <string>  
// using namespace std;  
  
// class Date  
//{  
// int day;  
// int month;  
// int year;  
  
// public:  
// // Default constructor  
// Date()  
//{  
// // We must define the default values for day, month, and year
```

```

// day = 0;
// month = 0;
// year = 0;
//}

/// Parameterized constructor
// Date(int d, int m, int y)
// {
//   // The arguments are used as values
//   day = d;
//   month = m;
//   year = y;
// }

// // A simple print function
// void printDate()
// {
//   cout << "Date: " << day << "/" << month << "/" << year << endl;
// }
//};

// int main()
//{
// // Call the Date constructor to create its object;

// Date d(1, 8, 2018); // Object created with specified values!
// d.printDate();
// }

// =====
/*
This is much more convenient than the default constructor!

```

this Pointer#

The this pointer exists for every class. It points to the class object itself. We use the pointer when we have an argument which has the same name as a data member. `this->memberName` specifies that we are accessing the memberName variable of the particular class.

Note: Since this is a pointer, we use the `->` operator to access members instead of ..
*/

```

// =====
// #include <iostream>
// #include <string>
// using namespace std;

// class Date {
//   int day;
//   int month;
//   int year;

```

```

// public:
// // Default constructor
// Date(){
//   // We must define the default values for day, month, and year
//   day = 0;
//   month = 0;
//   year = 0;
// }

// // Parameterized constructor
// Date(int day, int month, int year){
//   // Using this pointer
//   this->day = day;
//   this->month = month;
//   this->year = year;
// }

// // A simple print function
// void printDate(){
//   cout << "Date: " << day << "/" << month << "/" << year << endl;
// }
//};

// int main(){
//   // Call the Date constructor to create its object;

//   Date d(1, 8, 2018); // Object created with specified values!
//   d.printDate();
// }

// =====
/*

```

More Kinds of Constructors

Copy constructors#

The copy constructor allows a class to create an object by copying an existing object.

They expect a constant reference to another instance of the class as their argument.

```

class Account{
public:
  Account(const Account& other);
};

```

All the values of other can be copied into the new object. Both objects will have the same values afterward.

Move constructors#

The move constructor allows the data of one object to be transferred completely to another object.

They are a more efficient alternative to the copy constructor since everything is being moved instead of copied.

They expect a non-constant rvalue-reference to an instance of the class as their argument.

```
class Account{ public:  
    Account(Account&& other);  
};
```

After the move operation, other is in a moved-from state. Accessing it will result in undefined behavior. To use it again, we would have to re-initialize it.

Explicit constructors#

The explicit constructor is used to avoid implicit calls to a class's constructor.

Consider the following Account constructor:

```
public:  
    Account(double b): balance(b){}
private:  
    double balance;  
    std::string cur;
};
```

An instance can be created like this:

```
Account acc = 100.0;
```

A double is being assigned to an Account object, but the compiler implicitly calls the constructor that takes a double as an argument. Hence, the operation works without any errors.

If the constructor had been made explicit, this statement would not have worked. For this, we would have to use the explicit keyword.

```
class Account{
public:  
    explicit Account(double b): balance(b){}
    Account (double b, std::string c): balance(b), cur(c){}
private:  
    double balance;  
    std::string cur;
};
```

```
Account account = 100.0; // ERROR: implicit conversion
```

```
Account account(100.0); // OK: explicit invocation
```

```
Account account = {100.0,"EUR"}; // OK: implicit conversion
```

Now, the assignment operator won't work as it did before, though it still works for Account(double b, std::string c) since it has not been made explicit.

Example#

Here's a complete example showing the use of the explicit keyword:

```
*/  
// ======  
// #include <iostream>  
// #include <string>  
  
// class Account  
// {
```

```
// public:  
// Account(double b) : balance(b) {}  
// Account(double b, std::string c) : balance(b), cur(c) {}  
  
// private:  
// double balance;  
// std::string cur;  
// };  
  
// void strange(Account a)  
// {  
// std::cout << "It works!" << std::endl;  
// }  
  
// int main()  
// {  
  
// Account account = 100.0; // No ERROR  
// Account account1(100.0);  
// Account account2 = {100.0, "EUR"};  
// strange(100.0); // No ERROR  
// strange(false);  
// }  
// ======  
// #include <iostream>  
// #include <string>  
  
// class Account{  
// public:  
// explicit Account(double b): balance(b){}  
// Account(double b, std::string c):balance(b), cur(c){}  
  
// private:  
// double balance;  
// std::string cur;  
// };  
  
// void strange(Account a){  
// std::cout << "It works!" << std::endl;  
// }  
  
// int main(){  
  
// // Account account = 100.0; // ERROR  
// Account account1(100.0);  
// Account account2 = {100.0, "EUR"};  
// // strange(100.0); // ERROR  
// // strange(false);  
  
// }  
// ======
```

```
/*
```

In the implicit approach, the assignment operations in lines 20 and 22 do not cause an error.

In the implicit approach, the function `strange` has an `Account` parameter, but passing it a double or bool implicitly calls the `Account` constructor, as done in lines 23 and 24.

When the `explicit` keyword is introduced in the code, implicit constructor calls are restricted.

Uncomment the lines to observe the error shown by the compiler in the explicit code tab.

```
*/
```

```
/*
```

Member Initializer Lists

Member initializer list allows us to initialize data members of an object without writing assignment statements in a constructor. Without further ado, here is an example:

```
*/
```

```
// =====
// #include <iostream>
// #define PI 3.1416
// using namespace std;

// class Sphere
// {
// private:
//   const double density;
//   double radius;

// public:
//   Sphere(double r) : radius(r), density(4.3)
//   {
//     // The following initialization wouldn't work, because density is a const
//     // density = 4.3;
//   }
//   double volume()
//   {
//     return 4 * PI * radius * radius * radius / 3;
//   }
//   double mass()
//   {
//     return density * volume();
//   }
// };
// int main()
//{
//   // your code goes here
//   Sphere s(3.2);
//   cout << "Volume: " << s.volume() << ", mass: " << s.mass();
//   return 0;
//}
// =====
/*
```

Our Sphere class stores the radius and density of a sphere. We have defined functions to obtain the volume and mass of the sphere. The constructor uses member initializer list on line 10

10

to initialize the member variables.

Inside main(), we declare a Sphere object with radius 3.2

3.2

on line 23

23

and then displayed its volume and mass on line 24

24

.

One advantage of member initializer list is that it makes things compact. One no longer has to use assignment statements in the constructor. Another important point is that any const class members can't be initialized inside a constructor and must be initialized using member initializer lists.

Constructor Delegation

What it is#

Constructor delegation is when one constructor of a class invokes another constructor of the same class. There, that's simple enough!

How it is implemented#

```
/*
// =====
// #include <iostream>
// using namespace std;

// class Collector
// {
// private:
// int size;
// int capacity;
// int *list;

// public:
// Collector() : Collector(0) {}
// Collector(int cap) : capacity(cap), size(0)
// {
// if (cap > 0)
// {
// list = new int[cap];
// }
// else
// capacity = 0;
// }

// bool append(int v)
// {
// if (size < capacity)
```

```

// {
//   list[size++] = v;
//   return true;
// }
// else
//   return false;
// }

// ~Collector()
// {
//   if (size > 0)
//   {
//     delete[] list;
//   }
// }
//};

// int main()
//{
// // A useless Collector object of 0 capacity
// Collector c1;
// // Another Collector object, this time with a capacity of 10
// Collector c2(10);

// for (int i = 0; i < 15; i++)
// {
//   cout << c2.append(i) << endl;
// }

// return 0;
//}
//=====
/*

```

We have defined a parameterized constructor on lines 12

12

- 18

18

. The default constructor invokes the parameterized constructor with an argument of 0

0

in the member initializer list on line 11

11

.

Why is it useful?

Once you have defined and tested a parameterized constructor, it is safer to implement other constructors by invoking the tested and proven constructor with appropriate parameter values. Let one constructor handle initialization and implement others using it. If a bug is identified, we have only one place to fix the bug at, instead of in every constructor.

Destructors

What is a Destructor?

A destructor is the opposite of a constructor. It is called when the object of a class is no longer in use. The object is destroyed and the memory it occupied is now free for future use.

C++ does not have transparent garbage collection like Java. Hence, in order to efficiently free memory, we must specify our own destructor for a class.

In this destructor, we can specify the additional operations which need to be performed when a class object is deleted.

A class destructor can be created in a similar way to the constructor, except that the destructor is preceded by the ~ keyword.

Explicit Garbage Collection#

A small degree of garbage collection can be easily achieved through smart pointers. A smart pointer, the shared_ptr in particular, keeps a reference count for the object it points. When the counter comes down to 0, the object is deleted.

It's time to make a destructor and see how it behaves.

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

// class Collector {
//   int * list;
//   int size;
//   int capacity;

// public:
//   // Default constructor
//   Collector(){
//     // We must define the default values for the data members
//     list = nullptr;
//     size = 0;
//     capacity = 0;
//   }

//   // Parameterized constructor
//   Collector(int cap){
//     // The arguments are used as values
//     capacity = cap;
//     size = 0;
//     list = new int[capacity];
//   }

//   bool append(int v){
//     if (size < capacity) {
//       list [ size++ ] = v;
//     }
//     return true;
//   }
// }
```

```

// }
// return false;
// }

// // A simple print function
// void dump(){
//   for(int i = 0 ; i < size ; i++) {
//     cout << list[i] << " ";
//   }
//   cout << endl;
// }

// ~Collector(){
//   cout << "Deleting the object " << endl;
//   if (capacity > 0)
//     delete[] list;
// }
// };

// int main(){
// Collector c(10);
// for (int i = 0 ; i < 15 ; i++){
//   cout << c.append(i) << endl;
// }
// }

```

```

// =====
/*

```

Our Collector class has a heap-allocated array list, an int representing its capacity and the current number of elements in it (size). The default constructor sets the list pointer to a safe default (nullptr), size and capacity to 0
0
. An append() function is defined to append data to a Collector object. It returns true if space is available in the array, or false otherwise.

The destructor frees the list using the delete keyword, after checking that some space was actually allocated to it.

In main(), we are creating a Collector object with enough space for 10 integers, then try to push 15
15
integers into it while displaying the success (1
1
) or failure (0
0
) status.

As we can see, the destructor is automatically called when the Collector object c in main() goes out of scope. The memory is freed up. What's interesting is that the cout statement we specified is also executed on the destructor call.

Destructors and Pointers#

In the case of object pointers, destructors are called when we issue the delete command, as in the following program on line 56

```
56
:
*/
// =====
// #include <iostream>
// #include <string>
// using namespace std;

// class Collector {
//   int * list;
//   int size;
//   int capacity;

//   public:
//   // Default constructor
//   Collector(){
//     // We must define the default values for the data members
//     list = nullptr;
//     size = 0;
//     capacity = 0;
//   }

//   // Parameterized constructor
//   Collector(int cap){
//     // The arguments are used as values
//     capacity = cap;
//     size = 0;
//     list = new int[capacity];
//   }

//   bool append(int v){
//     if (size < capacity) {
//       list [ size++ ] = v;
//       return true;
//     }
//     return false;
//   }

//   // A simple print function
//   void dump(){
//     for(int i = 0 ; i < size ; i++) {
//       cout << list[i] << " ";
//     }
//     cout << endl;
//   }

//   ~Collector(){
//     cout << "Deleting the object " << endl;
//     if (capacity > 0)
```

```
//     delete[] list;
// }
// };

// int main(){
// Collector *c;
// c = new Collector(10);
// for (int i = 0 ; i < 15 ; i++){
// cout << c->append(i) << endl;
// }
// delete c;
// cout << "Exiting program" << endl;
// }

// =====
/*
```

Destroying Objects is Important#

If we don't deallocate the memory for the objects which are not in use, we will end up with memory leaks and no space for our application to work long term.

Request and Suppress Methods

Special methods#

Since C++11, there has been a list of special methods that the compiler can generate implicitly if we have not defined them:

Default constructors and destructors.

Copy/move constructors and copy/move assignment operators.

new and delete operators for objects and C arrays of objects.

The default and delete keywords can be used to guide the creation or suppression of these special methods.

default can only be assigned to special methods that do not have any default arguments. Hence, it wouldn't work with something like an ordinary class method or a parameterized constructor.

Let's suppose we have a parameterized constructor for our Account class but no default constructor. The compiler can easily generate it for us. All we need to do is assign default to the default constructor.

```
...
Account() = default;
Account (double balance){this->balance = balance;}
```

...
The behavior of the compiler varies based on what special members the user has defined. We can find details in the diagram by Howard Hinnant below:

Request methods: default#

The compiler generates the request methods when it has the following characteristics:

public access rights and are not virtual.

The copy constructor and copy assignment operator get constant lvalue references.

The move constructor and move assignment operator get nonconstant rvalue references.

The methods are not declared explicit and possess no exception specifications.

Suppress methods: delete#

By using delete, we can define purely declaratively that an automatically generated method from the compiler is not available.

We can simply tell the compiler what to do without explaining how to do it.

By using delete in combination with default, we can define whether or not a class's objects:

can be copied.

can only be created on the stack.

can only be created on the heap.

Apart from objects and pointers, delete is also applicable to functions.

Request and Suppress Examples

Constructors and destructors using default#

*/

```
// =====
// #include <iostream>

// class SomeType
// {
// public:
// // state the compiler generated default constructor
// SomeType() = default;

// // constructor for int
// SomeType(int value)
// {
// std::cout << "SomeType(int) " << std::endl;
// };

// // explicit Copy Constructor
// explicit SomeType(const SomeType &) = default;

// // virtual destructor
// virtual ~SomeType() = default;
// };

// int main()
// {

// std::cout << std::endl;
```

```
// SomeType someType;
// SomeType someType2(2);
// SomeType someType3(someType2);

// std::cout << std::endl;
//}
//=====
/*
Explanation#
```

In this example, we can see how default can be used to get the default implementations of constructors and destructors from the compiler.

Since we have defined a parameterized constructor in line 10, the compiler will not automatically create a default constructor.

Hence, we have to define it ourselves using default, as done in line 7.

default can automatically handle the copy constructor in line 15 and the destructor in line 18.

The explicit keyword is used in the copy constructor to avoid implicit conversions during copying.

We need the virtual destructor in case there is a derived class inheriting SomeType.

Restricting operations using delete#

The following code generates an error and the reasons why are mentioned below:

```
/*
//=====
// #include <iostream>

// class NonCopyableClass{
// public:

// // state the compiler generated default constructor
// NonCopyableClass()= default;

// // disallow copying
// NonCopyableClass& operator = (const NonCopyableClass&) = delete;
// NonCopyableClass (const NonCopyableClass&) = delete;

// // allow moving
// NonCopyableClass& operator = (NonCopyableClass&&) = default;
// NonCopyableClass (NonCopyableClass&&) = default;
// };

// class TypeOnStack {
// public:
// void * operator new(std::size_t)= delete;
// };

// class TypeOnHeap{
```

```
// public:  
// ~TypeOnHeap()= delete;  
// };  
  
// void onlyDouble(double){}  
// template <typename T>  
// void onlyDouble(T)=delete;  
  
// int main(){  
  
// NonCopyableClass nonCopyableClass;  
  
// TypeOnStack typeOnStack;  
  
// TypeOnHeap * typeOnHeap = new TypeOnHeap;  
  
// onlyDouble(3.14);  
  
// // force the compiler errors  
  
// NonCopyableClass nonCopyableClass2(nonCopyableClass); // cannot copy  
  
// TypeOnStack * typeOnHeap2 = new TypeOnStack; // cannot create on heap  
  
// TypeOnHeap typeOnStack2; // cannot create on stack  
  
// onlyDouble(2011); // int argument not accepted  
  
// }  
// ======  
/*
```

Explanation#

Here, we are prohibiting certain operations by using delete.

In lines 10 to 15, we have disabled copy operations for NonCopyableClass but enabled move operation. By assigning delete, we tell the compiler that the operation will not be accepted. Hence, an error will be thrown.

With delete, we can also prevent objects from being created on the heap or stack.

In the TypeOnStack class, we assign delete to the operator new on line 21. This means that an object of this class cannot occupy space on the heap.

Conversely, the TypeOnHeap class is not allowed to make objects on the stack. We simply define a destructor that calls delete in line 26.

Lastly, there is the onlyDouble() function. We have explicitly told the compiler to accept only double arguments.

For any other arguments, onlyDouble() will generate an error.

Lines 45 to 51 show various examples of the operations that have been restricted by delete. None of them will work.

```
*/  
/*  
Friend Functions  
We'll take a look at a special category of functions called friends.  
  
The private data members of a class are only accessible through the functions present in that class. Nothing from outside can manipulate the class object without using its functions.  
  
What if we need to access class variables in a function which is not a part of the class? That function would have to become a friend of the class.  
  
A friend function is an independent function which has access to the variables and methods of its befriended class.  
  
To create a friend function for a class, it must be declared in the class along with the friend keyword.  
  
Let's create a Ball class to explain this better:  
*/  
// ======  
// #include <iostream>  
// #include <string>  
  
// using namespace std;  
  
// class Ball  
// {  
//   double radius;  
//   string color;  
  
// public:  
//   Ball()  
//   {  
//     radius = 0;  
//     color = "";  
//   }  
  
//   Ball(double r, string c)  
//   {  
//     radius = r;  
//     color = c;  
//   }  
  
//   void printVolume();  
//   void printRadius();  
  
// // The friend keyword specifies that this is a friend function  
// friend void setRadius(Ball &b, double r);
```

```

//};

/// This is a member function that calculates the volume.
// void Ball::printVolume()
//{
// cout << (4 / 3) * 3.142 * radius * radius * radius << endl;
//}

// void Ball::printRadius()
//{
// cout << radius << endl;
//}

/// The implementation of our friend function
// void setRadius(Ball &b, double r)
//{
// b.radius = r;
//}

// int main()
//{
// Ball b(30, "green");
// cout << "Radius: ";
// b.printRadius();
// setRadius(b, 60);
// cout << "New radius: ";
// b.printRadius();
// cout << "Volume: ";
// b.printVolume();
//}
//=====================================================================
/*

```

In line 25, we can see that the Ball object is being passed by reference to the friend function. This is a crucial step in the functionality of the friend. If the object is not passed by reference, the changes made in the friend function will not work outside its scope. Basically, our object will not be altered.

The setRadius() function is completely independent of the Ball class, yet it has access to all the private variables. This is the beauty of the friend keyword.

```

*/
/*
```

Structs and Unions

Structs#

Introduction#

A Structure in C++ is a group of data elements grouped together under one name. These data elements, known as members, can be of different types and sizes. It is a user-defined data type that allows us to combine data items of different kinds.

The scope of Struct#

Structs are almost identical to classes. The default access specifier for a struct is public instead of private.

The default inheritance specifier is public instead of private.

Example#

Let's consider an example of a Person struct which contains age, size, weight, and name as members. A struct always ends with a ;.

```
struct Person{  
    int age;  
    int size;  
    int weight;  
    std::string name;  
};
```

Structs should be used instead of classes if the data type is a simple data holder.

Unions#

Introduction#

A union is a special data type where all members start at the same address. A union can only hold one type at a time, therefore, we can save memory. A tagged union is a union that keeps track of its types. By using union, we are actually pointing to the same memory for the different data types used.

Rules#

Unions are special class types.

Only one member can exist at any one point in time.

They only need as much space as the biggest member requires, which saves memory.

The access specifier is public by default.

They cannot have virtual methods like with Inheritance.

They cannot have references.

They cannot be inherited nor inherited from.

Example#

Let's take a look at an example of the union:

```
/*  
// ======  
// #include <iostream>
```

```
// union Value  
// {  
//     int i;  
//     double d;  
// }  
  
// int main()  
// {  
//     Value v = {123};      // now v holds an int  
//     std::cout << v.i << '\n'; // write 123  
//     v.d = 987.654;       // now v holds a double  
//     std::cout << v.d << '\n'; // write 987.654  
// }  
// ======  
/*
```

What is Data Hiding?

In this section, we will learn two concepts which help us create an efficient class in C++

We'll cover the following

Real Life Example

Components of Data Hiding

Data hiding is an essential process in the OOP cycle.

In layman's terms, data hiding refers to the concept of hiding the inner workings of a class and simply providing an interface through which the outside world can interact with the class without knowing what's going on inside.

Our goal is to make the interaction between different classes as simple as possible. This can only be achieved if the interfaces of the classes are simple. One class does not need to know anything about the underlying algorithms of another class. However, the two can still communicate.

Real Life Example#

Let's apply this to a real-world scenario. Take the doctor-patient model. In case of an illness, the patient consults the doctor, after which he or she is prescribed the appropriate medicine.

The patient only knows the process of going to the doctor. The logic and reasoning behind the doctor's prescription of a certain medicine are unknown to the patient. A patient will not understand the medical details the doctor uses to reach his/her decision on the treatment.

This is a classic example of the patient class interacting with the doctor class without knowing the inner workings of the doctor class.

The transaction shown above seems fairly simple. Data hiding is useful because it can apply the same simplicity to transactions between objects of different classes.

Components of Data Hiding#

Data hiding can be divided into two primary components:

Encapsulation

Abstraction

When used together, they allow us to make efficient classes for further use in our application.

*/

/*

Encapsulation

This lesson shows us how to implement the first component of data hiding: encapsulation.

We'll cover the following

A Real Life Example

Advantages of Encapsulation

A Real Life Example#

For the sake of explanation, we'll start off by creating a simple movie class which contains three members:

```
class Movie{  
    string title;  
    int year;  
    string genre;
```

```

public:
Movie(){
    title = "";
    year = -1;
    genre = "";
}

Movie(string t, int y, string g){
    title = t;
    year = y;
    genre = g;
}
;

```

There must be a way to interact with the title, year and genre variables. They hold all the information about a movie, but how do we access or modify them?

We could create a getTitle() method which would return the title to us. Similarly, the other two members could also have corresponding get functions.

By observing the emerging pattern, we can make a definitive conclusion. These functions should be part of the class of itself! Let's try it out.

```

*/
// =====
// #include <iostream>
// #include <string>
// using namespace std;

// class Movie{
//     string title;
//     int year;
//     string genre;

//     public:
//     Movie(){
//         title = "";
//         year = -1;
//         genre = "";
//     }

//     Movie(string t, int y, string g){
//         title = t;
//         year = y;
//         genre = g;
//     }

//     string getTitle(){
//         return title;
//     }

//     void setTitle(string t){

```

```

// title = t;
// }

// int getYear(){
//   return year;
// }
// void setYear(int y){
//   year = y;
// }

// string getGenre(){
//   return genre;
// }
// void setGenre(string g){
//   genre = g;
// }

// void printDetails(){
//   cout << "Title: " << title << endl;
//   cout << "Year: " << year << endl;
//   cout << "Genre: " << genre << endl;
// }
//};

// int main() {
// Movie m("The Lion King", 1994, "Adventure");
// m.printDetails();

// cout << "---" << endl;
// m.setTitle("Forrest Gump");
// cout << "New title: " << m.getTitle() << endl;
// }

// =====
/*

```

We have now provided an interface of public methods to interact with the Movie class. Our private variables cannot be accessed directly from the outside, but we have provided read and write functions which allow access those variables.

This, in essence, is data encapsulation.

```

*/
/*
Advantages of Encapsulation#
Classes are easier to change and maintain.
We can specify which data member we want to keep hidden or accessible.
We decide which variables have read/write privileges (increases flexibility).
In the next lesson, we'll discuss the second component of data hiding: abstraction.
```

Abstraction in Classes

This lesson will define what abstraction is and how it relates to data hiding.

Abstraction is the second component of data hiding in OOP. It is an extension of encapsulation and further simplifies the structure of programs.

What is Abstraction?#

Abstraction focuses on revealing only the relevant parts of the application while keeping the inner implementation hidden.

Users will perform actions and expect the application to respond accordingly. They will not be concerned with how the response is generated. Only the final outcome is relevant.

There are countless real life examples which follow the rules of abstraction. Take the Volume button on a television remote. With a click of a button, we request the T.V. to increase its volume. Let's say the button calls the volumeUp() function. The T.V. responds by producing a sound larger than before. We are oblivious to the fact how inner circuitry of the T.V. implements this, but we know the exposed function needed to interact with the T.V.'s volume.

svg viewer

Another instance of abstraction is our daily use of vehicles. To our general knowledge, the race peddle tells the car to consume fuel and increase its speed. We do not need to understand the mechanical process happening inside.

Class Abstraction#

So, let's put all this theory into practice. In the code below, we have a basic class for a circle:

```
class Circle{  
    double radius;  
    double pi;  
};
```

It has two variables, radius and pi. Now let's add the constructor and functions for the area and perimeter:

```
*/  
// ======  
// #include <iostream>  
// using namespace std;  
  
// class Circle  
// {  
//     double radius;  
//     double pi;  
  
// public:  
//     Circle()  
//     {  
//         radius = 0;  
//         pi = 3.142;  
//     }  
//     Circle(double r)  
//     {
```

```

// radius = r;
// pi = 3.142;
// }

// double area()
// {
//   return pi * radius * radius;
// }

// double perimeter()
// {
//   return 2 * pi * radius;
// }
//};

// int main()
//{
// Circle c(5);
// cout << "Area: " << c.area() << endl;
// cout << "Perimeter: " << c.perimeter() << endl;
//}
// =====
/*

```

As you can see, we only need to define the radius of the circle in the constructor. After that, the area() and perimeter() functions are available to us. This interface is part of encapsulation. However, the interesting part is what happens next.

All we have to do is call the functions and voila, we get the area and perimeter as we desire. Users would not know what computations were performed inside the function. Even the pi constant will be hidden to them. Only the input and the output matter. This is the process of abstraction using classes.

```

*/
/*
Abstraction in Header Files
Learn about implementing abstraction is creating header files. Find out more below!
```

We'll cover the following

Creating Header Files

When our goal is to hide the unnecessary details from the users, we can divide the code into different files. This is where header files come into play.

Creating Header Files#

Let's take a look at the Circle class below.

```

*/
// =====
// #include <iostream>
// using namespace std;

// class Circle
// {
//   double radius;
```

```

// double pi;

// public:
// Circle()
// {
//   radius = 0;
//   pi = 3.142;
// }
// Circle(double r)
// {
//   radius = r;
//   pi = 3.142;
// }

// double area()
// {
//   return pi * radius * radius;
// }

// double perimeter()
// {
//   return 2 * pi * radius;
// }
//};

// int main()
//{
// Circle c(5);
// cout << "Area: " << c.area() << endl;
// cout << "Perimeter: " << c.perimeter() << endl;
//}
// =====
/*

```

To hide our class, we will follow a few steps. The first step is to create a header file. This file will only contain the declaration of the class and its members. A header file always has the .h extension:

```

*/
/*
// Declare all the members of the class here.
class Circle{
  double radius;
  double pi;

  public:
  Circle ();
  Circle(double r);
  double area();
  double perimeter();
};

/*
// =====
// #include <iostream>

```

```

// using namespace std;

// class Circle{
//   double radius;
//   double pi;

// public:
//   Circle (){
//     radius = 0;
//     pi = 3.142;
//   }
//   Circle(double r){
//     radius = r;
//     pi = 3.142;
//   }

//   double area(){
//     return pi * radius * radius;
//   }

//   double perimeter(){
//     return 2 * pi * radius;
//   }
// };

// int main() {
//   Circle c(5);
//   cout << "Area: " << c.area() << endl;
//   cout << "Perimeter: " << c.perimeter() << endl;
// }
// =====
/*

```

As you can see, the header file isn't very useful if the complete implementation is still visible in our main file. Therefore, the second step is to move all the implementation to a separate file. Let's call this Circle.cpp.

In this file, we must include the header file. The include command should already be familiar to you. We use it all the time to include libraries like iostream or vector. We can include header files in the same way!

Since we're implementing all the methods of our Circle class in Circle.cpp, we must mention the name of the class along with the scope resolution operator (::). Let's do this now:

circle.h

```

// Declare all the members of the class here.
class Circle{
  double radius;
  double pi;

public:
  Circle ();
  Circle(double r);

```

```

double area();
double perimeter();
};

*/
/*

circle.cpp

#include "Circle.h"

Circle::Circle(){
    radius = 0;
    pi = 3.142;
}

Circle::Circle(double r){
    radius = r;
    pi = 3.142;
}

double Circle::area(){
    return pi * radius * radius;
}

double Circle::perimeter(){
    return 2 * pi * radius;
}

*/
// =====
// #include <iostream>
// #include "./Circle.h"

// using namespace std;

// int main() {
//     Circle c(5);
//     cout << "Area: " << c.area() << endl;
//     cout << "Perimeter: " << c.perimeter() << endl;
// }
// =====
/*

```

In the header file, we have two commands:

```

#ifndef CIRCLE_H
#define CIRCLE_H

```

These commands tell the compiler that this header file can be used in multiple places. The `#ifndef` command ends with `#endif`.

What we're seeing now is complete abstraction. None of the implementation is visible to users. If they need to know what methods are available in the Circle class, they can simply refer to the header file.

This ends our discussion on data hiding. Combining encapsulation and abstraction gives us a simple and reusable interface for our program.

```
*/  
/*
```

What is Inheritance?

In this lesson, we'll be learning about the core concept of the object-oriented paradigm, i.e., Inheritance and why there is a need for it?

We'll cover the following

Why do We Need Inheritance?

Vehicle Class

Implementation of Vehicle Class

Car Class

Implementation of Car Class

Ship Class

Implementation of Ship Class

Why do We Need Inheritance?#

In the classes chapter, we've covered the HAS-A relationship. We know a class HAS-A data members and member functions. Now, we want the data members, and member functions of the class are accessible from other classes. So, the capability of a class to derive properties and characteristics from another class is called Inheritance. In inheritance, we have IS-A relationship between classes e.g a car is a vehicle and a ship is a vehicle.

Let's take the example of Vehicle here.

Vehicle Class#

In a Vehicle class, we have many data members like Make, Color, Year and Model. A Vehicle HAS-A Model, Year, Color and Make.

Implementation of Vehicle Class#

Let's look at the implementation of Vehicle class:

```
*/
```

```
// ======  
// class Vehicle{  
//   protected:  
//   string Make;  
//   string Color;  
//   int Year;  
//   string Model;  
  
//   public:  
//   Vehicle(){  
//     Make = "";  
//     Color = "";  
//     Year = 0;  
//     Model = "";
```

```

// }

// Vehicle(string mk, string col, int yr, string mdl){
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
// }

// void print_details(){
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
//   cout << "Model: " << Model << endl;
// }
// };

// int main(){
//   Vehicle v("Ford Australia", "Yellow", 2008, "Falcon");
//   v.print_details();
// }
// =====
/*

```

The following illustration depicts the structure of the Vehicle class:

These attributes are also attributes of all Cars, Ships and Airplanes but every type of vehicle has some attributes that are different from other types of vehicles, as we will see in detail.

Car Class#

The implementation of a Car class needs the same data members and member functions of Vehicle class but we have to include them in the Car class. Cars do have a trunk and every trunk has a capacity to store things upto some limit.

Implementation of Car Class#

Let's look at the implementation of the Car class:

```

*/
// =====

// #include<iostream>
// #include<string>
// using namespace std;
// class Car
//{
//   string Make;
//   string Color;
//   int Year;
//   string Model;
//   string trunk_size;

// public:

```

```

// Car()
// {
//   Make = "";
//   Color = "";
//   Year = 0;
//   Model = "";
//   trunk_size = "";
// }

// Car(string mk, string col, int yr, string mdl, string ts)
// {
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
//   trunk_size = ts;
// }

// void print_details()
// {
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
//   cout << "Model: " << Model << endl;
//   cout << "Trunk size: " << trunk_size << endl;
// }
//};

// int main()
//{
//   Car car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
//   car.print_details();
// }
// =====
/*
Ship Class#

```

The implementation of a Ship class needs the same data members and member functions of Vehicle class but we have to include them in the Ship class. Ships do have anchors and they vary in numbers.

Implementation of Ship Class#

Let's look at the implementation of the Ship class:

```

// =====

// #include<iostream>
// #include<string>
// using namespace std;
// class Ship{
//   string Make;
//   string Color;

```

```

// int Year;
// string Model;
// int Number_of_Anchor;

// public:
// Ship(){
//   Make = "";
//   Color = "";
//   Year = 0;
//   Model = "";
//   Number_of_Anchor = 0;
// }

// Ship(string mk, string col, int yr, string mdl, int na){
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
//   Number_of_Anchor = na;
// }

// void print_details(){
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
//   cout << "Model: " << Model << endl;
//   cout << "Number of Anchors: " << Number_of_Anchor << endl;
// }
//};

// int main(){
//   Ship ship("Harland and Wolff, Belfast", "Black and white",
//             1912, "RMS Titanic", 3);
//   ship.print_details();
// }
// =====
/*
Base Class and Derived Class
In this lesson, we'll be learning about how a base class attributes are available to the derived classes and how to define base and a derived class.

```

We'll cover the following

Vehicle as a Base Class

Derived Classes

Modes of Inheritance

Public Inheritance

Explanation

In the last lesson, we have seen that Vehicle class attributes are shared by the other two classes(Car and Ship).

Vehicle as a Base Class#

We can consider the Vehicle class as a base class as it has common attributes.

Derived Classes#

Cars and Ships are considered as derived classes as they're inheriting properties from vehicle class.

Modes of Inheritance#

There are three modes of class inheritance: public, private and protected. The basic syntax for inheritance is given below:

```
class derivedClassName : modeOfInheritance baseClassName
```

We use the keyword public to implement public inheritance.

Now, the class Car has access to the public members of a base class Vehicle and the protected data is inherited as protected data, and the private data is not inherited, but it can be accessed directly by the public member functions of the class.

Public Inheritance#

We are updating our Car and Ship class so that Make, Color, Year, Model and the function void print_details() can be inherited from base class Vehicle. So, we have removed these variables and function from the derived classes. The following example shows the classes Car and Ship that inherits publicly from the base class Vehicle.

```
/*
// =====
// class Vehicle
//{
// protected:
//   string Make;
//   string Color;
//   int Year;
//   string Model;

// public:
//   Vehicle()
//   {
//     Make = "";
//     Color = "";
//     Year = 0;
//     Model = "";
//   }

//   Vehicle(string mk, string col, int yr, string mdl)
//   {
//     Make = mk;
//     Color = col;
//     Year = yr;
//     Model = mdl;
//   }
}
```

```
// void print_details()
// {
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
//   cout << "Model: " << Model << endl;
// }
//};

// class Cars : public Vehicle
//{
// string trunk_size;

// public:
// Cars()
// {
//   trunk_size = "";
// }

// Cars(string mk, string col, int yr, string mdl, string ts)
//   : Vehicle(mk, col, yr, mdl)
// {
//   trunk_size = ts;
// }

// void car_details()
// {
//   print_details();
//   cout << "Trunk size: " << trunk_size << endl;
// }
//};

// class Ships : public Vehicle
//{
// int Number_of_Anchor;

// public:
// Ships()
// {
//   Number_of_Anchor = 0;
// }

// Ships(string mk, string col, int yr, string mdl, int na)
//   : Vehicle(mk, col, yr, mdl)
// {
//   Number_of_Anchor = na;
// }

// void Ship_details()
// {
//   print_details();
// }
```

```

// cout << "Number of Anchors: " << Number_of_Anchors << endl;
// }
// };

// int main()
//{
// Cars car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
// car.car_details();

// cout << endl;

// Ships ship("Harland and Wolff, Belfast", "Black and white",
//           1912, "RMS Titanic", 3);
// ship.Ship_details();
//}

// =====
/*

```

Explanation#

Now the Ship and Car classes have access to public member functions of the base class Vehicle as shown in the above illustration. Protected and public data members are accessible to derived classes.

Now that we have learned about the base and derived classes. So, let's move to the next lesson in which we'll learn about the base class constructors and destructors.

```

*/
/*

```

Base Class Constructor and Destructor

In this lesson, we'll learn how constructors and destructors are called in derived and base classes during inheritance.

We'll cover the following

Base Class Constructor

Base Class Destructor

Base Class Constructor#

When we make an instance of the Derived class without parameters it will first call the default constructor of the Base class and then the Derived class. In the same way, when we call the parameterized constructor of the derived class, it will first call the parameterized constructor of the Base class and then Derived class.

The following code explains how this is done:

```

*/
// =====
// #include <iostream>
// using namespace std;

// // Base class
// class Base {

// public:
// Base(){
// cout << "Base class default constructor!" << endl;
// }

```

```

// // Base class's parameterised constructor
// Base(float i) {
//     cout << "Base class parameterized constructor" << endl;
// }
//;

/// // Derived class
// class Derived : public Base {
// public:
// Derived(){
//     cout << "Derived class default constructor!" << endl;
// }

// // Derived class's parameterised constructor
// Derived(float num): Base(num){
//     cout << "Derived class parameterized constructor" << endl;
// }
//;

/// // main function
// int main() {
// // creating object of Derived Class
// Derived obj;
// cout << endl;
// Derived obj1(10.2);
// }
// =====
/*
Base Class Destructor#
When we make an instance of the Derived class it will first call the destructor of the Derived class and then the Base class.

```

The following code explains how this is done:

```

*/
// =====
// #include <iostream>
// using namespace std;

/// // Base class
// class Base
// {

// public:
// ~Base()
// {
//     cout << endl
//     << "Base class Destructor!";
// }
//;

/// // Derived class

```

```

// class Derived : public Base
//{
// public:
// ~Derived()
// {
//   cout << endl
//   << "Derived class Destructor!";
// }
//};

/// main function
// int main()
//{
// // creating object of Derived Class
// Derived obj;
//}
//=====================================================================
/*
Function Overriding
Learn about overriding inherited functions

```

We'll cover the following

Overriding inherited functions

Example

An important observation

Overriding inherited functions#

When a derived class inherits from a base class, it may choose to change some of the inherited functionality. This is called function overriding, since the derived class is overriding the functionality of the base class.

Example#

Here's a simple example to demonstrate this:

*/

```

//=====================================================================
// #include <iostream>
// using namespace std;

// class Employee
//{
// protected:
// string name;
// int ID;
// int reportsTo;

// public:
// Employee(string name, int ID, int boss) : name(name), ID(ID), reportsTo(boss) {}
// string getName() { return name; }
// int getID() { return ID; }
// int getBoss() { return reportsTo; }
// void display()

```

```

// {
//   cout << ID << " " << name << " reports to " << reportsTo << endl;
// }

// void display(string salutation)
// {
//   cout << salutation << " ";
//   display();
// }
//};

// class Manager : public Employee
//{
// protected:
// string teamName;

// public:
// Manager(string name, int ID, int boss, string teamName) : Employee(name, ID, boss),
teamName(teamName) {}
// void display()
// {
//   Employee::display();
//   cout << " Heads the team " << teamName << endl;
// }
//};

// int main()
//{
// Employee worker("John Smith", 10, 2);
// Manager ceo("Jack Hobbs", 0, 0, "Eats R Us");
// Manager cto("Elizabeth Shaw", 2, 0, "IT");
// worker.display("Mr");
// ceo.display();
// cto.display();
// // ceo.display("Mr")
// return 0;
//}
//=====
/*

```

We have defined a base class to represent an Employee. Since a Manager is also an Employee, we have defined a class Manager that inherits from Employee. We have kept information that is common to all employees in the base class. A Manager also heads a team, so the team name is additionally placed in that class.

In the base class, we have defined two overloaded functions named display(). One of these takes no argument and displays all the members to the screen. The other overload accepts a string salutation as an argument and prepends it to the Employee's name.

In the Manager class, we want the display() method to show the team name as well, so we have overridden the base class functionality for display(). We want to display all the essential information from the base class

as well as the team name, here. Instead of re-inventing the wheel, we have just invoked the base class `display()` function by using the base class name (`Employee`) with the scope resolution operator (`::`) on line 31.

In `main()`, we have instantiated a worker, a ceo and a cto using parameterized constructors and then called their `display()` functions.

An important observation#

Note that calling `display("Mr")` on the worker object works fine. However, if you uncomment line number 45 and run the program again, it fails to compile. The reason is:

Overriding a function in a derived class hides all the overloads of the same function from the base class.

The overloaded `display(string salutation)` is not available to Manager class instances.

In the next lesson, we'll be learning about the public, protected and private inheritance.

```
*/  
/*
```

Modes of Inheritance

In this lesson, we'll learn about how Public, Private and Protected inheritance is done in C++.

We'll cover the following

private Mode of Inheritance

protected Mode of Inheritance

public Mode of Inheritance

Modes of Inheritance in Base Class

You are already familiar with Access Modifiers from the Classes chapter. By using these specifiers, we limit the access of the data members and member functions to the other classes and main.

private Mode of Inheritance#

By using private inheritance, the private data members and member functions of the base class are inaccessible in the derived class and in main. Protected and Public members of the base class are accessible to the derived class but not in main and become private members of the derived class.

Note: any classes inheriting from the above-derived class remain unaware (that is, do not have access) of the base class.

Let's look at the implementation using private inheritance:

```
*/  
// ======  
// class Vehicle  
// {  
  
//   string Make;  
//   string Color;  
//   int Year;  
  
// protected:  
//   string Model;  
  
// public:
```

```
// Vehicle()
// {
//   Make = "";
//   Color = "";
//   Year = 0;
//   Model = "";
// }

// Vehicle(string mk, string col, int yr, string mdl)
// {
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
// }

// void print_details()
// {
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
// }
//};

// class Car : private Vehicle
//{
// string trunk_size;

// public:
// Car()
// {
//   trunk_size = "";
// }

// Car(string mk, string col, int yr, string mdl, string ts)
//   : Vehicle(mk, col, yr, mdl)
// {
//   trunk_size = ts;
// }

// void car_details()
// {
//   print_details();
//   cout << "Trunk size: " << trunk_size << endl;
//   cout << "Model: " << Model << endl; // Model is protected and
//   // is accessible in derived class
// }
//};

// int main()
//{
```

```

// Car car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
// // car.Year = 2000; // this will give error as Year is private
// // car.Model = "Accord"; // this will give error as Model is protected

// car.car_details();
// // car.print_details(); // public functions of base class are inaccessible in main
// }
// =====
/*

```

protected Mode of Inheritance#

By using protected inheritance, the private members of the base class are inaccessible in the derived class and in main. Protected and Public members of the base class are accessible to the derived class but not in main and become protected members of the derived class.

Let's take an example of protected inheritance:

```

*/
// =====
// class Vehicle{

// string Make;
// string Color;
// int Year;

// protected:
// string Model;

// public:
// Vehicle(){
//   Make = "";
//   Color = "";
//   Year = 0;
//   Model = "";
// }

// Vehicle(string mk, string col, int yr, string mdl){
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
// }

// void print_details(){
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
// }
// };

// class Car: protected Vehicle{
// string trunk_size;

```

```

// public:
// Car(){}
//   trunk_size = "";
// }

// Car(string mk, string col, int yr, string mdl, string ts)
//   :Vehicle(mk, col, yr, mdl){
//   trunk_size = ts;
// }

// void car_details(){
//   print_details();
//   cout << "Trunk size: " << trunk_size << endl;
//   cout << "Model: " << Model << endl; // Model is protected and
//   // is accessible in derived class
// }
//};

// int main(){
//   Car car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
//   // car.Year = 2000; // this will give error as Year is private
//   // car.Model = "Accord"; // this will give error as Model is protected

//   car.car_details();
//   //car.print_details(); // public functions of base class are inaccessible in main
// }
// =====
/*
public Mode of Inheritance#
By using public inheritance, the private members of the base class are inaccessible in the derived class and in main. Protected members of the base class are accessible to the derived class but not in main. Public members of the base class are accessible to the derived class and in main.

```

Let's look at the implementation using public inheritance:

```

*/
// =====
// class Vehicle{

//   string Make;
//   string Color;
//   int Year;

//   protected:
//   string Model;

//   public:
//   Vehicle(){
//     Make = "";
//     Color = "";
//     Year = 0;
//     Model = "";
//   }
// };

```

```

// }

// Vehicle(string mk, string col, int yr, string mdl){
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
// }

// void print_details(){
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
// }
//};

// class Car: public Vehicle{
//   string trunk_size;

//   public:
//   Car(){
//     trunk_size = "";
//   }

//   Car(string mk, string col, int yr, string mdl, string ts)
//     :Vehicle(mk, col, yr, mdl){
//     trunk_size = ts;
//   }

//   void car_details(){
//     cout << "Trunk size: " << trunk_size << endl;
//     cout << "Model: " << Model << endl; // Model is protected and
//     // is accessible in derived class
//   }
// };

// int main(){
//   Car car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
//   // car.Year = 2000; // this will give error as Year is private
//   // car.Model = "Accord"; // this will give error as Model is protected

//   car.car_details();
//   car.print_details(); // public functions of base class are accessible in main
// }

// =====
/*
Modes of Inheritance in Base Class#
The given table depicts the access of members of our base class when we use specific modifiers and its
behavior.
*/
/*

```

Types of Inheritance

In this lesson, we'll learn about the types of inheritance which includes multiple inheritance and multilevel inheritance.

We'll cover the following

Multiple Inheritance

Example

Implementation

Multiple Inheritance#

We can inherit the base class attributes to the derived class if we want derived class to have access data members and member functions of the base class. But to inherit multiple classes data members and member functions to the derived, the concept of multiple inheritance comes in. We can inherit multiple classes as base classes separated by ,

```
class Derived : public Base1 , public Base2 , ...
```

Example#

Let's take the example of Vehicle and Car classes which acts as the base classes of the Honda class:

```
/*
// =====
// class Vehicle{
// protected:
// string Make;
// string Color;
// int Year;
// string Model;

// public:
// Vehicle(){
//   Make = "";
//   Color = "";
//   Year = 0;
//   Model = "";
// }

// Vehicle(string mk, string col, int yr, string mdl){
//   Make = mk;
//   Color = col;
//   Year = yr;
//   Model = mdl;
// }

// void print_details(){
//   cout << "Manufacturer: " << Make << endl;
//   cout << "Color: " << Color << endl;
//   cout << "Year: " << Year << endl;
//   cout << "Model: " << Model << endl;
// }
//};
```

```

// class Car{
// string trunk_size;

// public:
// Car(){
//   trunk_size = "";
// }

// Car(string ts){
//   trunk_size = ts;
// }

// void car_details(){
//   cout << "Trunk size: " << trunk_size << endl;
// }
//};

// class Honda: public Vehicle, public Car{
// int top_speed;

// public:
// Honda(){
//   top_speed = 0;
// }

// Honda(string mk, string col, int yr, string mdl, string na, int ts)
// :Vehicle(mk, col, yr, mdl), Car(na){
//   top_speed = ts;
// }

// void Honda_details(){
//   print_details();
//   car_details();
//   cout << "Top speed of the car: " << top_speed << endl;
// }
//};

// int main(){
// Honda car("Honda", "Black", 2006, "Accord", "14.7 cubic feet", 260);
// car.Honda_details();
// }
//=====
/*
Now, the Honda class object has access to all member functions of Car and Vehicle classes as they're now
base classes of Honda class. The highlighted lines in the code indicate how multiple inheritance is achieved.
*/
/*
Multi-level Inheritance
We'll cover the following

```

Example

Implementation

If we want to inherit data members and member functions of the base class which is already inherited from another class, the concept of multilevel inheritance comes in. This contains a more hierarchical approach.

```
class parent
class child : public parent
class grandChild : public child
```

Example#

Let's take the example of Vehicle class which acts as a parent to Car class. Now Car class acts as a parent to Honda class.

Implementation#

Implementation of the Honda class is given below:

vehicle.h

```
#include <iostream>
#include <string>
using namespace std;
class Vehicle {
protected:
    string Make;
    string Color;
    int Year;
    string Model;

public:
    Vehicle(){
        Make = "";
        Color = "";
        Year = 0;
        Model = "";
    }

    Vehicle(string mk, string col, int yr, string mdl){
        Make = mk;
        Color = col;
        Year = yr;
        Model = mdl;
    }

    void print_details(){
        cout << "Manufacturer: " << Make << endl;
        cout << "Color: " << Color << endl;
        cout << "Year: " << Year << endl;
        cout << "Model: " << Model << endl;
    }
}
```

```

};

class Car: public Vehicle{
    string trunk_size;

public:
Car(){
    trunk_size = "";
}

Car(string mk, string col, int yr, string mdl, string ts)
:Vehicle(mk, col, yr, mdl){
    trunk_size = ts;
}

void car_details(){
    cout << "Trunk size: " << trunk_size << endl;
}
};

class Honda: public Car{
int top_speed;

public:
Honda(){
    top_speed = 0;
}

Honda(string mk, string col, int yr, string mdl, string na, int ts)
:Car(mk, col, yr, mdl, na){
    top_speed = ts;
}

void Honda_details(){
    print_details();
    car_details();
    cout << "Top speed of the car: " << top_speed << endl;
}
};

// =====
// #include "vehicle.h"
// int main(){
//     Honda car("Honda", "Black", 2006, "Accord", "14.7 cubic feet", 260);
//     car.Honda_details();
// }

// =====
/*

```

Now, Honda class object has access to all member functions of Car class and the Car class has access to all members functions of the Vehicle class as they're now base classes of Honda class. The highlighted lines in the code indicate how multilevel inheritance is achieved.

In case of multi-level inheritance, a derived class' members are looked up in the immediate parent class and upwards until the ultimate base class until the member is found. Consider the following example:

```
/*
// =====
// #include <iostream>
// using namespace std;

// class A {
//   public:
//     void init() {
//       cout << "Class A initialized!" << endl;
//     }
//     void update() {
//       cout << "Class A updated!" << endl;
//     }
// };

// class B : public A {
//   public:
//     void update() {
//       cout << "Class B updated!" << endl;
//     }
// };

// class C : public B { };

// int main() {
//   // your code goes here
//   C c;
//   c.init();
//   c.update();
//   return 0;
// }
// =====
/*
```

Class A defines two functions, init() and update(). Class B is derived from class A and defines its own update() function. Class C is derived from class B and does not define any function.

The call to init() for an instance of C on line 26 results in a lookup of a matching function in C. Since that is not found, a matching function is looked up in B. Since one is not found, the lookup is performed in A and a match is found. That is why we see Class A initialized! on the console.

Similarly, when update() is called on line 27, a matching function is not found in C, so it is looked up in B. A match is found in B and the call resolves, resulting in Class B updated! on the console. The lookup proceeds no further.

Note that this is all happening at compile time. The function calls init() and update() in main() are bound to appropriate function definitions at compile time.

```
*/  
/*
```

The Diamond Problem

We'll cover the following

The diamond problem

Solutions to the diamond problem

The diamond problem#

When implementing multiple inheritance, you might run into a problem known as the diamond problem. The diamond problem occurs when a derived class inherits the same member from multiple parent classes. This causes ambiguity for the compiler.

Look at the following example. Press run and observe the compiler output.*/

```
// ======  
// #include<iostream>  
// using namespace std;  
  
// class A  
// {  
// protected:  
//   int ID;  
// public:  
//   A() : ID(0) {}  
// };  
  
// class B: public A  
// {  
// public:  
//   int length;  
// };  
  
// class C: public A  
// {  
// public:  
//   int radius;  
// };  
  
// class D: public B, public C  
// {  
// public:  
//   int getID() { return ID; }  
// };  
  
// int main(void)  
// {  
//   D d;  
//   cout << d.getID();  
//   return 0;
```

```
//}  
// =====  
/*
```

Notice that the compiler complains that in class D, the member ID is ambiguous. Both of its parent classes, B and C have that member. Hence a dilemma for the compiler. The situation is depicted in the following illustration. You'll notice the diamond shape of the figure, hence the name for the problem.

lass B gets its copy of the ID member and so does class C. Now, class D inherits two members with the same name.

Solutions to the diamond problem#

In the above code example, if you were to replace return ID; on line 27 with either return B::ID;, or return C::ID;, the program would compile and run fine. All that does is explicitly tell the compiler to fetch the value of the member named ID from class B or class C, depending on which solution you use.

However, there are still two copies of the ID member in this program. This may be OK for a simple program like this, but wouldn't be acceptable in larger programs.

One solution to the diamond problem is to use virtual inheritance, as shown in the following program.

```
*/  
// =====  
  
// #include <iostream>  
// using namespace std;  
  
// class A  
// {  
// protected:  
//   int ID;  
  
// public:  
//   A() : ID(0) {}  
// };  
  
// class B : virtual public A  
// {  
// public:  
//   int length;  
// };  
  
// class C : virtual public A  
// {  
// public:  
//   int radius;  
// };  
  
// class D : public B, public C  
// {  
// public:  
//   int getID() { return ID; }  
// };
```

```
// int main(void)
//{
// D d;
// cout << d.getID();
// return 0;
//}
//=====================================================================
/*
```

With virtual inheritance, only one instance of the base class A is inherited into the derived class D.

```
*/
```

```
/*
```

Override and Final

In this lesson, we'll discuss override and final in detail.

We'll cover the following

override

override

final

override#

The override keyword in a method declaration expresses that the method should override a virtual method of a base class. The compiler checks this assertion. It checks the parameter of the method, the return type of the method, and qualifiers like const and volatile. Of course, the compiler notices if the overridden method is not virtual.

The compiler verifies if the override annotated method actually overrides a virtual method of a base class.

The compiler checks for

The parameters and the return type.

The constness of the method.

The virtuality of the method.

The compiler ensures that the programmer obeys the contract.

By using the context-sensitive keywords override and final, we can explicitly manage the overriding of virtual functions. In particular, the keyword override solves a common bug present in object hierarchies: methods that should override methods of base classes but do not. The result is a syntactically but not semantically correct program. The program performs the wrong actions in the right way.

override#

To override a method, the signature of the overridden method of the base class has to match exactly.

Although this sounds easy in theory, it is often not in practice. If the signature of the method does not match exactly, the program will compile but have the wrong behavior... A different method than intended will be invoked.

final#

final supports two use cases. First, we can declare a method that cannot be overridden; second, we can define a class that cannot be derived from. The compiler uses the same rules to determine if a method of child class overrides a method of a base class. Of course, the strategy is inverted because the final specifier should disallow the overriding of a method. Therefore, the compiler checks the parameters of the method, its return type, and any const/volatile qualifiers.

A virtual method declared final must not be overridden.

The compiler checks for

The parameter.

The return type.

The constness of the method.

Methods and classes declared as final are an optimization opportunity for the compiler.

Both variants are equivalent:

void func() final;

```
virtual void func() final override;
```

The compiler ensures that the programmer obeys the contract.

To learn more about override, click [here](#).

To learn more about final, click [here](#).

In the next lesson, we'll look at some examples of override and final.

Examples

In this lesson, we'll discuss the examples of final and override.

We'll cover the following

Example 1: Override final

Explanation

Example 2: Override

Explanation

Example 3: final

Explanation

Example 1: Override final#

*/

// =====

// #include <iostream>

```
// class Sort
```

```
// {
```

```
// public:
```

```
// virtual void processData()
```

```
// {
```

```
//   readData();
```

```
//   sortData();
```

```
//   writeData();
```

```
// }
```

```
// private:
```

```
// virtual void readData() {}
```

```
// virtual void sortData() = 0;
```

```
// virtual void writeData() {}
```

```
//};
```

```
// class QuickSort : public Sort
```

```
// {
```

```
// private:
```

```
// void readData()
```

```
// {
```

```
//   std::cout << "readData" << std::endl;
```

```
// }
```

```
// void sortData()
```

```
// {
```

```
//   std::cout << "sortData" << std::endl;
```

```

// }
// void writeData()
// {
//   std::cout << "writeData" << std::endl;
// }
//};

// int main()
//{
// std::cout << std::endl;

// Sort *sort = new QuickSort;
// sort->processData();

// std::cout << std::endl;
//}

// =====
/*
Explanation#

```

We have implemented two classes named Sort and QuickSort.

We have created three private virtual methods and a public virtual method processData in the Sort class which calls the three private methods.

The QuickSort class publicly inherits from the Sort class.

We have overridden the methods of the Sort class in QuickSort.

By using a pointer to the Base class, we can access the overridden methods of the derived class.

Example 2: Override#

```

*/
// =====
// class Base {

// void func1();
// virtual void func2(float);
// virtual void func3() const;
// virtual long func4(int);

// virtual void f();

// };

// class Derived: public Base {

// // ill-formed; no virtual method func1 exists
// virtual void func1() override;

```

```

// // ill-formed: bad type
// virtual void func2(double) override;

// // ill-formed: const missing
// virtual void func3() override;

// // ill-formed: wrong return type
// virtual int func4(int) override;

// // well-formed: f override Base::f
// virtual void f() override;

//};

// int main(){

// Base base;
// Derived derived;

//};
//=====
/*
Explanation#

```

When we compile the program, the compiler will complain a lot. The error message is very specific.

The compiler complains in line 15 that the method func1 is not overriding a method. The same holds true for func2. It has the wrong parameter type. Continuing with the method func3, it complains that func3 has no const qualifier. func4 has the wrong return type. Only the method f in line 27 correctly overrides the method f of its base class.

final is the right tool for the job if a virtual method should not be overridden.

Example 3: final#

The given function causes a compilation error because the base method is declared final.

```

*/
//=====
// class Base {
// virtual void h(int) final;

// virtual void g(long int);
//};

// class Derived: public Base {

// // ill-formed: base method declared final
// virtual void h(int);

// // well-formed: a new virtual function
// virtual void h(double);

// virtual void g(long int) final;

```

```

// };

// class DerivedDerived: public Derived {
//   virtual void g(long int);
// };

// struct FinalClass final { };
// struct DerivedClass: FinalClass { };

// int main(){

//   Base base;
//   Derived derived;
//   DerivedDerived derivedDerived;

//   FinalClass finalClass;
//   DerivedClass derivedClass;

// };
// =====
/*
Explanation#

```

The compiler performs its job neatly. It complains that the method h in the class Base (line 2) is overridden by the method in class Derived (line 10). Of course, it's okay that the method h (line 13) in class Derived overloads h for the parameter type double. This method g (line 15) in the class Derived is quite interesting. The method overrides the method g (line 4) of the class Base and declares the method final. Therefore, g cannot be overridden in DerivedDerived (line 19).

DerivedClass cannot be derived from FinalClass, because the FinalClass is final

```

*/
/*
```

Advantages of Inheritance

In this lesson, you'll get to know about the advantages of Inheritance.

We'll cover the following

Avoiding Duplication of Code

Extensibility

Data Hiding

We have learned that we can implement inheritance which will result in avoiding redundant coding and will also save the programmer's time and effort.

Avoiding Duplication of Code#

Considering the previous example, if we have to implement another class for MoneyMarketAccount we don't need to duplicate the code for the deposit() and withdraw() methods inside this new Class because we can inherit and use the parent class's methods. In this way, we can avoid the duplication of code.

svg viewer

Extensibility#

Using inheritance, one can extend the base class logic as per the business logic of the derived class. This is an easy way to upgrade or enhance specific parts of a product without changing the core attributes. An existing class can act as a base class to derive a new class having upgraded features.

svg viewer

Data Hiding#

The base class can decide to keep some data private so that it cannot be altered by the derived class. This concept i.e. Encapsulation has already been discussed in the previous chapter.

Let's move on to quiz for checking your understanding of inheritance.

```
*/  
/*
```

Challenge 2: Implement an Animal Class

In this challenge, we'll implement a base class Animal and two derived classes Sheep and Dog.

We'll cover the following

Problem Statement

Input

Sample Input

Sample Output

Coding Exercise

Solution Review

Problem Statement#

The code below has:

A parent class named Animal.

Inside it define:

Name

Sound

void Animal_Details() function:

It prints the name and sound of the Animal.

Then there are two derived classes

Dog class

has a private member family

has a function named Dog_detail() which prints detail of the dog

Sheep class

has a private member color

has a function named Sheep_detail() which prints detail of the Sheep

The derived classes should

call the method of the Animal class which prints the name and the sound and for Dog class prints the family of dog that is Carnivores and for Sheep class prints the color of sheep White.

Input#

Name of Dog is set to Pongo and the Sound is set to woof woof in parametrized constructor of Dog object

Name of Sheep is set to Billy and the Sound is set to baaa baaa in parametrized constructor of Sheep object

Now, print Dog_detail and Sheep_detail from their respective objects

Here's a sample result which you should get.

Sample Input#
Dog d("Pongo", "Woof Woof");
d.Dog_detail();
Sheep s("Billy", "Baaa Baaa");
s.Sheep_detail();
Sample Output#
widget
Coding Exercise#
Implement the code in the problem tab.

Good Luck!

Problem

Solution

12345678

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

// Write classes code here

```
int main() {
    // Make classes objects here
}
```

Run

Save

Reset

Show Hint

Solution Review#

We have implemented Animal class which have Name and Sound variables, and a function Animal_detail() which prints Name and Sound of animal

Now implement Dog and Sheep classes inherited publicly from Animal class

Sheep has private string color variable and a function Sheep_detail() which calls Animal_detail() function and prints color of the sheep

Dog has private string family variable and a function Sheep_detail() which calls Animal_detail() function and prints family of the sheep

Create Dog and sheep object by calling parametrized constructors of the classes and print their traits by calling their respective functions

In the next challenge, we'll solve another exercise to get more grip on inheritance.

*/

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

// class Animal {
//   string Name;
//   string Sound;

// public:
//   Animal() {
//     Name = "";
//     Sound = "";
//   }

//   Animal(string nam, string soun) {
//     Name = nam;
//     Sound = soun;
//   }

//   void Animal_detail() {
//     cout << "Animal Name : " << Name << endl;
//     cout << "Animal Sound : " << Sound << endl;
//   }
// };

// class Dog : public Animal {

//   string family;

// public:
//   Dog(string N, string S): Animal(N, S) {
//     family = "Carnivores";
//   }

//   void Dog_detail() {
//     Animal_detail();
//     cout << "Dog's Family : " << family << endl;
//   }
// };

// class Sheep : public Animal {

//   string color;

// public:
//   Sheep(string N, string S): Animal(N, S) {
//     color = "White";
//   }

//   void Sheep_detail() {
//     Animal_detail();
//     cout << "Sheep Color: " << color << endl;
//   }
// }
```

```
// };

// int main() {
// Dog d("Pongo", "Woof Woof");
// d.Dog_detail();

// cout << endl;

// Sheep s("Billy", "Baaa Baaa");
// s.Sheep_detail();
//}
// =====
```

/*

Challenge 3: Implement a Father Class

In this challenge, we'll implement a base class father and derived classes, son and daughter.

We'll cover the following

Problem Statement

Input

Sample Input

Expected Output

Coding Exercise

Solution Review

Problem Statement#

Implement a code which have:

A parent class named Father.

Inside it define:

eye_color

hair_color

void Father_traits() function:

It prints the eye_color and hair_color of the called object

Then, there are two derived classes

Son class

has a private member name

has a function named Son_traits() which prints traits of the Son

Daughter class

has a private member name

has a function named Daughter_traits() which prints traits of the Daughter

The derived classes should

call the method of the Father class which prints the eye_color and the hair_color and for Son and Daughter classes prints the name of a respective object.

Input#

In Sonclass, eye_color is set to Brown and the hair_color is set to Black and name is set to Ralph in parametrized constructor of Son object

In Daughterclass, eye_color is set to Green and the hair_color is set to Golden and name is set to Rapunzel in parametrized constructor of Daughter object

Now, print Son_traits and Daughter_traits from their respective objects

Here's a sample result which you should get.

Sample Input#

```
Daughter obj("Rapunzel","Green","Golden");
obj.Daughter_traits();
```

```
Son Obj("Ralph","Brown","Black");
```

```
Obj.Son_traits();
```

Expected Output#

Coding Exercise#

Implement the code in the problem tab.

Good Luck!

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

```
// Write your classes here
```

```
int main() {
    // create classes objects here
    // call derived class member functions here
    return 0;
}
```

Solution Review#

We have implemented Father class which have eye_color and hair_color variables, and a function Father_traits() which prints eye_color and hair_color of animal

Now implement Daughter and Son classes inherited publicly from Father class

Daughter has private string name variable and a function Daughter_traits() which calls Father_traits() function and prints name of the Daughter

Son has private string name variable and a function Son_traits() which calls Father_traits() function and prints name of the Son

Create Son and Daughter object by calling parametrized constructors of the classes and print their traits by calling their respective functions

In the next chapter, we'll learn about a very important concept of OOP paradigm, polymorphism.

```
*/
```

```
// =====
// #include <iostream>
```

```
// using namespace std;

// class Father {
//   string eye_color;
//   string hair_color;
// public:
//   Father(string eye, string hair) {
//     eye_color = eye;
//     hair_color = hair;
//   }

//   void father_traits(){
//     cout << "Eye color: " << eye_color << endl;
//     cout << "Hair color: " << hair_color << endl;
//   }
// };

// class Daughter : public Father {
//   string name;
// public:
//   Daughter(string nam, string eye, string hair) : Father(eye, hair) {
//     name = nam;
//   }

//   void Daughter_traits(){
//     father_traits();
//     cout << name << " has long hair!\n";
//   }
// };

// class Son : public Father {
//   string name;
// public:
//   Son(string nam,string eye, string hair): Father(eye, hair) {
//     name = nam;
//   }

//   void Son_traits(){
//     father_traits();
//     cout << name << " has beard!\n";
//   }
// };

// int main ()
//{
//   Daughter obj("Rapunzel","Green","Golden");
//   obj.Daughter_traits();

//   cout << endl;

//   Son Obj("Ralph","Brown","Black");
```

```
//      Obj.Son_traits();  
//}  
//=====
```

```
/*  
Challenge 4: Implement Derived Class to Calculate Min/Max/Mean
```

In this exercise, you have to implement a derived class that will calculate the min, max and the mean of the data set given in the dynamic array.

We'll cover the following

Problem Statement

Input

Sample Input

Expected Output

Coding Exercise

Solution Review

Problem Statement#

In this challenge, you are given a class called DynamicArray which implements a dynamic array of integers that can grow in size.

This class has the following functions:

void append(int value): Adds a new value at the end of the array.

int get(int index): Returns the value at the given index.

int length(): Returns the current size of the array.

void resize(): Resizes the array when the maximum capacity is reached. The growth factor is two, therefore, it will each time double the capacity of the array.

Your task is to write a derived class called DynamicArrayWithStats which will implement the following functions:

int max(): Returns the maximum element in the dynamic array.

int min(): Returns the minimum element present in the dynamic array.

int mean(): Finds the mean value from the elements in the array and returns it.

Input#

The input will be an integer and it will be given using the append() function. The get() function will be used to print the values in the dynamic array. Then, all functions of the DynamicArrayWithStats class will be called.

Sample Input#

We want to execute the following instructions:

```
DynamicArrayWithStats arr = DynamicArrayWithStats();  
arr.append(2);  
arr.append(6);  
arr.append(4);  
arr.append(1);  
arr.append(3);
```

```
cout << "Array: "  
for(int i = 0; i < arr.length(); i++){  
    cout << arr.get(i) << " ";
```

```
}

cout << endl;

cout << "Max: " << arr.max() << endl;
cout << "Min: " << arr.min() << endl;
cout << "Mean: " << arr.mean() << endl;
```

Expected Output#

The following is the correct output:

Array: 2 6 4 1 3

Max: 6

Min: 1

Mean: 3

Coding Exercise#

Implement the code in the problem tab.

Good Luck!

```
/*
// =====
// #include <iostream>
// #include <assert.h>
// using namespace std;

// class DynamicArray
//{
//  int *array;
//  int capacity = 2;
//  int size;

// public:
//  DynamicArray()
//  {
//    array = new int[capacity];
//    size = 0;
//  }

//  void append(int element)
//  {
//    insertAt(element, size);
//  }

//  int length()
//  {
//    return size;
//  }

//  int get(int pos)
//  {
//    return array[pos];
//  }
```

```
// ~DynamicArray()
// {
//   delete[] array;
// }

// private:
// void insertAt(int element, int pos)
// {
//   assert(0 <= pos && pos <= size);
//   if (size == capacity)
//   {
//     resize();
//   }
//   for (int i = size; i > pos; i--)
//   {
//     array[i] = array[i - 1];
//   }
//   size++;
//   array[pos] = element;
// }

// void resize()
// {
//   capacity *= 2;
//   int *temp = new int[capacity];
//   copy(array, array + size, temp);
//   delete[] array;
//   array = temp;
// }
//};

// class DynamicArrayWithStats : public DynamicArray
//{
// public:
//   int max()
//   {
//     int max = get(0);
//     for (int i = 1; i < length(); i++)
//     {
//       if (get(i) > max)
//         max = get(i);
//     }
//     return max;
//   }
//   int min()
//   {
//     int min = get(0);
//     for (int i = 1; i < length(); i++)
//     {
//       if (get(i) < min)
```

```

//     min = get(i);
// }
// return min;
// }
// int mean()
// {
//     int sum = 0;
//     for (int i = 0; i < length(); i++)
//     {
//         sum += get(i);
//     }
//     int mean = sum / length();
//     return mean;
// }
//};

// int main()
//{
//    DynamicArrayWithStats arr = DynamicArrayWithStats();
//    arr.append(2);
//    arr.append(6);
//    arr.append(4);
//    arr.append(1);
//    arr.append(3);

//    cout << "Array: ";
//    for (int i = 0; i < arr.length(); i++)
//    {
//        cout << arr.get(i) << " ";
//    }
//    cout << endl;

//    cout << "Max: " << arr.max() << endl;
//    cout << "Min: " << arr.min() << endl;
//    cout << "Mean: " << arr.mean() << endl;
//    return 0;
//}
//=====
/*

```

Solution Review#

We have implemented a class called `DynamicArrayWithStats` and inherited it publicly from the `DynamicArray` class.

The `DynamicArrayWithStats` class has a public function called `max()` that finds the maximum element of the array. It uses `get()` function of `DynamicArray` to search through all elements of the array by index.

The `min()` function in `DynamicArrayWithStats` class finds the minimum element of the array. It also uses `get()` function of `DynamicArray` class.

The `mean()` function makes first calculates the sum of all elements and then calls `length()` to find the size of the array. The sum is divided by the size to find the mean value.

*/

```
/*
```

What is Polymorphism?

In this lesson, we will be learning about the basics of polymorphism with the implementation details.

The word Polymorphism is a combination of two Greek words, Poly means many and Morph means forms.

Definition#

When we say polymorphism in programming that means something which exhibits many forms or behaviors. So far, we have learned that we can add new data and functions to a class through inheritance. But what about if we want our derived class to inherit a method from the base class and have a different implementation for it? That is when polymorphism comes in, a fundamental concept in OOP paradigm.

Shape Class#

We are considering here the example of Shape class, which is base class for many shapes like Rectangle and Circle. This class contains a function `getArea()` which calculates the area for the derived classes.

Implementation#

Let's look at the implementation of Shape class:

```
// A simple Shape interface which provides a method to get the Shape's area
class Shape {
    public:
        float getArea(){}
};
```

Rectangle Class#

Consider the Rectangle class which is derived from Shape class. It has two data members, i.e., width and height and it returns the Area of the rectangle by using `getArea()` function.

Implementation#

Let's look at the implementation of the Rectangle class:

```
/*
// =====
// A Rectangle is a Shape with a specific width and height
// class Rectangle : public Shape { // derived form Shape class
//     private:
//         float width;
//         float height;

//     public:
//     Rectangle(float wid, float heigh) {
//         width = wid;
//         height = heigh;
//     }
//     float getArea(){
//         return width * height;
//     }
// };
// =====
/*
```

Circle Class#

Consider the Circle class which is derived from Shape class. It has one data member, i.e., radius and it returns the Area of the circle by using getArea() function.

```
/*
// =====
// A Circle is a Shape with a specific radius
// class Circle : public Shape {
// private:
// float radius;

// public:
// Circle(float rad){
//   radius = rad;
// }
// float getArea(){
//   return 3.14159f * radius * radius;
// }
// };
// =====
*/
```

Now, if we merge all the classes then by calling the getArea() function, let's see what happened:

```
/*
// =====
// #include <iostream>
// using namespace std;

/// A simple Shape interface which provides a method to get the Shape's area
class Shape {
public:
float getArea(){}
};

/// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape { // derived form Shape class
private:
float width;
float height;

public:
Rectangle(float wid, float heigh) {
width = wid;
height = heigh;
}
float getArea(){
return width * height;
}
};

/// A Circle is a Shape with a specific radius
class Circle : public Shape {
private:
float radius;
```

```

// public:
// Circle(float rad){
//   radius = rad;
// }
// float getArea(){
//   return 3.14159f * radius * radius;
// }
//};

// int main() {
// Rectangle r(2, 6); // Creating Rectangle object

// Shape* shape = &r; // Referencing Shape class to Rectangle object

// cout << "Calling Rectangle getArea function: " << r.getArea() << endl;
// cout << "Calling Rectangle from shape pointer: " << shape->getArea() << endl << endl;

// Circle c(5); // Creating Circle object

// shape = &c; // Referencing Shape class to Circle object

// cout << "Calling Circle getArea function: " << c.getArea() << endl;
// cout << "Calling Circle from shape pointer: " << shape->getArea() << endl << endl;
// =====
/*

```

Explanation of Code#

Polymorphism only works with a pointer and reference types, so we have created a Shape pointer, and pointed to the derived class objects. But due to multiple existences of the same functions in classes, it will get confused between which class getArea() function it's calling. The derived classes function has a different implementation but the same name and that's why we are not getting the expected output.

```
*/
/*
```

Overriding

In this lesson, we'll be learning about how overriding is done in C++.

We'll cover the following

getArea() Overridden Function

Implementation

Advantages of the Method Overriding

Key Features of Overriding

In object-oriented programming when we allow a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes is known as Function Overriding.

getArea() Overridden Function#

As you have already seen the implementation of the function `getArea()` in the previous lesson, which depicts the concept of overriding.

```
/*
// =====
// #include <iostream>
// using namespace std;

/// A simple Shape interface which provides a method to get the Shape's area
class Shape {
public:
    float getArea(){}
};

/// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape { // derived form Shape class
private:
    float width;
    float height;

public:
    Rectangle(float wid, float heigh) {
        width = wid;
        height = heigh;
    }
    float getArea(){
        return width * height;
    }
};

/// A Circle is a Shape with a specific radius
class Circle : public Shape {
private:
    float radius;

public:
    Circle(float rad){
        radius = rad;
    }
    float getArea(){
        return 3.14159f * radius * radius;
    }
};

int main() {
    Rectangle r(2, 6); // Creating Rectangle object

    Shape* shape = &r; // Referencing Shape class to Rectangle object

    cout << "Calling Rectangle getArea function: " << r.getArea() << endl; // Calls Rectangle.printArea()
}
```

```
// cout << "Calling Rectangle from shape pointer: " << shape->getArea() << endl << endl; // Calls shape's
dynamic-type's

// Circle c(5); // Creating Circle object

// shape = &c; // Referencing Shape class to Circle object

// cout << "Calling Circle getArea function: " << c.getArea() << endl;
// cout << "Calling Circle from shape pointer: " << shape->getArea() << endl << endl;
// */
// =====
/*
```

Advantages of the Method Overriding#

Method overriding is very useful in OOP and have many advantages. Some of them are stated below:

The derived classes can give its own specific implementation to inherited methods without modifying the parent class methods.

If a child class needs to use the parent class method, it can use it, and the other classes that want to have different implementation can use the overriding feature to make changes.

Key Features of Overriding#

Here are some key features of the Method Overriding:

Overriding needs inheritance and there should be at least one derived class.

Derived class/es must have the same declaration, i.e., name, same parameters and same return type of the function as of the base class.

The function in derived class/es must have different implementation from each other.

The method in the base class must need to be overridden in the derived class.

```
*/  
/*
```

Virtual Member Functions

In this lesson, we'll be learning about a very important concept of polymorphism, i.e., Virtual member.

We'll cover the following

Definition

Why Do We Need a Virtual Function?

Explanation

Virtual means existing in appearance but not in reality.

Definition#

A virtual function is a member function which is declared within the base class and is overridden by the derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call. They are mainly used to achieve Runtime polymorphism. Functions are declared with a virtual keyword in a base class. The function resolution call is done at run-time.

Why Do We Need a Virtual Function?

Suppose you have a number of objects of different classes like in this case, we have multiple shapes classes. You want to put them all in an array and perform a particular operation on them using the same function call. Given an example, we want to access the same function getArea() from multiple child classes.

```
/*
//=====
// #include <iostream>
// using namespace std;

/// A simple Shape interface which provides a method to get the Shape's area
class Shape {
public:
    virtual float getArea(){}
};

/// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape { // derived form Shape class
private:
    float width;
    float height;

public:
    Rectangle(float wid, float heigh) {
        width = wid;
        height = heigh;
    }
    float getArea(){
        return width * height;
    }
};

/// A Circle is a Shape with a specific radius
class Circle : public Shape {
private:
    float radius;

public:
    Circle(float rad){
        radius = rad;
    }
    float getArea(){
        return 3.14159f * radius * radius;
    }
};

int main() {
    Rectangle r(2, 6); // Creating Rectangle object
```

```

// Shape* shape = &r; // Referencing Shape class to Rectangle object

// cout << "Calling Rectangle from shape pointer: " << shape->getArea() << endl; // Calls shape's dynamic-
type's

// Circle c(5); // Creating Circle object
// shape = &c; // Referencing Shape class to Circle object

// cout << "Calling Circle from shape pointer: " << shape->getArea() << endl;

// }
// =====
/*
Explanation#
we have seen earlier when we're trying to print the child class function getArea() by referencing a parent class
pointer, it gave us an error. Just by writing the keyword virtual we can reference a parent class pointer to child
class object.
*/
/*

```

Pure Virtual Member Functions

In this lesson, we'll be learning about a very important concept of polymorphism, i.e., Pure Virtual Member Functions.

We'll cover the following

Abstract Class

How to Write a Pure Virtual Function?

=0 Sign

Overriding Virtual Function

Explanation

Abstract Class#

We can only make derived class's objects to access their functions, and we will never want to instantiate objects of a base class, we call it an abstract class. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects.

How to Write a Pure Virtual Function?#

It may also provide an interface for the class hierarchy by placing at least one pure virtual function in the base class. A pure virtual function is one with the expression =0 added to the declaration.

=0 Sign#

The equal sign = here has nothing to do with the assignment, the value 0 is not assigned to anything. The =0 syntax is simply how we tell the compiler that a virtual function will be pure.

Overriding Virtual Function#

Once you've placed a pure virtual function in the base class, you must override it in all the derived classes from which you want to instantiate objects. If a class doesn't override the pure virtual function, it becomes an abstract class itself, and you can't instantiate objects from it (although you might from classes derived from it). For consistency, you may want to make all the virtual functions in the base class pure.

```

*/
// =====
// #include <iostream>
```

```
// using namespace std;

// // A simple Shape interface which provides a method to get the Shape's area
// class Shape
// {
// public:
// virtual float getArea() = 0;
// };

// // A Rectangle is a Shape with a specific width and height
// class Rectangle : public Shape
// { // derived form Shape class
// private:
// float width;
// float height;

// public:
// Rectangle(float wid, float heigh)
// {
// width = wid;
// height = heigh;
// }
// float getArea()
// {
// return width * height;
// }
// };

// // A Circle is a Shape with a specific radius
// class Circle : public Shape
// {
// private:
// float radius;

// public:
// Circle(float rad)
// {
// radius = rad;
// }
// float getArea()
// {
// return 3.14159f * radius * radius;
// }
// };

// // A Square is a Shape with a specific length
// class Square : public Shape
// {
// private:
// float length;

// public:
```

```

// Square(float len)
// {
//   length = len;
// }
// float getArea()
// {
//   return length * length;
// }
//};

// int main()
//{
// Shape *shape[3]; // Referencing Shape class to Rectangle object
// // Shape * shape1 = new Shape(); //Instantiating the shape object

// Rectangle r(2, 6); // Creating Rectangle object
// shape[0] = &r; // Referencing Shape class to Rectangle object

// Circle c(5); // Creating Circle object
// shape[1] = &c; // Referencing Shape class to Circle object

// Square s(10); // Creating Square object
// shape[2] = &s; // Referencing Shape class to Circle object

```

```

// for (int i = 0; i < 3; i++)
//   cout << shape[i]->getArea() << endl;
//}
//=====
/*

```

Explanation#

Now in main(), when you attempt to create objects of a Shape class by uncommenting the line no.55, the compiler will complain that you're trying to instantiate an object of an abstract class. It will also tell you the name of the pure virtual function that makes it an abstract class. Notice that, although this is only a declaration, you never need to write a definition of the Shape class getArea(). Initialize the Shape class pointer and point it to objects of derived classes to access the getArea() function of respective classes.

```
*/
```

```
/*
```

Challenge 1: Implement an Account Class Using Virtual Functions

In this challenge, we'll implement an account class along with two derived classes saving and current.

We'll cover the following

Problem Statement

Input

Sample Input

Sample Output

Coding Exercise

Solution Review

Problem Statement#

Write a code that has:

A parent class named Account.

Inside it define:

a protected float member balance

We have three virtual functions:

void Withdraw(float amount)

void Deposit(float amount)

void printBalance()

Then, there are two derived classes

Savings class

has a private member interest_rate set to 0.8

Withdraw(float amount) deducts amount from balance with interest_rate

Deposit(float amount) adds amount in balance with interest_rate

printBalance() displays the balance in the account

Current class

Withdraw(float amount) deducts amount from balance

Deposit(float amount) adds amount in balance

printBalance() displays the balance in the account`

Input#

In Savings class, balance is set to 50000 in parametrized constructor of Savings object

In Current class, balance is set to 50000 in parametrized constructor of Current object

Here's a sample result which you should get.

Sample Input#

```
Savings s1(50000);
Account * acc = &s1;
acc->Deposit(1000);
acc->printBalance();
```

```
acc->Withdraw(3000);
acc->printBalance();
```

```
Current c1(50000);
acc = &c1;
acc->Deposit(1000);
acc->printBalance();
```

```
acc->Withdraw(3000);
acc->printBalance();
```

Sample Output#

widget

Coding Exercise#

Implement the code in the problem tab.

Good Luck!

```
*/
// =====
// #include <iostream>
```

```
// using namespace std;

// class Account {
// protected:
// float balance;

// public:
// Account(float bal) {
//   balance = bal;
// }

// virtual void Deposit(float amount){}
// virtual void Withdraw(float amount){}
// virtual void printBalance(){}
//};

// class Savings: public Account {
// float interest_rate = 0.8;

// public:
// Savings(int bal): Account(bal){}

// void Deposit(float amount) {
//   balance += amount + (amount * interest_rate);
// }

// void Withdraw(float amount) {
//   balance -= amount + (amount * interest_rate);
// }

// void printBalance() {
//   cout << "Balance in your saving account: " << balance << endl;
// }
//};

// class Current: public Account {

// public:
// Current(int bal): Account(bal){}

// void Deposit(float amount) {
//   balance += amount;
// }

// void Withdraw(float amount) {
//   balance -= amount;
// }

// void printBalance() {
//   cout << "Balance in your current account: " << balance << endl;
// }
//};
```

```

// };

// int main() {
//   // creating savings account object
//   Savings s1(50000);
//   Account * acc = &s1; // pointing acc to savings object
//   acc->Deposit(1000);
//   acc->printBalance();

//   acc->Withdraw(3000);
//   acc->printBalance();

//   cout << endl;

//   // creating current account object
//   Current c1(50000);
//   acc = &c1; // pointing acc to current object
//   acc->Deposit(1000);
//   acc->printBalance();

//   acc->Withdraw(3000);
//   acc->printBalance();
// }

//=====
/*

```

Solution Review#

We have implemented Account class which has balance float variable, and three virtual functions Deposit(float amount), Withdraw(amount) and printBalance()

Now implement Savings and Current classes inherited publicly from Account class

Savings has private float interest_rate variable and functions:

Withdraw(float amount) deducts amount from balance with interest_rate

Deposit(float amount) adds amount in balance with interest_rate

printBalance() displays the balance in the account

Current has functions:

Withdraw(float amount) deducts amount from balance

Deposit(float amount) adds amount in balance

printBalance() displays the balance in the account`

Create Savings and Current object by calling parametrized constructors of the classes and print their balance by calling their respective functions

In the next challenge, we'll be implementing this problem using a pure virtual function.

*/

/*

Challenge 2: Implement a Class Using Pure Virtual Functions

In this challenge, we'll implement an account class along with two derived classes saving and current.

We'll cover the following

Problem Statement

Input

Sample Input

Sample Output

Coding Exercise

Solution Review

Problem Statement#

Write a code that has:

A parent class named Account.

Inside it define:

a protected float member balance

We have three pure virtual functions:

void Withdraw(float amount)

void Deposit(float amount)

void printBalance()

Then, there are two derived classes

Savings class

has a private member interest_rate set to 0.8

Withdraw(float amount) deducts amount from balance with interest_rate

Deposit(float amount) adds amount in balance with interest_rate

printBalance() displays the balance in the account

Current class

Withdraw(float amount) deducts amount from balance

Deposit(float amount) adds amount in balance

printBalance() displays the balance in the account`

Input#

In Savings class, balance is set to 50000 in parametrized constructor of Savings object called by Account class

In Current class, balance is set to 50000 in parametrized constructor of Current object called by Account class

Here's a sample result which you should get.

Sample Input#

```
Account * acc[2];
acc[0] = new Savings(50000);
acc[0]->Deposit(1000);
acc[0]->printBalance();
```

```
acc[0]->Withdraw(3000);
acc[0]->printBalance();
```

```
acc[1] = new Current(50000);
acc[1]->Deposit(1000);
acc[1]->printBalance();
```

```
acc[1]->Withdraw(3000);
```

```
acc[1]->printBalance();
```

Sample Output#

widget

Coding Exercise#

Implement the code in the problem tab.

Good Luck!

```
/*
// =====
// #include <iostream>
// using namespace std;

// class Account {
// protected:
// float balance;

// public:
// Account(float bal) {
//   balance = bal;
// }

// virtual void Deposit(float amount) = 0;
// virtual void Withdraw(float amount) = 0;
// virtual void printBalance() = 0;
// };

// class Savings: public Account {
// float interest_rate = 0.8;

// public:
// Savings(int bal): Account(bal){}

// void Deposit(float amount) {
//   balance += amount + (amount * interest_rate);
// }

// void Withdraw(float amount) {
//   balance -= amount + (amount * interest_rate);
// }

// void printBalance() {
//   cout << "Balance in your saving account: " << balance << endl;
// }
// };

// class Current: public Account {

// public:
// Current(int bal): Account(bal){}

// void Deposit(float amount) {
```

```

// balance += amount;
// }

// void Withdraw(float amount) {
//   balance -= amount;
// }

// void printBalance() {
//   cout << "Balance in your current account: " << balance << endl;
// }
//};

// int main() {
//   // creating savings account object by calling account pointer
//   Account * acc[2];
//   acc[0] = new Savings(50000); // pointing acc to savings object
//   acc[0]->Deposit(1000);
//   acc[0]->printBalance();

//   acc[0]->Withdraw(3000);
//   acc[0]->printBalance();

//   cout << endl;

//   // creating current account object by calling account pointer

//   acc[1] = new Current(50000); // pointing acc to current object
//   acc[1]->Deposit(1000);
//   acc[1]->printBalance();

//   acc[1]->Withdraw(3000);
//   acc[1]->printBalance();
// }

=====
/*

```

Solution Review#

We have implemented Account class which has balance float variable, and three pure virtual functions Deposit(float amount), Withdraw(amount) and printBalance()

Now implement Savings and Current classes inherited publicly from Account class

Savings has private float interest_rate variable and functions:

Withdraw(float amount) deducts amount from balance with interest_rate

Deposit(float amount) adds amount in balance with interest_rate

printBalance() displays the balance in the account

Current has functions:

Withdraw(float amount) deducts amount from balance
Deposit(float amount) adds amount in balance
printBalance() displays the balance in the account`
Create Savings and Current object by calling parametrized constructors of the classes and print their balance by calling their respective functions

In the next chapter, we'll learn about the advanced concepts of Composition, Aggregation and

*/

/*

Operator Overloading

C++ allows us to overload operators. Let's find out how.

We'll cover the following

Definition

Rules

Example

Prohibited operators

Assignment operators

Example

Further information

Definition#

C++ allows us to define the behavior of operators for our own data types. This is known as operator overloading.

Operators vary in nature and therefore, require different operands. The number of operands for a particular operator depends on:

the kind of operator (infix, prefix, etc.)

whether the operator is a method or function.

```
struct Account{  
    Account& operator += (double b){  
        balance += b;  
        return *this; }....  
};
```

...

```
Account a;
```

```
a += 100.0;
```

We have already encountered function overloading. If the function is inside a class, it must be declared as a friend and all its arguments must be provided.

Rules#

To achieve perfect operator overloading, there is a large set of rules we have to follow. Here are some of the important ones.

We cannot change the precedence of an operator. The compiler computes all operators in order of precedence. We cannot alter this order. Hence, whatever operation our operator performs, it will be computed after the operator with higher precedence.

Derived classes inherit all the operators of their base classes except the assignment operator. Each class needs to overload the = operator.

All operators other than the function call operator cannot have default arguments.

Operators can be called explicitly. A benefit of overloading an operator is that it can be used directly with its operands. However, the compiler may cause some implicit conversion in this process. We can make explicit calls to the overloaded operator in the following format: a.operator += (b).

Example#

```
/*
// =====
// #include <iostream>

// class Account{

// public:
// explicit Account(double b): balance(b){}

// Account& operator += (double b){
//   balance += b;
//   return * this;
// }

// friend Account& operator += (Account& a, Account& b);
// friend std::ostream& operator << (std::ostream& os, const Account& a);

// private:
// double balance;

// };

// Account& operator += (Account& a, Account& b){
//   a.balance += b.balance;
//   return a;
// }

// std::ostream& operator << (std::ostream& os, const Account& a){
//   os << a.balance;
//   return os;
// }

// int main(){

// std::cout << std::endl;

// Account acc1(100.0);
// Account acc2(100.0);
// Account acc3(100.0);
```

```

// acc1 += 50.0;
// acc1 += acc1;

// acc2 += 50.0;
// acc2 += acc2;

// acc3.operator += (50.0);
// //acc3.operator += (acc3); ERROR

// std::cout << "acc1: " << acc1 << std::endl;
// std::cout << "acc2: " << acc2 << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

The `+=` operator is being overloaded for the `Account` class in lines 8 to 10. Now, we can add and assign a double to an `Account` object without any problems. However, this will not support `Account += Account`, where both operands are of the `Account` type.

For this purpose, we declare a friend function in line 13 to support the additional assignment between two `Account` objects. The friend function can access the private member, `balance`. Hence, we can perform `a.balance + b.balance`. The function has been implemented in lines 21 to 24.

The `<<` output operator has also been overloaded in a friend function on lines 26 to 29. Now, `std::cout << acc`; will print the balance of the `acc` object.

From lines 39 to 43, we can see examples of the additional assignment working between the `Account` and double types.

One thing to note is that the explicit operator call works when the argument is a double, as seen in line 45.

However, the call would not work for the `Account` argument in line 46. This is because the class doesn't support the conversion from `Account` to double.

Prohibited operators#

The following operators cannot be overloaded:

```
.
::
?:
sizeof
.*
typeof
```

Assignment operators#

We can overload the assignment operator by implementing it as a copy or move assignment operator. It has to be implemented in a class method. The implementation is very similar to a copy or move constructor.

If the assignment operator is not overloaded, the compiler creates one implicitly. This operator performs a member-wise assignment of all the values from the object to be assigned. This is very similar to the behavior

of the copy constructor, except that instead of a new object being created, the members of an existing object are updated.

Example#

```
/*
// =====
// #include <algorithm>
// #include <chrono>
// #include <iomanip>
// #include <iostream>

// class Account{
// public:

// Account()=default;
// Account(int numb): numberOf(numb), deposits(new double[numb]){}

// Account(const Account& other): numberOf(other.numberOf), deposits(new double[other.numberOf]){
//     std::copy(other.deposits, other.deposits + other.numberOf, deposits);
// }

// Account& operator = (const Account& other){
//     if (this != &other){
//         delete[] deposits;
//         numberOf = other.numberOf;
//         deposits = new double[other.numberOf];
//         std::copy(other.deposits, other.deposits + other.numberOf, deposits);
//     }
//     return *this;
// }

// Account(Account&& other):numberOf(other.numberOf), deposits(other.deposits){
//     other.deposits = nullptr;
//     other.numberOf = 0;
// }

// Account& operator =(Account&& other){
//     numberOf = other.numberOf;
//     deposits = other.deposits;
//     other.deposits = nullptr;
//     other.numberOf = 0;
//     return *this;
// }

// ~Account() noexcept {
//     delete [] deposits;
// }

// private:
// int numberOf;
// double * deposits;
//};
```

```

// int main(){

// std::cout << std::endl;
// std::cout << std::fixed << std::setprecision(10);

// Account account(200000000);
// Account account2(100000000);

// auto start = std::chrono::system_clock::now();
// account = account2;
// std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
// std::cout << "Account& operator = (const Account& other): " << dur.count() << " seconds" << std::endl;

// start = std::chrono::system_clock::now();
// account = std::move(account2);
// dur = std::chrono::system_clock::now() - start;
// std::cout << "Account& operator=(Account&& other):" << dur.count() << " seconds" << std::endl;

// std::cout << std::endl;
// }

// =====
/*

```

In this example, the assignment operator for the Account class is overloaded for both copy (line 16) and move (line 31) operations.

If the argument is an lvalue, a copy is performed. A new array on the heap is created, called deposits, and the contents of other's array is copied into it, as seen in lines 20 and 21.

If the argument is an rvalue, a move is performed. In this case, a new array is not created. This makes the move operation much faster.

This is evident in the main program.

The std::move call on line 61 returns an rvalue, hence the assignment operator will move the data from account2 to account.

This is significantly more efficient than the copy assignment on line 56.

Further information#
function overloading
*/

/*
Explicit Conversion Operators
This lesson explains how conversion operators can be overloaded explicitly in C++.

We'll cover the following

Asymmetry in C++98
Example

Asymmetry in C++98#

In C++98, the explicit keyword was only supported for conversion constructors. Conversion operators converted user-defined objects implicitly.

All this changed in C++11. Now, we can overload conversion operators to explicitly prevent and permit conversions.

Let's suppose that a class called MyClass can perform conversions from class A to MyClass and from MyClass to class B.

widget

Here is what myClass would look like:

```
class MyClass{
public:
    explicit MyClass(A){}    // C++98
    explicit operator B(){}   // C++11
};
```

MyClass(A): Converting constructor

operatorB(): Converting operator

As we can see, the explicit keyword can now be used when overloading the conversion operator, B().

One thing to keep in mind is that implicit conversions to bool are still possible, so be careful.

```
class MyBool{
public:
    explicit operator bool(){return true;}
};
```

...

MyBool myB;

if (myB){};

int a = (myB)? 3: 4;

int b = myB + a; // ERROR

We have defined that a MyBool object can be converted to bool but not to anything else.

Because of this, int b = myB + a; causes an error, since it is trying to implicitly convert myB to int.

Example#

```
/*
// =====
// #include <iostream>

// class A{};

// class B{};

// class MyClass{
// public:
//     MyClass(){}
//     explicit MyClass(A{})           // since C++98
```

```

// explicit operator B(){return B();} // new with C++11
// };

// void needMyClass(MyClass){};
// void needB(B){};

// struct MyBool{
// explicit operator bool(){return true;}
// };

// int main(){

// // A -> MyClass
// A a;

// // explicit invocation
// MyClass myClass1(a);
// // implicit conversion from A to MyClass
// MyClass myClass2=a;
// needMyClass(a);

// // MyClass -> B
// MyClass myCl;

// // explicit invocation
// B b1(myCl);
// // implicit conversion from MyClass to B
// B b2= myCl;
// needB(myCl);

// // MyBool -> bool conversion
// MyBool myBool;
// if (myBool){};
// int myNumber = (myBool)? 1998: 2011;
// // implicit conversion
// int myNewNumber = myBool + myNumber;
// auto myTen = (20*myBool -10*myBool)/myBool;

// std::cout << myTen << std::endl;

// }
// =====
/*

```

We have defined an explicit conversion constructor from A to MyClass in line 10.

The constructor call works fine in line 27, but the implicit conversions in lines 29 and 30 are rejected by the compiler.

needMyClass(a) will not be able to implicitly convert a to MyClass. This functionality has been available since C++98.

We have defined an explicit conversion operator from MyClass to B in line 11.

Lines 38 and 39 use an implicit conversion. Due to the explicit conversion operator B in line 11, this is not valid.

Because of this explicit definition, implicit conversions through the operator are rejected by the compiler, as seen in lines 46 and 47.

The explicit conversion feature was introduced in C++11.

```
*/  
/*  
Exercise#  
Adjust the following program so that all implicit conversions are possible. Does the program behave as  
expected?  
*/  
// ======  
// #include <iostream>  
  
// class A{};  
  
// class B{};  
  
// class MyClass{  
// public:  
//     MyClass(){}  
//     explicit MyClass(A{})           // since C++98  
//     explicit operator B(){return B();} // new with C++11  
// };  
  
// void needMyClass(MyClass){};  
// void needB(B){};  
  
// struct MyBool{  
//     explicit operator bool(){return true;}  
// };  
  
// int main(){  
//     // A -> MyClass  
//     A a;  
  
//     // explicit invocation  
//     MyClass myClass1(a);  
//     // implicit conversion from A to MyClass  
//     MyClass myClass2=a;  
//     needMyClass(a);  
  
//     // MyClass -> B  
//     MyClass myCl;  
  
//     // explicit invocation
```

```
// B b1(myCl);
// // implicit conversion from MyClass to B
// B b2= myCl;
// needB(myCl);

// // MyBool -> bool conversion
// MyBool myBool;
// if (myBool){};
// int myNumber = (myBool)? 1998: 2011;
// // implicit conversion
// int myNewNumber = myBool + myNumber;
// auto myTen = (20*myBool -10*myBool)/myBool;

// std::cout << myTen << std::endl;

//}

// explicit

// #include <iostream>

// class A{};

// class B{};

// class MyClass{
// public:
// MyClass(){}
// explicit MyClass(A{})           // since C++98
// explicit operator B(){return B();} // new with C++11
// };

// void needMyClass(MyClass){}
// void needB(B__);

// struct MyBool{
// explicit operator bool(){return true;}
// };

// int main(){

// // A -> MyClass
// A a;

// // explicit invocation
// MyClass myClass1(a);
// // implicit conversion from A to MyClass
// MyClass myClass2=a;
// needMyClass(a);

// // MyClass -> B
// MyClass myCl;
```

```
// // explicit invocation
// B b1(myCl);
// // implicit conversion from MyClass to B
// B b2= myCl;
// needB(myCl);

// // MyBool -> bool conversion
// MyBool myBool;
// if (myBool){};
// int myNumber = (myBool)? 1998: 2011;
// // implicit conversion
// int myNewNumber = myBool + myNumber;
// auto myTen = (20*myBool -10*myBool)/myBool;

// std::cout << myTen << std::endl;

//}

// implicit

// #include <iostream>

// class A{};

// class B{};

// class MyClass{
// public:
// MyClass(){}
// MyClass(A){}
// operator B(){ return B{}; }
// };

// void needMyClass(MyClass){};
// void needB(B){};

// struct MyBool{
// operator bool(){return true;}
// };

// int main(){

// // A -> MyClass
// A a;

// // explicit invocation
// MyClass myClass1(a);
// // implicit conversion from A to MyClass
// MyClass myClass2 = a;
// needMyClass(a);
```

```

// // MyClass -> B
// MyClass myCl;

// // explicit invocation
// B b1(myCl);
// // implicit conversion from MyClass to B
// B b2 = myCl;
// needB(myCl);

// // MyBool -> bool conversion
// MyBool myBool;
// if (myBool){};
// int myNumber = (myBool)? 1998: 2011;
// // implicit conversion
// int myNewNumber = myBool + myNumber;
// int myTen = (20 * myBool - 10 * myBool) / myBool;

// std::cout << "myTen: " << myTen << std::endl;

// }
// =====
/*
Explanation#
Recall that the explicit keyword is solely responsible for preventing implicit conversions.
```

Hence, the trick is to simply remove explicit from the conversion constructor and operator. This will enable implicit conversions again.

That brings us to the end of this topic. Next on our list is the call operator.

```
*/
/*
```

Call Operator

Let's take a deeper dive into what function objects are.

We'll cover the following

Functors

Further information

By overloading the function call operator, (), we can call objects like ordinary function objects that may or may not have arguments. These special objects are known as function objects or, wrongly, as functors.

The best feature of function objects is that they can have a state. Since they are objects, data is stored inside them, but they can also be used as functions to return data.

Functors#

Functors are very similar to lambda functions. We can say that lambdas actually create anonymous functors.

Because of this, functors are used frequently in STL algorithms as arguments. These functors can then be applied to the data being passed to the STL functions.

`std::add`, `std::transform`, and `std::reduce` are just a few of the functions that can use functors and apply them to data.

A functor that takes a single argument is a unary functor.

A functor that takes two arguments is a binary functor.

Further information#

lambda functions

```
*/  
/*
```

- Example

In contrast to a function, a function object can have a state. The example in this lesson explains the point.

We'll cover the following

Operator overloading using parentheses

Explanation

Operator overloading using parentheses#

```
*/  
// ======  
// #include <algorithm>  
// #include <iostream>  
// #include <vector>  
  
// class SumMe{  
// public:  
  
//   SumMe(): sum(0){};  
  
//   void operator()(int x){  
//     sum += x;  
//   }  
  
//   int getSum() const {  
//     return sum;  
//   }  
// private:  
//   int sum;  
// };  
  
// int main(){  
  
//   std::cout << std::endl;  
  
//   std::vector<int> intVec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
//   SumMe sumMe = std::for_each(intVec.begin(), intVec.end(), SumMe());  
//   std::cout << "sumMe.getSum(): " << sumMe.getSum() << std::endl;  
  
//   std::cout << std::endl;
```

```
//  
// ======  
/*  
Explanation#  
The std::for_each call in line 27 is a special algorithm of the Standard Template Library.
```

It can return its callable. We invoke std::for_each with the function object SumMe and can, therefore, store the result of the function call directly in the function object.

In line 28, we used the sum of all calls which is the state of the function object.

Note: Lambda functions can also have a state.

Exercise#

Implement the functionality in the example we saw in the previous lesson with the help of a lambda function.

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

```
class SumMe{  
public:  
  
    SumMe(): sum(0){};
```

```
    void operator()(int x){  
        sum += x;  
    }
```

```
    int getSum() const {  
        return sum;  
    }
```

```
private:  
    int sum;  
};
```

```
int main(){
```

```
    std::cout << std::endl;
```

```
    std::vector<int> intVec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
    // Use a lambda function here and class sumMe is no longer needed  
    SumMe sumMe = std::for_each(intVec.begin(), intVec.end(), SumMe());  
    std::cout << "sumMe.getSum(): " << sumMe.getSum() << std::endl;
```

```
    std::cout << std::endl;
```

```
}
```

```
/*  
// ======  
// #include <algorithm>
```

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

// int main()
//{
// std::cout << std::endl;

// std::vector<int> intVec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// int sum{0};
// std::for_each(intVec.begin(), intVec.end(), [&sum](int x)
// {
//     sum += x; });

// std::cout << "sum: " << sum << std::endl;

// std::cout << std::endl;
// }
// =====
/*
```

Explanation#

First of all, the variable sum represents the state of the lambda function.

With C++14, the so-called initialization capture of lambdas is supported. sum{0} declares and initializes a variable of type int which is only valid in the scope of the lambda function.

The lambda function is used in line 12. Note that they are constant by default.

&sum Stores the address of last updated value of sum.

In the next lesson, we'll discuss which access rights are available for members of the class.

*/

/*

A Brief Introduction

In this lesson, we will take a look at why composition, aggregation, and association are useful concepts in OOP.

We'll cover the following

Interaction Between Classes

Connecting OOP and OOD

Relationships Between Classes

Interaction Between Classes#

By now, we've learned all we need to know about the creation and the behavior of a class. The concepts of inheritance and polymorphism taught us how to create dependent classes out of a base class.

The next step after object-oriented programming is object-oriented design (OOD). OOD deals with the use of different class objects to create the design of an application. Naturally, this means that independent classes would have to find a way of interacting with each other.

Connecting OOP and OOD#

That interaction is what this section is all about. Composition, aggregation, and association are the final concepts of object-oriented programming. They act as a bridge between OOP and OOD. These three techniques are used to link different classes together through a variety of relationships. Before we delve into these techniques, let's understand the different types of relationships they use

Relationships Between Classes#

There are two main class relationships we need to know. We'll study them one by one.

Part-of

In this relationship, one class is a component of another class. Given two classes, class A and class B, they are in a part-of relation if class A is a part of class B or vice-versa.

An instance of the component class can only be created inside the main class. In the example to the right, class B and class C have their own implementations, but their objects are only created once a class A object is created. Hence, part-of is a dependent relationship.

svg viewer

Class A contains objects of Class B and C.

svg viewer

The object of class exists independent of A.

Has-a

This is a slightly less concrete relationship between two classes. Class A and class B have a has-a relationship if one or both need the other's object to perform an operation, but both classes can exist independently of each other.

This implies that a class has a reference to the object of the other class, but does not decide the lifetime of the other class's referenced object.

Now that we've understood the relationships relevant to this section, let's start off with composition.

*/

/*

Composition

In this lesson, we'll learn how can we achieve composition in C++.

We'll cover the following

Example

Implementation

Composition is accessing other classes objects in your class and owner class owns the object and is responsible for its lifetime. Composition relationships are Part-of relationships where the part must constitute part of the whole object. We can achieve composition by adding smaller parts of other classes to make a complex unit.

So, what makes composition unique?

In composition, the lifetime of the owned object depends on the lifetime of the owner.

Example#

A car is composed of an engine, tyres, and doors. In this case, a Car owned these objects so a Car is an Owner class and tyres,doors and engine classes are Owned classes.

```
/*
// =====
// #include <iostream>
// using namespace std;

// class Engine{
// int capacity;

// public:
// Engine(){
// capacity = 0;
// }

// Engine(int cap) {
// capacity = cap;
// }

// void Engine_details() {
// cout << "Engine details: " << capacity << endl;
// }
//};

// class Tyres{
// int No_of_tyres;

// public:
// Tyres(){
// No_of_tyres = 0;
// }

// Tyres(int nt) {
// No_of_tyres = nt;
// }

// void Tyre_details() {
// cout << "Number of tyres: " << No_of_tyres << endl;
// }
//};

// class Doors{
// int No_of_doors;

// public:
// Doors(){
// No_of_doors = 0;
// }

// Doors(int nod) {
// No_of_doors = nod;
// }
```

```

// }

// void Door_details() {
//   cout << "Number of Doors: " << No_of_doors << endl;
// }
//};

// class Car{
//   Engine Eobj;
//   Tyres Tobj;
//   Doors Dobj;
//   string color;

// public:
//   Car(Engine eng, Tyres tr, int dr, string col)
//     : Eobj(eng), Tobj(tr), Dobj(dr){

//   color = col;
// }

// void Car_detail(){
//   Eobj.Engine_details();
//   Tobj.Tyre_details();
//   Dobj.Door_details();
//   cout << "Car color: " << color << endl;
// }
//};

// int main(){
//   Engine Eobj(1600);
//   Tyres Tobj(4);
//   Doors Dobj(4);
//   Car Cobj(Eobj, Tobj, 4, "Black");
//   Cobj.Car_detail();
// }
// =====
/*

```

We have created a Car class which contains the objects of Engine, Tyres and Doors classes. Car class owns the objects and is responsible for their lifetime. When Car dies, so does tyre, engine and doors too.

```

*/
/*

```

Aggregation

In this lesson, we'll learn a new way of linking different classes.

We'll cover the following

Independent Lifetimes

Example

Aggregation is very similar to composition. It also follows the Has-A model. This creates a parent-child relationship between two classes, with one class owning the object of another.

So, what makes aggregation unique?

Independent Lifetimes#

In aggregation, the lifetime of the owned object does not depend on the lifetime of the owner.

The owner object could get deleted but the owned object can continue to exist in the program. In composition, the parent contains a child object. This bounds the child to its parent. In aggregation, the parent only contains a reference to the child, which removes the child's dependency.

You can probably guess from the illustration above that we'll need pointers to implement aggregation.

Example#

Let's take the example of people and their country of origin. Each person is associated with a country, but the country can exist without that person:

```
/*
// =====
// #include <iostream>
// #include <string>
// using namespace std;

// class Country{
//   string name;
//   int population;

//   public:
//   Country(string n, int p){
//     name = n;
//     population = p;
//   }
//   string getName(){
//     return name;
//   }
// };

// class Person {
//   string name;
//   Country* country; // A pointer to a Country object

//   public:
//   Person(string n, Country* c){
//     name = n;
//     country = c;
//   }

//   string printDetails(){
//     cout << "Name: " << name << endl;
//     cout << "Country: " << country->getName() << endl;
//   }
// };

// int main() {
```

```

// Country* country = new Country("Utopia", 1);
// {
//   Person user("Darth Vader", country);
//   user.printDetails();
// }
// // The user object's lifetime is over

// cout << country->getName() << endl; // The country object still exists!
//}
=====
/*

```

As we can see, the country object lives on even after the user goes out of scope. This creates a looser relationship between the two in comparison to composition.

In the next lesson, we will explore the third type of linkage between classes; association.

```

*/
/*

```

Association

In this lesson, we'll learn about the relationship between two unrelated objects that is the association in C++.

We'll cover the following

Example

In object-oriented programming, association is the relationship between the unrelated objects of the classes. Objects lifespan are not directly tied to each other like in composition and aggregation. To clearly understand the concept of the association we'll take an example of student and teacher class.

Example#

The teacher has a relationship with his students, but it's not a part-of relationship as in composition. A teacher can add many students to a class, and a student can be registered in many courses. Neither of the objects(teacher and student) lifespan is tied to the other as any student can leave the class of specific teacher and similarly any teacher can change the course which he's currently teaching. Let's look at the implementation:

```

*/
//
=====
=====
// class Teacher; // Making this friend of a student class

// class Student {
// private:
//   string Std_name;
//   vector<Teacher *> tr;
//   void addTeacher(Teacher * teach);

// public:
//   Student(string name) {
//     Std_name = name;
//   }

//   string getName() const {

```

```
//    return Std_name;
// }

// friend ostream& operator<<(ostream &out, const Student &std);

// // Making teacher friend of this class to access addTeacher function
//     friend class Teacher;
// };

// class Teacher {
// private:
//     string tr_name;
//     vector<Student *> stdnt;

// public:
//     Teacher (string name) {
//         tr_name = name;
//     }

//     void addStudent(Student *st) {
//         // Teacher will add this student to course
//         // stdnt.push_back(st);

//         // Student will also add this teacher for connection
//         // st->addTeacher(this);
//     }

//     friend ostream& operator<<(ostream &out, const Teacher &tchr) {
//         int length = tchr.stdnt.size();
//         if (length == 0) {
//             out << tchr.tr_name << " is not teaching any class";
//             return out;
//         }

//         out << tchr.tr_name << " is teaching: ";
//         for (int count = 0; count < length; ++count)
//             out << tchr.stdnt[count]->getName() << ' ';

//         return out;
//     }

//     string getName() const {
//         return tr_name;
//     }
// };

// void Student::addTeacher(Teacher *teach) {
//     tr.push_back(teach);
// }

// ostream& operator<<(ostream &out, const Student &std) {
```

```

//     int length = std.tr.size();
//     if (length == 0) {
//         out << std.getName() << " is not registered in any course";
//         return out;
//     }

//     out << std.Std_name << " is taught by: ";
//     for (int count = 0; count < length; ++count)
//         out << std.tr[count]->getName() << ' ';

//     return out;
// }

// int main() {
//     // Creating a Students outside the scope of the Teacher
//     Student *s1 = new Student("John");
//     Student *s2 = new Student("Stacy");
//     Student *s3 = new Student("Sarah");

//     Teacher *t1 = new Teacher("Henry");
//     Teacher *t2 = new Teacher("Neil");
//     Teacher *t3 = new Teacher("Steve");

//     t1->addStudent(s1);
//     t2->addStudent(s1);
//     t1->addStudent(s3);

//     cout << *t1 << endl;
//     cout << *t2 << endl;
//     cout << *t3 << endl;
//     cout << *s1 << endl;
//     cout << *s2 << endl;
//     cout << *s3 << endl;
// }
// =====
/*

```

In the above example, student and teacher objects are sharing a connection between each other. As clearly seen in the example that teacher and student can exist independently too.

```

*/
// =====
=====
// =====
// =====
// =====
=====
```

//

=====

=====

//

=====

=====

/*

C++ STL*****

The History#

C++ and therefore the standard library have a long history. C++ started in the 1980s of the last millennium and ended now in 2017. Anyone who knows about software development knows how fast our domain evolves. So 30 years is a very long period. You may not be so astonished that the first components of C++, like I/O streams, were designed with a different mindset than the modern Standard Template Library (STL). This evolution in the area of software development in the last 30 years, which you can observe in the C++ standard library, is also an evolution in the way software problems are solved. C++ started as an object-oriented language, then incorporated generic programming with the STL and now has adopted a lot of functional programming ideas.

widget

The first C++ standard library from 1998 had three components. Those were the previously mentioned I/O streams, mainly for file handling, the string library, and the Standard Template Library. The Standard Template Library facilitates the transparent application of algorithms on containers.

The history continues in the year 2005 with Technical Report 1 (TR1). The extension to the C++ library ISO/IEC TR 19768 was not an official standard, but almost all of the components became part of C++11. These were, for example, the libraries for regular expressions, smart pointers, hash tables, random numbers and time, based on the boost libraries.

In addition to the standardization of TR1, C++11 got one new component: the multithreading library.

C++14 was only a small update to the C++11 standard. Therefore only a few improvements to the already existing libraries for smart pointers, tuples, type traits, and multithreading were added.

What comes next in the C++ standard library? With C++17 and C++20 we will get two new standards. C++17 is already done. C++17 includes libraries for the file system and the two new data types std::any and std::optional. With C++20 we might get libraries for network programming; with Concepts Lite we might get a type system for template parameters and better support for multithreading.

Now that we know the history of C++, let's talk about the various utilities it provides.

Utilities

To use C++ to its full potential, we must use the multitude of utilities it provides.

As C++11 has a lot of libraries, it is often not so easy to find the convenient one for each use case.

Utilities are libraries which have a general focus and therefore can be applied in many contexts.

Calculating the Minimum and Maximum#

Examples of utilities are functions to calculate the minimum or maximum of values or functions to swap or move values.

Functional Programming#

Other utilities are std::function and std::bind. With std::bind you can easily create new functions from existing ones. In order to bind them to a variable and invoke them later, you have std::function.

Pairs#

With `std::pair` and its generalization `std::tuple` you can create heterogeneous pairs and tuples of arbitrary length.

Reference Wrappers#

The reference wrappers `std::ref` and `std:: cref` are pretty handy. One can use them to create a reference wrapper for a variable, which for `std:: cref` is `const`.

Smart Pointers#

Of course, the highlights of the utilities are the smart pointers. They allow explicit automatic memory management in C++. You can model the concept of explicit ownership with `std::unique_ptr` and model shared ownership with `std::shared_ptr`. `std::shared_ptr` uses reference counting for taking care of its resource. The third one, `std::weak_ptr`, helps to break the cyclic dependencies among `std::shared_ptr`s, the classic problem of reference counting.

Type Traits#

The type traits library is used to check, compare and manipulate type information at compile time.

Multithreading#

The `time` library is an important addition of the new multithreading capabilities of C++. But it is also quite handy to make performance measurements.

Values of Datatypes#

With `std::any`, `std::optional`, and `std::variant`, we get with C++17 three special datatypes that can have any, an optional value, or a variant of values respectively.

Now, let's talk about the components of the C++ Standard Library.

Components

The three components of the STL are:

The Standard Template Library (STL) consists of three components from a bird's-eye view. Those are containers, algorithms that run on the containers, and iterators that connect both of them. This abstraction of generic programming enables you to combine algorithms and containers uniquely. The containers have only minimal requirements for their elements.

Containers#

The C++ Standard Library has a rich collection of containers. From a bird's eye, we have sequential and associative containers. Associative containers can be classified as ordered or unordered associate containers.

Sequential Containers#

Each of the sequential containers has a unique domain, but in 95% of the use cases `std::vector` is the right choice. `std::vector` can dynamically adjust its size, automatically manages its memory and provides you with outstanding performance. In contrast, `std::array` is the only sequential container that cannot adjust its size at runtime. It is optimized for minimal memory and performance overhead. While `std::vector` is good at putting new elements at its end, you should use `std::deque` to put an element also at the beginning. With `std::list` being a doubly-linked list and `std::forward_list` as a singly linked list, we have two additional containers that are optimized for operations at arbitrary positions in the container, with high performance.

Associative Containers#

Associative containers are containers of key-value pairs. They provide their values by their respective key. A typical use case for an associative container is a phone book, where you use the key family name to retrieve

the value phone number. C++ has eight different associative containers. On one side there are the associative containers with ordered keys: std::set, std::map, std::multiset and std::multimap. On the other side there are the unordered associative containers: std::unordered_set, std::unordered_map, std::unordered_multiset and std::unordered_multimap.

Ordered and Unordered Associative Containers

Let's look first at the ordered associative containers. The difference between std::set and std::map is that the former has no associated value. The difference between std::map and std::multimap is, that the latter can have more than one identical key. This naming conventions also holds for the unordered associative containers, which have a lot in common with the ordered ones. The key difference is performance. While the ordered associative containers have an access time depending logarithmically on the number of elements, the unordered associative containers allow constant access time. Therefore the access time of the unordered associative containers is independent of their size.

Container adapters#

Container adapters provide a simplified interface to the sequential containers. C++ has std::stack, std::queue and std::priority_queue.

Iterators#

Iterators act as glue between the containers and the algorithms. The container creates them. As generalized pointers, you can use them to iterate forward and backward or to an arbitrary position in the container. The type of iterator you get depends on the container. If you use an iterator adapter, you can directly access a stream.

Algorithms#

The STL gives you more than 100 algorithms. By specifying the execution policy, you can run most of the algorithms sequential, parallel, or parallel and vectorized. Algorithms operate on elements or a range of elements. Two iterators define a range. The first one defines the beginning, the second one, called end iterator, defines the end of the range. It's important to know that the end iterator points to one element past the end of the range.

The algorithms can be used in a wide range of applications. You can find elements or count them, find ranges, compare or transform them. There are algorithms to generate, replace or remove elements from a container. Of course, you can sort, permute or partition a container or determine the minimum or maximum element of it. A lot of algorithms can be further customized by callables like functions, function objects or lambda-functions. The callables provide special criteria for search or elements transformation. They highly increase the power of the algorithm.

In the next lesson, I will give an overview of the C++ Standard Library.

Numeric Functions

Numerical computations are very powerful in programming, and C++ provides a diverse range of mathematical functions.

There are two libraries for numerics in C++: the random numbers library and the mathematical functions, which C++ inherited from C.

The random numbers library consists of two parts. On one side there is the random number generator, on the other hand, the distribution of the generated random numbers. The random number generator generates a stream of numbers between a minimum and a maximum value, which the random number distribution maps onto the concrete distribution.

Because of C, C++ has a lot of standard mathematical functions. For example, there are logarithmic, exponential and trigonometric functions.

Text Processing

C++ handles text by using `std::string` and `std::string_view` utilities.

We'll cover the following

String

String View

Regular Expression

With strings and regular expressions, C++ has two powerful libraries to process text.

String#

`std::string` possesses a rich collection of methods to analyze and modify its text. Because it has a lot in common with an `std::vector` of characters, the algorithms of the STL can be used for `std::string`. `std::string` is the successor of the C string but a lot easier and safer to use. C++ strings manage their memory themselves.

String View#

In contrast to a `std::string` a `std::string_view` is quite cheap to copy. A `std::string_view` is a non-owning reference to a `std::string`.

Regular Expression#

Regular expression is a language for describing text patterns. You can use regular expressions to determine whether a text pattern is present once or more times in a particular text. But that's not all. Regular expressions can be used to replace the content of the matched patterns with a text.

In the next lesson, we will talk about input and output in the C++ Standard Library.

Input, Output and Filesystems

iostream Library#

I/O streams library is a library, present from the beginning of C++, that allows communication with the outside world.

Communication means in this concrete case, that the extraction operator (`>>`) enables it to read formatted or unformatted data from the input stream, and the insertion operator (`<<`) enables it to write the data on the output stream. Data can be formatted using manipulators.

The stream classes have an elaborate class hierarchy. Two stream classes are significant: First, string streams allow you to interact with strings and streams. Second, file streams allow you to read and write files easily. The state of streams is kept in flags, which you can read and manipulate.

By overloading the input operator and output operator, your class can interact with the outside world like a fundamental data type.

Filesystem Library#

In contrast to the I/O streams library, filesystem library was added to the C++ Standard with C++17. The library is based on the three concepts file, file name, and path. Files can be directories, hard links, symbolic links or regular files. Paths can be absolute or relative.

The filesystem library supports a powerful interface for reading and manipulating the filesystem.

Multithreading

C++ gets with the 2011 published C++ standard a multithreading library. This library has basic building blocks like atomic variables, threads, locks, and condition variables. That's the base on which future C++ standards can build higher abstractions. But C++11 already knows tasks, which provide a higher abstraction than the cited basic building blocks.

At a low level, C++11 provides for the first time a memory model and atomic variables. Both components are the foundation for well-defined behavior in multithreading programming.

Threads#

A new thread in C++ will immediately start its work. It can run in the foreground or background and gets its data by copy or reference.

Shared Variables#

The access to shared variables between threads has to be coordinated. This coordination can be done in different ways with mutexes or locks. But often it's sufficient to protect the initialization of the data as it will be immutable during its lifetime.

Thread-Local Variable#

Declaring a variable as thread-local ensures that a thread gets its own copy, so there is no conflict.

Condition variables#

Condition variables are a classic solution to implement sender-receiver workflows. The key idea is that the sender notifies the receiver when it's done with its work so that the receiver can start.

Tasks#

Tasks have a lot in common with threads. But while a programmer explicitly creates a thread, a task will be implicitly created by the C++ runtime. Tasks are like data channels. The promise puts data into the data channel; the future picks the value up. The data can be a value, an exception or simply a notification.

Introduction

In this lesson, we'll define the three steps needed to use libraries in C++.

To use a library in a file you have to perform three steps. At first, you have to include the header files with the `#include` statement, so the compiler knows the names of the library. Because the names of the C++ standard library are in the namespace `std`, you can use them in the second step fully qualified or you have to import them in the global namespace. The third and final step is to specify the libraries for the linker to get an executable. This third step is often not necessary. The three steps are explained in the next couple of lesson.

Including Header Files

The first step in using libraries is to include the header files. Let's find out how.

The preprocessor includes the file, following the `#include` statement. That is most of the time a header file. The header files will be enclosed in angular brackets:

Specify all necessary header files

The compiler is free to add additional headers to the header files. So your program may have all the necessary headers although you didn't specify all of them. It's not recommended to rely on this feature. All needed headers should always be explicitly specified. Otherwise, a compiler upgrade or code porting may provoke a compilation error.

In the next lesson, I will discuss in detail the second step of using applications in the C++ Standard Library – using namespaces.

Using Namespaces

Namespaces have to be written exactly as they are, however, the 'using' method allows us to make namespaces simpler.

If you use qualified names, you have to use them exactly as defined. For each namespace you must put the scope resolution operator `::`. More libraries of the C++ standard library use nested namespaces.

```
#include <iostream>
#include <chrono>
...
std::cout << "Hello world:" << std::endl;
auto timeNow= std::chrono::system_clock::now();
```

Unqualified Use of Names#

You can use names in C++ with the using declaration and the using directive.

Using Declaration#

A using declaration adds a name to the visibility scope, in which you applied the using declaration:

```
#include <iostream>
#include <chrono>
...
using std::cout;
using std::endl;
using std::chrono::system_clock;
...
cout << "Hello world:" << endl; // unqualified name
auto timeNow= now();          // unqualified name
```

The application of a using declaration has the following consequences:

An ambiguous lookup and therefore a compiler error occurs if the same name was declared in the same visibility scope.

If the same name was declared in a surrounding visibility scope, it will be hidden by the using declaration

Using Directive#

The using directive permits it to use all names of a namespace without qualification.

```
#include <iostream>
#include <chrono>
...
using namespace std;
...
cout << "Hello world:" << endl;      // unqualified name
auto timeNow= chrono::system_clock::now(); // partially qualified name
```

i Use using directives with great care in source files

using directives should be used with great care in source files, because by the directive using namespace std all names from std becomes visible. That includes names, which accidentally hide names in the local or surrounding namespace.

Don't use using directives in header files. If you include a header with the using namespace std directive, all names from std become visible.

Namespace Alias#

A namespace alias defines a synonym for a namespace. It's often convenient to use an alias for a long namespace or nested namespaces:

```
#include <chrono>
...
namespace sysClock= std::chrono::system_clock;
auto nowFirst= sysClock::now();
auto nowSecond= std::chrono::system_clock::now();
```

Building an Executable

In this lesson, we'll briefly examine the final step of library usage: linking our executable with libraries.

It is only seldom necessary to link explicitly against a library. That sentence is platform dependent. For example, with the current g++ or clang++ compiler, you have to link against the pthread library to get the multithreading functionality.

```
g++ -std=c++14 thread.cpp -o thread -pthread
```

Now that we know how to use libraries, let's dive into the C++ Standard library and learn about the various utilities it has to offer.

The min, max and minmax functions

This family of functions allows us to find the minimum and maximum in a set of data. Let's find out how.

Required Headers#

The many variations of the min, max, and minmax functions apply to values and initializer lists. These functions need the header <algorithm>. Nearly the same holds for the functions std::move, std::forward and std::swap. You can apply them to arbitrary values. These three functions are defined in the header <utility>.

std::min, std::max and std::minmax#

The functions std::min, std::max and std::minmax, defined in the header <algorithm>, act on values and initialiser lists and give you the requested value back as result. In the case of std::minmax, you get an std::pair. The first element of the pair is the minimum, the second is the maximum of the values. By default, the less operator (<) is used, but you can specify your comparison operator. This function needs two arguments and returns a boolean. Functions that either return true or false are called predicates.

```
*/
```

```
// =====
// minMax.cpp
// #include <iostream>
// #include <algorithm>
// ...
// using std::cout;
// ...
// int main()
//{
// cout << "std::min(2011, 2014):\t\t\t";
// cout << std::min(2011, 2014) << "\n"; // 2011
//
// cout << "std::min({3, 1, 2011, 2014, -5}):\t";
// cout << std::min({3, 1, 2011, 2014, -5}) << "\n"; // -5
//
// cout << "std::min(-10, -5, [](...)){...}:\t\t";
// cout << std::min(-10, -5, [](int a, int b)
// { return std::abs(a) < std::abs(b); })
// << "\n\n"; // -5
//
// std::pair<int, int> pairInt = std::minmax(2011, 2014);
// auto pairSeq = std::minmax({3, 1, 2011, 2014, -5});
// auto pairAbs = std::minmax({3, 1, 2011, 2014, -5}, [](int a, int b)
```

```

// { return std::abs(a) < std::abs(b); });

// cout << "pairInt.first, pairInt.second:\t\t";
// cout << pairInt.first << ", " << pairInt.second << "\n"; // 2011,2014

// cout << "pairSeq.first, pairSeq.second:\t\t";
// cout << pairSeq.first << ", " << pairSeq.second << "\n"; // -5,2014

// cout << "pairAbs.first, pairAbs.second:\t\t ";
// cout << pairAbs.first << ", " << pairAbs.second << "\n"; // 1,2014

// return 0;
//}
//=====
/*

```

The table provides an overview of the functions `std::min`, `std::max` and `std::minmax`

Function	Description
<code>min(a, b)</code>	Returns the minimal value of a and b.
<code>min(a, b, comp)</code>	Returns the minimal value of a and b according to the predicate comp.
<code>min(initializer list)</code>	Returns the minimal value of the initializer list.
<code>min(initializer list, comp)</code>	Returns the minimal value of the initializer list according to the predicate comp.
<code>max(a, b)</code>	Returns the maximal value of a and b.
<code>max(a, b, comp)</code>	Returns the maximal value of a and b according to the predicate comp.
<code>max(initializer list)</code>	Returns the maximal value of the initializer list.
<code>max(initializer list, comp)</code>	Returns the maximal value of the initializer list according to the predicate comp.
<code>minmax(a, b)</code>	Returns the minimal and maximal value of a and b.
<code>minmax(a, b, comp)</code>	Returns the minimal and maximal value of a and b according to the predicate comp according to the predicate comp.
<code>minmax(initializer list)</code>	Returns the minimal and maximal value of the initializer list.
<code>minmax(initializer list, comp)</code>	Returns the minimal and maximal value of the initializer list according to the predicate comp.

The variations of `'std::min'`, `'std::max'` and `'std::minmax'`

Now, let's talk about another useful function the `std::move`.

Move vs. Copy

The function `std::move`, defined in the header `<utility>`, empowers the compiler to move its resource. In the so-called move semantic, the values from the source object are moved to the new object. Afterward, the source is in a well-defined but not specified state. Most of the times that is the default state of the source. By using `std::move`, the compiler converts the source arg to a rvalue reference:
`static_cast<std::remove_reference<decltype(arg)>::type&&>(arg)`.

The subtle difference is that if we create a new object based on an existing one, the copy semantic will copy the elements of the existing resource, whereas the move semantic will move the elements of the resource. So, of course, copying is expensive and moving is cheap. But there are additional serious consequences.

With the copy semantic, it is possible that a `std::bad_alloc` will be thrown because our program is out of memory.

The source of the move operation is in a “valid but unspecified state” afterward.

The second point can be explained well by std::string example below.

If the compiler can not apply the move semantic, it falls back to the copy semantic.

Vector elements#

```
#include <utility>
//...
std::vector<int> myBigVec(10000000, 2011);
std::vector<int> myVec;

myVec = myBigVec;      // copy semantic
myVec = std::move(myBigVec); // move semantic
*/
// =====
// #include <iostream>
// #include <utility>

/// Driver code
// int main()
//{
// std::string str1 = "abcd";
// std::string str2 = "efgh";
// std::cout << "str1: " << str1 << std::endl;
// std::cout << "str2: " << str2 << "\n\n";

// // Copying
// str2 = str1; // copy semantic
// std::cout << "After copying" << std::endl;
// std::cout << "str1: " << str1 << std::endl;
// std::cout << "str2: " << str2 << "\n\n";

// str1 = "abcd";
// str2 = "efgh";

// // Moving
// str2 = std::move(str1);
// std::cout << "After moving" << std::endl;
// std::cout << "str1: " << str1 << std::endl;
// std::cout << "str2: " << str2 << "\n\n";
// }

// =====
/*
```

In line 22, str1 is empty after the move operation. This is not guaranteed but is often the case. We explicitly requested the move semantic with the function std::move. The compiler will automatically perform the move semantic if it is sure that the source of the move semantic is not needed anymore.

Classes#

A class supports copy semantics if the class has a copy constructor and a copy assignment operator.

A class supports move semantics if the class has a move constructor and a move assignment operator.

If a class has a copy constructor, it should also have a copy assignment operator. The same holds true for the move constructor and move assignment operator.

User-defined data types#

User-defined data types can support the move and copy semantics as well.

```
class MyData{  
    MyData(MyData&& m) = default; // move constructor  
    MyData& operator = (MyData&& m) = default; // move assignment  
    MyData(const MyData& m) = default; // copy constructor  
    MyData& operator = (const MyData& m) = default; // copy assignment  
};
```

 To move is cheaper than to copy

The move semantic has two advantages. Firstly, it is often a good idea to use cheap moving instead of expensive copying. So there is no superfluous allocation and deallocation of memory necessary. Secondly, there are objects, which can not be copied, e.g., a thread or a lock.

The function std::forward, defined in the header <utility>, empowers you to write function templates, which can identically forward their arguments. Typical use cases for std::forward are factory functions or constructors. Factory functions are functions which create an object and must therefore identically pass the arguments. Constructors often use their arguments to initialize their base class with identical arguments. So std::forward is the perfect tool for authors of generic libraries:

```
*/  
/*  
// forward.cpp  
...  
#include <utility>  
...  
using std::initialiser_list;  
  
struct MyData{  
    MyData(int, double, char){};  
};  
  
template <typename T, typename... Args>  
T createT(Args&&... args){  
    return T(std::forward<Args>(args)... );  
}  
...  
  
int a= createT<int>();  
int b= createT<int>(1);  
  
std::string s= createT<std::string>("Only for testing.");  
MyData myData2= createT<MyData>(1, 3.19, 'a');  
  
typedef std::vector<int> IntVec;  
IntVec intVec= createT<IntVec>(initialiser_list<int>({1, 2, 3}));
```

The function template `createT` has to take their arguments as a universal reference: `Args&&... args``. A universal reference or also called forwarding reference is an rvalue reference in a type deduction context.

 `std::forward` in combination with variadic templates allows completely generic functions
If you use `std::forward` together with variadic templates, you can define completely generic function templates. Your function template can accept an arbitrary number of arguments and forward them unchanged.

Swap

Swapping is made simple in C++ using `std::swap`.

With the function `std::swap` defined in the header `<utility>`, you can easily swap two objects. The generic implementation in the C++ standard library internally uses the function `std::move`.

```
// swap.cpp
...
#include <utility>
...
template <typename T>
inline void swap(T& a, T& b){
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

std::bind and std::function

Programmers can use this pair of utilities to create and bind functions to variables.

The two functions `std::bind` and `std::functions` fit very well together. While `std::bind` enables you to create new function objects on the fly, `std::function` takes these temporary function objects and binds them to a variable. Both functions are powerful tools from functional programming and need the header `<functional>`.

```
//=====================================================================
// #include <iostream>
// #include <functional>

/// for placeholder _1 and _2
// using namespace std::placeholders;

// using std::bind;
// using std::function;

// double divMe(double a, double b) { return a / b; }

// int main()
//{
// std::cout << std::boolalpha;
```

```

// function<double(double, double)> myDiv1 = bind(divMe, _1, _2);
// function<double(double)> myDiv2 = bind(divMe, 2000, _1);
// std::cout << (divMe(2000, 10) == myDiv1(2000, 10)) << '\n';
// std::cout << (myDiv1(2000, 10) == myDiv2(10));
// }
// =====
/*

```

Behavior of std::bind and std::function

Let's take a step deeper into the workings of std::bind and std::function.

We'll cover the following

std::bind
std::function
std::bind#

Because of std::bind, you can create function objects in a variety of ways:

bind the arguments to an arbitrary position,
change the order of the arguments,
introduce placeholders for arguments,
partially evaluate functions,
invoke the newly created function objects, use them in the algorithm of the STL or store them in std::function.
std::function#

std::function can store arbitrary callables in variables. It's a kind of polymorphic function wrapper. A callable may be a lambda function, a function object, or a function. std::function is always necessary and can't be replaced by auto, if you have to specify the type of the callable explicitly.

```

*/
// =====
// dispatchTable.cpp
// #include <iostream>
// #include <map>
// #include <functional>
// using std::make_pair;
// using std::map;

// int main()
// {
//   map<const char, std::function<double(double, double)>> tab;
//   tab.insert(make_pair('+', [](double a, double b)
//   //           { return a + b; }));
//   tab.insert(make_pair('-', [](double a, double b)
//   //           { return a - b; }));
//   tab.insert(make_pair('*', [](double a, double b)
//   //           { return a * b; }));
//   tab.insert(make_pair('/', [](double a, double b)
//   //           { return a / b; }));

//   std::cout << "3.5 + 4.5\t= " << tab['+'](3.5, 4.5) << "\n"; // 3.5 + 4.5      = 8
//   std::cout << "3.5 - 4.5\t= " << tab['-'](3.5, 4.5) << "\n"; // 3.5 - 4.5      = -1
//   std::cout << "3.5 * 4.5\t= " << tab['*'](3.5, 4.5) << "\n"; // 3.5 * 4.5     = 15.75

```

```
// std::cout << "3.5 / 4.5\t= " << tab['/'](3.5, 4.5) << "\n"; // 3.5 / 4.5      = 0.777778
```

```
// return 0;  
//}  
// ======  
/*
```

The type parameter of `std::function` defines the type of callables `std::function` will accept.

Function type	Return type	Type of the arguments
double(double, double)		double double
int()	int	
double(int, double)	double	int, double
void()		

Return type and type of the arguments

Pairs

The idea of a pair of values often comes handy in programming. C++ allows us to make these pairs.

We'll cover the following

`std::make_pair`

With `std::pair`, you can build pairs of arbitrary types. The class template `std::pair` needs the header `<utility>`. `std::pair` has a default, copy and move constructor. Pair objects can be swapped: `std::swap(pair1, pair2)`.

Pairs will often be used in the C++ library. For example, the function `std::minmax` returns its result as a pair, the associative container `std::map`, `std::unordered_map`, `std::multimap` and `std::unordered_multimap` manage their key/value association in pairs.

To get the elements of a pair `p`, you can either access it directly or via an index. So, with `p.first` or `std::get<0>(p)` you get the first, with `p.second` or `std::get<1>(p)` you get the second element of the pair.

Pairs support the comparison operators `==`, `!=`, `<`, `>`, `<=` and `>=`. If you compare two pairs for identity, at first the members `pair1.first` and `pair2.first` will be compared and then `pair1.second` and `pair2.second`. The same strategy holds for the other comparison operators.

`std::make_pair#`

C++ has the practical help function `std::make_pair` to generate pairs, without specifying their types. `std::make_pair` automatically deduces their types.

```
*/
```

```
// ======  
// pair.cpp  
// #include <iostream>  
// #include <utility>  
// using namespace std;  
  
// int main()  
// {  
//     pair<const char *, double> charDoub("str", 3.14);  
//     pair<const char *, double> charDoub2 = make_pair("str", 3.14);
```

```

// auto charDoub3 = make_pair("str", 3.14);

// cout << charDoub.first << ", " << charDoub.second << "\n"; // str, 3.14
// charDoub.first = "Str";
// get<1>(charDoub) = 4.14;
// cout << charDoub.first << ", " << charDoub.second << "\n"; // Str, 4.14

// return 0;
//}
//=====
/*

```

Tuples

Tuples extend the principles of a pair to a broader perspective. Find out more in this lesson.

You can create tuples of arbitrary length and types with `std::tuple`. The class template needs the header `<tuple>`. `std::tuple` is a generalization of `std::pair`. You can convert between tuples with two elements and pairs. The tuple has, like his younger brother `std::pair`, a default, a copy, and a move constructor. You can swap tuples with the function `std::swap`.

The i -th element of a tuple t can be referenced by the function template `std::get: std::get<i-1>(t)`. By `std::get<type>(t)` you can directly refer to the element of the type `type`.

Tuples support the comparison operators `==`, `!=`, `<`, `>`, `<=` and `>=`. If you compare two tuples, the elements of the tuples will be compared lexicographically. The comparison starts at index 0.

`std::make_tuple`

The helper function `std::make_tuple` is quite convenient for the creation of tuples. You don't have to provide the types. The compiler automatically deduces them.

```

*/
//=====
// tuple.cpp
// #include <iostream>
// #include <tuple>
// using std::get;

// int main()
//{
// std::tuple<std::string, int, float> tup1("first", 3, 4.17f);
// auto tup2 = std::make_tuple("second", 4, 1.1);

// std::cout << get<0>(tup1) << ", " << get<1>(tup1) << ", "
// << get<2>(tup1) << std::endl; // first, 3, 4.17

// std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
// << get<2>(tup2) << std::endl; // second, 4, 1.1

// std::cout << (tup1 < tup2) << std::endl; // true

```

```

// get<0>(tup2) = "Second";

// std::cout << get<0>(tup2) << "," << get<1>(tup2) << ","
//      << get<2>(tup2) << std::endl; // Second, 4, 1.1

// std::cout << (tup1 < tup2) << std::endl; // false

// auto pair = std::make_pair(1, true);
// std::tuple<int, bool> tup = pair;

// std::cout << get<0>(tup) << std::endl;
// std::cout << get<1>(tup) << std::endl;

// std::cout << std::boolalpha;
// std::cout << get<1>(tup) << std::endl;

// return 0;
//}

// =====
/*
std::tie and std::ignore#
std::tie enables you to create tuples, whose elements reference variables. With std::ignore you can explicitly ignore elements of the tuple.
*/
// =====
// tupleTie.cpp
// #include <iostream>
// #include <tuple>
// using namespace std;

// int main()
//{
//    int first = 1;
//    int second = 2;
//    int third = 3;
//    int fourth = 4;

//    cout << first << " " << second << " "
//        << third << " " << fourth << endl; // 1 2 3 4

//    auto tup = tie(first, second, third, fourth) // bind the tuple
//        = std::make_tuple(101, 102, 103, 104); // create the tuple
//        // and assign it
//    cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
//        << " " << get<3>(tup) << endl; // 101 102 103 104

//    cout << first << " " << second << " " << third << " "
//        << fourth << endl; // 101 102 103 104

//    first = 201;

```

```

// // get<1>(tup) = 202; //not neccessary as the first=201 had made the neccesary changes available

// cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
//     << " " << get<3>(tup) << endl; // 201 202 103 104

// cout << first << " " << second << " " << third << " "
//     << fourth << endl; // 201 202 103 104

// int a, b;
// tie(std::ignore, a, std::ignore, b) = tup;
// cout << a << " " << b << endl; // 202 104

// return 0;
// }
// =====
/*

```

Introduction

C++ takes reference functionality one step higher by introducing reference wrappers!

A reference wrapper is a copy-constructible and copy-assignable wrapper for a object of type `&`, which is defined in the header `<functional>`. So you have an object, that behaves like a reference, but can be copied. Contrary to classic references, `std::reference_wrapper` objects support two additional use cases:

You can use them in containers of the Standard Template Library. `std::vector<std::reference_wrapper<int>> myIntRefVector`

You can copy instances of classes, which have `std::reference_wrapper` objects. That is in general not possible with references.

To access the reference of a `std::reference_wrapper<int> myInt(1)`, the `get` method can be used: `myInt.get()`. You can use a reference wrapper to encapsulate and invoke a callable.

```

*/
// =====
// referenceWrapperCallable.cpp
// #include <iostream>
// #include <functional>

// void foo(){
//   std::cout << "Invoked" << std::endl;
// }

// int main() {
//   typedef void callableUnit();
//   std::reference_wrapper<callableUnit> refWrap(foo);

//   refWrap(); // Invoked
//   return 0;
// }
// =====
/*
std::ref and std::cref

```

In the last lesson, we were introduced to reference wrappers. Now, we will learn how to create them.

With the helper functions std::ref and std::cref you can easily create reference wrappers to variables. std::ref will create a non constant reference wrapper, std::cref a constant one:

```
/*
// =====
// referenceWrapperRefCref.cpp
// #include <iostream>
// #include <functional>

// void invokeMe(const std::string &s)
//{
// std::cout << s << ": const " << std::endl;
//}

// template <typename T>
// void doubleMe(T t)
//{
// t *= 2;
//}

// int main()
//{
// std::string s{"string"};

// invokeMe(std::cref(s)); // string

// int i = 1;
// std::cout << i << std::endl; // 1

// doubleMe(i);
// std::cout << i << std::endl; // 1S

// doubleMe(std::ref(i));
// std::cout << i << std::endl; // 2

// return 0;
//}
// =====
/*
```

So it's possible to invoke the function invokeMe, which gets a constant reference to an std::string, with a non-constant std::string s, which is wrapped in a std::cref(s). If I wrap the variable i in the helper function std::ref, the function template doubleMe will be invoked with a reference. So the variable i will be doubled.

Introduction

This section will deal with a very powerful new category called smart pointers. Let's begin.

```
/*
// =====
=====
/*
/*
```

Smart pointers are one of the most important additions to C++ because they empower you to implement explicit memory management in C++. Beside the deprecated `std::auto_ptr`, C++ offers three different smart pointers. They are defined in the header `<memory>`.

Firstly there is the `std::unique_ptr`, which models the concept of exclusive ownership. Secondly, there is the `std::shared_ptr`, who models the concept of shared ownership. Lastly, there is the `std::weak_ptr`. `std::weak_ptr` is not so smart, because it has only a thin interface. Its job is it to break cycles of `std::shared_ptr`. It models the concept of temporary ownership.

The smart pointers manage their resource according to the RAII idiom. So the resource is automatically released if the smart pointer goes out of scope.

i Resource Acquisition Is Initialization Resource Acquisition Is Initialization, short RAII, stands for a popular technique in C++, in which the resource acquisition and release are bound to the lifetime of an object. This means for the smart pointer that the memory is allocated in the constructor and deallocated in the destructor. You can use this technique in C++ because the destructor is called when the object goes out of scope.

Name	Standard	Description
<code>std::auto_ptr</code> (deprecated)	C++98	Owns exclusively the resource. Moves the resource while copying.
<code>std::unique_ptr</code>	C++11	Owns exclusively the resource. Can't be copied.
<code>std::shared_ptr</code>	C++11	Has a reference counter for the shared variable. Manages the reference counter automatically. Deletes the resource, if the reference counter is 0.
<code>std::weak_ptr</code>	C++11	Helps to break cycles of <code>std::shared_ptr</code> . Doesn't modify the reference counter.

Overview smart pointers

Unique Pointers

Introduction#

An `std::unique_ptr` automatically and exclusively manages the lifetime of its resource according to the RAII idiom. `std::unique_ptr` should be our first choice since it functions without memory or performance overhead.

`std::unique_ptr` exclusively controls its resource. It automatically releases the resource if it goes out of scope. No copy semantics are required, and it can be used in containers and algorithms of the Standard Template Library. `std::unique_ptr` is as cheap and fast as a raw pointer when no special delete function is used.

Characteristics#

Before we go into the usage of `std::unique_ptr`, here are its characteristics in a few bullet points.

The `std::unique_ptr`:

- can be instantiated with and without a resource.
- manages the life cycle of a single object or an array of objects.
- transparently offers the interface of the underlying resource.
- can be parametrized with its own deleter function.
- can be moved (move semantics).
- can be created with the helper function `std::make_unique`.

Replacement for `std::auto_ptr`#

⚠ Don't use `std::auto_ptr`

Classical C++98 has a smart pointer `std::auto_ptr`, which exclusively addresses the lifetime of a resource. `std::auto_ptr` has a conceptual issue. If we implicitly or explicitly copy an `std::auto_ptr`, the resource is moved. Therefore, rather than utilizing copy semantic, we have a hidden move semantic, leading to undefined behavior. So `std::auto_ptr` is deprecated in C++11 and removed in C++17. Instead, use `std::unique_ptr`. We can neither implicitly nor explicitly copy an `std::unique_ptr`; only moving it is possible.

The code below will generate an error since we are using the deprecated `auto_ptr`. These are the methods of `std::unique_ptr`:

Name	Description
<code>get</code>	Returns a pointer to the resource.
<code>get_deleter</code>	Returns the delete function.
<code>release</code>	Returns a pointer to the resource and releases it.
<code>reset</code>	Resets the resource.
<code>swap</code>	Swaps the resources.

Methods of `std::unique_ptr`

In the next lesson, we can examine the application of these methods.

Special Deleters#

`std::unique_ptr` can be parametrized with special deleters: `std::unique_ptr<int, MyIntDeleter>` up(new int(2011), myIntDeleter()). `std::unique_ptr` uses, by default, the deleter of the resource.

`std::make_unique`#

The helper function `std::make_unique` was unlike its sibling `std::make_shared`, and was “forgotten” in the C++11 standard. Therefore, `std::make_unique` was added with the C++14 standard. `std::make_unique` enables us to create an `std::unique_ptr` in a single step:

```
std::unique_ptr<int> uniqPtr1= std::make_unique<int>(2011);
auto uniqPtr2= std::make_unique<int>(2014);
```

Using `std::make_unique` in combination with automatic type deduction means typing is reduced to its bare minimum.

Always use `std::make_unique`.

If we use

```
func(std::make_unique<int>(2014), functionMayThrow());
func(std::unique_ptr<int>(new int(2011)), functionMayThrow());
```

and `functionMayThrow` throws, we have a memory leak with `new int(2011)` for this possible sequence of calls:

```
new int(2011)
functionMayThrow()
std::unique_ptr<int>(...)
```

Rarely, when we create `std::unique_ptr` in an expression and the compiler optimizes this expression, a memory leak may occur with an `std::unique_ptr` call. Using `std::make_unique` guarantees that no memory leak will occur.

Under the hood, `std::unique_ptr` uses the perfect forwarding pattern. The same holds for the other factory methods such as `std::make_shared`, `std::make_tuple`, `std::make_pair`, or an `std::thread` constructor.

Further information#

std::unique_ptr

std::make_unique

std::make_shared

- Examples

The key question of the std::unique_ptr is when to delete the underlying resource. This occurs when the std::unique_ptr goes out of scope or receives a new resource. Let's look at two use cases to better understand this concept.

```
/*
// =====
// uniquePtr.cpp

// #include <iostream>
// #include <memory>
// #include <utility>

// struct MyInt
//{
//  MyInt(int i) : i_(i) {}

//  ~MyInt()
//  {
//    std::cout << "Good bye from " << i_ << std::endl;
//  }

//  int i_;
//};

// int main()
//{
//  std::cout << std::endl;

//  std::unique_ptr<MyInt> uniquePtr1{new MyInt(1998)};

//  std::cout << "uniquePtr1.get(): " << uniquePtr1.get() << std::endl;

//  std::unique_ptr<MyInt> uniquePtr2;
//  uniquePtr2 = std::move(uniquePtr1);
//  std::cout << "uniquePtr1.get(): " << uniquePtr1.get() << std::endl;
//  std::cout << "uniquePtr2.get(): " << uniquePtr2.get() << std::endl;

//  std::cout << std::endl;
// }
```

```

// std::unique_ptr<MyInt> localPtr{new MyInt(2003)};
// }

// std::cout << std::endl;

// uniquePtr2.reset(new MyInt(2011));
// MyInt *myInt = uniquePtr2.release();
// delete myInt;

// std::cout << std::endl;

// std::unique_ptr<MyInt> uniquePtr3{new MyInt(2017)};
// std::unique_ptr<MyInt> uniquePtr4{new MyInt(2022)};

// std::cout << "uniquePtr3.get(): " << uniquePtr3.get() << std::endl;
// std::cout << "uniquePtr4.get(): " << uniquePtr4.get() << std::endl;

// std::swap(uniquePtr3, uniquePtr4);

// std::cout << "uniquePtr3.get(): " << uniquePtr3.get() << std::endl;
// std::cout << "uniquePtr4.get(): " << uniquePtr4.get() << std::endl;

// std::cout << std::endl;
// }

// =====
/*

```

Explanation#

The class `MyInt` (lines 7 -17) is a simple wrapper for a number. We have adjusted the destructor in line 11 - 13 for observing the life cycle of `MyInt`.

We create, in line 24, an `std::unique_ptr` and return, in line 26, the address of its resource, `new MyInt(1998)`. Afterward, we move the `uniquePtr1` to `uniquePtr2` (line 29). Therefore, `uniquePtr2` is the owner of the resource. That is shown in the output of the program in lines 30 and 31.

In line 37, the local `std::unique_ptr` reaches its valid range at the end of the scope. Therefore, the destructor of the local `ptr` – meaning the destructor of the resource `new MyInt(2003)` – will be executed.

The most interesting lines are lines 42 to 44. First, we assign a new resource to the `uniquePtr2`. Therefore, the destructor of `MyInt(1998)` will be executed. After the resource in line 43 is released, we can explicitly invoke the destructor.

The rest of the program is quite easy to understand. In lines 48 - 58, we create two `std::unique_ptr` and swap their resources. `std::swap` uses move semantics since `std::unique_ptr` doesn't support copy semantics. With the end of the main function, `uniquePtr3` and `uniquePtr4` go out of scope, and their destructors will be automatically executed.

Now that we have a sense of this technique, let's dig into a few details of `std::unique_ptr` in the example below.

```

*/
/*

```

Example 2#

std::unique_ptr has a specialization for arrays. The access is transparent, meaning that if the std::unique_ptr manages the lifetime of an object, the operators for the object access are overloaded (operator* and operator->). If std::unique_ptr manages the lifetime of an array, the index operator [] is overloaded. The invocations of the operators are, therefore, transparently forwarded to the underlying resource.

```
/*
// =====
// uniquePtrArray.cpp

// #include <iomanip>
// #include <iostream>
// #include <memory>

// class MyStruct{
// public:
//   MyStruct(){
//     std::cout << std::setw(15) << std::left << (void*) this << " Hello " << std::endl;
//   }
//   ~MyStruct(){
//     std::cout << std::setw(15) << std::left << (void*)this << " Good Bye " << std::endl;
//   }
// };

// int main(){

// std::cout << std::endl;

// std::unique_ptr<int> uniqInt(new int(2011));
// std::cout << "*uniqInt: " << *uniqInt << std::endl;

// std::cout << std::endl;

// {
//   std::unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[5]};
// }

// std::cout << std::endl;

// {
//   std::unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[1]};
//   MyStruct myStruct;
//   myUniqueArray[0]=myStruct;
// }

// std::cout << std::endl;

// {
//   std::unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[1]};
//   MyStruct myStruct;
//   myStruct= myUniqueArray[0];
// }
```

```
// }  
  
// std::cout << std::endl;  
  
//}  
// ======  
/*
```

Explanation#

We dereference (line 22) an std::unique_ptr and get the value of its resource.

In lines 7 - 15, MyStruct acts as the base of an array of std::unique_ptr's. If we instantiate a MyStruct object, we will get its address. The destructor gives the output. Now it is easy to observe the life cycle of the objects.

In lines 26 - 28, we create and destroy five instances of MyStruct.

The lines 32 - 36 are more interesting. We create a MyStruct instance on the heap (line 33) and on the stack (line 34). Therefore, both objects have addresses from different ranges.

Afterward, we assign the local object to the std::unique_ptr (line 35). The lines 40 - 44 follows a similar strategy. Now we assign the local object, the first element of myUniqueArray. The index access to the std::unique_ptr in the lines 35 and 43 feels like familiar to index access to an array.

Let's move on to the second type of smart pointers in this section, called shared pointers.

Shared Pointers

Next, we will go over shared pointers, which follow the principle of keeping a reference count to maintain the count of its copies. The lesson below elaborates further.

Introduction#

std::shared_ptr shares ownership of the resource. They have two handles: one for the resource, and one for the reference counter. By copying an std::shared_ptr, the reference count is increased by one. It is decreased by one if the std::shared_ptr goes out of scope. If the reference counter becomes the value 0, the C++ runtime automatically releases the resource, since there is no longer an std::shared_ptr referencing the resource. The release of the resource occurs exactly when the last std::shared_ptr goes out of scope. The C++ runtime guarantees that the call of the reference counter is an atomic operation. Due to this management, std::shared_ptr consumes more time and memory than a raw pointer or std::unique_ptr.

Methods#

In the following table, we will see the methods of std::shared_ptr.

Name Description

get Returns a pointer to the resource.

get_deleter Returns the delete function.

reset Resets the resource.

swap Swaps the resources.

unique Checks if the std::shared_ptr is the exclusive owner of the resource.

use_count Returns the value of the reference counter.

Methods of std::shared_ptr

std::make_shared#

The helper function `std::make_shared` creates the resource and returns it in an `std::shared_ptr`. Use `std::make_shared` rather than directly creating an `std::shared_ptr` because `std::make_shared` is much faster. Additionally, such as in the case of `std::make_unique`, `std::make_shared` guarantees no memory leaks.

`std::shared_ptr` from this#

This unique technique, in which a class derives from a class template having itself as a parameter, is called CRTP and stands for Curiously Recurring Template Pattern.

Using the class `std::enable_shared_from_this`, we can create objects that return an `std::shared_ptr` to themselves. To do so, we must publicly derive the class from `std::enable_shared_from_this`. So the class ShareMe support the method `shared_from_this`, and return `std::shared_ptr`:

```
/*
// =====
// enableShared.cpp

// #include <iostream>
// #include <memory>

// class ShareMe: public std::enable_shared_from_this<ShareMe>{
// public:
//   std::shared_ptr<ShareMe> getShared(){
//     return shared_from_this();
//   }
// };

// int main(){

// std::cout << std::endl;

// std::shared_ptr<ShareMe> shareMe(new ShareMe);
// std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();
// {
//   auto shareMe2(shareMe1);
//   std::cout << "shareMe.use_count(): " << shareMe.use_count() << std::endl;
// }
// std::cout << "shareMe.use_count(): " << shareMe.use_count() << std::endl;

// shareMe1.reset();

// std::cout << "shareMe.use_count(): " << shareMe.use_count() << std::endl;

// std::cout << std::endl;

// }
// =====
/*
```

The smart pointer `shareMe` (line 17) is copied by `shareMe1` (line 18) and `shareMe2` (line 20), and all of them

reference the very same resource.

increment and decrement the reference counter.

The call `shareMe->getShared()` in line 18 creates a new smart pointer. `getShared()` (line 9) internally uses the function `shared_from_this`.

Further information#

`std::shared_ptr`

`std::make_shared`

CRTPO

`std::enable_shared_from_this`

Example 1#

To get a visual idea of the life cycle of the resource, there is a short message in the constructor and destructor of `MyInt` (lines 8 - 16).

`*/`

```
// =====
// sharedPtr.cpp

// #include <iostream>
// #include <memory>

// using std::shared_ptr;

// struct MyInt
// {
//   MyInt(int v) : val(v)
//   {
//     std::cout << " Hello: " << val << std::endl;
//   }
//   ~MyInt()
//   {
//     std::cout << " Good Bye: " << val << std::endl;
//   }
//   int val;
// };

// int main()
// {
//   std::cout << std::endl;

//   shared_ptr<MyInt> sharPtr(new MyInt(1998));
//   std::cout << " My value: " << sharPtr->val << std::endl;
//   std::cout << "sharedPtr.use_count(): " << sharPtr.use_count() << std::endl;

//   {
//     shared_ptr<MyInt> locSharPtr(sharPtr);
//     std::cout << "locSharPtr.use_count(): " << locSharPtr.use_count() << std::endl;
//   }
// }
```

```

// std::cout << "sharPtr.use_count(): " << sharPtr.use_count() << std::endl;
// shared_ptr<MyInt> globSharPtr = sharPtr;
// std::cout << "sharPtr.use_count(): " << sharPtr.use_count() << std::endl;
// globSharPtr.reset();
// std::cout << "sharPtr.use_count(): " << sharPtr.use_count() << std::endl;

// sharPtr = shared_ptr<MyInt>(new MyInt(2011));

// std::cout << std::endl;
//}
//=====================================================================
/*

```

Explanation#

In line 22, we create `MyInt(1998)`, which is the resource that the smart pointer should address. By using `sharPtr->val`, we have direct access to the resource (line 23).

The output of the program shows the number of references counted. It starts in line 24 with 1. It then has a local copy `shartPtr` in line 28 and goes to 2. The program then returns to 1 after the block (lines 27-30).

The copy assignment call in line 33 modifies the reference counter. The expression `sharPtr=shared_ptr<MyInt>(new MyInt(2011))` in line 38 is more interesting.

First, the resource `MyInt(2011)` is created in line 38 and assigned to `sharPtr`. Consequently, the destructor of `sharPtr` is invoked. `sharedPtr` was the exclusive owner of the resource `new MyInt(1998)` (line 22).

The last new resource `MyInt(2011)` will be destroyed at the end of `main`.

Weak Pointers

`std::weak_ptr` is the last component of the smart pointers family. Its purpose is limited compared to the other smart pointers, and we will examine why in this lesson.

Introduction#

To be honest, `std::weak_ptr` is not a classic smart pointer, since it supports no transparent access to the resource; it only borrows the resource from an `std::shared_ptr`.

Methods#

The table provides an overview of the methods of `std::weak_ptr`.

Name Description

<code>expired</code>	Checks if the resource was deleted.
<code>lock</code>	Creates a <code>std::shared_ptr</code> on the resource.
<code>reset</code>	Resets the resource.
<code>swap</code>	Swaps the resources.
<code>use_count</code>	Returns the value of the reference counter.

Methods of `std::weak_ptr`

There is one main reason for the existence and use of `std::weak_ptr`. It breaks the cycle of `std::shared_ptr`. We will discuss these cyclic references in detail in the next lessons.

Further information#

```
std::weak_ptr
```

```
cyclic references
```

```
Example#
```

```
/*
// =====
// weakPtr.cpp

// #include <iostream>
// #include <memory>

// int main()
//{
// std::cout << std::boolalpha << std::endl;

// auto sharedPtr = std::make_shared<int>(2011);
// std::weak_ptr<int> weakPtr(sharedPtr);

// std::cout << "weakPtr.use_count(): " << weakPtr.use_count() << std::endl;
// std::cout << "sharedPtr.use_count(): " << sharedPtr.use_count() << std::endl;
// std::cout << "weakPtr.expired(): " << weakPtr.expired() << std::endl;

// if (std::shared_ptr<int> sharedPtr1 = weakPtr.lock())
// {
// std::cout << "*sharedPtr: " << *sharedPtr << std::endl;
// std::cout << "sharedPtr1.use_count(): " << sharedPtr1.use_count() << std::endl;
// }
// else
// {
// std::cout << "Don't get the resource!" << std::endl;
// }

// weakPtr.reset();
// if (std::shared_ptr<int> sharedPtr1 = weakPtr.lock())
// {
// std::cout << "*sharedPtr: " << *sharedPtr << std::endl;
// std::cout << "sharedPtr1.use_count(): " << sharedPtr1.use_count() << std::endl;
// }
// else
// {
// std::cout << "Don't get the resource!" << std::endl;
// }

// std::cout << std::endl;
//}

// =====
/*
```

```
Explanation#
```

```
In line 11, we create an std::weak_ptr that borrows the resource from the std::shared_ptr.
```

The output of the program shows that the reference counter is 1 (line 13 and 14), meaning that std::weak does not increment the counter.

The call weakPtr.expired() checks if the resource was already deleted. That is equivalent to the expression weakPtr.use_count() == 0.

If the std::weak_ptr shared a resource, we could use weakPtr.lock() at line 17 to create an std::shared_ptr out of it.

The reference counter will now be increased to 2 (line 18). After resetting the weakPtr (line 25), the call weakPtr.lock() fails.

That was almost the whole story for the std::weak_ptr. Almost, because the std::weak_ptr has a special job: it helps to break the cyclic references of std::shared_ptr.

Cyclic References

We get cyclic references of std::shared_ptr if they refer to each other.

The issue#

If we have a cyclic reference of std::shared_ptr, the reference counter will never become 0. We can break this cycle if by embedding an std::weak_ptr in the cycle. std::weak_ptr does not modify the reference counter.

Theoretically, we can use a raw pointer to break the cycle of std::shared_ptr's, but a raw pointer has two disadvantages. First, they don't have a well-defined interface. Second, they don't support an interface that can create an std::shared_ptr out of it.

There are two cycles in the graphic below: first, between the mother and her daughter; second, between the mother and her son. The subtle difference is that the mother references her daughter with an std::weak_ptr. Therefore, the std::shared_ptr cycle is broken.

Example#

```
/*
// cyclicReference.cpp

// #include <iostream>
// #include <memory>

// struct Son;
// struct Daughter;

// struct Mother
//{
// ~Mother()
// {
// std::cout << "Mother gone" << std::endl;
// }
// void setSon(const std::shared_ptr<Son> s)
```

```

// {
//   mySon = s;
// }
// void setDaughter(const std::shared_ptr<Daughter> d)
// {
//   myDaughter = d;
// }
// std::shared_ptr<const Son> mySon;
// std::weak_ptr<const Daughter> myDaughter;
//};

// struct Son
//{
// Son(std::shared_ptr<Mother> m) : myMother(m) {}
// ~Son()
// {
//   std::cout << "Son gone" << std::endl;
// }
// std::shared_ptr<const Mother> myMother;
//};

// struct Daughter
//{
// Daughter(std::shared_ptr<Mother> m) : myMother(m) {}
// ~Daughter()
// {
//   std::cout << "Daughter gone" << std::endl;
// }
// std::shared_ptr<const Mother> myMother;
//};

// int main()
//{
// std::cout << std::endl;
// {
//   std::shared_ptr<Mother> mother = std::shared_ptr<Mother>(new Mother);
//   std::shared_ptr<Son> son = std::shared_ptr<Son>(new Son(mother));
//   std::shared_ptr<Daughter> daughter = std::shared_ptr<Daughter>(new Daughter(mother));
//   mother->setSon(son);
//   mother->setDaughter(daughter);
// }
// std::cout << std::endl;
//}
// =====
/*
Explanation#
In line 41 – 47, due to the artificial scope, the lifetime of the mother, the son, and the daughter are limited. In other words, mother, son, and daughter go out of scope, and therefore the destructor of the class Mother (line 10 - 12), Son (line 25 - 27), and Daughter (line 33 - 35) should automatically be invoked.
```

We state should, because only the destructor of the class Daughter is called.

The graphic of the source code shows that we have a cyclic reference of std::shared_ptr between mother and son. Therefore, the reference counter is always greater than 0, and the destructor will not automatically be invoked.

That observation does not hold true for mother and daughter. If the daughter goes out of scope, the reference counter of the std::shared_ptr myMother (line 36) becomes 0 and the resource will automatically be deleted.

Performance Comparison

A simple performance test should give an idea of the overall performance.

Run the code in the tabs below to see the performance of each pointer.

Test Code#

 The codes might take some time to execute.

```
/*
// =====
// all.cpp

// #include <chrono>
// #include <iostream>
// #include <memory>
// static const long long numInt = 100000000;

// int main()
//{
//    auto start = std::chrono::system_clock::now();

//    for (long long i = 0; i < numInt; ++i)
//    {
//        int *tmp(new int(i));
//        delete tmp;
//        // std::shared_ptr<int> tmp(new int(i));
//        // std::shared_ptr<int> tmp(std::make_shared<int>(i));
//        // std::unique_ptr<int> tmp(new int(i));
//        // std::unique_ptr<int> tmp(std::make_unique<int>(i));
//    }

//    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
//    std::cout << "time native: " << dur.count() << " seconds" << std::endl;
//}

/*
Explanation#
In this test, we compare the explicit calls of new and delete (line 13 and 14) with the usage of std::shared_ptr (line 15), std::make_shared (line 16), std::unique_ptr (line 17), and std::make_unique (line 18).
```

The handling of smart pointers (line 15 - 18) is now much simpler since the smart pointer automatically releases its dynamically created int variable if it goes out of scope.

The two functions ::make_shared (line 16) and std::make_unique (line 18) are useful, for they create the smart pointers respectively.

There are more memory allocations necessary for the creation of an std::shared_ptr. Memory is necessary for the managed resource and reference counters. std::make_shared makes one memory allocation out of these counters.

Passing Smart Pointers

Passing smart pointers is an important topic that is seldom addressed. This chapter ends with the C++ core guidelines since they have six rules for passing std::shared_ptr and std::unique_ptr.

The Six Rules#

The following six rules violate the important DRY (don't repeat yourself) principle for software development. In the end, we only have six rules, which makes life as a software developer a lot easier. Here are the rules:

- R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget.
- R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget.
- R.34: Take a shared_ptr<widget> parameter to express that a function is part owner.
- R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer.
- R.36: Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object.
- R.37: Do not pass a pointer or reference obtained from an aliased smart pointer.

Let's start with the first two rules for std::unique_ptr.

R.32#

If a function should take ownership of a Widget, take the std::unique_ptr<Widget> by copy. The consequence is that the caller has to move the std::unique_ptr<Widget> to make the code run.

*/

```
// =====
// #include <memory>
// #include <utility>

// struct Widget
//{
//    Widget(int) {}
//};

// void sink(std::unique_ptr<Widget> uniqPtr)
//{
//    // do something with uniqPtr
//}

// int main()
//{
//    auto uniqPtr = std::make_unique<Widget>(1998);
```

```

// sink(std::move(uniqPtr)); // (1)
// // sink(uniqPtr);      // (2) ERROR
// }
// =====
/*

```

The call in line 15 is fine but the call line 16 breaks because we cannot copy an std::unique_ptr. If the function only wants to use the Widget, it should take its parameter by pointer or by reference. A pointer can be a null pointer, but a reference cannot.

```

void useWidget(Widget* wid);
void useWidget(Widget& wid);

```

R.33#

Sometimes a function wants to reseat a Widget. In this case, pass the std::unique_ptr<Widget> by a non-const reference.

```

*/
// =====
// #include <memory>
// #include <utility>

// struct Widget{
//   Widget(int){}
// };

// void reseat(std::unique_ptr<Widget>& uniqPtr){
//   uniqPtr.reset(new Widget(2003)); // (0)
//   // do something with uniqPtr
// }

// int main(){
//   auto uniqPtr = std::make_unique<Widget>(1998);

//   // // reseat(std::move(uniqPtr));    // (1) ERROR
//   reseat(uniqPtr);           // (2)
// }
// =====
/*

```

Now, the call in line 16 fails because we cannot bind an rvalue to a non-const lvalue reference. This will not hold for the copy in line 17. An lvalue can be bound to an lvalue reference. The call in line 9 will not only construct a new Widget(2003), but it will also destruct the old Widget(1998).

The next three rules of std::shared_ptr repeat each other, so we will only discuss one.

R.34, R.35, and R.36#

Here are the three function signatures that we have to address.

```

*/
/*
void share(std::shared_ptr<Widget> shaWid);
void reseat(std::shared_ptr<Widget>& shadWid);
void mayShare(const std::shared_ptr<Widget>& shaWid);

```

We will take a look at each function signature in isolation, but what does this mean from the function perspective? Let's find out!

`void share(std::shared_ptr shaWid):` For the lifetime of the function body, this method is a shared owner of the Widget. At the start of the function body, we will increase the reference counter; at the end of the function, we will decrease the reference counter; therefore, the Widget will stay alive, as long as we use it.

`void reseat(std::shared_ptr& shaWid):` This function isn't a shared owner of the Widget because we will not change the reference counter. We have not guaranteed that the Widget will stay alive during the execution of the function, but we can reseat the resource. A non-const lvalue reference is more like borrowing the resource with the ability to reseat it.

`void mayShare(const std::shared_ptr& shaWid):` This function only borrows the resource. Neither can we extend the lifetime of the resource nor can we reseat the resource. To be honest, we should use a pointer (`Widget*`) or a reference (`Widget&`) as a parameter instead, because there is no added value in using an `std::shared_ptr`.

R.37#

Let's take a look at a short code snippet to make the rule clearer.

```
void oldFunc(Widget* wid){  
    // do something with wid  
}  
  
void shared(std::shared_ptr<Widget>& shaPtr){    // (2)  
  
    oldFunc(*shaPtr);                // (3)  
  
    // do something with shaPtr  
  
}  
  
auto globShared = std::make_shared<Widget>(2011); // (1)  
  
...  
  
shared(globShared);
```

In line 13, `globShared` is a globally shared pointer. The function `shared` takes its argument by reference in line 5. Therefore, the reference counter of `shaPtr` will not be increased and the function `shared` will not extend the lifetime of `Widget(2011)`. The issue begins on line 7. `oldFunc` accepts a pointer to the Widget; therefore, `oldFunc` has no guarantee that the Widget will stay alive during its execution. `oldFunc` only borrows the Widget.

The solution is quite simple. We must ensure that the reference count of `globShared` is increased before the call to the function `oldFunc`, meaning that we must make a copy of `std::shared_ptr`:

Pass the `std::shared_ptr` by copy to the function `shared`:

```
void shared(std::shared_ptr<Widget> shaPtr){
```

```
oldFunc(*shaPtr);
// do something with shaPtr
}
```

Make a copy of the shaPtr in the function shared:

```
void shared(std::shared_ptr<Widget>& shaPtr){
    auto keepAlive = shaPtr;
    oldFunc(*shaPtr);
    // do something with keepAlive or shaPtr
}
```

The same reasoning also applies to std::unique_ptr, but there isn't a simple solution since we cannot copy an std::unique_ptr. Rather, we can clone the std::unique_ptr and make a new std::unique_ptr.

Further information#
don't repeat yourself

R.32

R.33

R.34

R.35

R.36

R.37

'

TYPE TRIATS:

Introduction

The type traits library helps us optimize our code. This section will cover the library in depth.

The type traits library enables you to check, compare and modify types at compile time. So, there is no overhead on the runtime of your program. There are two reasons for using the type traits library: Optimization and Correctness. Optimization, because the introspection capabilities of the type traits library make it possible to choose the faster code automatically. Correctness, because you can specify requirements for your code, which is checked at compile time.

 The type traits library and static_assert are a powerful pair

The type traits library and the function static_assert are a powerful pair. On one side, the functions of the type traits library provide the type information at compile time. On the other side, the static_assert function checks the given information at compile time. All this happens transparently to the runtime of the program:

```
#include <type_traits>
template <typename T> T fac(T a){
    static_assert(std::is_integral<T>::value, "T not integral");
```

```

//...
}
fac(10);
fac(10.1); // with T= double; T not integral

```

The GCC compiler quits the function invocation fac(10.1). The message at compile is that T is of type double and therefore no integral type.

Primary Type Categories#

There are 14 different type categories. They are complete and don't overlap. So each type is only a member of one type category. If you check a type category for your type, the request is independent of the const or volatile qualifiers.

```

template <class T> struct is_void;
template <class T> struct is_null_pointer;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
*/

```

```

// =====
// typeCategories.cpp
// #include <iostream>
// #include <type_traits>
// using std::cout;
// int main()
//{
//  // out put 1 means that the function returns true
//  cout << "is_void: " << std::is_void<void>::value << "\n";           // 1
//  cout << "is_integral: " << std::is_integral<short>::value << "\n";      // 1
//  cout << "is_floating_point: " << std::is_floating_point<double>::value << "\n"; // 1
//  cout << "is_array: " << std::is_array<int[]>::value << "\n";           // 1
//  cout << "is_pointer: " << std::is_pointer<int *>::value << "\n";         // 1
//  cout << "is_reference: " << std::is_reference<int &>::value << "\n";       // 1

//  struct A
//  {
//    int a;
//    int f(int) { return 2011; }
//  };
//  cout << "is_member_object_pointer: " << std::is_member_object_pointer<int A::*>::value << "\n";
//  */

```

```

// cout << "is_member_function_pointer: " << std::is_member_function_pointer<int (A::*)(int)>::value << "\n"; // 1

// enum E
// {
//   e = 1,
// };
// cout << "is_enum: " << std::is_enum<E>::value << "\n"; // 1

// union U
// {
//   int u;
// };
// cout << "is_union: " << std::is_union<U>::value << "\n"; // 1

// cout << "is_class: " << std::is_class<std::string>::value << "\n";           // 1
// cout << "is_function: " << std::is_function<int *(double)>::value << "\n";    // 1
// cout << "is_lvalue_reference: " << std::is_lvalue_reference<int &>::value << "\n"; // 1
// cout << "is_rvalue_reference: " << std::is_rvalue_reference<int &&>::value << "\n"; // 1

// return 0;
//}

```

// =====
/*

Composite Type Categories#

Based on the 14 primary type categories, there are 6 composite type categories.

Composite type categories	Primary type category
is_arithmetic	is_floating_point or is_integral
is_fundamental	is_arithmetic or is_void
is_object	is_arithmetic or is_enum or is_pointer or is_member_pointer
is_reference	is_lvalue_reference or is_rvalue_reference
is_compound	complement of is_fundamental
is_member_pointer	is_member_object_pointer or is_member_function_pointer

Composite type categories

Type Properties#

In addition to the primary and composite type categories, there are type properties

```

template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;

```

```

template <class T> struct is_signed;
template <class T> struct is_unsigned;

template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;

template <class T, class U> struct is_assignable;
template <class T> struct is_copyAssignable;
template <class T> struct is_moveAssignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copyAssignable;
template <class T> struct is_trivially_moveAssignable;

template <class T> struct is_trivially_destructible;

template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;

template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copyAssignable;
template <class T> struct is_nothrow_moveAssignable;

template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;

```

Type Comparisons and Modifications

Sometimes we need to manipulate or compare different types. We can use the type traits library for that!

Type Comparisons#

The library supports three kinds of type comparisons:

Function	Description
template <class Base, class Derived>	Checks if Derived is derived from Base.
struct is_base_of	
template <class From, class To>	Checks if From can be converted to To.
struct is_convertible	
template <class T, class U>	Checks if the types T and U are the same.
struct is_same	

Type Modifications#

The type traits library enables you to modify types during compile time. So you can modify the constness of a type:

*/

```

// =====
// typeTraitsModifications.cpp
// #include <iostream>
// #include <type_traits>
// using namespace std;

/// output 0 if the function returns false and 1 if the function returns true
/// int main()
//{
// cout << is_const<int>::value << "\n"; // 0
// cout << is_const<const int>::value << "\n"; // 1
// cout << is_const<add_const<int>::type>::value << "\n"; // 1

// typedef add_const<int>::type myConstInt;
// cout << is_const<myConstInt>::value << "\n"; // 1

// typedef const int myConstInt2;
// cout << is_same<myConstInt, myConstInt2>::value << "\n"; // 1

// cout << is_same<int, remove_const<add_const<int>::type>::type>::value << "\n"; // 1
// cout << is_same<const int, add_const<int>::type>::value << "\n"; // 1

// return 0;
//}
// =====
/*

```

The function `std::add_const` adds the constness to a type, while `std::remove_const` removes it.

There are a lot more functions available in the type traits library. So you can modify the const-volatile properties of a type.

```

template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;

template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

```

You can change at compile time the sign,

```

template <class T> struct make_signed;
template <class T> struct make_unsigned;

```

or the reference or pointer properties of a type.

```

template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

```

```
template <class T> struct remove_pointer;
template <class T> struct add_pointer;
```

The three following functions are especially valuable for the writing of generic libraries.

```
template <class B> struct enable_if;
template <class B, class T, class F> struct conditional;
template <class... T> common_type;
```

You can conditionally hide with std::enable_if a function overload or template specialization from overload resolution. std::conditional provides you with the ternary operator at compile time and std::common_type gives you the type, to which all type parameters can be implicitly converted to. std::common_type is a variadic template, therefore the number of type parameters can be arbitrary.

Let's say we want to write code for the euclid algorithm to calculate the greatest common divisor of two numbers. We can incorporate enable_if and conditional in our code:

```
/*
// =====
// #include <iostream>
// #include <type_traits>

// template <typename T1, typename T2,
//           typename std::enable_if<std::is_integral<T1>::value, T1>::type = 0,
//           typename std::enable_if<std::is_integral<T2>::value, T2>::type = 0,
//           typename R = typename std::conditional<(sizeof(T1) < sizeof(T2)), T1, T2>::type>
// R gcd(T1 a, T2 b)
//{
//   if (b == 0)
//   {
//     return a;
//   }
//   else
//   {
//     return gcd(b, a % b);
//   }
//}

// int main()
//{
// std::cout << "gcd(100, 10)= " << gcd(100, 10) << std::endl;
// std::cout << "gcd(100, 33)= " << gcd(100, 33) << std::endl;
// std::cout << "gcd(3.5, 4.0)= " << gcd(35, 40) << std::endl;
//}
// =====
/*
```

Lines 5 and 6 are the key lines of the program above. The expression std::is_integral determines whether the type parameter T1 and T2 are integrals. If T1 and T2 are not integrals, and therefore they return false, we will not get a template instantiation. This is the decisive observation.

If `std::enable_if` returns true as the first parameter, `std::enable_if` will have a public member `typedef type`. This type is used in lines 5 and 6. If `std::enable_if` returns false as first parameter, `std::enable_if` will have no member type. Therefore, lines 5 and 6 are not valid. This is not an error but a common technique in C++: SFINAE. SFINAE stands for Substitution Failure Is Not An Error. Only the template for exactly this type will not be instantiated and the compiler tries to instantiate the template in another way.

 C++14 has a shorthand for `::type`

If you want to get a `const int` from an `int` you have to ask for the type: `std::add_const<int>::type`. With the C++14 standard use simply `std::add_const_t<int>` instead of the verbose form: `std::add_const<int>::type`. This rule works for all type traits functions.

```
*/
```

```
/*
time library:
```

Introduction

This library gives us the power to run multithreads based on our time requirements.

The time library is a key component of the new multithreading capabilities of C++. So you can put the current thread by `std::this_thread::sleep_for(std::chrono::milliseconds(15))` for 15 milliseconds to sleep, or you try to acquire a lock for 2 minutes: `lock.try_lock_until(now + std::chrono::minutes(2))`. Beside that, the chrono library makes it easy to perform simple performance tests:

```
/*
// =====
// performanceMeasurement.cpp
// #include <iostream>
// #include <vector>
// #include <chrono>
// using namespace std;

// int main()
//{
// std::vector<int> myBigVec(10000000, 2011);
// std::vector<int> myEmptyVec1;

// auto begin = std::chrono::high_resolution_clock::now();
// myEmptyVec1 = myBigVec;
// auto end = std::chrono::high_resolution_clock::now() - begin;

// auto timeInSeconds = std::chrono::duration<double>(end).count();
// std::cout << timeInSeconds << std::endl; // 0.0150688800 <- may vary from execution to execution

// return 0;
//}
// =====
/*
```

The time library consists of the three components, time point, time duration and clock.

Time point:#

Time point is defined by a starting point, the so-called epoch, and an additional time duration.

Time duration:#

Time duration is the difference between two time-points. It is given by the number of ticks.

Clock:#

A clock consists of a starting point (epoch) and a tick, so that the current time point can be calculated.

Time Point

Now, we will study the first component of the chrono library.

A duration consists of a span of time, defined as some number of ticks of some time unit. A time point consists of a clock and a time duration. This time duration can be positive or negative.

```
template <class Clock, class Duration= typename Clock::duration>
class time_point;
```

The epoch is not defined for the clocks std::chrono::steady_clock, std::chrono::high_resolution_clock and std::chrono::system. But on the popular platform the epoch of std::chrono::system is usually defined as 1.1.1970. You can calculate the time since 1.1.1970 in the resolutions nanoseconds, seconds and minutes.
*/

```
// =====
// epoch.cpp
// #include <iostream>
// #include <chrono>

// int main(){
//   auto timeNow= std::chrono::system_clock::now();
//   auto duration= timeNow.time_since_epoch();
//   std::cout << duration.count() << "ns\n";    // nanoseconds (default)

//   // duration_cast converts one type into the other
//   auto durationSeconds = std::chrono::duration_cast<std::chrono::seconds>(duration).count();
//   std::cout << durationSeconds << "s\n";    // seconds

//   auto durationMinutes = std::chrono::duration_cast<std::chrono::minutes>(duration).count();
//   std::cout << durationMinutes << "m\n";    // minutes

//   return 0;
// }
```

```
/*
Time Duration
```

The time duration is a measure of how many ticks have passed since a certain time point. The implementation is presented in this lesson.

Time duration#

Time duration is the difference between the two time-points. Time duration is measured in the number of ticks.

```
typedef duration<signed int, nano> nanoseconds;
typedef duration<signed int, micro> microseconds;
typedef duration<signed int, milli> milliseconds;
typedef duration<signed int> seconds;
typedef duration<signed int, ratio< 60>> minutes;
typedef duration<signed int, ratio<3600>> hours;
```

How long can a time duration be? The C++ standard guarantees that the predefined time durations can store +/- 292 years. You can easily define your own time duration like a German school hour: `typedef std::chrono::duration<double, std::ratio<2700>> MyLessonTick`. Time durations in natural numbers have to be explicitly converted to time durations in floating pointer numbers. The value will be truncated:

```
/*
// =====
```

```
// duration.cpp
// #include <iostream>
// #include <chrono>
// #include <ratio>
// using namespace std;
// using namespace std::chrono;
// template <class Rep, class Period = ratio<1>> class duration;

// int main(){
//   typedef std::chrono::duration<long long, std::ratio<1>> MySecondTick;
//   MySecondTick aSecond(1);

//   milliseconds milli(aSecond);
//   std::cout << milli.count() << " ms\n";    // 1000 milli

//   seconds seconds(aSecond);
//   std::cout << seconds.count() << " sec\n";    // 1 sec

//   minutes minutes(duration_cast<minutes>(aSecond));
//   std::cout << minutes.count() << " min\n";    // 0 min

//   typedef std::chrono::duration<double, std::ratio<2700>> MyLessonTick;
//   MyLessonTick myLesson(aSecond);
//   std::cout << myLesson.count() << " less\n";    // 0.00037037 less
```

```
//   return 0;
// }
```

```
// =====
```

```
/*
i std::ratio
```

`std::ratio` supports arithmetic at compile time with rational numbers. A rational number has two template arguments. One is the nominator, the other the denominator. C++11 predefines lots of rational numbers.

```
typedef ratio <1, 10000000000000000000> atto;
typedef ratio <1, 1000000000000000> femto;
typedef ratio <1, 1000000000000> pico;
typedef ratio <1, 1000000000> nano;
typedef ratio <1, 1000000> micro;
typedef ratio <1, 1000> milli;
typedef ratio <1, 100> centi;
typedef ratio <1, 10> deci;
typedef ratio < 10, 1> deca;
typedef ratio < 100, 1> hecto;
typedef ratio < 1000, 1> kilo;
typedef ratio < 1000000, 1> mega;
typedef ratio < 1000000000, 1> giga;
typedef ratio < 1000000000000000, 1> tera;
typedef ratio < 10000000000000000000, 1> peta;
typedef ratio < 10000000000000000000000000, 1> exa;
```

C++14 has built-in literals for the most used time durations.

Type	Suffix	Example
------	--------	---------

std::chrono::hours	h	5h
std::chrono::minutes	min	5min
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns

Built-in literals for time durations

Clock

Here, we briefly discuss the different types of clocks in C++.

The clock consists of a starting point and a tick. So you can get the current time with the method now.

std::chrono::system_clock

System time, which you can synchronize with the external clock.

std::chrono::steady_clock

Clock, which can not be adjusted.

std::chrono::high_resolution_clock:

System time with the greatest accuracy.

`std::chrono::system_clock` will refer typically to the 1.1.1970. You can not adjust `std::steady_clock` forward or backward in opposite to two other clocks. The methods `to_time_t` and `from_time_t` can be used to convert between `std::chrono::system_clock` and `std::time_t` objects.

`std::any`

C++17 allows us to put our value in a safe container which can be accessed only when its type is specified. Welcome to std::any.

The new C++17 data types `std::any`, `std::optional`, and `std::variant` are all based on the Boost libraries.

`std::any` is a type-safe container for single values of any type which is copy-constructible. There are a few ways to create a `std::any` container `any`. You can use the various constructors or the factory function `std::make_any`. By using `any.emplace`, you directly construct one value into `any`. `any.reset` lets you destroy the contained object. If you want to know whether the container `any` has a value, use the method `any.has_value`. You can even get the `typeid` of the container object via `any.type`. Thanks to the generic function `std::any_cast` you have access to the contained object. If you specify the wrong type, you will get a `std::bad_any_cast` exception.

Here is a code snippet showing the basic usage of `std::any`.

```
/*
// =====
// #include <any>
// #include <iostream>
// #include <vector>

// using namespace std;
// struct MyClass
//{
//};

// int main()
//{
//    std::vector<std::any> anyVec{true, 2017, std::string("test"), 3.14, MyClass()};
//    std::cout << std::any_cast<bool>(anyVec[0]) << endl; // true
//    int myInt = std::any_cast<int>(anyVec[1]);
//    std::cout << myInt << std::endl
//    << endl; // 2017

//    std::cout << anyVec[0].type().name() << endl; // b
//    std::cout << anyVec[1].type().name();      // i

//    return 0;
//}
// =====
/*
```

The program snippet defines a `std::vector<std::any>`. To get one of its elements, you have to use `std::any_cast`. As mentioned, if you use the wrong type, you will get a `std::bad_any_cast` exception.

i The string representation of the typeid

The string representation of the typeid is implementation defined. If `anyVec[1]` is of type `int` the expression `anyVec[1].type().name()` will return `i` with the GCC C++ compiler and `int` with the Microsoft Visual C++ compiler.

`std::optional`

`std::optional` is very convenient when the value of our object can be null or empty.

`std::optional` is quite comfortable for calculations such as database queries that may have a result.

Don't use no-results

Before C++17 it was common practice to use a special value such as a null pointer, an empty string, or a unique integer to denote the absence of a result. These special values or no-results are very error-prone because you have to misuse the type system to check the return value. This means that for the type system that you have to use a regular value such as an empty string to define an irregular value.

The various constructors and the convenience function `std::make_optional` let you define an optional object `opt` with or without a value. `opt.emplace` will construct the contained value in-place and `opt.reset` will destroy the container value. You can explicitly ask a `std::optional` container if it has a value or you can check it in a logical expression. `opt.value` returns the value and `opt.value_or` returns the value or a default value. If `opt` has no contained value, the call `opt.value` will throw a `std::bad_optional_access` exception.

Here is a short example of using `std::optional`.

```
/*
// =====

// optional.cpp
// #include <iostream>
// #include <optional>
// #include <vector>

// std::optional<int> getFirst(const std::vector<int> &vec)
// {
//   if (!vec.empty())
//     return std::optional<int>(vec[0]);
//   else
//     return std::optional<int>();
// }

// int main()
// {

//   std::vector<int> myVec{1, 2, 3};
//   std::vector<int> myEmptyVec;

//   auto myInt = getFirst(myVec);

//   if (myInt)
//   {
//     std::cout << *myInt << std::endl;      // 1
//     std::cout << myInt.value() << std::endl; // 1
//     std::cout << myInt.value_or(2017) << std::endl; // 1
//   }

//   auto myEmptyInt = getFirst(myEmptyVec);

//   if (!myEmptyInt)
//   {
//     std::cout << myEmptyInt.value_or(2017) << std::endl; // 2017
//   }
}
```

```
// return 0;
// }
// =====
/*
```

I use std::optional in the function getFirst. getFirst returns the first element if it exists. If not, you will get a std::optional<int> object. The main function has two vectors. Both invoke getFirst and return a std::optional object. In the case of myInt the object has a value; in the case of myEmptyInt, the object has no value. The program displays the value of myInt and myEmptyInt. myInt.value_or(2017) returns the value, but myEmptyInt.value_or(2017) returns the default value.

std::variant, explained in the next section, can have more than one value.

```
*/
/*
std::variant
```

The last part of this section deals with std::variant which allows us to create a variable from any of the types specified in the std::variant container.

std::variant is a type-safe union. An instance of std::variant has a value from one of its types. The type must not be a reference, array or void. A std::variant can have a type more than once. A default-initialised std::variant is initialised with its first type; therefore, its first type must have a default constructor. By using var.index you get the zero-based index of the alternative held by the std::variant var.

var.valueless_by_exception returns false if the variant holds a value. By using var.emplace you can create a new value in-place. There are a few global functions used to access a std::variant. The function template var.holds_alternative lets you check if the std::variant holds a specified alternative. You can use std::get with an index and with a type as argument. By using an index, you will get the value. If you invoke std::get with a type, you only will get the value if it is unique. If you use an invalid index or a non-unique type, you will get a std::bad_variant_access exception. In contrast to std::get which eventually returns an exception, std::get_if returns a null pointer in the case of an error.

The following code snippet shows you the usage of a std::variant.

```
/*
// #include <variant>
// #include <string>
// #include <cassert>

// using namespace std::literals;

// int main()
//{
// std::variant<int, float> v, w;
// v = 12; // v contains int
// int i = std::get<int>(v);
// w = std::get<int>(v);
// w = std::get<0>(v); // same effect as the previous line
// w = v;           // same effect as the previous line

// // std::get<double>(v); // error: no double in [int, float]
// // std::get<3>(v);    // error: valid index values are 0 and 1
```

```

// try
// {
//   std::get<float>(w); // w contains int, not float: will throw
// }
// catch (const std::bad_variant_access &)
// {
// }

// std::variant<std::string> x("abc"); // converting constructors work when unambiguous
// x = "def";                      // converting assignment also works when unambiguous

// std::variant<std::string, bool> y("abc"); // casts to bool when passed a char const *
// assert(std::holds_alternative<bool>(y)); // succeeds
// y = "xyz"s;
// assert(std::holds_alternative<std::string>(y)); // succeeds
// }
// =====
/*

```

v and w are two variants. Both can have an int and a float value. Their default value is 0. v becomes 12 and the following call std::get<int>(v) returns the value. The next three lines show three possibilities to assign the variant v to w, but you have to keep a few rules in mind. You can ask for the value of a variant by type std::get<double>(v) or by index: std::get<3>(v). The type must be unique and the index valid. The variant w holds an int value; therefore, I get a std::bad_variant_access exception if I ask for a float type. If the constructor call or assignment call is unambiguous, a conversion can take place. This is the reason that it's possible to construct a std::variant<std::string> from a C-string or assign a new C-string to the variant.

std::variant has an interesting non-member function std::visit that allows you to execute a callable on a list of variants. A callable is something which you can invoke. Typically this can be a function, a function object, or lambda expression. For simplicity reasons, I use a lambda function in this example.

```

*/
// =====
// visit.cpp
// #include <iostream>
// #include <variant>
// #include <vector>

// using namespace std;

// int main()
// {
//   std::vector<std::variant<char, long, float, int, double, long long>>
//     vecVariant = {5, '2', 5.4, 100ll, 2011l, 3.5f, 2017};

//   for (auto &v : vecVariant)
//   {
//     std::visit([](auto &&arg)
//     {
//       std::cout << arg << " ";
//     },
//     v);
//   } // 5 2 5.4 100 2011 3.5 2017
// }
```

```

// cout << std::endl;

// // display each type
// for (auto &v : vecVariant)
// {
//   std::visit([](auto &&arg)
//   {
//     std::cout << typeid(arg).name() << " ";
//   },
//   v);
//   // i c d x l f i (these letters refer to int char double __int64 long float int respectively
// }
// cout << endl;

// // get the sum
// std::common_type<char, long, float, int, double, long long>::type res{};

// std::cout << typeid(res).name() << std::endl; // d (for double)

// for (auto &v : vecVariant)
// {
//   std::visit([&res](auto &&arg)
//   {
//     res += arg;
//   },
//   v);
// }
// std::cout << "res: " << res << std::endl; // res: 4191.9

// // double each value
// for (auto &v : vecVariant)
// {
//   std::visit([](auto &&arg)
//   {
//     arg *= 2;
//   },
//   v);
//   std::visit([](auto &&arg)
//   {
//     std::cout << arg << " ";
//   },
//   v);
//   // 10 d 10.8 200 4022 7 4034
// }
// return 0;
// */
// =====
/*

```

Each variant in this example can hold a char, long, float, int, double, or long long. The first visitor [](auto&& arg){std::cout << arg << " ";} will output the various variants. The second visitor std::cout << typeid(arg).name() << " ";} will display its types.

Now I want to sum up the elements of the variants. First I need the right result type at compile time. std::common_type from the type traits library will provide it. std::common_type gives the type to which all types char, long, float, int, double, and long long can implicitly be converted to. The final {} in res{} causes it to be initialised to 0.0. res is of type double.

The visitor [&res](auto&& arg){arg *= 2;} doubles each element and the following line displays the result.
*/

```
/*
container in general:
```

Introduction

This chapter deals with the different features that are present in all types of C++ containers.

Although the sequential and associative containers of the Standard Template library are two very different classes of containers, they have a lot in common. For example, the operations used to create or delete a container, to determine its size, to access its elements, to assign or swap, are all independent of the type of elements in a container. It is common for the containers to be defined with an arbitrary size. The reason is that each container has an allocator, hence the size of a container can be adjusted at runtime. The allocator works in the background most of the time. This can be seen for an `std::vector`. The call `std::vector<int>` results in a call `std::vector<int, std::allocator<int>>`. Because of the `std::allocator` we can adjust the size of all containers dynamically, except for `std::array`. However, they have even more in common. We can access the elements of a container quite easily with an iterator.

Having so much in common, the containers differ in their details. The chapters Sequential Containers and Associative Containers in General provide these details.

C++ covers the sequential containers `std::array`, `std::vector`, `std::deque`, `std::list`, and `std::forward_list`, in detail.

The same holds true for associative containers, which can be classified in the ordered and unordered associative container.

Further information#

Sequential Containers

Associative Containers in General

`std::array`

`std::vector`

`std::deque`

`std::list`

`std::forward_list`

Create and Delete

Below, we examine the different methods available for constructing and destroying containers with particular parameters.

We can construct each container using a multitude of constructors. To delete all elements of a container `cont`, use `cont.clear()`. It makes no difference if you create and delete a container or if we add and remove elements. Each time the container takes care of memory management.

The table shows the constructors and destructors of a container. All these functions can be used with `std::vector`.

```

Default std::vector<int> vec1
Range std::vector<int> vec2(vec1.begin(), vec1.end())
Copy std::vector<int> vec3(vec2)
Copy std::vector<int> vec3 = vec2
Move std::vector<int> vec4(std::move(vec3))
Move std::vector<int> vec4 = std::move(vec3)
Sequence (Initializer list) std::vector<int> vec5 {1, 2, 3, 4, 5}
Sequence (Initializer list) std::vector<int> vec5 = {1, 2, 3, 4, 5}
Destructor vec5.~vector()
Delete elements vec5.clear()
Creation and deletion of a container

```

Because `std::array` is generated at compile-time, there are a few things that are special. `std::array` has no move constructor and can't be created with a range or with an initializer list. However, an `std::array` can be initialized with an aggregate initialization. Also, `std::array` has no method for removing its elements.

Now we can use the different constructors on the different containers.

```

*/
// =====
// containerConstructor.cpp
// #include <iostream>
// #include <map>
// #include <unordered_map>
// #include <vector>
// using namespace std;

// int main()
//{
// vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// map<string, int> m = {"bart", 12345}, {"jenne", 34929}, {"huber", 840284};
// unordered_map<string, int> um{m.begin(), m.end()};

// for (auto v : vec)
//   cout << v << " "; // 1 2 3 4 5 6 7 8 9
// cout << "\n";
// for (auto p : m)
//   cout << p.first << "," << p.second << " "; // bart,12345 huber,840284 jenne,34929
// cout << "\n";
// for (auto p : um)
//   cout << p.first << "," << p.second << " "; // bart,12345 jenne,34929 huber,840284
// cout << "\n";

// vector<int> vec2 = vec;
// cout << vec.size() << endl; // 9
// cout << vec2.size() << endl; // 9

// vector<int> vec3 = move(vec);
// cout << vec.size() << endl; // 0
// cout << vec3.size() << endl; // 9

```

```

// for (int i = 0; i < vec3.size(); i++)
// {
//   cout << " my code: " << i << " index: " << vec3.at(i) << endl;
// }

// vec3.clear();
// cout << vec3.size() << endl; // 0
// return 0;
//}

//=====
/*
Size

```

For a container cont, use cont.empty() to see if the container is empty. cont.size() returns the current number of elements, and cont.max_size() returns the maximum number of elements cont can have. The maximum number of elements is implementation defined.

```

*/
//=====
// containerSize.cpp
// #include <iostream>
// #include <map>
// #include <set>
// #include <vector>

// using namespace std;

// int main()
// {
//   vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
//   map<string, int> str2Int = {"bart", 12345},
//                      {"jenne", 34929},
//                      {"huber", 840284};
//   set<double> douSet{3.14, 2.5};
//   cout << boolalpha;
//   cout << intVec.empty() << endl; // false
//   cout << str2Int.empty() << endl; // false
//   cout << douSet.empty() << endl; // false

//   cout << intVec.size() << endl; // 9
//   cout << str2Int.size() << endl; // 3
//   cout << douSet.size() << endl; // 2

//   cout << intVec.max_size() << endl; // 4611686018427387903
//   cout << str2Int.max_size() << endl; // 256204778801521550

//   // min_size does not exist
//   cout << douSet.max_size() << endl; // 461168601842738790
//   return 0;
// }
```

```
// =====
/*
 Use cont.empty() instead of cont.size()
For a container cont, use the method cont.empty() instead of (cont.size() == 0) to determine if the container is empty. First, cont.empty() is, in general, faster than (cont.size() == 0); second, the container std::forward_list has no method size().
```

To access the elements of a container, we can use an iterator. A begin and end iterator forms a range, which can be processed further. For a container cont, cont.begin() is the begin iterator and cont.end() is the end iterator, which defines a half-open range. It is half-open because the begin iterator belongs to the range, the end iterator refers to a position past the range. With the iterator pair cont.begin() and cont.end() we can modify the elements.

Iterator	Description
cont.begin() and cont.end()	Pair of iterators to iterate forward.
cont.cbegin() and cont.cend()	Pair of iterators to iterate const forward.
cont.rbegin() and cont.rend()	Pair of iterators to iterate backward.
cont.crbegin() and cont.crend()	Pair of iterators to iterate const backward.
Functions available for iterators in containers	
*	

```
// =====
// containerAccess.cpp
```

```
// here all begin end etc are the address of that index.
```

```
// so must use a pointer .
```

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

```
// struct MyInt
//{
// MyInt(int i) : myInt(i){};
// int myInt;
//};
```

```
// int main()
//{
// std::vector<MyInt> myIntVec;
// myIntVec.push_back(MyInt(5));
// myIntVec.emplace_back(1);
// std::cout << myIntVec.size() << std::endl; // 2
```

```
// std::vector<int> intVec;
// intVec.assign({1, 2, 3});
// for (auto v : intVec)
// std::cout << v << " "; // 1 2 3
```

```

// cout << std::endl;

// intVec.insert(intVec.begin(), 0);
// for (auto v : intVec)
//   std::cout << v << " "; // 0 1 2 3
// cout << std::endl;

// intVec.insert(intVec.begin() + 4, 4);
// for (int v : intVec)
//   std::cout << v << " "; // 0 1 2 3 4
// cout << std::endl;

// cout << *(intVec.begin() + 3) << endl;

// intVec.insert(intVec.end(), {5, 6, 7, 8, 9, 10, 11});

// for (auto v : intVec)
//   std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10 11
// cout << std::endl;

// for (auto revIt = intVec.rbegin(); revIt < intVec.rend(); ++revIt)
//   std::cout << *revIt << " "; // 11 10 9 8 7 6 5 4 3 2 1 0
// cout << std::endl;

// intVec.pop_back();
// for (auto v : intVec)
//   std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10
// cout << std::endl;

// return 0;
//}
// =====
/*
Assign and Swap

```

This lesson deals with ways to update and swap values in containers.

We can assign new elements to an existing container and, if required, swap two containers as well. If we want to assign a container `cont2` to another container `cont1`, we can do so either through copy assignment `cont1=cont2` or move assignment `cont1=std::move(cont2)`. In a move assignment, the value of `cont2` is copied to `cont1` and `cont2` becomes empty. A special form of assignment is the one with an initializer list: `cont={1, 2, 3, 4, 5}`. In the case of `std::array`, an initializer list is not possible, hence we use aggregate initialization. The function `swap` exists in two forms. It is a method `cont1(swap(cont2))` and also a function template `std::swap(cont1, cont2)`.

```
*/
// =====
```

```
// containerAssignmentAndSwap.cpp
// #include <iostream>
// #include <set>

// int main(){
```

```

// std::set<int> set1{0, 1, 2, 3, 4, 5};
// std::set<int> set2{6, 7, 8, 9};

// for (auto s: set1) std::cout << s << " "; // 0 1 2 3 4 5
// std::cout << "\n";
// for (auto s: set2) std::cout << s << " "; // 6 7 8 9
// std::cout << "\n";

// set1= set2;
// for (auto s: set1) std::cout << s << " "; // 6 7 8 9
// std::cout << "\n";
// for (auto s: set2) std::cout << s << " "; // 6 7 8 9
// std::cout << "\n";

// set1= std::move(set2); //moves value of set2 in set1 and set2 becomes empty
// for (auto s: set1) std::cout << s << " "; // 6 7 8 9
// std::cout << "\n";
// for (auto s: set2) std::cout << s << " "; // prints null since set2 becomes empty after the move operation

// set2= {60, 70, 80, 90};
// for (auto s: set1) std::cout << s << " "; // 6 7 8 9
// std::cout << "\n";
// for (auto s: set2) std::cout << s << " "; // 60 70 80 90
// std::cout << "\n";

// std::swap(set1, set2);
// for (auto s: set1) std::cout << s << " "; // 60 70 80 90
// std::cout << "\n";
// for (auto s: set2) std::cout << s << " "; // 6 7 8 9
// std::cout << "\n";
// return 0;
// }

//=====
/*
Compare
All the general comparison operators work on containers.

Containers support the comparison operators ==, !=, <, >, <=, and >=. The comparison of two containers
happens on the elements of the containers. When associative containers are compared, their keys are
compared. Unordered associative containers support only the comparison operator == and !=.
*/

```

```
//=====
```

```

// containerComparison.cpp
// #include <iostream>
// #include <array>
// #include <set>
// #include <unordered_map>
// #include <vector>

```

```

// using namespace std;

// // output 1 represents true and 0 represents false
// int main()
//{
// vector<int> vec1{1, 2, 3, 4};
// vector<int> vec2{1, 2, 3, 4};
// cout << (vec1 == vec2) << endl; // 1

// array<int, 4> arr1{1, 2, 3, 4};
// array<int, 4> arr2{1, 2, 3, 4};
// cout << (arr1 == arr2) << endl; // 1

// set<int> set1{1, 2, 3, 4};
// set<int> set2{4, 3, 2, 1};
// cout << (set1 == set2) << endl; // 1

// set<int> set3{1, 2, 3, 4, 5};
// cout << (set1 < set3) << endl; // 1

// set<int> set4{1, 2, 3, -3};
// cout << (set1 > set4) << endl; // 1

// unordered_map<int, string> uSet1{{1, "one"}, {2, "two"}};
// unordered_map<int, string> uSet2{{1, "one"}, {2, "Two"}};
// cout << (uSet1 == uSet2) << endl; // 0

// return 0;
//}
//=====================================================================
/*
sequential containers:
```

Introduction

The first category of containers we'll study are sequential containers. Listed below are the major features for various types of sequential containers.

The sequential containers have a lot in common, but each container has its special domain. Before we dive into the details, we'll look at an overview of all five sequential containers of the std namespace.

Criteria	Array	Vector	Deque	List	Forward List
Size	static	dynamic	dynamic	dynamic	dynamic
Implementation	static array	dynamic array	sequence of arrays	doubled linked list	single linked list
Access	random	random	random	forward and backward	forward
Optimized for insert and delete at	—	—	end: O(1)	begin and end: O(1)	begin and end: O(1); arbitrary: O(1); begin: O(1); arbitrary: O(1)
Memory reservation	—	yes	no	no	no
Release of memory	—	shrink_to_fit	shrink_to_fit	always	always

Strength** no memory allocation; minimal memory requirements 95% solution insertion and deletion at the begin and end insertion and deletion at an arbitrary position fast insertion and deletion; minimal memory requirements

Weakness no dynamic memory; memory allocation Insertion and deletion; at an arbitrary position: O(n) Insertion and deletion; at an arbitrary position: O(n) no random access no random access

The sequential containers

A few additional remarks to bring to the table.

O(1) stands for the complexity (runtime) of an operation. So O(1) means that the runtime of an operation on a container is constant and is independent of the size of the container. Opposite to that, O(n) means that the runtime depends linearly on the number of the elements of the container. What does that mean for an std::vector? The access time on an element is independent of the size of the std::vector, but the insertion or deletion of an arbitrary element with k-times more elements is k-times slower.

Although the random access on the elements of an std::vector has the same complexity, O(1), as the random access on the element of an std::deque, that doesn't mean, that both operations are equally fast.

The complexity guarantee O(1) for the insertion or deletion of a double (std::list) or single linked list (std::forward_list) is only guaranteed if the iterator points to the right element.

i std::string is like std::vector

Of course std::string is not a container of the standard template library. But from a behavioral point of view, it is like a sequential container, specifically an std::vector<char>. Therefore, we will treat std::string like an std::vector<char>.

*/

/*

=====

=====

Arrays

The array type is perhaps the most popular sequential container. This lesson will cover its properties in detail.

This is what an array looks like: |1|2|3|4|5|

std::array is a homogeneous container of fixed length. It requires the header <array>. An instance of std::array combines the memory and runtime characteristic of a C array with the interface of an std::vector. In particular, an std::array knows its size. We can use STL algorithms on instances of std::array.

Keep a few special rules in mind for initializing an std::array.

std::array<int, 10> arr: The 10 elements are not initialized.

std::array<int, 10> arr{}: The 10 elements initialized to 0 by default.

std::array<int, 10> arr{1, 2, 3, 4, 5}: The unspecified elements are initialized to 0 by default.

std::array supports three types of index access.

```
- arr[n];
- arr.at(n);
- std::get<n>(arr);
```

The most commonly used first type of index access using angle brackets does not check the boundaries of the arr. This is in contrast to arr.at(n). We will eventually get an std::range-error exception. The last form in the above snippet shows the relationship of std::array with the std::tuple, because both are containers of fixed length.

Here is a little bit of arithmetic using std::array:

```
/*
// =====

// array.cpp
// #include <iostream>
// #include <array>
// #include <vector>
// #include <numeric>

// using namespace std;

// int main()
//{
// std::array<int, 10> arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// for (auto a : arr)
// std::cout << a << " "; // 1 2 3 4 5 6 7 8 9 10
// cout << "\n";

// double sum = accumulate(arr.begin(), arr.end(), 0);
// std::cout << sum << std::endl; // 55

// double mean = sum / arr.size();
// std::cout << mean << std::endl;      // 5.5
// std::cout << (arr[0] == std::get<0>(arr)); // 1 (1 represents true)

// return 0;
//}
// =====
/*
```

Vectors

Vectors are more refined version of arrays. They simplify the insertion and deletion of values.

```
/*
/*
```

std::vector is a homogeneous container, for which it's length can be adjusted at runtime. std::vector needs the header <vector>. As it stores its elements contiguously in memory, std::vector supports pointer arithmetic.

```
for (int i= 0; i < vec.size(); ++i){
    std::cout << vec[i] == *(vec + i) << std::endl; // true
}
```

Make sure to distinguish the round and curly braces in the creation of an std::vector

If we construct a std::vector, we must keep a few things in mind. The constructor with round braces in the following example creates an std::vector with a capacity of 10 elements, while the constructor with curly braces creates an std::vector with the element 10.

```
std::vector<int> vec(10);
std::vector<int> vec{10};
```

The same rules hold true for the expressions std::vector<int>(10, 2011) or std::vector<int>{10, 2011}. In the first case, we get an std::vector with 10 elements, initialised to 2011. In the second case, we get an std::vector with the elements 10 and 2011. The reason for this behaviour is that curly braces are interpreted as initialiser lists so the sequence constructor is used.

```
/*
// #include <utility>
// #include <vector>

// int main()
//{
//    std::vector<int> first;
//    std::vector<int> second(4, 2011);
//    std::vector<int> third(second.begin(), second.end());
//    std::vector<int> forth(second);
//    std::vector<int> fifth(std::move(second));
//    std::vector<int> sixth{1, 2, 3, 4, 5};
//}
/*
*/
```

Size vs. Capacity#

The number of elements an std::vector has usually take up less space than what is already reserved. There is a simple reason for this. With extra memory already allocated, the size of the std::vector can increase without an expensive allocation of new memory.

There are a few methods for smartly handling memory:

Method	Description
vec.size()	Returns the number of elements of vec.
vec.capacity()	Returns the number of elements, which vec can have without reallocation.
vec.resize(n)	vec will be increased to n elements.
vec.reserve(n)	Reserve memory for at least n elements.
vec.shrink_to_fit()	Reduces capacity of vec to the size.

Memory management of std::vector

The call vec.shrink_to_fit() is not binding. That means the runtime can ignore it. But on popular platforms, I always observed the desired behavior.

```
/*
// =====
// vector.cpp
```

```

// #include <iostream>
// #include <vector>

// int main()
//{
// std::vector<int> intVec1(5, 2011);
// intVec1.reserve(10);
// std::cout << intVec1.size() << std::endl; // 5
// std::cout << intVec1.capacity() << std::endl; // 10

// intVec1.shrink_to_fit();
// std::cout << intVec1.capacity() << std::endl; // 5

// std::vector<int> intVec2(10);
// std::cout << intVec2.size() << std::endl; // 10

// std::vector<int> intVec3{10};
// std::cout << intVec3.size() << std::endl; // 1

// std::vector<int> intVec4{5, 2011};
// std::cout << intVec4.size() << std::endl; // 2
// return 0;
//}
//=====================================================================
/*

```

std::vector vec has a few methods to access its elements. vec.front(), yields the first element, and vec.back() yields the last element of vec. To read or write the $(n+1)$ -th element of vec, we can use the index operator vec[n] or the method vec.at(n). The second one checks the boundaries of vec, so that we eventually get an std::range_error exception.

Besides the index operator, std::vector offers additional methods to assign, insert, create or remove elements. See the following overview.

Method	Description
vec.assign(...)	Assigns one or more elements, a range or an initializer list.
vec.clear()	Removes all elements from vec.
vec.emplace(pos, args ...)	Creates a new element before poswith the args in vec and returns the new position of the element.
vec.emplace_back(args ...)	Creates a new element in vec with args
vec.erase(...)	Removes one element or a range and returns the next position.
vec.insert(pos, ...)	Inserts one or more elements, a range or an initializer list and returns the new position of the element.
vec.pop_back()	Removes the last element.
vec.push_back(elem)	Adds a copy of elem at the end of vec.

Modify the elements of a std::vector

Further information#

std::vector

std::vector

C++ Containers library std::vector

```

Defined in header <vector>
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
(1)
namespace pmr {
    template< class T >
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
(2) (since C++17)

```

1) std::vector is a sequence container that encapsulates dynamic size arrays.

2) std::pmr::vector is an alias template that uses a polymorphic allocator.

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

The storage of the vector is handled automatically, being expanded as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using capacity() function. Extra memory can be returned to the system via a call to shrink_to_fit(). (since C++11)

Reallocations are usually costly operations in terms of performance. The reserve() function can be used to eliminate reallocations if the number of elements is known beforehand.

The complexity (efficiency) of common operations on vectors is as follows:

Random access - constant O(1)

Insertion or removal of elements at the end - amortized constant O(1)

Insertion or removal of elements - linear in the distance to the end of the vector O(n)

std::vector (for T other than bool) meets the requirements of Container, AllocatorAwareContainer, SequenceContainer , ContiguousContainer (since C++17) and ReversibleContainer.

Member functions of std::vector are constexpr: it is possible to create and use std::vector objects in the evaluation of a constant expression.

However, std::vector objects generally cannot be constexpr, because any dynamically allocated storage must be released in the same evaluation of constant expression.

(since C++20)

Template parameters

T - The type of the elements.

T must meet the requirements of CopyAssignable and CopyConstructible. (until C++11)

The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of Erasable, but many member functions impose stricter requirements. (since C++11)

(until C++17)

The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type meets the requirements of Erasable, but many member

functions impose stricter requirements. This container (but not its members) can be instantiated with an incomplete element type if the allocator satisfies the allocator completeness requirements.

Feature-test macro	Value	Std	Comment
<code>__cpp_lib_incomplete_container_elements</code>	201505L	(C++17)	Minimal incomplete type support (since C++17)
Allocator	-		An allocator that is used to acquire/release memory and to construct/destroy the elements in that memory. The type must meet the requirements of Allocator. The behavior is undefined (until C++20)The program is ill-formed (since C++20) if Allocator::value_type is not the same as T.
Specializations			The standard library provides a specialization of std::vector for the type bool, which may be optimized for space efficiency.
vector<bool>			
space-efficient dynamic bitset (class template specialization)			
Iterator invalidation			
Operations	Invalidated		
All read only operations	Never		
swap, std::swap	end()		
clear, operator=, assign	Always		
reserve, shrink_to_fit	If the vector changed capacity, all of them. If not, none.		
erase	Erased elements and all elements after them (including end())		
push_back, emplace_back	If the vector changed capacity, all of them. If not, only end().		
insert, emplace	If the vector changed capacity, all of them. If not, only those at or after the insertion point (including end()).		
resize	If the vector changed capacity, all of them. If not, only end() and any elements erased.		
pop_back	The element erased and end().		
Member types			
Member type	Definition		
value_type	T		
allocator_type	Allocator		
size_type	Unsigned integer type (usually std::size_t)		
difference_type	Signed integer type (usually std::ptrdiff_t)		
reference	value_type&		
const_reference	const value_type&		
pointer			
Allocator::pointer	(until C++11)		
std::allocator_traits<Allocator>::pointer	(since C++11)		
const_pointer			
Allocator::const_pointer	(until C++11)		
std::allocator_traits<Allocator>::const_pointer	(since C++11)		
iterator			
LegacyRandomAccessIterator and LegacyContiguousIterator	to value_type		
(until C++20)			
LegacyRandomAccessIterator, contiguous_iterator, and ConstexprIterator	to value_type		

(since C++20)

`const_iterator`

`LegacyRandomAccessIterator` and `LegacyContiguousIterator` to `const value_type`

(until C++20)

`LegacyRandomAccessIterator`, `contiguous_iterator`, and `ConstexprIterator` to `const value_type`

(since C++20)

`reverse_iterator` `std::reverse_iterator<iterator>`

`const_reverse_iterator` `std::reverse_iterator<const_iterator>`

Member functions

(constructor)

constructs the vector

(public member function)

(destructor)

destructs the vector

(public member function)

`operator=`

assigns values to the container

(public member function)

`assign`

assigns values to the container

(public member function)

`get_allocator`

returns the associated allocator

(public member function)

Element access

`at`

access specified element with bounds checking

(public member function)

`operator[]`

access specified element

(public member function)

`front`

access the first element

(public member function)

`back`

access the last element

(public member function)

`data`

direct access to the underlying array

(public member function)

Iterators

begin

cbegin

(C++11)

returns an iterator to the beginning

(public member function)

end

cend

(C++11)

returns an iterator to the end

(public member function)

rbegin

crbegin

(C++11)

returns a reverse iterator to the beginning

(public member function)

rend

crend

(C++11)

returns a reverse iterator to the end

(public member function)

Capacity

empty

checks whether the container is empty

(public member function)

size

returns the number of elements

(public member function)

max_size

returns the maximum possible number of elements

(public member function)

reserve

reserves storage

(public member function)

capacity

returns the number of elements that can be held in currently allocated storage

(public member function)

`shrink_to_fit`

(C++11)

reduces memory usage by freeing unused memory

(public member function)

Modifiers

`clear`

clears the contents

(public member function)

`insert`

inserts elements

(public member function)

`emplace`

(C++11)

constructs element in-place

(public member function)

`erase`

erases elements

(public member function)

`push_back`

adds an element to the end

(public member function)

`emplace_back`

(C++11)

constructs an element in-place at the end

(public member function)

`pop_back`

removes the last element

(public member function)

`resize`

changes the number of elements stored

(public member function)

`swap`

swaps the contents

(public member function)

Non-member functions

`operator==`

```
operator!=  
operator<  
operator<=  
operator>  
operator>=  
operator<=>
```

```
(removed in C++20)  
(C++20)
```

lexicographically compares the values in the vector
(function template)
`std::swap(std::vector)`

specializes the `std::swap` algorithm
(function template)
`erase(std::vector)`
`erase_if(std::vector)`

(C++20)

Erases all elements satisfying specific criteria
(function template)
Deduction guides(since C++17)

Example

Run this code

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

```
int main()  
{  
    // Create a vector containing integers  
    std::vector<int> v = {7, 5, 16, 8};  
  
    // Add two more integers to vector  
    v.push_back(25);  
    v.push_back(13);
```

```
    // Print out the vector  
    std::cout << "v = { ";  
    for (int n : v)  
        std::cout << n << ", ";  
    std::cout << "}; \n";  
}
```

Output:

```
v = { 7, 5, 16, 8, 25, 13, };
```

```
*/  
/*  
Deques:  
std::deque(double ended queue), which consist of a sequence of arrays, is quite similar to std::vector.  
std::deque needs the header <deque>. The std::deque has three additional methods deq.push_front(elem),  
deq.pop_front(), and deq.emplace_front(args... ) to add or remove elements at the beginning.  
  
*/  
  
// ======  
// deque.cpp  
// #include <iostream>  
// #include <deque>  
// using namespace std;  
  
// class MyInt  
// {  
// private:  
//   int myInt;  
  
// public:  
//   MyInt(int i) : myInt(i){};  
//   friend ostream &operator<<(ostream &os, const MyInt &m)  
//   {  
//     os << m.myInt << " ";  
//     return os;  
//   }  
// };  
  
// int main()  
// {  
//   std::deque<MyInt> myIntDeq;  
  
//   myIntDeq.push_back(MyInt(5));  
//   myIntDeq.emplace_back(1);  
//   std::cout << myIntDeq.size() << std::endl; // 2  
  
//   std::deque<MyInt> intDeq;  
  
//   intDeq.assign({1, 2, 3});  
//   for (auto v : intDeq)  
//     cout << v << " "; // 1 2 3  
//   cout << endl;  
  
//   intDeq.insert(intDeq.begin(), 0);  
//   for (auto v : intDeq)  
//     cout << v << " "; // 0 1 2 3  
//   cout << endl;
```

```

// intDeq.insert(intDeq.begin() + 4, 4);
// for (auto v : intDeq)
//   cout << v << " "; // 0 1 2 3 4
// cout << endl;

// intDeq.insert(intDeq.end(), {5, 6, 7, 8, 9, 10, 11});
// for (auto v : intDeq)
//   cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10 11
// cout << endl;

// for (auto revIt = intDeq.rbegin(); revIt != intDeq.rend(); ++revIt)
//   std::cout << *revIt << " "; // 11 10 9 8 7 6 5 4 3 2 1 0
// cout << endl;

// intDeq.pop_back();
// for (auto v : intDeq)
//   cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10
// cout << endl;

// intDeq.push_front(-1);
// for (auto v : intDeq)
//   cout << v << " "; // -1 0 1 2 3 4 5 6 7 8 9 10
// cout << endl;

// return 0;
//}
// =====
/*

```

list:

std::list is a doubled linked list. std::list needs the header <list>.

Although it has a similar interface to std::vector or std::deque, std::list is quite different from both of them. That's due to its structure.

std::list makes the following points unique:

It supports no random access.

Accessing an arbitrary element is slow because we might have to iterate through the whole list.

Adding or removing an element is fast, if the iterator points to the right place.

If we add or remove an element, the iterator remains valid.

Because of its special structure, std::list has a few special methods.

Special methods of std::list#

Method	Description
--------	-------------

lis.merge(c) Merges the sorted list c into the sorted list lis, so that lis remains sorted.

lis.merge(c, op) Merges the sorted list c into the sorted list lis, so that lis remains sorted. Uses op as sorting criteria.

lis.remove(val) Removes all elements from lis with value val.

```

lis.remove_if(pre)    Removes all elements from lis, fulfilling the predicate pre.
lis.splice(pos, ... ) Splits the elements in lis before pos. The elements can be single elements, ranges or
lists.
lis.unique()    Removes adjacent element with the same value.
lis.unique(pre)Removes adjacent elements, fulfilling the predicate pre.
*/
// =====
// #include <iostream>
// #include <list>
// #include <algorithm>

// int main()
//{
// std::list<int> list1{15, 2, 18, 19, 4, 15, 1, 3, 18, 5,
//                     4, 7, 17, 9, 16, 8, 6, 6, 17, 1, 2};

// list1.sort();
// for (auto l : list1)
//   std::cout << l << " ";
// // 1 1 2 2 3 4 4 5 6 6 7 8 9 15 15 16 17 17 18 18 19
// std::cout << std::endl;

// list1.unique();
// for (auto l : list1)
//   std::cout << l << " ";
// // 1 2 3 4 5 6 7 8 9 15 16 17 18 19
// std::cout << std::endl;

// std::list<int> list2{10, 11, 12, 13, 14};

// list1.splice(std::find(list1.begin(), list1.end(), 15), list2);
// for (auto l : list1)
//   std::cout << l << " ";
// // 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

// return 0;
//}
// =====
/*

```

Forward Lists

A forward list is the primitive form of the list structure we studied in the previous lesson. Nevertheless, forward lists are still useful.

`std::forward_list` is a singly linked list, which needs the header `<forward_list>`. `std::forward_list` has a drastically reduced interface and is optimized for minimal memory requirements.

`std::forward_list` has a lot in common with `std::list`:

It doesn't support the random access.

The access of an arbitrary element is slow because in the worst case, we have to iterate forward through the whole list.

To add or remove an element is fast, if the iterator points to the right place.

If we add or remove an element, the iterator remains valid.

Operations always refer to the beginning of the std::forward_list or the position past the current element.

Being able to iterate through an std::forward_list forward has a great impact. The iterators cannot be decremented and therefore, operations like -- (decrement) on iterators are not supported. For the same reason, std::forward_list has no backward iterator. std::forward_list is the only sequential container which doesn't know its size.

 std::forward_list has a very special domain

std::forward_list is the replacement for single linked lists. It's optimized for minimal memory management and performance if the insertion, extraction, or movement of elements only affects adjacent elements. This is typical for sorting algorithms.

The following are the special methods of std::forward_list:

Method Description

forw.before_begin() Returns an iterator before the first element.

forw.emplace_after(pos, args...) Creates an element after pos with the arguments args....

forw.emplace_front(args...) Creates an element at the beginning of forw with the arguments args....

forw.erase_after(pos, ...) Removes from forw the element pos or a range of elements, starting with pos.

forw.insert_after(pos, ...) Inserts new elements after pos. These elements can be single elements, ranges or initialiser lists.

forw.merge(c) Merges the sorted forward list c into the sorted forward list forw, so that forw keeps sorted.

forw.merge(c, op) Merges the forward sorted list c into the forward sorted list forw, so that forw keeps sorted. Uses op as sorting criteria.

forw.splice_after(pos, ...) Splits the elements in forw before pos. The elements can be single elements, ranges or lists.

forw.unique() Removes adjacent element with the same value.

forw.unique(pre) Removes adjacent elements, fulfilling the predicate pre.

Special methods of std::forward_list

*/

```
// =====
// forwardList.cpp
// #include <iostream>
// #include <algorithm>
// #include <forward_list>

// using std::cout;

// int main(){
//   std::forward_list<int> forw;
//   std::cout << forw.empty() << std::endl; // 1 (1 denoted true)

//   forw.push_front(7);
//   forw.push_front(6);
//   forw.push_front(5);
//   forw.push_front(4);
//   forw.push_front(3);
```

```

// forw.push_front(2);
// forw.push_front(1);
// for (auto i: forw) cout << i << " "; // 1 2 3 4 5 6 7
// cout<<"\n";

// forw.erase_after(forw.before_begin());
// cout<< forw.front(); // 2
// cout<<"\n";

// std::forward_list<int> forw2;
// forw2.insert_after(forw2.before_begin(), 1);
// forw2.insert_after(++forw2.before_begin(), 2);
// forw2.insert_after(++(++(forw2.before_begin()))), 3);
// forw2.push_front(1000);
// for (auto i= forw2.cbegin();i != forw2.cend(); ++i) cout << *i << " "; // 1000 1 2 3
// cout<<"\n";

// auto IteratorTo5= std::find(forw.begin(), forw.end(), 5);
// forw.splice_after(IteratorTo5, std::move(forw2));
// for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " "; // 2 3 4 5 1000 1 2 3 6 7
// cout<<"\n";

// forw.sort();
// for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// // 1 2 2 3 3 4 5 6 7 1000
// cout<<"\n";

// forw.reverse();
// for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// // 1000 7 6 5 4 3 3 2 2 1
// cout<<"\n";

// forw.unique();
// for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// // 1000 7 6 5 4 3 2 1
// cout<<"\n";

// return 0;
// }
// =====
/*
assocaitive container in general :
```

C++ has eight different associative containers. Four of them are associative containers with sorted keys: `std::set`, `std::map`, `std::multiset`, and `std::multimap`. The other four are associative containers with unsorted keys: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset`, and `std::unordered_multimap`. The associative containers are special containers. That means they support all of the operations described in the chapter containers in general.

Overview#

All eight ordered and unordered containers have one thing in common: they associate a key with a value. We can use the key to get the value. To classify the associative containers, three simple questions need to be answered:

Are the keys sorted?

Does the key have an associated value?

Can a key appear more than once?

The following table with 2^3

2

3

= 8 rows gives the answers to the three questions. The answer to a fourth question is in the table. How fast is the access time of a key in the best case?

Associative container	Sorted	Associated value	More identical keys	Access time
std::set	yes	no	no	logarithmic
std::unordered_set	no	no	no	constant
std::map	yes	yes	no	logarithmic
std::unordered_map	no	yes	no	constant
std::multiset	yes	no	yes	logarithmic
std::unordered_multiset	no	no	yes	constant
std::multimap	yes	yes	yes	logarithmic
std::unordered_multimap	no	yes	yes	constant

Characteristics for associative containers

Since C++98, there have been ordered associative containers; with C++11 came unordered associative containers. Both classes have a very similar interface. That's the reason that the following code sample is identical for std::map and std::unordered_map. To be more precise, the interface of std::unordered_map is a superset of the interface of std::map. The same holds true for the remaining three unordered associative containers. So, porting your code from the ordered to unordered containers is quite easy.

We can initialize the containers with an initializer list and add new elements with the index operator. To access the first element of the key-value pair p, we have p.first, and for the second element, we have p.second. p.first is the key and p.second is the associated value of the pair.

```
/*
// =====
// orderedUnorderedComparison.cpp
// #include <iostream>
// #include <map>
// #include <unordered_map>

/// std::map
int main(){
    std::map<std::string, int> m {"Dijkstra", 1972}, {"Scott", 1976};
    m["Ritchie"] = 1983;
    std::cout << m["Ritchie"];      // 1983
    std::cout << "\n";

    for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
    //      // {Dijkstra,1972},{Ritchie,1983},{Scott,1976}
    std::cout << "\n";
```

```

// m.erase("Scott");
// for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
// // {Dijkstra,1972},{Ritchie,1983}
// std::cout << "\n";

// m.clear();
// std::cout << m.size() << std::endl; // 0

// // std::unordered_map

// std::unordered_map<std::string, int> um {"Dijkstra", 1972}, {"Scott", 1976};
// um["Ritchie"] = 1983;
// std::cout << um["Ritchie"]; // 1983
// std::cout << "\n";

// for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
// // {Ritchie,1983},{Scott,1976},{Dijkstra,1972}
// std::cout << "\n";

// um.erase("Scott");
// for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
// // {Ritchie,1983},{Dijkstra,1972}
// std::cout << "\n";

// um.clear();
// std::cout << um.size() << std::endl; // 0

// return 0;
//}
//=====
/*

```

There is a subtle difference between the two program executions: The keys of the std::map are ordered, the keys of the std::unordered_map are unordered. The question is: Why do we have such similar containers in C++? We already pointed this in the table above. The reason is so often the same: performance. The access time to the keys of an unordered associative container is constant and therefore independent of the size of the container. If the containers are big enough, the performance difference is significant. Have a look at the section about performance.

```

*/
// =====
// associativeContainerModify.cpp
// #include <iostream>
// #include <set>
// #include <array>

// int main(){
// std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};

// for (auto s: mySet) std::cout << s << " "; // 1 1 2 2 3 3 3 4 4 4 4 5 5 6 6 7
// std::cout << "\n";

```

```

// mySet.insert(8);
// std::array<int, 5> myArr{10, 11, 12, 13, 14};
// mySet.insert(myArr.begin(), myArr.begin() + 3);
// mySet.insert({22, 21, 20});
// for (auto s: mySet) std::cout << s << " ";
// // 1 1 2 2 3 3 3 4 4 4 5 5 6 6 7 10 11 12 20 21 22
// std::cout << "\n";

// std::cout << mySet.erase(4); // 4
// mySet.erase(mySet.lower_bound(5), mySet.upper_bound(15));
// for (auto s: mySet) std::cout << s << " ";
// // 1 1 2 2 3 3 3 20 21 22
// std::cout << "\n";

// return 0;
//}
// =====
/*

```

In the next two chapters, we will learn about the two kinds of associative containers:

ordered associative containers
 unordered associative containers
*/

/*
 ordered associative containers:

The ordered associative containers `std::map` and `std::multimap` associate their key with a value. Both are defined in the header `<map>`. `std::set` and `std::multiset` need the header `<set>`.

All four ordered containers are parametrized by their type, their allocator, and their comparison function. The containers have default values for the allocator and the comparison function, depending on the type. The declaration of `std::map` and `std::set` show this very nicely.

```
template < class key, class val, class Comp = less<key>,
          class Alloc = allocator<pair<const key, val> >
class map;
```

```
template < class T, class Comp = less<T>,
          class Alloc = allocator<T> >
class set;
```

The declaration of both associative containers shows that `std::map` has an associated value. The key and the value are used for the default allocator: `allocator<pair<const key, val>>`. With a little bit more imagination, we can derive more from the allocator. `std::map` has pairs of the type `std::pair<const key, val>`. The associated value `val` does not matter for the sort criteria: `less<key>`. All observations also hold for `std::multimap` and `std::multiset`.

In the next lesson, we'll discuss the properties of keys and values.

Keys and Values

The properties of keys and values are listed below.

There are special rules for the key and the value of an ordered associative container.

The key has to be

sortable (by default, they are sorted in ascending order)
copyable and moveable

The value has to be

default constructible
copyable and moveable

The key associated with the value builds a pair p so that we get a member with p.first and the value with p.second.

*/

```
// =====
// #include <iostream>
// #include <map>

// int main(){
// std::multimap<char, int> multiMap= {'a', 10}, {'a', 20}, {'b', 30};
// for (auto p: multiMap) std::cout << "{" << p.first << "," << p.second << "}" ;
//           // {a,10} {a,20} {b,30}
// return 0;
// }
// =====
/*
```

The Comparison Criterion

This lesson talks about the rules followed by ordered associative containers when comparing the values inside them.

The default comparison criterion of the ordered associative containers is `std::less`. If we want to use a user-defined type as the key, we have to overload the operator `<`. It's sufficient to overload the operator `<` for our data type because the C++ runtime compares, with the help of the relation `(! (elem1<elem2 || elem2<elem1))`, two elements for equality.

We can specify the sorting criterion as a template argument that must implement a strict weak ordering.

i Strict weak ordering

Strict weak ordering for a sorting criterion on a set S is given if the following requirements are met:

For every s from S it has to hold that $s < s$ is not possible.

For all s_1 and s_2 from S it must hold: If $s_1 < s_2$, then $s_2 < s_1$ is not possible.

For all s_1 , s_2 and s_3 with $s_1 < s_2$ and $s_2 < s_3$ the following must hold: $s_1 < s_3$.

For all s_1 , s_2 and s_3 with s_1 not comparable with s_2 and s_2 not comparable with s_3 the following must hold: s_1 is not comparable with s_3 .

In contrast to the definition of the strict weak ordering, the usage of a comparison criterion with strict weak ordering is a lot simpler for an `std::map`.

```
/*
// =====
```

```

// #include <iostream>
// #include <map>

// int main(){
//   std::map<int, std::string, std::greater<int>> int2Str{
//     {5, "five"}, {1, "one"}, {4, "four"}, {3, "three"},
//     {2, "two"}, {7, "seven"}, {6, "six"} };
//   for (auto p: int2Str) std::cout << "{" << p.first << "," << p.second << "}";
//   // {7,seven} {6,six} {5,five} {4,four} {3,three} {2,two} {1,one}

//   return 0;
// }
// =====
/*

```

Special Search Functions

The functions listed in this lesson make searching more efficient.

Ordered associative containers are optimized for searching, and so they offer unique search functions.

Search function Description

ordAssCont.count(key) Returns the number of values with the key.

ordAssCont.find(key) Returns the iterator of key in ordAssCont. If there is no key in ordAssCont it returns ordAssCont.end().

ordAssCont.lower_bound(key) Returns the iterator to the first key in ordAssCont in which key would be inserted.

ordAssCont.upper_bound(key) Returns the last position of key in ordAssCont in which key would be inserted.

ordAssCont.equal_range(key) Returns the range ordAssCont.lower_bound(key) and ordAssCont.upper_bound(key) in a std::pair.

Special search functions of the ordered associative containers

Now, the application of the special search functions.

*/

```

// =====
// associativeContainerSearch.cpp
// #include <iostream>
// #include <set>

// int main(){
//   std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};

//   for (auto s: mySet) std::cout << s << " "; // 1 1 2 2 3 3 3 4 4 4 4 5 5 6 6 7
//   std::cout << "\n";

//   mySet.erase(mySet.lower_bound(4), mySet.upper_bound(4));
//   for (auto s: mySet) std::cout << s << " "; // 1 1 2 2 3 3 3 5 5 6 6 7
//   std::cout << "\n";

//   std::cout << mySet.count(3) << std::endl; // 4
//   std::cout << *mySet.find(3) << std::endl; // 3

```

```

// std::cout << *mySet.lower_bound(3) << std::endl; // 3
// std::cout << *mySet.upper_bound(3) << std::endl; // 5
// auto pair= mySet.equal_range(3);
// std::cout << "(" << *pair.first << "," << *pair.second << ")"; // (3,5)

// return 0;
//}
//=====
/*
Maps

```

Now, we shall look at the features of `std::map` which make it such a popular container.

`std::map` is by far the most frequently used associative container. The reason is simple; It combines adequate performance with a very convenient interface. We can access its elements via the index operator. If the key doesn't exist, `std::map` creates a key-value pair. For the value, the default constructor is used.

 Consider `std::map` as a generalization of `std::vector`

Often, `std::map` is called an associative array because `std::map` supports the index operator like a sequential container. The subtle difference is that its index is not restricted to a number like in the case of `std::vector`. Its index can be almost any arbitrary type.

The same observations hold for its namesake `std::unordered_map`.

In addition to the index operator, `std::map` supports the `at` method. The compiler checks the `at` function to make sure it is not out of bounds. So if the request key doesn't exist in the `std::map`, an `std::out_of_range` exception is thrown.

In the next lesson, we'll analyze some code to better understand this concept.

```

*/
// =====
// #include <regex>
// #include <algorithm>
// #include <cstdlib>
// #include <fstream>
// #include <iostream>
// #include <sstream>
// #include <string>
// #include <map>
// #include <unordered_map>
// #include <utility>

// using str2Int = std::unordered_map<std::string, size_t>;
// using intAndWords = std::pair<std::size_t, std::vector<std::string>>;
// using int2Words = std::map<std::size_t, std::vector<std::string>>;

/// // count the frequency of each word
// str2Int wordCount(const std::string& text){
// std::regex wordReg(R"(\w+)");
// std::sregex_iterator wordItBegin(text.begin(), text.end(), wordReg);
// const std::sregex_iterator wordItEnd;
```

```

// str2Int allWords;
// for (; wordItBegin != wordItEnd; ++wordItBegin){
//   ++allWords[wordItBegin->str()];
// }
// return allWords;
// }

/// get to all frequencies the appropriate words
// int2Words frequencyOfWords(str2Int& wordCount){
// int2Words freq2Words;
// for ( auto wordIt: wordCount ){
//   auto freqWord= wordIt.second;
//   if ( freq2Words.find(freqWord) == freq2Words.end() ){
//     freq2Words.insert( intAndWords(freqWord, {wordIt.first} ) );
//   }
//   else {
//     freq2Words[freqWord].push_back(wordIt.first);
//   }
// }
// return freq2Words;
// }

// int main(int argc, char* argv[]){
// std::cout << std::endl;

// // get the filename
// std::string myFile = "Test.rtf";

// // open the file
// std::ifstream file(myFile, std::ios::in);
// if (!file){
//   std::cerr << "Can't open file " + myFile + "!" << std::endl;
//   exit(EXIT_FAILURE);
// }

// // read the file
// std::stringstream buffer;
// buffer << file.rdbuf();
// std::string text(buffer.str());

// // get the frequency of each word
// auto allWords= wordCount(text);

// std::cout << "The first 20 (key, value)-pairs: " << std::endl;
// auto end= allWords.begin();
// std::advance(end, 20);
// for (auto pair= allWords.begin(); pair != end; ++pair){
//   std::cout << "(" << pair->first << ":" << pair->second << ")";
// }
// std::cout << "\n\n";

```

```

// std::cout << "allWords[Web]: " << allWords["Web"] << std::endl;
// std::cout << "allWords[The]: " << allWords["The"] << "\n\n";

// std::cout << "Number of unique words: ";
// std::cout << allWords.size() << "\n\n";

// size_t sumWords=0;
// for (auto wordIt: allWords) sumWords+= wordIt.second;
// std::cout << "Total number of words: " << sumWords << "\n\n";

// auto allFreq= frequencyOfWords(allWords);

// std::cout << "Number of different frequencies: " << allFreq.size() << "\n\n";

// std::cout << "All frequencies: ";
// for (auto freqIt: allFreq) std::cout << freqIt.first << " ";
// std::cout << "\n\n";

// std::cout << "The most frequently occurring word(s): " << std::endl;
// auto atTheEnd= allFreq.rbegin();
// std::cout << atTheEnd->first << ":";
// for (auto word: atTheEnd->second) std::cout << word << " ";
// std::cout << "\n\n";

// std::cout << "All word which appears more then 1000 times:" << std::endl;
// auto biggerIt= std::find_if(allFreq.begin(), allFreq.end(), [](intAndWords iAndW){return iAndW.first > 1000;});
// if (biggerIt == allFreq.end()){
//   std::cerr << "No word appears more then 1000 times !" << std::endl;
//   exit(EXIT_FAILURE);
// }
// else{
//   for (auto allFreqIt= biggerIt; allFreqIt != allFreq.end(); ++allFreqIt){
//     std::cout << allFreqIt->first << ":";
//     for (auto word: allFreqIt->second) std::cout << word << " ";
//     std::cout << std::endl;
//   }
// }
// std::cout << std::endl;
// */

```

Overview

The main difference between unordered and ordered associative containers is the idea of sorted keys. Let's find out how unordered containers handle keys.

With the new C++11 standard, C++ has four unordered associative containers: `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set`, and `std::unordered_multiset`. They have a lot in common with

their namesakes, the ordered associative containers. The difference is that the unordered ones have a richer interface and their keys are not sorted.

This shows the declaration of an std::unordered_map.

```
/*
// =====
// template <class key, class val, class Hash = std::hash<key>,
//     class KeyEqual = std::equal_to<key>,
//     class Alloc = std::allocator<std::pair<const key, val>>>
// class unordered_map;
// =====
/*
```

Like std::map, std::unordered_map has an allocator, but std::unordered_map needs no comparison function. Instead std::unordered_map needs two additional functions: One to determine the hash value of its key: std::hash<key> and a second to compare the keys for equality: std::equal_to<key>. Because of the three default template parameters, we only have to provide the type of the key and the type of the value. For example, declaration of std::unordered_map would be std::unordered_map<char,int> unordMap.

Keys and Values

This lesson is about the different properties of keys and values in this type of associative container.

There are special rules for the key and value of an unordered associative container.

The key has to be

comparable
available as hash value
copyable or moveable
The value has to be

default constructible
copyable or moveable

In the next lesson, we'll discuss how the performance of unordered containers is better than ordered containers.

Performance

The unordered associative container type is more optimized when compared to its ordered sibling.

Performance. That's the simple reason why the unordered associative containers were so long missed in C++. In the example below, one million randomly created values are read from an std::map and std::unordered_map with 10 million values. The impressive result is that the linear access time of an unordered associative container is 20 times faster than the access time of an ordered associative container. That is just the difference between constant and logarithmic complexity O(log n) of these operations.

Note: The given code might take more time to execute than normal.

```
/*
// =====
// #include <chrono>
// #include <iostream>
// #include <map>
```

```

// #include <random>
// #include <unordered_map>

// static const long long mapSize = 10000000;
// static const long long accSize = 1000000;

// int main()
//{
// std::cout << std::endl;

// std::map<int, int> myMap;
// std::unordered_map<int, int> myHash;

// for (long long i = 0; i < mapSize; ++i)
// {
//   myMap[i] = i;
//   myHash[i] = i;
// }

// std::vector<int> randValues;
// randValues.reserve(accSize);

// // random values
// std::random_device seed;
// std::mt19937 engine(seed());
// std::uniform_int_distribution<> uniformDist(0, mapSize);
// for (long long i = 0; i < accSize; ++i)
//   randValues.push_back(uniformDist(engine));

// auto start = std::chrono::system_clock::now();
// for (long long i = 0; i < accSize; ++i)
// {
//   myMap[randValues[i]];
// }
// std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
// std::cout << "time for std::map: " << dur.count() << " seconds" << std::endl;

// auto start2 = std::chrono::system_clock::now();
// for (long long i = 0; i < accSize; ++i)
// {
//   myHash[randValues[i]];
// }
// std::chrono::duration<double> dur2 = std::chrono::system_clock::now() - start2;
// std::cout << "time for std::unordered_map: " << dur2.count() << " seconds" << std::endl;

// std::cout << std::endl;
//}

//=====
/*
The Hash Function

```

Here, we will understand why hash functions are essential in unordered associative pairs.

We'll cover the following

The hash function

The reason for the constant access time of an unordered associative container is the hash function. The hash function maps the key to its value (its hash value). A hash function is good if it produces as few collisions as possible and equally distributes the keys onto the buckets. Because the execution of the hash function takes a constant amount of time, accessing the elements in the base case is also constant.

The hash function#

is already defined for the built-in types like boolean, natural numbers, and floating point numbers.

is available for std::string and std::wstring.

generates, for a C string, const char a hash value of the pointer address.

can be defined for user-defined data types.

For user-defined types, which are used as a key for an unordered associative container, we have to keep two requirements in mind: They need a hash function and an equality operator to be defined in order for them to be compared.

*/

```
// =====
// unorderedMapHash.cpp
// #include <iostream>
// #include <unordered_map>

// struct MyInt{
//   MyInt(int v):val(v){}
//   bool operator==(const MyInt& other) const {
//     return val == other.val;
//   }
//   int val;
// };

// struct MyHash{
//   std::size_t operator()(MyInt m) const {
//     std::hash<int> hashVal;
//     return hashVal(m.val);
//   }
// };

// std::ostream& operator<< (std::ostream& st, const MyInt& myIn){
//   st << myIn.val ;
//   return st;
// }

// int main(){
//   typedef std::unordered_map<MyInt, int, MyHash> MyIntMap;
```

```

// MyIntMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

// for(auto m : myMap) std::cout << "{" << m.first << "," << m.second << "}";
//   // {MyInt(1),1} {MyInt(0),0} {MyInt(-1),-1} {MyInt(-2),-2}

// std::cout << myMap[MyInt(-2)] << std::endl; // -2
// return 0;
// }
// =====
/*

```

The Details

We'll finish this section by learning the terminology present in unordered associative containers.

The unordered associative containers store their indices in buckets. In which bucket the index goes depends on the hash function, which maps the key to the index. If different keys are mapped to the same index, it's called a collision. The hash function tries to avoid this.

Indices are typically stored in the bucket as a linked list. Since the access to the bucket is constant, the access to the linked list in the bucket is linear. The number of buckets is called capacity, the average number of elements for each bucket is called the load factor. In general, the C++ runtime generates new buckets if the load factor is greater than 1. This process is called rehashing and can also be triggered explicitly:

```

*/
// =====
// hashInfo.cpp
// #include <iostream>
// #include <unordered_set>
// using namespace std;

// void getInfo(const unordered_set<int> &hash)
// {
//   cout << "hash.bucket_count(): " << hash.bucket_count();
//   cout << "hash.load_factor(): " << hash.load_factor();
// }

// int main()
// {

//   // Create an unoredered set and initialize it with the array
//   // Set will contain only random elements
//   int arr[100];
//   for (int i = 0; i < 100; i++)
//     arr[i] = (rand() % 100) + 1;
//   unordered_set<int> hash(arr, arr + sizeof(arr) / sizeof(int));
//   cout << "hash.max_load_factor():\t" << hash.max_load_factor() << endl; // hash.max_load_factor():1

//   getInfo(hash);
//   // hash.bucket_count(): 103hash.load_factor(): 0.660194
//   cout << endl;

//   hash.insert(500);
//   cout << "hash.bucket(500):\t" << hash.bucket(500) << endl; // 88

```

```

// getInfo(hash);
// cout << endl;
// // hash.bucket_count(): 103hash.load_factor(): 0.669903

// hash.rehash(500);

// getInfo(hash);
// // hash.bucket_count(): 503hash.load_factor(): 0.137177500

// cout << endl
//     << "hash.bucket(500):\t" << hash.bucket(500); // hash.bucket(500):500

// return 0;
// }
// =====
/*
adaptors for containers
introduction

```

In this section, we will tackle three advanced types of sequential containers which are built on std::deque.

C++ has with std::stack, std::queue and std::priority_queue three special sequential containers. I guess, most of you know these classic data structures from your education.

The adaptors for containers

support a reduced interface for existing sequential containers,
can not be used with algorithms of the Standard Template Library,
are class templates which are parametrised by their data type and their container (std::vector, std::list and std::deque),
use by default std::deque as the internal sequential container:

```
template <typename T, typename Container= deque<T>>
class stack;
```

Stack

It's time to study the behavior of this popular data structure.

widget

The std::stack follows the LIFO principle (Last In First Out). The stack sta, which needs the header <stack>, has three special methods.

With sta.push(e) you can insert a new element e at the top of the
*/

```
// =====
// stack.cpp
// #include <iostream>
```

```

// #include <stack>

// int main()
//{
// std::stack<int> myStack;

// std::cout << myStack.empty() << std::endl; // true
// std::cout << myStack.size() << std::endl; // 0

// myStack.push(1);
// myStack.push(2);
// myStack.push(3);
// std::cout << myStack.top() << std::endl; // 3

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

// std::cout << std::endl
// << myStack.empty() << std::endl; // 1 (denotes true)
// std::cout << myStack.size() << std::endl; // 0

```

```

// return 0;
//}
//=====
/*
```

Queue

A queue follows the opposite principle of stack. It is a very powerful data structure in its own right.

widget

The std::queue follows the FIFO principle (First In First Out). The queue que, which needs the header <queue>, has four special methods.

With que.push(e) you can insert an element e at the end of the queue and remove the first element from the queue with que.pop(). que.back() enables you to refer to the last element in the que, que.front() to the first element in the que. std::queue has similar characteristics as std::stack. So you can compare std::queue instances and get their sizes. The operations of the queue have constant complexity.

```

*/
//=====
// #include <iostream>
// #include <queue>
```

```

// int main()
//{
// std::queue<int> myQueue;

// std::cout << myQueue.empty() << std::endl; // true
// std::cout << myQueue.size() << std::endl; // 0
```

```

// myQueue.push(1);
// myQueue.push(2);
// myQueue.push(3);
// std::cout << myQueue.back() << std::endl; // 3
// std::cout << myQueue.front() << std::endl; // 1

// while (!myQueue.empty())
// {
//   std::cout << myQueue.back() << " ";
//   std::cout << myQueue.front() << ":" ;
//   myQueue.pop();
// } // 3 1 : 3 2 : 3 3

// std::cout << std::endl
//       << myQueue.empty() << std::endl; // 1 (denotes true)
// std::cout << myQueue.size() << std::endl; // 0

// return 0;
// }

=====
/*
Priority Queue
By combining a queue and order, we get priority queues!

```

widget

The `std::priority_queue` is a reduced `std::queue`. It needs the header `<queue>`.

The difference to the `std::queue` is, that their biggest element is always at the top of the priority queue. `std::priority_queue` `pri` uses by default the comparison operator `std::less`. Similar to `std::queue`, `pri.push(e)` inserts a new element `e` into the priority queue. `pri.pop()` removes the first element of the `pri`, but does that with logarithmic complexity. With `pri.top()` you can reference the first element in the priority queue, which is the greatest one. The `std::priority_queue` knows its size, but didn't support the comparison operator on their instances.

*/

```

=====
// priorityQueue.cpp
// #include <iostream>
// #include <queue>

// int main(){
//   std::priority_queue<int> myPriorityQueue;

//   std::cout << "is empty:\t" << myPriorityQueue.empty() << std::endl; // 1 (denotes true)
//   std::cout << "size:\t\t" << myPriorityQueue.size() << std::endl; // 0

//   myPriorityQueue.push(3);
//   myPriorityQueue.push(1);
//   myPriorityQueue.push(2);
//   std::cout << "top:\t\t" << myPriorityQueue.top() << std::endl; // 3

```

```

// std::cout << "Data:\t";
// while (!myPriorityQueue.empty()){
//   std::cout << myPriorityQueue.top() << " ";
//   myPriorityQueue.pop();
// }                                // 3 2 1
// std::cout << std::endl;

// std::cout << "is empty:\t" << myPriorityQueue.empty() << std::endl; // 1 (denotes true)
// std::cout << "size:\t\t" << myPriorityQueue.size() << std::endl; // 0

// std::priority_queue<std::string, std::vector<std::string>,
//                     std::greater<std::string>> myPriorityQueue2;

// myPriorityQueue2.push("Only");
// myPriorityQueue2.push("for");
// myPriorityQueue2.push("testing");
// myPriorityQueue2.push("purpose");
// myPriorityQueue2.push(".");

// while (!myPriorityQueue2.empty()){
//   std::cout << myPriorityQueue2.top() << " ";
//   myPriorityQueue2.pop();
// }                                // . Only for purpose testing
// return 0;
// }
// =====
/*

```

iterators:

Introduction

In this part of the course, we will learn to implement iterators and use them to traverse our data.

On the one hand, iterators are generalizations of pointers which represents positions in a container. On the other hand, they provide powerful iteration and random access in a container.

Iterators are the glue between the generic containers and the generic algorithms of the Standard Template Library.

Iterators support the following operations:

`*:` Returns the element at the current position.

`==, !=:` Compares two positions.

`=:` Assigns a new value to an iterator.

The range-based for-loop uses the iterators implicitly.

Because iterators are not checked, they have the same issues as pointers.

`*/`

```

// =====
// #include <iostream>
```

```

// #include <vector>
// #include <queue>
// using namespace std;

// int main()
//{
// std::vector<int> vec{1, 23, 3, 3, 3, 4, 5};
// std::deque<int> deq;

// // Start iterator bigger than end iterator
// std::copy(vec.begin() + 2, vec.begin(), deq.begin());

// // Target container too small
// std::copy(vec.begin(), vec.end(), deq.end());
// return 0;
//}
//=====
/*

```

Categories

Iterators can be categorized into three primary types, each with its own advantages.

Their capabilities can categorize iterator. C++ has forward, bidirectional and random access iterators. With the forward iterator, you can iterate the container forward, with the bidirectional iterator, in both directions. With the random access iterator, you can directly access an arbitrary element. In particular, this means for the last one, that you can use iterator arithmetic and ordering comparisons (e.g.: `<`). The category of an iterator depends on the type of container used.

In the table below is a representation of containers and their iterator categories. The bidirectional iterator includes the forward iterator functionalities, and the random access iterator includes the forward and the bidirectional iterator functionalities. It and It2 are iterators, n is a natural number.

The iterator categories of the container:

Iterator category	Properties	Container
Forward iterator	<code>++It, It++, *It</code>	unordered associative container
<code>It == It2, It != It2</code>	<code>std::forward_list</code>	
Bidirectional iterator	<code>--It, It--</code>	ordered associative container
<code>std::list</code>		
Random access iterator	<code>It[i]</code>	<code>std::array</code>
<code>It+= n, It-= n</code>	<code>std::vector</code>	
<code>It+n, It-n</code>	<code>std::deque</code>	
<code>n+It</code>	<code>std::string</code>	
<code>It-It2</code>		
<code>It < It2, It <= It2, It > It2</code>		
<code>It >= It2</code>		

The iterator categories of the container

The input iterator and the output iterator are special forward iterators: they can read and write their pointed element only once.

```

*/
/*

```

Iterator Creation

In this lesson, we'll observe how a map creates and handles its iterator.

Each container generates its suitable iterator on request. For example, an std::unordered_map generates constant and non-constant forward iterators.

```
std::unordered_map<std::string, int>::iterator unMapIt= unordMap.begin();
std::unordered_map<std::string, int>::iterator unMapIt= unordMap.end();
```

```
std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cbegin();
std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cend();
```

In addition, std::map supports the backward iterators:

```
std::map<std::string, int>::reverse_iterator mapIt= map.rbegin();
std::map<std::string, int>::reverse_iterator mapIt= map.rend();
```

```
std::map<std::string, int>::const_reverse_iterator mapIt= map.rcbegin();
std::map<std::string, int>::const_reverse_iterator mapIt= map.rcend();
```

Use auto for iterator definition

Iterator definition is very labour intensive. The automatic type deduction with auto reduces the writing to the bare minimum.

```
std::map<std::string, int>::const_reverse_iterator
mapIt= map.rcbegin();
auto mapIt2= map.rcbegin();
*/
// =====
// #include <iostream>
// #include <string>
// #include <unordered_map>
// #include <vector>

// int main()
//{
//   std::cout << std::endl;

//   std::unordered_map<std::string, int> unordMap{{"Rainer", 1966}, {"Beatrix", 1966}, {"Juliette", 1997},
// {"Marius", 1999};

//   std::unordered_map<std::string, int>::const_iterator endMapIt = unordMap.end();
//   std::unordered_map<std::string, int>::iterator mapIt;

//   for (mapIt = unordMap.begin(); mapIt != endMapIt; ++mapIt)
//     std::cout << "{" << mapIt->first << ", " << mapIt->second << "}" << std::endl;

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

//   std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```

// std::vector<int>::const_iterator vecEndIt = myVec.end();
// std::vector<int>::iterator vecIt;
// for (vecIt = myVec.begin(); vecIt != vecEndIt; ++vecIt)
//   std::cout << *vecIt << " ";

// std::cout << std::endl;

// for (const auto v : myVec)
//   std::cout << v << " ";

// std::cout << std::endl;

// std::vector<int>::const_reverse_iterator vecEndRevIt = myVec.rend();
// std::vector<int>::reverse_iterator vecRevIt;
// for (vecRevIt = myVec.rbegin(); vecRevIt != vecEndRevIt; ++vecRevIt)
//   std::cout << *vecRevIt << " ";

// std::cout << "\n\n";
//}
//=====
/*

```

Useful Functions

C++ offers several tools to make the iteration process simpler yet safer.

The global functions `std::begin`, `std::end`, `std::prev`, `std::next`, `std::distance` and `std::advance` make your handling of the iterators a lot easier. Only the function `std::prev` requires a bidirectional iterator. All functions need the header `<iterator>`. The table gives you an overview:

Global function	Description
<code>std::begin(cont)</code>	Returns a begin iterator to the container cont.
<code>std::end(cont)</code>	Returns an end iterator to the container cont.
<code>std::rbegin(cont)</code>	Returns a reverse begin iterator to the container cont.
<code>std::rend(cont)</code>	Returns a reverse end iterator to the container cont.
<code>std::cbegin(cont)</code>	Returns a constant begin iterator to the container cont.
<code>std::cend(cont)</code>	Returns a constant end iterator to the container cont.
<code>std::crbegin(cont)</code>	Returns a reverse constant begin iterator to the container cont.
<code>std::crend(cont)</code>	Returns a reverse constant end iterator to the container cont.
<code>std::prev(it)</code>	Returns an iterator, which points to a position before it
<code>std::next(it)</code>	Returns an iterator, which points to a position after it.
<code>std::distance(fir, sec)</code>	Returns the number of elements between fir and sec.
<code>std::advance(it, n)</code>	Puts the iterator it n positions further.

Useful functions for iterators

Now, the application of the useful functions.

*/

```

// =====
// #include <array>
// #include <iostream>
// #include <iterator>

```

```

// #include <string>
// #include <unordered_map>

// int main()
//{
// std::cout << std::endl;

// std::unordered_map<std::string, int> myMap{{"Rainer", 1966}, {"Beatrix", 1966}, {"Juliette", 1997},
// {"Marius", 1999}};

// for (auto m : myMap)
// std::cout << "{" << m.first << ", " << m.second << "}" << std::endl;

// std::cout << std::endl;

// auto mapItBegin = std::begin(myMap);
// std::cout << "{" << mapItBegin->first << ", " << mapItBegin->second << "}" << std::endl;
// auto mapIt = std::next(mapItBegin);
// std::cout << "{" << mapIt->first << ", " << mapIt->second << "}" << std::endl;

// auto dist = std::distance(mapItBegin, mapIt);
// std::cout << "std::distance(mapItBegin, mapIt): " << dist << std::endl;

// std::cout << std::endl;

// std::array<int, 10> myArr{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// for (auto a : myArr)
// std::cout << a << " ";

// std::cout << std::endl;

// auto arrItEnd = std::end(myArr);
// auto arrIt = std::prev(arrItEnd);
// std::cout << *arrIt << std::endl;

// std::advance(arrIt, -5);
// std::cout << *arrIt << std::endl;

// std::cout << std::endl;
//}

=====
/*
Adaptors
Iterators can do more than just search through data, they can now insert values!

```

We'll cover the following

Insert iterators

Stream Iterators

Iterator adaptors enable the use of iterators in insert mode or with streams. They need the header <iterator>.

Insert iterators#

With the three insert iterators std::front_inserter, std::back_inserter and std::inserter you can insert an element into a container at the beginning, at the end, or an arbitrary position respectively. The memory for the elements will automatically be provided. The three map their functionality on the underlying methods of the container cont.

The table below gives you two pieces of information: Which methods of the containers are internally used and which iterators can be used depends on the container's type.

Name	Internally-used Method	Container
std::front_inserter(val)	cont.push_front(val)	std::deque
std::list		
std::back_inserter(val)	cont.push_back(val)	std::vector
std::deque		
std::list		
std::string		
std::inserter(val, pos)	cont.insert(pos, val)	std::vector
std::deque		
std::list		
std::string		
std::map		
std::set		

The three insert iterators

You can combine the algorithms in the STL with the three insert iterators.

```
/*
// =====
// #include <iostream>
// #include <iterator>
// #include <queue>
// #include <vector>
// #include <unordered_map>
// #include <algorithm>

// int main(){

// std::deque<int> deq{5, 6, 7, 10, 11, 12};
// std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

// std::copy(std::find(vec.begin(), vec.end(), 13), vec.end(), std::back_inserter(deq));

// for (auto d: deq) std::cout << d << " "; // 5 6 7 10 11 12 13 14 15
// std::cout << std::endl;

// std::copy(std::find(vec.begin(), vec.end(), 8),
// std::find(vec.begin(), vec.end(), 10),
// std::inserter(deq, std::find(deq.begin(), deq.end(), 10)));
// for (auto d: deq) std::cout << d << " "; // 5 6 7 8 9 10 11 12 13 14 15
// std::cout << std::endl;
```

```

// std::copy(vec.rbegin() + 11, vec.rend(),
// std::front_inserter(deq));
// for (auto d: deq) std::cout << d << " "; // 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
// std::cout << std::endl;

// return 0;
//}
// =====
/*

```

Stream Iterators#

Stream iterator adaptors can use streams as a data source or data sink. C++ offers two functions to create istream iterators and two to create ostream iterators. The created istream iterators behave like input iterators, the ostream iterators like insert iterators.

Function Description

`std::istream_iterator<T>` Creates an end-of-stream iterator.

`std::istream_iterator<T>(istream)` Creates an istream iterator for istream.

`std::ostream_iterator<T>(ostream)` Creates an ostream iterator for ostream

`std::ostream_iterator<T>(ostream, delim)` Creates an ostream iterator for ostream with the delimiter `delim`.

The four stream iterators

Thanks to the stream iterator adapter you can directly read from or write to a stream.

The following interactive program fragment reads in an endless loop natural numbers from `std::cin` and pushes them onto the vector `myIntVec`. If the input is not a natural number, an error in the input stream will occur. All numbers in `myIntVec` are copied to `std::cout`, separated by `:`. Now you can see the result on the console.

`*/`

```

// =====
// #include <iostream>
// #include <iterator>
// #include <queue>
// #include <vector>
// #include <unordered_map>
// #include <algorithm>
// #include <iterator>

// int main(){
// std::vector<int> myIntVec;
// std::istream_iterator<int> myIntStreamReader(std::cin);
// std::istream_iterator<int> myEndIterator;

// // Possible input
// // 1
// // 2
// // 3
// // 4
// // z
// while(myIntStreamReader != myEndIterator){

```

```

// myIntVec.push_back(*myIntStreamReader);
// ++myIntStreamReader;
// }

// std::copy(myIntVec.begin(), myIntVec.end(), std::ostream_iterator<int>(std::cout, ":"));
//      // 1:2:3:4:

// return 0;
//}
//=====
/*

```

callable objects:

Introduction

Functions, function objects and lambda functions are all part of the callable units class. They are 'called' to retrieve data or perform an action.

 This chapter is intentionally not exhaustive

This course is about the C++ Standard library, therefore, it will no go into the details of callable units. I provide only as much information as it's necessary to use them correctly in the algorithms of the Standard Template Library. An exhaustive discussion of callable units should be part of a course about the C++ core language.

Many of the STL algorithms and containers can be parametrized with callable units or short callables. A callable is something that behaves like a function. Not only are these functions but also function objects and lambda functions. Predicates are special functions that return a boolean as the result. If a predicate has one argument, it's called a unary predicate if a predicate has two arguments, it's called a binary predicate. The same holds for functions. A function taking one argument is a unary function; a function taking two arguments is a binary function.

 To change the elements of a container, your algorithm must get them by reference

Callables can receive their arguments by value or by reference from their container. To modify the elements of the container, they must address them directly, so it is necessary that the callable gets them by reference.

```

*/
/*
Functions and Function Objects
Functions#

```

Functions are the simplest callables. They can have - apart from static variables - no state. Because the definition of a function is often widely separated from its use or even in a different translation unit, the compiler has fewer opportunities to optimize the resulting code.

```

*/
// =====

// #include <iostream>
// #include <vector>
// #include <algorithm>

// void square(int& i) { i = i * i; }
```

```
// int main(){  
// std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
// std::for_each(myVec.begin(), myVec.end(), square);  
// for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

```
// return 0;  
// }
```

```
=====
```

```
/*
```

Function Objects#

At first, don't call them functors. That's a well-defined term from the category theory.

Function objects are objects that behave like functions. They achieve this due to their call operator being implemented. As function objects are objects, they can have attributes and therefore state.

```
*/
```

```
=====
```

```
// #include <iostream>
```

```
// #include <vector>
```

```
// #include <algorithm>
```

```
// struct Square
```

```
// {
```

```
// void operator()(int &i) { i = i * i; }
```

```
// };
```

```
// int main()
```

```
// {
```

```
// std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
// std::for_each(myVec.begin(), myVec.end(), Square());
```

```
// for (auto v : myVec)
```

```
// std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

```
// return 0;
```

```
// }
```

```
=====
```

```
/*
```

💡 Instantiate function objects to use them

It's a common error that only the name of the function object (Square) and not the instance of function object (Square()) is used in an algorithm: `std::for_each(myVec.begin(), myVec.end(), Square)`. That's of course an error. You have to use the instance: `std::for_each(myVec.begin(), myVec.end(), Square())`

```
*/
```

```
/*
```

Predefined Function Objects#

C++ offers a bunch of predefined function objects. They need the header `<functional>`. These predefined function objects are very useful to change the default behaviour of the containers. For example, the keys of the ordered associative containers are by default sorted with the predefined function object `std::less`. But you may want to use `std::greater` instead:

```
std::map<int, std::string> myDefaultMap; // std::less<int>
```

```
std::map<int, std::string, std::greater<int>> mySpecialMap; // std::greater<int>
```

There are function objects in the Standard Template Library for arithmetic, logic, and bitwise operations, and also for negation and comparison. Here are a few of these predefined function objects:

Function object for Representative

Negation std::negate<T>()

Arithmetic std::plus<T>(), std::minus<T>()

std::multiplies<T>(), std::divides<T>()

std::modulus<T>()

Comparison std::equal_to<T>(), std::not_equal_to<T>()

std::less<T>(), std::greater<T>()

std::less_equal<T>(), std::greater_equal<T>()

Logical std::logical_not<T>()

std::logical_and<T>(), std::logical_or<T>()

Bitwise std::bit_and<T>(), std::bit_or<T>()

std::bit_xor<T>()

Predefined function objects

Lambda Functions

Lambda functions provide us all the functionality we need, on the go. They are faster than usual functions.

Lambda functions provide in-place functionality. The compiler gets a lot of insight and has therefore great optimization potential. Lambda functions can receive their arguments by value or by reference. They can capture their environment by value, by reference, and with C++14 by move.

```
*/
```

```
// =====
```

```
// #include <iostream>
```

```
// #include <vector>
```

```
// #include <algorithm>
```

```
// int main(){
```

```
// std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
// std::for_each(myVec.begin(), myVec.end(), [](int& i){
```

```
// i= i*i;
```

```
// std::cout << i << " ";
```

```
// }); // 1 4 9 16 25 36 49 64 81 100
```

```
// return 0;
```

```
// }
```

```
// =====
```

```
/*
```

 Lambda functions should be your first choice

If the functionality of your callable is short and self-explanatory, use a lambda function. A lambda function is, in general, faster and easier to understand.

```
*/
```

```
/*
```

algorithms:

Introduction

This chapter brings us to the various computing algorithms supported by C++17.

The Standard Template Library has a large number of algorithms to work with containers and their elements. As the algorithms are function templates, they are independent of the type of container elements. The glue between the containers and algorithms are the iterators. If your container supports the interface of an STL container, you can apply the algorithms to your container.

```
/*
// =====
// algorithm.cpp
// #include <iostream>
// #include <algorithm>
// #include <deque>
// #include <iostream>
// #include <list>
// #include <string>
// #include <vector>

// template <typename Cont, typename T>
// void doTheSame(Cont cont, T t){
//   for (const auto c: cont) std::cout << c << " ";
//   std::cout << std::endl;
//   std::cout << cont.size() << std::endl;
//   std::reverse(cont.begin(), cont.end());
//   for (const auto c: cont) std::cout << c << " ";
//   std::cout << std::endl;
//   std::reverse(cont.begin(), cont.end());
//   for (const auto c: cont) std::cout << c << " ";
//   std::cout << std::endl;
//   auto lt= std::find(cont.begin(), cont.end(), t);
//   std::reverse(lt, cont.end());
//   for (const auto c: cont) std::cout << c << " ";
// }

// int main(){
//   std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
//   std::deque<std::string> myDeq({"A", "B", "C", "D", "E", "F", "G", "H", "I"});
//   std::list<char> myList({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'});

//   doTheSame(myVec, 5);
//   std::cout << "\n\n";
//   // 1 2 3 4 5 6 7 8 9 10
//   // 10
//   // 10 9 8 7 6 5 4 3 2 1
//   // 1 2 3 4 5 6 7 8 9 10
//   // 1 2 3 4 10 9 8 7 6 5

//   doTheSame(myDeq, "D");
//   std::cout << "\n\n";
//   // A B C D E F G H I
//   // 9
//   // I H G F E D C B A
```

```

// // A B C D E F G H I
// // A B C I H G F E D

// doTheSame(myList, 'd');
// std::cout << "\n\n";
// // a b c d e f g h
// // 8
// // h g f e d c b a
// // a b c d e f g h
// // a b c h g f e d

// return 0;
//}
//=====
/*

```

Conventions

What are the rules and terminologies needed to run algorithms? Let's find out.

To use the algorithms, you have to keep a few rules in your head.

The algorithms are defined in various headers.

<algorithm>:

Contains the general algorithms.

<numeric>:

Contains the numeric algorithms. Many of the algorithms have the name suffix `_if` and `_copy`.

`_if`:

The algorithm can be parametrized by a predicate.

`_copy`:

The algorithm copies its elements in another range.

Algorithms like `auto num = std::count(Init first, Init last, const T& val)` return the number of elements that are equal to `val`. `num` is of type `iterator_traits<Init>::difference_type`. You have the guarantee that `num` is sufficient to hold the result. Because of the automatic return type deduction with `auto`, the compiler will give you the right types.

 If the container uses an additional range, it has to be valid

The algorithm `std::copy_if` uses an iterator to the beginning of its destination range. This destination range has to be valid.

Naming conventions for the algorithms

I use a few naming conventions for the type of arguments and the return type of the algorithms to make them easier to read.

Name Description

`Init` [Input iterator]

`FwdIt` [Forward iterator]

`Bilt` [Bidirectional iterator]

UnFunc [Unary callable]

BiFunc [Binary callable]

UnPre [Unary predicate]

BiPre [Binary predicate]

Search The searcher encapsulates the search algorithm.

ValType From the input range automatically deduced value type.

ExePol Execution policy

Signature of the algorithms

*/

/*

Iterators are the glue

Without iterators, there would be no way for us to move through the container and alter it according the algorithm. Hence, the iterator forms the backbone of this process.

Iterators define the range of the container on which the algorithms work. They describe a half-open range. In a half-open range the begin iterator points to the beginning, and the end iterator points to one position after the range.

The iterators can be categorized based on their capabilities. See the Categories section of the chapter on Iterators. The algorithms provide conditions to the iterators. Like in the case of std::rotate, most of the times a forward iterator is sufficient. But that doesn't hold for std::reverse. std::reverse requires a bidirectional iterator.

Sequential, parallel, or parallel execution with vectorisation

We will now learn how to execute our algorithm according to a certain execution policy.

We'll cover the following

Execution Policies

Without Optimisation

With maximum Optimisation

By using an execution policy in C++17, you can specify whether the algorithm should run sequentially, in parallel, or in parallel with vectorization.

Execution Policies#

The policy tag specifies whether an algorithm should run sequentially, in parallel, or in parallel with vectorization.

std::execution::seq: runs the algorithm sequentially

std::execution::par: runs the algorithm in parallel on multiple threads

std::execution::par_unseq: runs the algorithm in parallel on multiple threads and allows the interleaving of individual loops; permits a vectorised version with SIMD (Single Instruction Multiple Data) extensions.

The following code snippet shows all execution policies.

*/

// =====

// #include <iostream>

// #include <vector>

```

// #include <algorithm>

// int main(){
// std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};

// // standard sequential sort
// std::sort(v.begin(), v.end());

// // sequential execution
// std::sort(std::execution::seq, v.begin(), v.end());

// // permitting parallel execution
// std::sort(std::execution::par, v.begin(), v.end());

// // permitting parallel and vectorised execution
// std::sort(std::execution::par_unseq, v.begin(), v.end());
// return 0;
//}

/*

```

The example shows that you can still use the classic variant of `std::sort` without execution policy. In addition, in C++17 you can specify explicitly whether the sequential, parallel, or the parallel and vectorised version should be used.

i Parallel and Vectorised Execution

Whether an algorithm runs in a parallel and vectorized way depends on many factors. For example, it depends on whether the CPU and the operating system support SIMD instructions. Additionally, it depends on the compiler and the optimization level that you use to translate your code.

The following example shows a simple loop for creating a new vector.

```

const int SIZE= 8;

int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
int res[] = {0, 0, 0, 0, 0, 0, 0, 0};

int main(){
    for (int i = 0; i < SIZE; ++i) {
        res[i] = vec[i] + 5;
    }
}

```

The expression `res[i] = vec[i] + 5` is the key line in this small example. Thanks to the compiler Explorer we can have a closer look at the assembler instructions generated by x86-64 clang 3.6.

Without Optimisation#

Here are the assembler instructions. Each addition is done sequentially.

```

movslq -8(%rbp), %rax
movl vec(%rax,4), %ecx
addl $5, %ecx
movslq -8(%rbp), %rax

```

```
movl %ecx, res(,%rax,4)
```

With maximum Optimisation#

By using the highest optimisation level, -O3, special registers such as xmm0 that can hold 128 bits or 4 ints are used. This means that the addition takes place in parallel on four elements of the vector.

```
movdqa .LCPIO_0(%rip), %xmm0 # xmm0 = [5,5,5,5]
movdqa vec(%rip), %xmm1
padd %xmm0, %xmm1
movdqa %xmm1, res(%rip)
padd vec+16(%rip), %xmm0
movdqa %xmm0, res+16(%rip)
xorl %eax, %eax
*/
/*
```

Algorithms with Parallelized Versions

A list of all the algorithms which can be parallelized.

Here are the 77 algorithms with parallelized versions.

```
std::adjacent_difference    std::adjacent_find    std::all_of
std::any_of     std::copy      std::copy_if
std::copy_n     std::count     std::count_if
std::equal       std::exclusive_scan  std::fill
std::fill_n     std::find      std::find_end
std::find_first_of   std::find_if    std::find_if_not
std::for_each    std::for_each_n   std::generate
std::generate_n   std::includes   std::inclusive_scan
std::inner_product std::inplace_merge std::is_heap
std::is_heap_until  std::is_partitioned std::is_sorted
std::is_sorted_until std::lexicographical_compare std::max_element
std::merge        std::min_element  std::minmax_element
std::mismatch    std::move      std::none_of
std::nth_element   std::partial_sort std::partial_sort_copy
std::partition    std::partition_copy std::reduce
std::remove       std::remove_copy  std::remove_copy_if
std::remove_if std::replace    std::replace_copy
std::replace_copy_if std::replace_if std::reverse
std::reverse_copy  std::rotate     std::rotate_copy
std::search        std::search_n   std::set_difference
std::set_intersection std::set_symmetric_difference std::set_union
std::sort          std::stable_partition std::stable_sort
std::swap_ranges   std::transform  std::transform_exclusive_scan
std::transform_inclusive_scan std::transform_reduce std::uninitialized_copy
std::uninitialized_copy_n  std::uninitialized_fill std::uninitialized_fill_n
std::unique        std::unique_copy
```

The 77 algorithms with parallelised versions

Availability of the Parallel STL

As far as I know, at the time this book is being updated to C++17 (October 2017), there is no standard-conforming implementation of the parallel STL available yet. Therefore, you have to install third-party frameworks such as HPX. The HPX (High-Performance ParalleX) is a framework that is a general-purpose C++

runtime system for parallel and distributed applications of any scale. HPX has already implemented the parallel STL in a different namespace.

```
*/  
/*  
for_each
```

As the name suggests, this method picks up each value in our container and performs the desired action.

std::for_each applies a unary callable to each element of its range. The range is given by the input iterators.

```
UnFunc std::for_each(InplIt first, InplIt second, UnFunc func)  
void std::for_each(ExePol pol, FwdIt first, FwdIt second, UnFunc func)
```

std::for_each when used without an explicit execution policy is a special algorithm because it returns its callable argument. If you invoke std::for_each with a function object, you can store the result of the function call directly in the function object.

```
InplIt std::for_each_n(InplIt first, Size n, UnFunc func)  
FwdIt std::for_each_n(ExePol pol, FwdIt first, Size n, UnFunc func)
```

std::for_each_n is new with C++17 and applies a unary callable to the first n elements of its range. The range is given by an input iterator and a size.

```
*/  
// ======  
// #include <array>  
// #include <algorithm>  
// #include <iostream>  
// #include <vector>  
  
// template <typename T>  
// class ContainerInfo  
// {  
// public:  
// void operator()(T t)  
// {  
// num++;  
// sum += t;  
// }  
  
// int getSum() const  
// {  
// return sum;  
// }  
  
// int getSize() const { return num; }  
  
// double getMean() const  
// {  
// return static_cast<double>(sum) / static_cast<double>(num);  
// }  
  
// private:
```

```

// T sum{0};
// int num{0};
// };

// int main()
//{
// std::cout << std::endl;

// std::vector<double> myVec{1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};

// auto vecInfo = std::for_each(myVec.begin(), myVec.end(), ContainerInfo<double>());
// std::cout << "vecInfo.getSum(): " << vecInfo.getSum() << std::endl;
// std::cout << "vecInfo.getSize(): " << vecInfo.getSize() << std::endl;
// std::cout << "vecInfo.getMean(): " << vecInfo.getMean() << std::endl;

// std::cout << std::endl;

// std::array<int, 100> myArr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// auto arrInfo = std::for_each(myArr.begin(), myArr.end(), ContainerInfo<int>());
// std::cout << "arrInfo.getSum(): " << arrInfo.getSum() << std::endl;
// std::cout << "arrInfo.getSize(): " << arrInfo.getSize() << std::endl;
// std::cout << "arrInfo.getMean(): " << arrInfo.getMean() << std::endl;

// std::cout << std::endl;
//}

// =====
/*
non modifyable algorithm:
Non-modifying algorithms are algorithms for searching and counting elements. However, you can also check properties on ranges, compare ranges or search for ranges within ranges.

```

Search Elements

This lesson will cover the different ways of searching for values using 'find'.

You can search for elements in three different ways.

Returns an element in a range:

```
InPlt find(InPlt first, InPlt last, const T& val)
InPlt find(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

```
InPlt find_if(InPlt first, InPlt last, UnPred pred)
InPlt find_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
```

```
InPlt find_if_not(InPlt first, InPlt last, UnPred pre)
InPlt find_if_not(ExePol pol, FwdIt first, FwdIt last, UnPred pre)
```

Returns the first element of a range in a range:

```
FwdIt1 find_first_of(Inplt1 first1, Inplt1 last1, FwdIt2 first2, FwdIt2 last2)
FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2)
```

```
FwdIt1 find_first_of(Inplt1 first1, Inplt1 last1, FwdIt2 first2, FwdIt2 last2, BiPre pre)
FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, BiPre pre)
```

Returns identical, adjacent elements in a range:

```
FwdIt adjacent_find(FwdIt first, FwdIt last)
FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last)
```

```
FwdIt adjacent_find(FwdIt first, FwdIt last, BiPre pre)
FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

The algorithms require input or forward iterators as arguments and return an iterator on the element when successfully found. If the search is not successful, they return an end iterator.

```
/*
// =====
// #include <iostream>
// #include <algorithm>
// #include <set>
// #include <list>
// using namespace std;

// bool isVowel(char c){
//   string myVowels{"aeiouäöü"};
//   set<char> vowels(myVowels.begin(), myVowels.end());
//   return (vowels.find(c) != vowels.end());
// }

// int main(){
//   list<char> myCha{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
//   int cha[] = {'A', 'B', 'C'};

//   cout << *find(myCha.begin(), myCha.end(), 'g') << endl;      // g
//   cout << *find_if(myCha.begin(), myCha.end(), isVowel) << endl;  // a
//   cout << *find_if_not(myCha.begin(), myCha.end(), isVowel) << endl; // b

//   auto iter= find_first_of(myCha.begin(), myCha.end(), cha, cha + 3);
//   if (iter == myCha.end()) cout << "None of A, B or C." << endl; // None of A, B or C.

//   auto iter2= find_first_of(myCha.begin(), myCha.end(), cha, cha+3,
//                           [] (char a, char b){ return toupper(a) == toupper(b); });
//   if (iter2 != myCha.end()) cout << *iter2 << endl;                // a
//   auto iter3= adjacent_find(myCha.begin(), myCha.end());
//   if (iter3 == myCha.end()) cout << "No same adjacent chars." << endl;
//   // No same adjacent chars.

//   auto iter4= adjacent_find(myCha.begin(), myCha.end(),
//                           [] (char a, char b){ return isVowel(a) == isVowel(b); });
//   if (iter4 != myCha.end()) cout << *iter4;                         // b
```

```
// }
// =====
/*
Count Elements
Count algorithms assist us in counting the number of elements in a range which satisfy a certain predicate.
```

You can count elements with the STL with and without a predicate.

Returns the number of elements:

```
Num count(Inplt first, Inplt last, const T& val)
Num count(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

```
Num count_if(Inplt first, Inplt last, UnPred pre)
Num count_if(ExePol pol, FwdIt first, FwdIt last, UnPred pre)
```

Count algorithms take input iterators as arguments and return the number of elements matching val or the predicate.

```
*/
// =====
// #include <algorithm>
// #include <cctype>
// #include <iostream>
// #include <string>

// int main(){

// std::cout << std::endl;

// std::string str{"abcdabAAAaefaBqeBCQEaadsfdewAAQAAafbd"};
// std::cout << "count: " << std::count(str.begin(), str.end(), 'a') << std::endl;
// std::cout << "count_if: " << std::count_if(str.begin(), str.end(), [](char a){ return std::isupper(a);}) <<
std::endl;

// std::cout << std::endl;

// }
```

```
/*
Check Conditions on Ranges
```

C++17 contains several algorithms to check whether a value or values in a range fulfill our given condition.

Let's look at these algorithms now.

The three functions `std::all_of`, `std::any_of` and `std::none_of` answer the question, if all, at least one or no element of a range satisfies the condition. The functions need as arguments input iterators and a unary predicate and return a boolean.

Checks if all elements of the range satisfy the condition:

```
*/  
/*  
bool all_of(InPlt first, InPlt last, UnPre pre)  
bool all_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Checks if at least one element of the range satisfies the condition:

```
bool any_of(InPlt first, InPlt last, UnPre pre)  
bool any_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Checks if no element of the range satisfies the condition:

```
bool none_of(InPlt first, InPlt last, UnPre pre)  
bool none_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)  
*/  
// ======  
// #include <algorithm>  
// #include <iostream>  
// #include <vector>  
  
// int main()  
// {  
  
// std::cout << std::boolalpha << std::endl;  
  
// auto even = [](int i)  
// { return i % 2; };  
  
// std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};  
  
// std::cout << "std::any_of:\t" << std::any_of(myVec.begin(), myVec.end(), even) << std::endl;  
// std::cout << "std::all_of:\t" << std::all_of(myVec.begin(), myVec.end(), even) << std::endl;  
// std::cout << "std::none_of:\t" << std::none_of(myVec.begin(), myVec.end(), even) << std::endl;
```

```
// std::cout << std::endl;  
// }  
// ======  
/*
```

Compare Ranges

The functions describe below allow us to check the degree of equality between ranges.

With `std::equal` you can compare ranges on equality. With `std::lexicographical_compare` and `std::mismatch` you discover which range is the smaller one.

`equal`: checks if both ranges are equal

```
bool equal(InPlt first1, InPlt last1, InPlt first2)  
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2)
```

```

bool equal(Inplt first1, Inplt last1, Inplt first2, BiPre pred)
bool equal(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, BiPre pred)

bool equal(Inplt first1, Inplt last1, Inplt first2, Inplt last2)
bool equal(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, Fwdlt last2)

bool equal(Inplt first1, Inplt last1, Inplt first2, Inplt last2, BiPre pred)
bool equal(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, Fwdlt last2, BiPre pred)
*/
/*
lexicographical_compare: checks if the first range is smaller than the secondc

bool lexicographical_compare(Inplt first1, Inplt last1, Inplt first2, Inplt last2)
bool lexicographical_compare(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, Fwdlt last2)

bool lexicographical_compare(Inplt first1, Inplt last1, Inplt first2, Inplt last2, BiPre pred)
bool lexicographical_compare(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, Fwdlt last2, BiPre pred)

pair: finds the first position at which both ranges are not equal

pair<Inplt, Inplt> mismatch(Inplt first1, Inplt last1, Inplt first2)
pair<Inplt, Inplt> mismatch(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2)

pair<Inplt, Inplt> mismatch(Inplt first1, Inplt last1, Inplt first2, BiPre pred)
pair<Inplt, Inplt> mismatch(ExePol pol, Fwdlt first1, Fwdlt last2, Fwdlt first2, BiPre pred)

pair<Inplt, Inplt> mismatch(Inplt first1, Inplt last1, Inplt first2, Inplt last2)
pair<Inplt, Inplt> mismatch(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, Fwdlt last2)

pair<Inplt, Inplt> mismatch(Inplt first1, Inplt last1, Inplt first2, Inplt last2, BiPre pred)
pair<Inplt, Inplt> mismatch(ExePol pol, Fwdlt first1, Fwdlt last1, Fwdlt first2, Fwdlt last2, BiPre pred)

The algorithms take input iterators and eventually a binary predicate. std::mismatch returns as its result a pair
pa of input iterators. pa.first holds an input iterator for the first element that is not equal. pa.second holds the
corresponding input iterator for the second range. If both ranges are identical, you get two end iterators.
*/
// =====

// #include <algorithm>
// #include <cctype>
// #include <iostream>
// #include <string>

// int main(){
//   std::cout << std::boolalpha << std::endl;
// 
//   std::string str1{"Only For Testing Purpose."};
//   std::string str2{"only for testing purpose."};
// 
//   std::cout << "str1: " << str1 << std::endl;

```

```

// std::cout << "str2: " << str2 << std::endl;
// std::cout << std::endl;

// std::cout << "std::equal(str1.begin(), str1.end(), str2.begin()): " << std::equal(str1.begin(), str1.end(),
str2.begin()) << std::endl;
// std::cout << "std::equal(str1.begin(), str1.end(), str2.begin(), [](char c1, char c2){ return std::toupper(c1) ==
std::toupper(c2);}): "
//           << std::equal(str1.begin(), str1.end(), str2.begin(), [](char c1, char c2){ return std::toupper(c1) ==
std::toupper(c2);}) << std::endl;

// std::cout << std::endl;

// str1= {"Only for testing Purpose."};
// str2= {"Only for testing purpose."};

// std::cout << "str1: " << str1 << std::endl;
// std::cout << "str2: " << str2 << std::endl;

// std::cout << std::endl;

// auto pair= std::mismatch(str1.begin(), str1.end(), str2.begin());
// if ( pair.first == str1.end() ){
//   std::cout << "str1 and str2 are equal" << std::endl;
// }
// else{
//   std::cout << "str1 and str2 are different at position " << std::distance(str1.begin(), pair.first)
//   << " with (" << *pair.first << ", " << *pair.second << ")" << std::endl;
// }

// auto pair2= std::mismatch(str1.begin(), str1.end(), str2.begin(), [](char c1, char c2){ return std::toupper(c1)
// == std::toupper(c2);});
// if ( pair2.first == str1.end() ){
//   std::cout << "str1 and str2 are equal" << std::endl;
// }
// else{
//   std::cout << "str1 and str2 are different at position " << std::distance(str1.begin(), pair2.first)
//   << " with(" << *pair2.first << ", " << *pair2.second << ")" << std::endl;
// }

// std::cout << std::endl;

// */

```

Search for Ranges within Ranges

Need to acquire a sub-range from your existing range? std::search solves the problem efficiently.

std::search searches for a range in another range from the beginning, std::find_end from the end.
 std::search_n searches for n consecutive elements in the range.

All algorithms take a forward iterator, can be parametrized by a binary predicate, and return an end an iterator for the first range, if the search was unsuccessful.

Searches the second range in the first one and returns the position. Starts at the beginning:

```
FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2)  
FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2)
```

```
FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, BiPre pre)  
FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, BiPre pre)
```

```
FwdIt1 search(FwdIt1 first, FwdIt last1, Search search)
```

Searches the second range in the first one and returns the positions. Starts at the end:

```
*/  
/*  
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2 FwdIt2 last2)  
FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2 FwdIt2 last2)
```

```
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, BiPre pre)  
FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, BiPre pre)
```

Searches count consecutive values in the first range:

```
FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value)  
FwdIt search_n(ExePol pol, FwdIt first, FwdIt last, Size count, const T& value)
```

```
FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value, BiPre pre)  
FwdIt search_n(ExePol pol, FwdIt first, FwdIt last, Size count, const T& value, BiPre pre)
```

 The algorithm search_n is very special

The algorithm FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value, BiPre pre) is very special. The binary predicate BiPre uses as the first argument the values of the range and as second argument the value value.

```
*/  
// ======  
// #include <algorithm>  
// #include <array>  
// #include <cmath>  
// #include <iostream>  
  
// int main(){  
  
// std::cout << std::endl;  
  
// std::array<int, 10> arr1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
// std::array<int, 5> arr2{3, 4, -5, 6, 7};  
  
// auto fwdIt= std::search(arr1.begin(), arr1.end(), arr2.begin(), arr2.end());  
  
// if (fwdIt == arr1.end()) std::cout << "arr2 not in arr1." << std::endl;
```

```

// else{
//   std::cout << "arr2 at position " << std::distance(arr1.begin(), fwdIt) << " in arr1." << std::endl;
// }

// auto fwdIt2= std::search(arr1.begin(), arr1.end(), arr2.begin(), arr2.end(), [](int a, int b){ return std::abs(a)
// == std::abs(b); });

// if (fwdIt2 == arr1.end()) std::cout << "arr2 not in arr1." << std::endl;
// else{
//   std::cout << "arr2 at position " << std::distance(arr1.begin(), fwdIt2) << " in arr1." << std::endl;
// }

// std::cout << std::endl;

// }
// =====
/*

```

Copy Elements and Ranges

Learn how to perform various copy operations on a given range.

You can copy ranges forward with `std::copy`, backward with `std::copy_backward` and conditionally with `std::copy_if`. If you want to copy `n` elements, you can use `std::copy_n`.

`copy`: copies the range:

`OutIt copy(InIt first, InIt last, OutIt result)`
`FwdIt2 copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)`

`copy_n`: copies `n` elements:

`OutIt copy_n(InIt first, Size n, OutIt result)`
`FwdIt2 copy_n(ExePol pol, FwdIt first, Size n, FwdIt2 result)`

`copy_if`: Copies the elements dependent on the predicate `pre`.

`OutIt copy_if(InIt first, InIt last, OutIt result, UnPre pre)`
`FwdIt2 copy_if(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, UnPre pre)`

`Bilt`: Copies the range backward:

`Bilt copy_backward(Bilt first, Bilt last, Bilt result)`

The algorithms need input iterators and copy their elements to result. They return an end iterator to the destination range.

*/

```

// =====
// #include <algorithm>
// #include <iostream>
// #include <string>
// #include <vector>

// int main(){

// std::cout << std::endl;
```

```

// std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
// std::vector<int> myVec2(10);

// std::copy_if(myVec.begin(), myVec.end(), myVec2.begin() + 3, [](int a){ return a%2; });
// for ( auto v: myVec2 ) std::cout << v << " ";

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

// std::string str{"lambdastring1"};
// std::string str2{"Hellostring-----2"};

// std::cout << str2 << std::endl;
// std::copy_backward(str.begin(), str.begin() + 5, str.end());
// std::cout << str2 << std::endl;

// std::cout << std::endl;

// std::cout << str << std::endl;
// std::copy_backward(str.begin(), str.begin() + 5, str.end());
// std::cout << str << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

Replace Elements and Ranges

Let's look at the functions we can use to update and replace values in ranges.

You have with `std::replace`, `std::replace_if`, `std::replace_copy` and `std::replace_copy_if` four variations to replace elements in a range. The algorithms differ in two aspects. First, does the algorithm need a predicate? Second, does the algorithm copy the elements in the destination range?

`replace`: Replaces the old elements in the range with `newValue`, if the old element has the value `old`.

```
void replace(FwdIt first, FwdIt last, const T& old, const T& newValue)
void replace(ExePol pol, FwdIt first, FwdIt last, const T& old, const T& newValue)
```

`replace_if`: Replaces the old elements of the range with `newValue`, if the old element fulfils the predicate `pred`:

```
void replace_if(FwdIt first, FwdIt last, UnPred pred, const T& newValue)
void replace_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred, const T& newValue)
```

`replace_copy`: Replaces the old elements in the range with `newValue`, if the old element has the value `old`. Copies the result to `result`:

```
OutIt replace_copy(InIt first, InIt last, OutIt result, const T& old, const T& newValue)
FwdIt2 replace_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, const T& old, const T& newValue)
```

```

replace_copy_if: Replaces the old elements of the range with newValue, if the old element fulfills the
predicate pred. Copies the result to result:
OutIt replace_copy_if(InIt first, InIt last, OutIt result, UnPre pred, const T& newValue)
FwdIt2 replace_copy_if(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, UnPre pred, const T& newValue)
*/
// =====
// #include <algorithm>
// #include <cctype>
// #include <iostream>
// #include <string>

// int main(){

// std::cout << std::endl;

// std::string str{"Only for testing purpose."};

// std::cout << str << std::endl;

// std::replace(str.begin(), str.end(), ' ', '1');
// std::cout << str << std::endl;

// std::replace_if(str.begin(), str.end(), [](char c){ return c == '1'; }, '2');
// std::cout << str << std::endl;

// std::string str2;
// std::replace_copy(str.begin(), str.end(), std::back_inserter(str2), '2', '3');
// std::cout << str2 << std::endl;

// std::string str3;
// std::replace_copy_if(str2.begin(), str2.end(), std::back_inserter(str3), [](char c){ return c == '3'; }, '4');
// std::cout << str3 << std::endl;

// std::cout << std::endl;

// }
// =====
/*
Remove Elements and Ranges

```

Apart from insertion, copying and replacement, we can also delete elements completely.

The four variations `std::remove`, `std::remove_if`, `std::remove_copy` and `std::remove_copy_if` support two kinds of operations. On one hand, remove elements with and without a predicate from a range. On the other hand, copy the result of your modification to a new range.

`remove`: Removes the elements from the range, having the value val:

```

FwdIt remove(FwdIt first, FwdIt last, const T& val)
FwdIt remove(ExePol pol, FwdIt first, FwdIt last, const T& val)

```

`remove_if`: Removes the elements from the range, fulfilling the predicate pred:

```
FwdIt remove_if(FwdIt first, FwdIt last, UnPred pred)
FwdIt remove_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
```

remove_copy: Removes the elements from the range, having the value val. Copies the result to result:
OutIt remove_copy(InIt first, InIt last, OutIt result, const T& val)
FwdIt2 remove_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, const T& val)

remove_copy_if: Removes the elements from the range, which fulfill the predicate pred. Copies the result to result.

```
OutIt remove_copy_if(InIt first, InIt last, OutIt result, UnPre pred)
FwdIt2 remove_copy_if(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, UnPre pred)
```

The algorithms need input iterators for the source range and an output iterator for the destination range. They return as a result an end iterator for the destination range.

⚠ Apply the erase-remove idiom

The remove variations don't remove an element from the range. They return the new logical end of the range. You have to adjust the size of the container with the erase-remove idiom.

```
*/
```

```
// =====

// #include <algorithm>
// #include <cctype>
// #include <iostream>
// #include <string>
// #include <vector>

// int main(){

// std::cout << std::endl;

// std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// for (auto v: myVec) std::cout << v << " ";
// std::cout << std::endl;
// auto newIt= std::remove_if(myVec.begin(), myVec.end(), [](int a){ return a % 2; } );
// for (auto v: myVec) std::cout << v << " ";
// std::cout << std::endl;

// myVec.erase(newIt, myVec.end());
// for (auto v: myVec) std::cout << v << " ";

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

// std::string str{"Only for Testing Purpose."};
// std::cout << str << std::endl;
// str.erase(std::remove_if(str.begin(), str.end(), [](char c){ return std::isupper(c); } ), str.end());
// std::cout << str << std::endl;
```

```
// std::cout << std::endl;  
// }  
// ======  
/*
```

Fill and Create Ranges

Next in the line of modifying algorithms, we have the 'fill' and 'generate' functions.

You can fill a range with `std::fill` and `std::fill_n`; you can generate new elements with `std::generate` and `std::generate_n`.

`fill`: Fills a range with elements:

```
void fill(FwdIt first, FwdIt last, const T& val)  
void fill(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

`fill_n`: Fills a range with n new elements:

```
OutIt fill_n(OutIt first, Size n, const T& val)  
FwdIt fill_n(ExePol pol, FwdIt first, Size n, const T& val)
```

`generate`: Generates a range with a generator gen:

```
void generate(FwdIt first, FwdIt last, Generator gen)  
void generate(ExePol pol, FwdIt first, FwdIt last, Generator gen)
```

`generate_n`: Generates n elements of a range with the generator gen:

```
OutIt generate_n(OutIt first, Size n, Generator gen)  
FwdIt generate_n(ExePol pol, FwdIt first, Size n, Generator gen)
```

The algorithms expect the value `val` or a generator `gen` as an argument. `gen` has to be a function taking no argument and returning the new value. The return value of the algorithms `std::fill_n` and `std::generate_n` is an output iterator, pointing to the last created element.

```
*/  
// ======  
// #include <algorithm>  
// #include <iostream>  
// #include <list>  
// #include <vector>  
  
// int getNext()  
// {  
//     static int next{0};  
//     return ++next;  
// }  
  
// int main()  
// {  
  
//     std::cout << std::endl;  
  
//     std::vector<int> vec(20);  
//     std::fill(vec.begin(), vec.end(), 2011);
```

```

// for (auto v : vec)
// std::cout << v << " ";
// std::cout << std::endl;

// std::generate_n(vec.begin(), 15, getNext);
// for (auto v : vec)
// std::cout << v << " ";

// std::cout << "\n\n";
//}
//=====
/*

```

Move Ranges

In C++, we can move data from one range to another. Read the lesson for more details.

`std::move` moves the ranges forward; `std::move_backward` moves the ranges backwards.

`move`: moves the range forward:

`InIt move(InIt first, InIt last, OutIt result)`

`FwdIt2 move(ExePol pol, FwdIt first, FwdIt last, Fwd2It result)`

`move_backward`: Moves the range backward:

`Bilt move_backward(Bilt first, Bilt last, Bilt result)`

Both algorithms need a destination iterator `result`, to which the range is moved. In the case of the `std::move` algorithm, this is an output iterator, and in the case of the `std::move_backward` algorithm, this is a bidirectional iterator. The algorithms return output or a bidirectional iterator, pointing to the initial position in the destination range.

 The source range may be changed

`std::move` and `std::move_backward` apply move semantics. Therefore the source range is valid but does have not necessarily the same elements afterward.

`*/`

```

// =====
// #include <algorithm>
// #include <iostream>
// #include <string>
// #include <vector>

// int main()
// {
// std::cout << std::endl;

// std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
// std::vector<int> myVec2(10);

// std::move(myVec.begin(), myVec.end(), myVec2.begin());
// for (auto v : myVec2)

```

```

// std::cout << v << " ";
// std::cout << "\n\n";
// std::string str{"abcdefghijklmnp"};
// std::string str2{"-----"};
// std::cout << str2 << std::endl;
// std::move_backward(str.begin(), str.end(), str2.end());
// std::cout << str2 << std::endl;
// std::cout << std::endl;
//}
// =====
/*
Swap Ranges
Along with moving data between ranges, we can also swap their values with one another.

```

`std::swap` and `std::swap_ranges` can swap objects and ranges.

`swap`: swaps objects:

`void swap(T& a, T& b)`

`swap_ranges`: swaps ranges:

`FwdIt swap_ranges(FwdIt1 first1, FwdIt1 last1, FwdIt first2)`
`FwdIt swap_ranges(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt first2)`

The returned iterator points to the last swapped element in the destination range.

 The ranges must not overlap.

```

#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

int main(){

    std::cout << std::endl;

    std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
    std::vector<int> myVec2(10);

    for (auto v: myVec) std::cout << v << " ";
    for (auto v: myVec2) std::cout << v << " ";

    std::cout << std::endl;
    std::swap(myVec, myVec2);

    for (auto v: myVec) std::cout << v << " ";

```

```

for (auto v: myVec2) std::cout << v << " ";
std::cout << "\n\n";
std::string str{"abcdefghijklmnp"};
std::string str2{"-----"};
std::cout << str << std::endl;
std::cout << str2 << std::endl;

std::swap_ranges(str.begin(), str.begin() + 5, str2.begin() + 5);

std::cout << str << std::endl;
std::cout << str2 << std::endl;

std::cout << std::endl;
}

*/
/*
Transform Ranges

```

Now we will study `std::transform` which is used to perform transformations on a range.

The `std::transform` algorithm applies a unary or binary callable to a range and copies the modified elements to the destination range.

Applies the unary callable fun to the elements of the input range and copies the result to result:

```

OutIt transform(InIt first1, InIt last1, OutIt result, UnFun fun)
FwdIt2 transform(ExePol pol, FwdIt first1, FwdIt last1, FwdIt2 result, UnFun fun)

```

Applies the binary callable fun to both input ranges and copies the result to result:

```

OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2, OutIt result, BiFun fun)
FwdIt3 transform(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt3 result, BiFun fun)

```

The difference between the two versions is that the first version applies the callable to each element of the range; the second version applies the callable to pairs of both ranges in parallel. The returned iterator points to one position after the last transformed element.

```

*/
// =====
// #include <algorithm>
// #include <cctype>
// #include <iostream>
// #include <string>
// #include <vector>

// int main()
//{
// std::cout << std::endl;

```

```

// std::string str{"abcdefghijklmnoprstuvwxyz"};
// std::cout << str << std::endl;

// std::transform(str.begin(), str.end(), str.begin(), [](char c)
// { return std::toupper(c); });

// std::cout << str << std::endl;

// std::cout << std::endl;

// std::vector<std::string> vecStr{"Only", "for", "testing", "purpose", ".};
// std::vector<std::string> vecStr2(5, "-");

// std::vector<std::string> vecRes;

// std::transform(vecStr.begin(), vecStr.end(),
// vecStr2.begin(),
// std::back_inserter(vecRes),
// [](std::string a, std::string b)
// { return std::string(b) + a + b; });

// for (auto str : vecRes)
// std::cout << str << std::endl;

// std::cout << std::endl;
// */
// =====
/*
```

Reverse Ranges

There were always roundabout ways of reversing a range. Now we have a predefined function to do that.

`std::reverse` and `std::reverse_copy` invert the order of the elements in their range.

Reverses the order of the elements in the range:

```
void reverse(Bil first, Bil last)
void reverse(ExePol pol, Bil first, Bil last)
```

Reverses the order of the elements in the range and copies the result to result:

```
OutIt reverse_copy(Bil first, Bil last, OutIt result)
FwdIt reverse_copy(ExePol pol, Bil first, Bil last, FwdIt result)
```

Both algorithms require bidirectional iterators. The returned iterator points to the position of the output range result before the elements were copied.

```
/*
// =====
// #include <algorithm>
```

```

// #include <deque>
// #include <iostream>
// #include <list>
// #include <string>
// #include <vector>

// template <typename Cont, typename T>
// void doTheSame(Cont cont, T t)
// {

//   for (auto c : cont)
//     std::cout << c << " ";
//   std::cout << std::endl;
//   std::cout << "cont.size(): " << cont.size() << std::endl;
//   std::reverse(cont.begin(), cont.end());
//   for (auto c : cont)
//     std::cout << c << " ";
//   std::cout << std::endl;
//   std::reverse(cont.begin(), cont.end());
//   for (auto c : cont)
//     std::cout << c << " ";
//   std::cout << std::endl;
//   auto lt = std::find(cont.begin(), cont.end(), t);
//   std::reverse(lt, cont.end());
//   for (auto c : cont)
//     std::cout << c << " ";
// }

// int main()
// {

//   std::cout << std::endl;

//   std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
//   std::deque<std::string> myDeque({"A", "B", "C", "D", "E", "F", "G", "H", "I"});
//   std::list<char> myList({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'});

//   doTheSame(myVec, 5);
//   std::cout << "\n\n";
//   doTheSame(myDeque, "D");
//   std::cout << "\n\n";
//   doTheSame(myList, 'd');

//   std::cout << "\n\n";
// }

// =====
/*
Rotate Ranges
We can rotate our data such that every element now lies at a different index, which is decided by the rotation offset.

```

`std::rotate` and `std::rotate_copy` rotate their elements.

Rotates the elements in such a way that middle becomes the new first element:

```
FwdIt rotate(FwdIt first, FwdIt middle, FwdIt last)  
FwdIt rotate(ExePol pol, FwdIt first, FwdIt middle, FwdIt last)
```

Rotates the elements in such a way that middle becomes the new first element. Copies the result to result:

```
OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt result)  
FwdIt2 rotate_copy(ExePol pol, FwdIt first, FwdIt middle, FwdIt last, FwdIt2 result)
```

Both algorithms need forward iterators. The returned iterator is an end iterator for the copied range.
*/

```
// =====  
// #include <algorithm>  
// #include <iostream>  
// #include <string>  
  
// int main()  
{  
  
// std::string str{"123456789"};  
  
// auto endIt = str.end();  
// for (auto middleIt = str.begin(); middleIt != endIt; ++middleIt)  
// {  
//   std::rotate(str.begin(), middleIt, str.end());  
//   std::cout << str << std::endl;  
// }  
// }  
// =====  
/*
```

Randomly Shuffle Ranges

Rearrange the values in your range randomly, using `std::random_shuffle` and `std::shuffle`.

You can randomly shuffle ranges with `std::random_shuffle` and `std::shuffle`.

Randomly shuffles the elements in a range:

```
void random_shuffle(RanIt first, RanIt last)
```

Randomly shuffles the elements in the range, by using the random number generator gen:

```
void random_shuffle(RanIt first, RanIt last, RanNumGen&& gen)
```

```
void random_shuffle(RanIt first, RanIt last, RanNumGen&& gen)  
void shuffle(RanIt first, RanIt last, URNG&& gen)
```

The algorithms need random access iterators. `RanNumGen&& gen` has to be callable, taking an argument and returning a value within its arguments. `URNG&& gen` has to be a uniform random number generator.

Prefer std::shuffle

Use `std::shuffle` instead of `std::random_shuffle`. `std::random_shuffle` has been deprecated since C++14 and removed in C++17, because it uses the C function `rand` internally.

*/

```
// =====
```

```
// #include <algorithm>
// #include <chrono>
// #include <iostream>
// #include <random>
// #include <vector>

// int main()
//{
//    std::cout << std::endl;

//    std::vector<int> vec1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
//    std::vector<int> vec2(vec1);

//    for (auto v : vec1)
//        std::cout << v << " ";

//    std::cout << std::endl;

//    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();

//    std::cout << std::endl;

//    std::random_shuffle(vec1.begin(), vec1.end());
//    for (auto v : vec1)
//        std::cout << v << " ";

//    std::cout << std::endl;

//    std::shuffle(vec2.begin(), vec2.end(), std::default_random_engine(seed));
//    for (auto v : vec2)
//        std::cout << v << " ";

//    std::cout << "\n\n";
//}
// =====
```

/*

Remove Duplicates

Next, we'll study ways to make sure each element in our range is unique.

With the algorithms `std::unique` and `std::unique_copy` you have more opportunities to remove adjacent duplicates. This can be done with and without a binary predicate.

Removes adjacent duplicates:

```
FwdIt unique(FwdIt first, FwdIt last)
FwdIt unique(ExePol pol, FwdIt first, FwdIt last)
```

Removes adjacent duplicates, satisfying the binary predicate:

```
FwdIt unique(FwdIt first, FwdIt last, BiPred pre)
FwdIt unique(ExePol pol, FwdIt first, FwdIt last, BiPred pre)
```

Removes adjacent duplicates and copies the result to result:

```
OutIt unique_copy(InIt first, InIt last, OutIt result)
FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

Removes adjacent duplicates, satisfying the binary predicate and copies the result to result:

```
OutIt unique_copy(InIt first, InIt last, OutIt result, BiPred pre)
FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, BiPred pre)
```

 The unique algorithms return the new logical end iterator

The unique algorithms return the logical end iterator of the range. The elements have to be removed with the erase-remove idiom.

```
*/
```

```
// =====
// #include <algorithm>
// #include <iostream>
// #include <vector>

// int main()
//{
//    std::cout << std::endl;

//    std::vector<int> myVec{0, 0, 1, 1, 2, 2, 3, 4, 4, 5, 3, 6, 7, 8, 1, 3, 3, 8, 8, 9};

//    for (auto v : myVec)
//        std::cout << v << " ";
//    std::cout << std::endl;
//    // auto newIt= std::unique(myVec.begin(), myVec.end(), [](int a){ return a%2; });
//    myVec.erase(std::unique(myVec.begin(), myVec.end()), myVec.end());
//    for (auto v : myVec)
//        std::cout << v << " ";

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

//    std::vector<int> myVec2{1, 4, 3, 3, 3, 5, 7, 9, 2, 4, 1, 6, 8, 0, 3, 5, 7, 8, 7, 3, 9, 2, 4, 2, 5, 7, 3};
//    std::vector<int> resVec;
//    resVec.reserve(myVec2.size());
//    std::unique_copy(myVec2.begin(), myVec2.end(), std::back_inserter(resVec),
//                    [] (int a, int b)
//                    { return (a % 2) == (b % 2); });

//    for (auto v : myVec2)
```

```

// std::cout << v << " ";
// std::cout << std::endl;
// for (auto v : resVec)
// std::cout << v << " ";

// std::cout << "\n\n";
//}
// =====
/*

```

Partition

This algorithm allows us to divide or split ranges into separate sets.

i What is a partition?

A partition of a set is a decomposition of a set in subsets so that each element of the set is precisely in one subset. The subsets are defined in C++ by a unary predicate so that the members of the first subset fulfill the predicate. The remaining elements are in the second subset.

C++ offers a few functions for dealing with partitions. All of them need a unary predicate `pre`. `std::partition` and `std::stable_partition` partition a range and returns the partition point. With `std::partition_point` you can get the partition point of a partition. Afterwards you can check the partition with `std::is_partitioned` or copy it with `std::partition_copy`.

Checks if the range is partitioned:

```

bool is_partitioned(InIt first, InIt last, UnPre pre)
bool is_partitioned(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

Partitions the range:

```

FwdIt partition(FwdIt first, FwdIt last, UnPre pre)
FwdIt partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

Partitions the range stable:

```

Bilt stable_partition(FwdIt first, FwdIt last, UnPre pre)
Bilt stable_partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

Copies a partition in two ranges:

```

pair<OutIt1, OutIt2> partition_copy(InIt first, InIt last, OutIt1 result_true, OutIt2 result_false, UnPre pre)
pair<FwdIt1, FwdIt2> partition_copy(ExePol pol, FwdIt1 first, FwdIt1 last, FwdIt2 result_true, FwdIt3
result_false, UnPre pre)

```

Returns the partition point:

```

FwdIt partition_point(FwdIt first, FwdIt last, UnPre pre)

```

A `std::stable_partition` guarantees, in contrary to a `std::partition`, that the elements preserve their relative order. The returned iterator `FwdIt` and `Bilt` points to the initial position in the second subset of the partition. The pair `std::pair<OutIt, OutIt>` of the algorithm `std::partition_copy` contains the end iterator of the subsets `result_true` and `result_false`. The behavior of `std::partition_point` is undefined if the range is not partitioned.

```

*/
// =====
// #include <algorithm>
// #include <cctype>

```

```

// #include <deque>
// #include <iostream>
// #include <list>
// #include <string>
// #include <vector>

// bool isOdd(int i){ return (i%2); }

// int main(){

// std::cout << std::boolalpha << std::endl;

// std::vector<int> vec{1, 4, 3, 4, 5, 6, 7, 3, 4, 5, 6, 0, 4, 8, 4, 6, 6, 5, 8, 8, 3, 9, 3, 7, 6, 4, 8};

// for ( auto v: vec ) std::cout << v << " ";

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

// auto parPoint= std::partition(vec.begin(), vec.end(), isOdd);

// for (auto v: vec) std::cout << v << " ";
// std::cout << std::endl;
// for (auto v= vec.begin(); v != parPoint; ++v) std::cout << *v << " ";
// std::cout << std::endl;
// for (auto v= parPoint; v != vec.end(); ++v) std::cout << *v << " ";
// std::cout << std::endl;

// std::cout << std::endl;

// std::cout << "std::is_partitioned: " << std::is_partitioned(vec.begin(), vec.end(), isOdd) << std::endl;

// std::cout << "std::partition_point: " << (std::partition_point(vec.begin(), vec.end(), isOdd) == parPoint) << std::endl;

// std::cout << std::endl;

// std::list<int> li;
// std::list<int> de;
// std::partition_copy(vec.begin(), vec.end(), std::back_inserter(li), std::back_inserter(de), [](int i) { return i < 5; });

// for (auto v: li) std::cout << v << " ";
// std::cout << std::endl;
// for (auto v: de) std::cout << v << " ";

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

// }

// =====
/*
Sort

```

Sorting and verifying the sortedness of your data has been made very easy in C++. Let's find out how.

You can sort a range with `std::sort` or `std::stable_sort` or `sort until a position with std::partial_sort`. In addition `std::partial_sort_copy` copies the partially sorted range. With `std::nth_element` you can assign an element the sorted position in the range. You can check with `std::is_sorted` if a range is sorted. If you want to know until which position a range is sorted, use `std::is_sorted_until`.

Per default the predefined function object `std::less` is used as sorting criterion. However, you can use your sorting criterion. This has to obey the strict weak ordering.

Sorts the elements in the range:

```
*/  
/*  
void sort(Ralt first, Ralt last)  
void sort(ExePol pol, Ralt first, Ralt last)  
  
void sort(Ralt first, Ralt last, BiPre pre)  
void sort(ExePol pol, Ralt first, Ralt last, BiPre pre)
```

Sorts the elements in the range stable:

```
void stable_sort(Ralt first, Ralt last)  
void stable_sort(ExePol pol, Ralt first, Ralt last)  
  
void stable_sort(Ralt first, Ralt last, BiPre pre)  
void stable_sort(ExePol pol, Ralt first, Ralt last, BiPre pre)
```

Sorts partially the elements in the range until middle:

```
void partial_sort(Ralt first, Ralt middle, Ralt last)  
void partial_sort(ExePol pol, Ralt first, Ralt middle, Ralt last)  
  
void partial_sort(Ralt first, Ralt middle, Ralt last, BiPre pre)  
void partial_sort(ExePol pol, Ralt first, Ralt middle, Ralt last, BiPre pre)
```

Sorts partially the elements in the range and copies them in the destination ranges `result_first` and `result_last`:

```
Ralt partial_sort_copy(Init first, Init last, Ralt result_first, Ralt result_last)  
Ralt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last, Ralt result_first, Ralt result_last)  
  
Ralt partial_sort_copy(Init first, Init last, Ralt result_first, Ralt result_last, BiPre pre)  
Ralt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last, Ralt result_first, Ralt result_last, BiPre pre)
```

Checks if a range is sorted:

```
bool is_sorted(FwdIt first, FwdIt last)  
bool is_sorted(ExePol pol, FwdIt first, FwdIt last)  
  
bool is_sorted(FwdIt first, FwdIt last, BiPre pre)  
bool is_sorted(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

Returns the position to the first element that doesn't satisfy the sorting criterion:

```
FwdIt is_sorted_until(FwdIt first, FwdIt last)  
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last)
```

```
FwdIt is_sorted_until(FwdIt first, FwdIt last, BiPre pre)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

Reorders the range, so that the n-th element has the right (sorted) position:

```
void nth_element(Ralt first, Ralt nth, Ralt last)
void nth_element(ExePol pol, Ralt first, Ralt nth, Ralt last)
```

```
void nth_element(Ralt first, Ralt nth, Ralt last, BiPre pre)
void nth_element(ExePol pol, Ralt first, Ralt nth, Ralt last, BiPre pre)
*/
// =====
// #include <algorithm>
// #include <iostream>
// #include <string>
// #include <vector>

// int main(){

// std::cout << std::boolalpha << std::endl;

// std::string str{"RUdAjdDkaACsdfjwIdXmnEiVSEZTiepfgOlkue"};

// std::cout << str << std::endl;

// std::cout << "std::is_sorted(str.begin(), str.end()): " << std::is_sorted(str.begin(), str.end()) << std::endl;

// std::cout << std::endl;

// std::partial_sort(str.begin(), str.begin() + 30, str.end());
// std::cout << str << std::endl;
// auto sortUntil= std::is_sorted_until(str.begin(), str.end());
// std::cout << "Sorted until: " << *sortUntil << std::endl;
// for (auto charIt= str.begin(); charIt != sortUntil; ++charIt) std::cout << *charIt;

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

// std::vector<int> vec{1, 0, 4, 3, 5};

// auto vecIt= vec.begin();
// while( vecIt != vec.end() ){
//   std::nth_element(vec.begin(), vecIt++, vec.end());
//   std::cout << std::distance(vec.begin(), vecIt) << "-th ";
//   for (auto v: vec) std::cout << v;
//   std::cout << std::endl;
// }

// std::cout << std::endl;

// }
// =====
```

```
/*
Binary Search
```

The fast search which has a search time of $O(\log n)$ has been predefined in C++.

The binary search algorithms use the fact that the ranges are already sorted. To search for an element, use `std::binary_search`. With `std::lower_bound` you get an iterator for the first element, being no smaller than the given value. With `std::upper_bound` you get an iterator back for the first element, which is bigger than the given value. `std::equal_range` combines both algorithms.

If the container has n elements, you need on average $\log_2(n)$ comparisons for the search. The binary search requires that you use the same comparison criterion that you used for sorting the container. Per default the comparison criterion is `std::less`, but you can adjust it. Your sorting criterion has to obey the strict weak ordering. If not, the program is undefined.

If you have an unordered associative container, the methods of the unordered associative container are in general faster.

Searches the element `val` in the range:

```
/*
bool binary_search(FwdIt first, FwdIt last, const T& val)
bool binary_search(FwdIt first, FwdIt last, const T& val, BiPre pre)
```

Returns the position of the first element of the range, being not smaller than `val`:

```
FwdIt lower_bound(FwdIt first, FwdIt last, const T& val)
FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)
```

Returns the position of the first element of the range, being bigger than `val`:

```
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val)
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)
```

Returns the pair `std::lower_bound` and `std::upper_bound` for the element `val`:

```
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val)
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val, BiPre pre)
```

Finally, the code snippet.

```
/*
// =====
// #include <algorithm>
// #include <cmath>
// #include <iostream>
// #include <vector>

// bool isLessAbs(int a, int b){
//   return std::abs(a) < std::abs(b);
// }

// int main(){
//   std::cout << std::boolalpha << std::endl;
```

```

// std::vector<int> vec{-3, 0, -3, 2, -3, 5, -3, 7, -0, 6, -3, 5, -6, 8, 9, 0, 8, 7, -7, 8, 9, -6, 3, -3, 2};

// for ( auto v: vec ) std::cout << v << " ";

// std::sort(vec.begin(), vec.end(), isLessAbs);
// std::cout << std::endl;
// for ( auto v: vec ) std::cout << v << " ";

// std::cout << std::endl;

// std::cout << std::endl;
// std::cout << "std::binary_search(vec.begin(), vec.end(), -5, isLessAbs): " << std::binary_search(vec.begin(), vec.end(), -5, isLessAbs) << std::endl;
// std::cout << "std::binary_search(vec.begin(), vec.end(), 5, isLessAbs): " << std::binary_search(vec.begin(), vec.end(), 5, isLessAbs) << std::endl;

// auto pair= std::equal_range(vec.begin(), vec.end(), 3, isLessAbs);

// std::cout << std::endl;

// std::cout << "Position of first 3: " << std::distance(vec.begin(), pair.first) << std::endl;
// std::cout << "Position of last 3: " << std::distance(vec.begin(), pair.second)-1 << std::endl;
// for ( auto threelt= pair.first; threelt != pair.second ; ++threelt ) std::cout << *threelt << " ";

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

// }
// =====
/*

```

Merge Operations

In this lesson, we'll learn different ways of combining ranges.

Merge operations empower you to merge sorted ranges in a new sorted range. The algorithm requires that the ranges and the algorithm use the same sorting criterion. If not, the program is undefined. Per default the predefined sorting criterion `std::less` is used. If you use your sorting criterion, it has to obey the strict weak ordering. If not, the program is undefined.

You can merge two sorted ranges with `std::inplace_merge` and `std::merge`. You can check with `std::includes` if one sorted range is in another sorted range. You can merge with `std::set_difference`, `std::set_intersection`, `std::set_symmetric_difference` and `std::set_union` two sorted ranges in a new sorted range.

Merges in place two sorted sub ranges [first, mid) and [mid, last):

```

*/
/*
void inplace_merge(Bilt first, Bilt mid, Bilt last)
void inplace_merge(ExePol pol, Bilt first, Bilt mid, Bilt last)

void inplace_merge(Bilt first, Bilt mid, Bilt last, BiPre pre)
void inplace_merge(ExePol pol, Bilt first, Bilt mid, Bilt last, BiPre pre)

```

Merges two sorted ranges and copies the result to result:

```
OutIt merge(InIt first1, InIt last1, InIt first2, InIt last2, OutIt result)
FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, FwdIt3 result)
```

```
OutIt merge(InIt first1, InIt last1, InIt first2, InIt last2, OutIt result, BiPre pre)
FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, FwdIt3 result, BiPre pre)
```

Checks if all elements of the second range are in the first range:

```
bool includes(InIt first1, InIt last1, InIt1 first2, InIt1 last2)
bool includes(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2)
```

```
bool includes(InIt first1, InIt last1, InIt first2, InIt last2, BinPre pre)
bool includes(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2, BinPre pre)
```

Copies these elements of the first range to result, being not in the second range:

```
OutIt set_difference(InIt first1, InIt last1, InIt1 first2, InIt2 last2, OutIt result)
FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1, FWdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_difference(InIt first1, InIt last1, InIt1 first2, InIt2 last2, OutIt result, BiPre pre)
FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

Determines the intersection of the first with the second range and copies the result to result:

```
OutIt set_intersection(InIt first1, InIt last1, InIt1 first2, InIt2 last2, OutIt result)
FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1, FWdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_intersection(InIt first1, InIt last1, InIt1 first2, InIt2 last2, OutIt result, BiPre pre)
FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

Determines the union of the first with the second range and copies the result to result:

```
OutIt set_union(InIt first1, InIt last1, InIt1 first2, InIt2 last2, OutIt result)
FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1, FWdIt1 first2, FwdIt1 last2, FwdIt2 result)
```

```
OutIt set_union(InIt first1, InIt last1, InIt1 first2, InIt2 last2, OutIt result, BiPre pre)
FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

The returned iterator is an end iterator for the destination range. The destination range of std::set_difference has all the elements in the first, but not the second range. On the contrary, the destination range of std::symmetric_difference has only the elements that are elements of one range, but not both. std::union determines the union of both sorted ranges.

```
/*
// =====
// #include <algorithm>
// #include <deque>
// #include <iostream>
// #include <iterator>
// #include <vector>

// int main(){
//   std::cout << std::boolalpha;
```

```

// std::vector<int> vec1{1, 1, 4, 3, 5, 8, 6, 7, 9, 2};
// std::vector<int> vec2{1, 2, 3};

// std::cout << "vec1:\t\t\t\t";
// for (auto v: vec1) std::cout << v << " ";
// std::cout << std::endl;
// //vec1:    1 1 4 3 5 8 6 7 9 2
// std::cout << "vec2:\t\t\t\t";
// for (auto v: vec2) std::cout << v << " ";
// std::cout << std::endl;
// //vec2:    1 2 3

// std::sort(vec1.begin(), vec1.end());
// std::vector<int> vec(vec1);

// std::cout << std::endl;
// std::cout << "vec1 includes vec2: " << std::includes(vec1.begin(), vec1.end(), vec2.begin(), vec2.end()) << std::endl;
// //vec1 includes vec2: true
// std::cout << std::endl;

// vec1.reserve(vec1.size() + vec2.size());
// vec1.insert(vec1.end(), vec2.begin(), vec2.end());

// std::cout << "vec1:\t\t\t\t";
// for (auto v: vec1) std::cout << v << " ";
// std::cout << std::endl;

// std::inplace_merge(vec1.begin(), vec1.end() - vec2.size(), vec1.end());
// std::cout << "vec1:\t\t\t\t";
// for ( auto v: vec1 ) std::cout << v << " ";

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

// vec2.push_back(10);

// std::cout << "vec:\t\t\t\t";
// for (auto v: vec) std::cout << v << " ";
// std::cout << std::endl;
// std::cout << "vec2:\t\t\t\t";
// for (auto v: vec2) std::cout << v << " ";

// std::vector<int> res;
// std::set_union(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
//                std::back_inserter(res));
// std::cout << "\n" << "set_union:\t\t\t\t";
// for (auto v : res) std::cout << v << " ";

// res={};
// std::set_intersection(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
//                      std::back_inserter(res));

```

```

// std::cout << "\n" << "set_intersection:\t\t\t";
// for (auto v : res) std::cout << v << " ";

// res={};
// std::set_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
//         std::back_inserter(res));
// std::cout << "\n" << "set_difference:\t\t\t";
// for (auto v : res) std::cout << v << " ";

// res={};
// std::set_symmetric_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
//         std::back_inserter(res));
// std::cout << "\n" << "set_symmetric_difference:\t\t";
// for (auto v : res) std::cout << v << " ";

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

```

```

//}
//=====
/*

```

Heaps

This is another popular data structure implemented in C++ using a range.

```

*/
/*

```

i What is a heap?

A heap is a binary search tree in which parent elements are always bigger than its child elements. Heap trees are optimized for the efficient sorting of elements.

You can create with `std::make_heap` a heap. You can push with `std::push_heap` new elements on the heap. On the contrary, you can pop the largest element with `std::pop_heap` from the heap. Both operations respect the heap characteristics. `std::push_heap` moves the last element of the range on the heap; `std::pop_heap` moves the biggest element of the heap to the last position in the range. You can check with `std::is_heap` if a range is a heap. You can determine with `std::is_heap_until` until which position the range is a heap. `std::sort_heap` sorts the heap.

The heap algorithms require that the ranges and the algorithm use the same sorting criterion. If not, the program is undefined. Per default, the predefined sorting criterion `std::less` is used. If you use your sorting criterion, it has to obey the strict weak ordering. If not, the program is undefined.

Creates a heap from the range:

```

void make_heap(Ralt first, Ralt last)
void make_heap(Ralt first, Ralt last, BiPre pre)

```

Checks if the range is a heap:

```

bool is_heap(Ralt first, Ralt last)
bool is_heap(ExePol pol, Ralt first, Ralt last)

```

```

bool is_heap(Ralt first, Ralt last, BiPre pre)
bool is_heap(ExePol pol, Ralt first, Ralt last, BiPre pre)

```

Determines until which position the range is a heap:

```
bool is_heap_until(Ralt first, Ralt last)
```

```
bool is_heap_until(ExePol pol, Ralt first, Ralt last)
```

```
bool is_heap_until(Ralt first, Ralt last, BiPre pre)
```

```
bool is_heap_until(ExePol pol, Ralt first, Ralt last, BiPre pre)
```

Sorts the heap:

```
void sort_heap(Ralt first, Ralt last)
```

```
void sort_heap(Ralt first, Ralt last, BiPre pre)
```

Pushes the last element of the range onto the heap. [first, last-1) has to be a heap.

```
void push_heap(Ralt first, Ralt last)
```

```
void push_heap(Ralt first, Ralt last, BiPre pre)
```

Removes the biggest element from the heap and puts it to the end of the range:

```
void pop_heap(Ralt first, Ralt last)
```

```
void pop_heap(Ralt first, Ralt last, BiPre pre)
```

With std::pop_heap you can remove the biggest element from the heap. Afterwards, the biggest element is the last element of the range. To remove it from the heap h, use h.pop_back.

```
/*
// =====
// #include <algorithm>
// #include <iostream>
// #include <vector>

// int main(){
    std::cout << std::boolalpha << std::endl;

    std::vector<int> vec{4, 3, 2, 1, 5, 6, 7, 9, 10};
    for (auto v: vec) std::cout << v << " ";
    std::cout << std::endl;

    std::make_heap(vec.begin(), vec.end());
    for (auto v: vec) std::cout << v << " ";
    std::cout << std::endl;

    std::cout << "std::is_heap(vec.begin(), vec.end()): " << std::is_heap(vec.begin(), vec.end()) << std::endl;

    std::cout << std::endl;

    vec.push_back(100);
    std::cout << "std::is_heap(vec.begin(), vec.end()): " << std::is_heap(vec.begin(), vec.end()) << std::endl;
    std::cout << "*std::is_heap_until(vec.begin(), vec.end()): " << *std::is_heap_until(vec.begin(), vec.end()) << std::endl;
    for (auto v: vec) std::cout << v << " ";
    std::push_heap(vec.begin(), vec.end());
    std::cout << "std::is_heap(vec.begin(), vec.end()): " << std::is_heap(vec.begin(), vec.end()) << std::endl;
    std::cout << std::endl;
```

```

// for (auto v: vec) std::cout << v << " ";
// std::cout << "\n\n";
// std::pop_heap(vec.begin(), vec.end());
// for (auto v: vec) std::cout << v << " ";
// std::cout << std::endl;
// std::cout << "*std::is_heap_until(vec.begin(), vec.end()): " << *std::is_heap_until(vec.begin(), vec.end()) << std::endl;
// vec.resize(vec.size() - 1);
// std::cout << "std::is_heap(vec.begin(), vec.end()): " << std::is_heap(vec.begin(), vec.end()) << std::endl;
// std::cout << std::endl;
// std::cout << "vec.front(): " << vec.front() << std::endl;
// std::cout << std::endl;
// }
// =====
/*

```

Min and Max

Let's take a look at the functions C++ provides to check the minimum and maximum in a range.

You can determine the minimum, the maximum and the minimum and maximum pair of a range with the algorithms `std::min_element`, `std::max_element` and `std::minmax_element`. Each algorithm can be configured with a binary predicate.

Returns the minimum element of the range:

*/

/*

```
constexpr FwdIt min_element(FwdIt first, FwdIt last)
FwdIt min_element(ExePol pol, FwdIt first, FwdIt last)
```

```
constexpr FwdIt min_element(FwdIt first, FwdIt last, BinPre pre)
FwdIt min_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

Returns the maximum element of the range:

```
constexpr FwdIt max_element(FwdIt first, FwdIt last)
FwdIt max_element(ExePol pol, FwdIt first, FwdIt last)
```

```
constexpr FwdIt max_element(FwdIt first, FwdIt last, BinPre pre)
FwdIt max_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

Returns the pair `std::min_element` and `std::max_element` of the range:

```
constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last)
pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last)
```

```
constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last, BinPre pre)
```

```
pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

If the range has more than one minimum or maximum element, the first one is returned.

```
*/
```

```
// =====
```

```
// #include <algorithm>
```

```
// #include <cstdlib>
```

```
// #include <iostream>
```

```
// #include <string>
```

```
// #include <vector>
```

```
// #include <sstream>
```

```
// std::string toString(int i)
```

```
// {
```

```
// std::stringstream buff;
```

```
// buff.str("");
```

```
// buff << i;
```

```
// std::string val = buff.str();
```

```
// return val;
```

```
// }
```

```
// int tolnt(const std::string &s)
```

```
// {
```

```
// std::stringstream buff;
```

```
// buff.str("");
```

```
// buff << s;
```

```
// int value;
```

```
// buff >> value;
```

```
// return value;
```

```
// }
```

```
// int main()
```

```
// {
```

```
// std::cout << std::endl;
```

```
// std::vector<int> myInts;
```

```
// std::vector<std::string> myStrings{"94", "5", "39", "-4", "-49", "1001", "-77", "23", "0", "84", "59", "96", "6", "-94", "87"};
```

```
// std::transform(myStrings.begin(), myStrings.end(), std::back_inserter(myInts), tolnt);
```

```
// for (auto i : myInts)
```

```
// std::cout << i << " ";
```

```
// std::cout << "\n\n";
```

```
// auto paInt = std::minmax_element(myInts.begin(), myInts.end());
```

```
// std::cout << "std::minmax_element(myInts.begin(), myInts.end()): "
```

```
// << "(" << *paInt.first << ", " << *paInt.second << ")" << std::endl;
```

```
// auto paStr = std::minmax_element(myStrings.begin(), myStrings.end());
```

```

// std::cout << "std::minmax_element(myStrings.begin(), myStrings.end()): "
//           << "(" << *paStr.first << ", " << *paStr.second << ")" << std::endl;

// auto paStrAsInt = std::minmax_element(myStrings.begin(), myStrings.end(), [](std::string a, std::string b)
//                                         { return tolnt(a) < tolnt(b); });
// std::cout << "std::minmax_element(myStrings.begin(), myStrings.end()): "
//           << "(" << *paStr.first << ", " << *paStr.second << ")" << std::endl;

// std::cout << std::endl;
//}
// =====
/*

```

Permutations

We can see the different permutations in a range using C++.

`std::prev_permutation` and `std::next_permutation` return the previous smaller or next bigger permutation of the newly ordered range. If a smaller or bigger permutation is not available, the algorithms return false. Both algorithms need bidirectional iterators. Per default the predefined sorting criterion `std::less` is used. If you use your sorting criterion, it has to obey the strict weak ordering. If not, the program is undefined.

Applies the previous permutation to the range:

```

bool prev_permutation(Bilt first, Bilt last)
bool prev_permutation(Bilt first, Bilt last, BiPred pre))
*/
/*

```

Applies the next permutation to the range:

```

bool next_permutation(Bilt first, Bilt last)
bool next_permutation(Bilt first, Bilt last, BiPred pre)
*/
// =====

```

```
// #include <algorithm>
```

```
// #include <iostream>
```

```
// #include <vector>
```

```
// int main()
```

```
//{
```

```
// std::cout << std::endl;
```

```
// std::vector<int> myInts{1, 2, 3};
```

```
// std::cout << "All 3! permutations"
```

```
//           << "\n\n";
```

```
// std::cout << "forwards" << std::endl;
```

```
// do
```

```
// {
```

```
//   for (auto i : myInts)
```

```
//     std::cout << i << " ";
```

```
//     std::cout << std::endl;
```

```
// } while (std::next_permutation(myInts.begin(), myInts.end()));
```

```

// std::cout << std::endl;

// std::reverse(myInts.begin(), myInts.end());

// std::cout << "backwards" << std::endl;

// do
// {
//   for (auto i : myInts)
//     std::cout << i << " ";
//   std::cout << std::endl;
// } while (std::prev_permutation(myInts.begin(), myInts.end()));

// std::cout << std::endl;
// }
// =====
/*

```

Numeric

The numeric library is host to several numeric functions. We'll look at a few of them in this lesson.

The numeric algorithms `std::accumulate`, `std::adjacent_difference`, `std::partial_sum`, `std::inner_product` and `std::iota` and the six additional C++17 algorithms `std::exclusive_scan`, `std::inclusive_scan`, `std::transform_exclusive_scan`, `std::transform_inclusive_scan`, `std::reduce`, and `std::transform_reduce` are special. All of them are defined in the header `<numeric>`. They are widely applicable, because they can be configured with a callable.

Accumulates the elements of the range. init is the start value:

```

T accumulate(Inplt first, Inplt last, T init)
T accumulate(Inplt first, Inplt last, T init, BiFun fun)
*/
/*

```

Calculates the difference between adjacent elements of the range and stores the result in result:

```

Outlt adjacent_difference(Inplt first, Inplt last, Outlt result)
Fwdlt2 adjacent_difference(ExePol pol, Fwdlt first, Fwdlt last, Fwdlt2 result)

```

`Outlt adjacent_difference(Inplt first, Inplt last, Outlt result, BiFun fun)`

`Fwdlt2 adjacent_difference(ExePol pol, Fwdlt first, Fwdlt last, Fwdlt2 result, BiFun fun)`

Calculates the partial sum of the range:

```

Outlt partial_sum(Inplt first, Inplt last, Outlt result)
Outlt partial_sum(Inplt first, Inplt last, Outlt result, BiFun fun)

```

Calculates the inner product (scalar product) of the two ranges and returns the result:

```

T inner_product(Inplt first1, Inplt last1, Outlt first2, T init)
T inner_product(Inplt first1, Inplt last1, Outlt first2, T init, BiFun fun1, BiFun fun2)

```

Assigns each element of the range a by 1 sequentially increasing value. The start value is val:

`void iota(Fwdlt first, Fwdlt last, T val)`

The algorithms are not so easy to get.

std::accumulate without callable uses the following strategy:

```
result = init;  
result += *(first+0);  
result += *(first+1);
```

std::adjacent_difference without callable uses the following strategy:

```
*(result) = *first;  
*(result+1) = *(first+1) - *(first);  
*(result+2) = *(first+2) - *(first+1);
```

std::partial_sum without callable uses the following strategy:

```
*(result) = *first;  
*(result+1) = *first + *(first+1);  
*(result+2) = *first + *(first+1) + *(first+2)
```

The challenging algorithm variation inner_product(InplI, InplI, OutlI, T, BiFun fun1, BiFun fun2) with two binary callables uses the following strategy: The second callable fun2 is applied to each pair of the ranges to generate the temporary destination range tmp, and the first callable is applied to each element of the destination range tmp for accumulating them and therefore generating the final result.

```
*/  
// ======  
// #include <array>  
// #include <iostream>  
// #include <numeric>  
// #include <vector>  
  
// int main()  
// {  
  
//   std::cout << std::endl;  
  
//   std::array<int, 9> arr{1, 2, 3, 4, 5, 6, 7, 8, 9};  
  
//   std::cout << "std::accumulate(arr.begin(), arr.end(), 0): " << std::accumulate(arr.begin(), arr.end(), 0) <<  
//   std::endl;  
//   std::cout << "std::accumulate(arr.begin(), arr.end(), 1, [](int a, int b){ return a+b;}): " <<  
//   std::accumulate(arr.begin(), arr.end(), 1, [](int a, int b)  
//                           { return a * b; })  
//   << std::endl;  
  
//   std::cout << std::endl;  
  
//   std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};  
//   std::vector<int> myVec;  
  
//   std::cout << "adjacent_difference: " << std::endl;  
//   std::adjacent_difference(vec.begin(), vec.end(), std::back_inserter(myVec), [](int a, int b)  
//                           { return a * b; });  
//   for (auto v : vec)  
//     std::cout << v << " ";
```

```

// std::cout << std::endl;
// for (auto v : myVec)
//   std::cout << v << " ";

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

// std::cout << "std::inner_product(vec.begin(), vec.end(), arr.begin(), 0): " << std::inner_product(vec.begin(),
vec.end(), arr.begin(), 0) << std::endl;

// std::cout << std::endl;

// myVec = {};
// std::partial_sum(vec.begin(), vec.end(), std::back_inserter(myVec));
// std::cout << "partial_sum: ";
// for (auto v : myVec)
//   std::cout << v << " ";

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

// std::cout << "iota: ";
// std::vector<int> myLongVec(100);
// std::iota(myLongVec.begin(), myLongVec.end(), 2000);

// for (auto v : myLongVec)
//   std::cout << v << " ";

// std::cout << "\n\n";
// }
// =====
/*
Reduce

```

We often need to reduce a numeric range. That is where `std::reduce` comes in handy.

We'll cover the following

`reduce`

`transform_reduce`:

The six new algorithms that are typically used for parallel execution are also known under the name prefix sum. If the given binary callables are not associative and commutative, the behavior of the algorithms is undefined.

`reduce#`

This reduces the elements of the range. `init` is the start value.

Behaves the same as `std::accumulate` but the range may be rearranged.

`ValType reduce(Inplt first, Inplt last)`

`ValType reduce(ExePol pol, Inplt first, Inplt last)`

`T reduce(Inplt first, Inplt last, T init)`

`T reduce(ExePol pol, Inplt first, Inplt last, T init)`

```

T reduce(InPlt first, InPlt last, T init, BiFun fun)
T reduce(ExePol pol, InPlt first, InPlt last, T init, BiFun fun)
*/
/*
transform_reduce:#
```

This transforms and reduces the elements of one or two ranges. init is the start value.

Behaves similarly to std::inner_product but the range may be rearranged.

If applied to two ranges

if not provided, multiplication is used for transforming the ranges into one range, and addition is used to reduce the intermediate-range into the result

if provided, fun1 is used for the transforming step and fun2 is used for the reducing step

If applied to a single range

fun2 is used for transforming the given range

```
T transform_reduce(InPlt first, InPlt last, InPlt first2, T init)
```

```
T transform_reduce(InPlt first, InPlt last, InPlt first2, T init, BiFun fun1, BiFun fun2)
```

```
T transform_reduce(FwdIt first, FwdIt last, FwdIt first2, T init)
```

```
T transform_reduce(ExePol pol, FwdIt first, FwdIt last, FwdIt first2, T init, BiFun fun1, BiFun fun2)
```

```
T transform_reduce(InPlt first, InPlt last, T init, BiFun fun1, UnFun fun2)
```

```
T transform_reduce(ExePol pol, FwdIt first, FwdIt last, T init, BiFun fun1, UnFun fun2)
```

i MapReduce in C++17

The Haskell function map is called std::transform in C++. When you substitute transform with map in the name std::transform_reduce, you will get std::map_reduce. MapReduce is the well-known parallel framework that first maps each value to a new value, then reduces in the second phase all values to the result.

The algorithm is directly applicable in C++17. In the following example, in the map phase, each word is mapped to its length, and the lengths of all words are then reduced to their sum during the reduce phase. The result is the sum of the length of all words.

```

std::vector<std::string> str{"Only", "for", "testing", "purpose"};

std::size_t result = std::transform_reduce(
    std::execution::par, str.begin(), str.end(), 0,
    [](std::size_t a, std::size_t b){ return a + b; },
    [](std::string s){ return s.length(); }
);

std::cout << result << std::endl; // 21
*/
// =====
// =====
/*
Scan
Scan operations are useful when working with prefix sums.
```

exclusive_scan: computes the exclusive prefix sum using a binary operation

Behaves similar to std::reduce, but provides a range of all prefix sums
excludes the last element in each iteration

OutIt exclusive_scan(InIt first, InIt last, OutIt first, T init)
FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FWdIt last, FwdIt2 first2, T init)

OutIt exclusive_scan(InIt first, InIt last, OutIt first, T init, BiFun fun)
FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2, T init, BiFun fun)

inclusive_scan: computes the inclusive prefix sum using a binary operation

Behaves similar to std::reduce, but provides a range of all prefix sums
includes the last element in each iteration

OutIt inclusive_scan(InIt first, InIt last, OutIt first2)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2)

OutIt inclusive_scan(InIt first, InIt last, OutIt first, BiFun fun)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2, BiFun fun)

OutIt inclusive_scan(InIt first, InIt last, OutIt firs2t, BiFun fun, T init)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2, BiFun fun, T init)
*/
/*

transform_exclusive_scan: first transforms each element and then computes the exclusive prefix sums

OutIt transform_exclusive_scan(InIt first, InIt last, OutIt first2, T init, BiFun fun, UnFun fun2)
FwdIt2 transform_exclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2, T init, BiFun fun, UnFun fun2)

transform_inclusive_scan: first transforms each element of the input range and then computes the inclusive prefix sums
OutIt transform_inclusive_scan(InIt first, InIt last, OutIt first2, BiFun fun, UnFun fun2)
FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt first2, BiFun fun, UnFun fun2)
OutIt transform_inclusive_scan(InIt first, InIt last, OutIt first2, BiFun fun, UnFun fun2, T init)
FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt first2, BiFun fun, UnFun fun2, T init)
*/

// =====

// #include <iostream>
// #include <numeric>

// #include <string>

// #include <vector>

// int main(){

// std::cout << std::endl;

// // for_each_n

// std::vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

```

// std::for_each_n(std::execution::par,
//                 intVec.begin(), 5, [](int& arg){ arg *= arg; });

// std::cout << "for_each_n: ";
// for (auto v: intVec) std::cout << v << " ";
// std::cout << "\n\n";

// // exclusive_scan and inclusive_scan
// std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
// std::exclusive_scan(std::execution::par,
//                     resVec.begin(), resVec.end(), resVec.begin(), 1,
//                     [](int fir, int sec){ return fir * sec; });

// std::cout << "exclusive_scan: ";
// for (auto v: resVec) std::cout << v << " ";
// std::cout << std::endl;

// std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8, 9};

// std::inclusive_scan(std::execution::par,
//                     resVec2.begin(), resVec2.end(), resVec2.begin(),
//                     [](int fir, int sec){ return fir * sec; }, 1);

// std::cout << "inclusive_scan: ";
// for (auto v: resVec2) std::cout << v << " ";
// std::cout << "\n\n";

// // transform_exclusive_scan and transform_inclusive_scan
// std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8, 9};
// std::vector<int> resVec4(resVec3.size());
// std::transform_exclusive_scan(std::execution::par,
//                             resVec3.begin(), resVec3.end(),
//                             resVec4.begin(), 0,
//                             [](int fir, int sec){ return fir + sec; },
//                             [](int arg){ return arg *= arg; });

// std::cout << "transform_exclusive_scan: ";
// for (auto v: resVec4) std::cout << v << " ";
// std::cout << std::endl;

// std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};
// std::vector<int> resVec5(strVec.size());

// std::transform_inclusive_scan(std::execution::par,
//                             strVec.begin(), strVec.end(),
//                             resVec5.begin(), 0,
//                             [] (auto fir, auto sec){ return fir + sec; },
//                             [] (auto s){ return s.length(); });

// std::cout << "transform_inclusive_scan: ";
// for (auto v: resVec5) std::cout << v << " ";

```

```

// std::cout << "\n\n";
// // reduce and transform_reduce
// std::vector<std::string> strVec2{"Only", "for", "testing", "purpose"};
// std::string res = std::reduce(std::execution::par,
//                             strVec2.begin() + 1, strVec2.end(), strVec2[0],
//                             [](auto fir, auto sec){ return fir + ":" + sec; });
// std::cout << "reduce: " << res << std::endl;
// std::size_t res7 = std::parallel::transform_reduce(std::execution::par,
//                                                 strVec2.begin(), strVec2.end(), 0,
//                                                 [] (std::size_t a, std::size_t b){ return a + b; },
//                                                 [] (std::string s){ return s.length(); });
// std::cout << "transform_reduce: " << res7 << std::endl;
// std::cout << std::endl;
// }
// =====
/*
numeric:
```

Random Numbers

There's a whole library dedicated to the generation and usage of random numbers.

We'll cover the following

Random number generator

Random number distribution

C++ inherits the numeric functions from C and has a random number library.

Random numbers are necessary for many domains, e.g., to test software, to generate cryptographic keys or for computer games. The random number facility of C++ consists of two components. There is the generation of the random numbers, and there is the distribution of these random numbers. Both components need the header `<random>`.

Random number generator#

The random number generator generates a random number stream between a minimum and maximum value. This stream is initialized by a “so-called” seed, guaranteeing different sequences of random numbers.

1234

```
#include <random>
```

...

```
std::random_device seed;
```

```
std::mt19937 generator(seed());
```

A random number generator gen of type Generator supports four different requests:

Generator::result_type: Data type of the generated random number.

`gen()`: Returns a random number.

`gen.min()`: Returns the minimum random number that can be returned by `gen()`.

`gen.max()`: Returns the maximum random number that can be returned by `gen()`.

The random number library supports several random number generators. The best known are the Mersenne Twister, the `std::default_random_engine` that is chosen by the implementation and `std::random_device`. `std::random_device` is the only true random number generator, but not all platforms offer it.

Random number distribution#

The random number distribution maps the random number with the help of the random number generator `gen` to the selected distribution.

```
#include <random>
...
std::random_device seed;
std::mt19937 generator(seed);
```

A random number generator `gen` of type `Generator` supports four different requests:

`Generator::result_type`: Data type of the generated random number.

`gen()`: Returns a random number.

`gen.min()`: Returns the minimum random number that can be returned by `gen()`.

`gen.max()`: Returns the maximum random number that can be returned by `gen()`.

The random number library supports several random number generators. The best known are the Mersenne Twister, the `std::default_random_engine` that is chosen by the implementation and `std::random_device`. `std::random_device` is the only true random number generator, but not all platforms offer it.

Random number distribution#

The random number distribution maps the random number with the help of the random number generator `gen` to the selected distribution.

```
#include <random>
...
std::random_device seed;
std::mt19937 gen(seed());
std::uniform_int_distribution<> unDis(0, 20); // distribution between 0 and 20
unDis(gen); // generates a random number
```

C++ has several discrete and continuous random number distributions. The discrete random number distribution generates integers, the continuous random number distribution generates floating point numbers.

```
class bernoulli_distribution;
template<class T = int> class uniform_int_distribution;
```

```

template<class T = int> class binomial_distribution;
template<class T = int> class geometric_distribution;
template<class T = int> class negative_binomial_distribution;
template<class T = int> class poisson_distribution;
template<class T = int> class discrete_distribution;
template<class T = double> class exponential_distribution;
template<class T = double> class gamma_distribution;
template<class T = double> class weibull_distribution;
template<class T = double> class extreme_value_distribution;
template<class T = double> class normal_distribution;
template<class T = double> class lognormal_distribution;
template<class T = double> class chi_squared_distribution;
template<class T = double> class cauchy_distribution;
template<class T = double> class fisher_f_distribution;
template<class T = double> class student_t_distribution;
template<class T = double> class piecewise_constant_distribution;
template<class T = double> class piecewise_linear_distribution;
template<class T = double> class uniform_real_distribution;

```

Class templates with a default template argument int are discrete. The Bernoulli distribution generates booleans.

Here is an example using the Mersenne Twister std::mt19937 as the pseudo random-number generator for generating 1 million random numbers. The random number stream is mapped to the uniform and normal (or Gaussian) distribution.

```

#include <cstdlib>
#include <fstream>
#include <iostream>
#include <map>
#include <random>

static const int NUM=1000000;

void writeToFile(const char* fileName , const std::map<int, int>& data ){

    std::ofstream file(fileName);

    if ( !file ){
        std::cerr << "Could not open the file " << fileName << ".";
        exit(EXIT_FAILURE);
    }

    // print the datapoints to the file
    for ( auto mapIt: data) file << mapIt.first << " " << mapIt.second << std::endl;

}

int main(){

    std::random_device seed;

```

```

// default generator
std::mt19937 engine(seed());

// distributions

// min= 0; max= 20
std::uniform_int_distribution<> uniformDist(0, 20);
// mean= 50; sigma= 8
std::normal_distribution<> normDist(50, 8);
// mean= 6;
std::poisson_distribution<> poiDist(6);
// alpha= 1;
std::gamma_distribution<> gammaDist;

std::map<int, int> uniformFrequency;
std::map<int, int> normFrequency;
std::map<int, int> poiFrequency;
std::map<int, int> gammaFrequency;

for ( int i=1; i<= NUM; ++i){
    ++uniformFrequency[uniformDist(engine)];
    ++normFrequency[round(normDist(engine))];
    ++poiFrequency[poiDist(engine)];
    ++gammaFrequency[round(gammaDist(engine))];
}

writeToFile("uniform_int_distribution.txt", uniformFrequency);
writeToFile("normal_distribution.txt", normFrequency);
writeToFile("poisson_distribution.txt", poiFrequency);
writeToFile("gamma_distribution.txt", gammaFrequency);

}
*/
/*

```

Functions Inherited from C

As C++ evolved from C, many of the functions have been passed down to the new language. We'll discuss them in this lesson.

C++ inherited many numeric functions from C. They need the header < cmath >. The table below shows the names of these functions:

pow	sin	tanh	asinh	fabs
exp	cos	asin	aconst	fmod
sqrt	tan	acos	atanh	frexp
log	sinh	atan	ceil	ldexp
log10	cosh	atan2	floor	modf

Mathematical functions in the cmath library

Additionally, C++ inherits further mathematical functions from C. They are defined in the header < cstdlib >. Once more, the names of mathematical functions in < cstdlib >:

```
abs    llabs   ldiv    srand
labs    div    lldiv   rand
Mathematical functions in < cstdlib >
```

All functions for integers are available for the types int, long and long long; all functions for floating-point numbers are available for the types float, double, and long double.

The numeric functions need to be qualified with the namespace std.

```
/*
// =====
// #include <cmath>
// #include <ctime>
// #include <cstdlib>
// #include <iostream>

// int main(){

// std::cout << std::endl;

// std::cout << "cmath" << std::endl;

// std::cout << "std::pow(2, 10): " << std::pow(2, 10) << std::endl;
// std::cout << "std::pow(2, 0.5): " << std::pow(2, 0.5) << std::endl;
// std::cout << "std::exp(1): " << std::exp(1) << std::endl;
// std::cout << "std::ceil(5.5): " << std::ceil(5.5) << std::endl;
// std::cout << "std::floor(5.5): " << std::floor(5.5) << std::endl;
// std::cout << "std::fmod(5.5, 2): " << std::fmod(5.5, 2) << std::endl;
// double intPart;
// auto fracPart= std::modf(5.7, &intPart);
// std::cout << "fmod(5.7, &intPart): " << intPart << " + " << fracPart << std::endl;

// std::cout << "\ncstdlib: " << "\n\n";
// std::div_t divresult= std::div(14, 5);
// std::cout << "std::div(14, 5): " << divresult.quot << " remainder: " << divresult.rem << std::endl;

// // seed
// std::srand(time(nullptr));
// for ( int i=0; i <= 10; ++i){
//   std::cout << "Dice: " << (rand()%6 + 1) << std::endl;
// }

// std::cout << std::endl;

//}

// =====
/*
strings:
```

Introduction

In this chapter, we'll explore one of the most prominent classes in C++: Strings.

Let's begin!

A string is a sequence of characters. C++ has many methods to analyze or to change a string. C++ strings are the safe replacement for C Strings: `const char*`. Strings need the header `<string>`.

svg viewer

i A string is very similar to an `std::vector`

A string is similar to an `std::vector` containing characters. It supports a very similar interface. This means that in addition to the methods of the string class, we can access the algorithms of the Standard Template Library to work with the string.

The following code snippet has the variable `std::string name` with the value `RainerGrimm`. We have used the STL algorithm `std::find_if` to get the upper letter and then extract my first and last name into the variables `firstName` and `lastName`. The expression `name.begin() + 1` shows that strings support random access iterators:

```
//string versus vector
...
#include <algorithm>
#include <string>

std::string name{"RainerGrimm"};
auto strIt= std::find_if(name.begin() + 1, name.end(),
    [] (char c){ return std::isupper(c); });
if (strIt != name.end()){
    firstName= std::string(name.begin(), strIt);
    lastName= std::string(strIt, name.end());
}
```

Strings are class templates parametrized by their character, their character trait and their allocator. The character trait and the allocator have defaults.

```
template <typename charT, typename traits= char_traits<charT>, typename Allocator= allocator<charT> >
class basic_string;
```

C++ has synonyms for the character types `char`, `wchar_t`, `char16_t`, and `char32_t`.

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

i `std::string` is the string

When speaking about strings in C++, we refer with 99% probability to the specialization `std::basic_string` for the character type `char`. This statement is also true for this course.

*/

/*

Create and Delete

Let's start things off by learning how to create and destroy strings.

C++ offers many methods to create strings from C or C++ strings. Under the hood, there is always a C string involved for creating a C++string. That changes with C++14, because the new C++ standard supports C++ string literals: std::string str{"string"s}. With the suffix s, the C string literal "string literal" becomes a C++string literal:"string literal"s.

The table gives us an overview of the methods to create and delete a C++string.

Methods	Example
Default	std::string str
Copies from a C++ string	std::string str(oth)
Moves from a C++ string	std::string str(std::move(oth))
From the range of a C++ string	std::string(oth.begin(), oth.end())
From a substring of a C++ string	std::string(oth, otherIndex)
From a substring of a C++ string	std::string(oth, otherIndex, strlen)
From a C string	std::string str("c-string")
From a C array	std::string str("c-array", len)
From characters	std::string str(num, 'c')
From a initializer list	std::string str({'a', 'b', 'c', 'd'})
From a substring	str= other.substring(3, 10)
Destructor	str::~string()
Methods to create and delete a string	
*/	
// =====	
// #include <iostream>	
// #include <string>	
// #include <utility>	
// int main(){	
// std::cout << std::endl;	
// std::string defaultString;	
// std::cout << "From C-String" << std::endl;	
// std::string other{"123456789"};	
// std::cout << "other: " << other << std::endl;	
// std::cout << std::endl;	
// std::cout << "From C++-string" << std::endl;	
// std::string str1(other);	
// std::string tmp(other);	
// std::string str2(std::move(tmp));	
// std::string str3(other.begin(), other.end());	
// std::string str4(other, 2);	
// std::string str5(other, 2, 5);	
// std::cout << "str1: " << str1 << std::endl;	

```

// std::cout << "str2: " << str2 << std::endl;
// std::cout << "str3: " << str3 << std::endl;
// std::cout << "str4: " << str4 << std::endl;
// std::cout << "str5: " << str5 << std::endl;

// std::cout << std::endl;

// std::cout << "From C-String" << std::endl;

// std::string str6("123456789", 5);
// std::string str7(5, '1');
// std::string str8({'1', '2', '3', '4', '5', '6', '7', '8', '9'});

// std::cout << "str6: " << str6 << std::endl;
// std::cout << "str7: " << str7 << std::endl;
// std::cout << "str8: " << str8 << std::endl;

// std::cout << std::endl;

// std::cout << "As Part of a C++-String" << std::endl;
// std::cout << "str6.substr(): " << str6.substr() << std::endl;
// std::cout << "str6.substr(1): " << str6.substr(1) << std::endl;
// std::cout << "str6.substr(1, 2): " << str6.substr(1, 2) << std::endl;

// std::cout << std::endl;
// }
// =====
/*

```

Conversion Between C++ and C Strings

There are several ways to convert C++ strings to C strings. Let's go through them now.

While the conversion of a C string in a C++ string is done implicitly, we must explicitly request conversion from a C++ string into a C string. `str.copy()` copies the content of a C++ string without the terminating \0 character. `str.data()` and `str.c_str()` include the terminating null character.

 Be careful with `str.data()` and `str.c_str()`

The return value of the two methods `str.data()` and `std.c_str()` becomes invalid if `str` is modified.

```

*/
// =====
=====

// #include <iostream>
// #include <string>

// int main(){

// std::cout << std::endl;

// std::string str{"C++-String"};
// std::cout << str << std::endl;
// str += " C-String";

```

```

// std::cout << str << std::endl;

// const char* cString= str.c_str();

// char buffer[10];
// str.copy(buffer, 10);

// str+= "works";
// const char* cString2= cString;

// std::string str2(buffer, buffer+10);
// std::cout << str2 << std::endl;

// std::cout << std::endl;
//}
//
=====
=====

/*
Size versus Capacity
We'll test and alter the capacity of a string.

```

The number of elements a string has (`str.size()`) is in general smaller than the number of elements for which space is reserved: `str.capacity()`. Therefore if we add elements to a string, new memory will not necessarily be allocated. `std::max_size()` return the maximum amount of elements a string can have. For the three methods the following relation holds: `str.size() <= str.capacity() <= str.max_size()`.

The following table shows the methods for dealing with memory management of strings.

```

*/
=====
=====

/*
Methods      Description
str.empty()   Checks if str has elements.
str.size(), str.length() Number of elements of the str.
str.capacity() Number of elements str can have without reallocation.
str.max_size() Number of elements str can maximal have.
str.resize(n) Increases str to n elements.
str.reserve(n) Reserves memory for a least n elements.
str.shrink_to_fit()    Adjusts the capacity of the string to it's size.

```

The request `str.shrink_to_fit()` is, as in the case of `std::vector`, non-binding.

```

*/
=====
// #include <iostream>
// #include <string>

// void showStringInfo(const std::string& s){

```

```

// std::cout << s << std::endl;
// std::cout << "s.size(): " << s.size() << std::endl;
// std::cout << "s.capacity(): " << s.capacity() << std::endl;
// std::cout << "s.max_size(): " << s.max_size() << std::endl;
// std::cout << std::endl;

//}

// int main(){

// std::string str;
// showStringInfo(str);

// str += "12345";
// showStringInfo(str);

// str.resize(30);
// showStringInfo(str);

// str.reserve(1000);
// showStringInfo(str);

// str.shrink_to_fit();
// showStringInfo(str);

//}

// =====
/*

```

Comparison and Concatenation

Can we merge and compare strings like we did with ranges? This lesson shows us how.

We'll cover the following

Comparison

String concatenation

Comparison#

Strings support the well-known comparison operators ==, !=, <, >, >=. The comparison of two strings takes place on their elements.

*/

```

// =====
// #include <iostream>
// #include <string>

// int main()
// {

// std::cout << std::boolalpha << std::endl;

// std::string first{"aaa"};
// std::string second{"aaaa"};
```

```

// std::cout << "first < first :" << (first < first) << std::endl;
// std::cout << "first <= first :" << (first <= first) << std::endl;
// std::cout << "first < second :" << (first < second) << std::endl;

// std::cout << std::endl;

// std::string one{"1"};
// std::string oneOneOne = one + std::string("1") + "1";

// std::cout << "1 + 1 + 1: " << oneOneOne << std::endl;

// std::cout << std::endl;
// }
// =====
/*
String concatenation#
The + operator is overloaded for strings, so we can add strings.

```

 The + operator is only overloaded for C++ strings

The C++ type system permits concatenation of C++ and C strings into C++ strings, but not concatenation of C++ and C strings into C strings. The reason is that the + operator is overloaded for C++ strings. Therefore only the second line is valid C++, because the C string is implicitly converted to a C++ string:

```

//...
#include <string>
//...
std::string wrong= "1" + "1"; // ERROR
std::string right= std::string("1") + "1"; // 11
*/
// =====
// =====
/*

```

Element Access

Accessing a character in a string is very easy and similar to element access in arrays.

We'll cover the following

Access the elements of the string

Access to the elements of a string str is very convenient because the strings support random access iterators. We can access with str.front() the first character and with str.back() the last character of the string. With str[n] and str.at(n) we get the n-th element by index.

The following table provides an overview.

Access the elements of the string#

Methods	Example
---------	---------

str.front() Returns the first character of str.

str.back() Returns the last character of str.

str[n] Returns the n-th character of str. The string boundaries will not be checked.

str.at(n) Returns the n-th character of str. The string boundaries will be checked. If the boundaries are violated a std::out_of_range exception is thrown.

```

*/
// =====
// #include <iostream>
// #include <stdexcept>
// #include <string>
// #include <vector>

// int main(){

// std::cout << std::endl;

// std::string str= {"0123456789"};
// std::cout << "str.front(): " << str.front() << std::endl;
// std::cout << "str.back(): " << str.back() << std::endl;

// std::cout << std::endl;

// for (int i=0; i <= 10; ++i){
//   std::cout << "str[" << i << "]: " << str[i] << std::endl;
// }

// std::cout << std::endl;

// try{
//   str.at(10);
// }
// catch (const std::out_of_range& e){
//   std::cerr << "Exception: " << e.what() << std::endl;
// }

// std::cout << std::endl;

// std::cout << "*(&str[0]+5): " << *(&str[0]+5) << std::endl;
// std::cout << "*(&str[5]): " << *(&str[5]) << std::endl;
// std::cout << "str[5] : " << str[5] << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

It is particularly interesting to see in the example that the compiler performs the invocation `str[10]`. The access outside the string boundaries is undefined behavior. In contrast, the C++ runtime throws an `std::out_of_range` exception for the call `str.at(10)`.

In the next lesson, we'll discuss how we can use strings for taking input and output the data.

```

*/
// =====
// =====
/*
Input and Output

```

Strings are well-known for their use in input and output of data. Let's look at how this is done.

A string can read from an input stream via `>>` and write to an output stream via `<<`.

The global function `getline` empowers us to read from an input stream line by line until the end-of-file character.

There are four variations of the `getline` function available. The first two arguments are the input stream `is` and the string `line` holding the line read. Optionally we can specify a special line separator. The function returns by reference to the input stream.

```
istream& getline (istream& is, string& line, char delim);
istream& getline (istream&& is, string& line, char delim);
istream& getline (istream& is, string& line);
istream& getline (istream&& is, string& line);
```

`getline` consumes the whole line including empty spaces. Only the line separator is ignored. The function needs the header `<string>`.

Before executing the code, first write the filename (`string.txt`) using STDIN button in the given space.

```
/*
// =====
// #include <fstream>
// #include <iostream>
// #include <string>
// #include <vector>

// std::vector<std::string> readFromFile(const char *fileName)
//{
//    std::ifstream file(fileName);

//    if (!file)
//    {
//        std::cerr << "Could not open the file " << fileName << ".";
//        exit(EXIT_FAILURE);
//    }

//    std::vector<std::string> lines;
//    std::string line;
//    while (getline(file, line))
//        lines.push_back(line);

//    return lines;
//}

// int main()
//{
//    std::cout << std::endl;
```

```

// std::string fileName;
// std::cout << "Your filename: ";
// std::cin >> fileName;

// std::vector<std::string> lines = readFromFile(fileName.c_str());

// int num{0};
// for (auto line : lines)
//   std::cout << ++num << ":" << line << std::endl;

// std::cout << std::endl;
//}
//=====
/*
string.txt

```

A thrifty alligator's tiger comes with it the thought that the dazzling goldfish is a turtle.

The program displays the lines of an arbitrary file including line numbers. The expression `std::cin >> fileName` reads the file name. The function `readFromFile` reads with `getline` all file lines and pushes them into the vector.

```

*/
//=====
//=====
/*
Search

```

In this lesson, we'll examine the different search features available in the string class.

C++ offers many ways to search in a string. Each way exists in various overloaded forms.

i Search is called find

It seems a bit odd but the algorithms for searching in a string start with the prefix “find”. If the search was successful, we get an index of type `std::string::size_type`, if not, we get the constant `std::string::npos`. The first character has the index 0.

The find algorithms support:

searching for a character from a C or C++ string.

searching forward and backward.

searching for the presence or lack of characters in a C or C++ string.

starting the search at an arbitrary position in the string.

The arguments of all six variations of the find function follow a similar pattern. The first argument is the text we are searching for. The second argument holds the start position of the search, and the third represents for the number of characters starting from the second argument.

Here are the six variations:

Methods	Description
str.find(...)	Returns the first position of a character, a C or C++ string in str.
str.rfind(...)	Returns the last position of a character, a C or C++ string in str.
str.find_first_of(...)	Returns the first position of a character from a C or C++ string in str.
str.find_last_of(...)	Returns the last position of a character from a C or C++ string in str.
str.find_first_not_of(...)	Returns the first position of a character in str, which is not from a C or C++ string.
str.find_last_not_of(...)	Returns the last position of a character in str, which is not from a C or C++ string.
Find variations of the string	
/*	
// =====	
// #include <iostream>	
// #include <string>	
// int main()	
// {	
// std::string str;	
// auto idx = str.find("no");	
// if (idx == std::string::npos)	
// std::cout << "not found"; // not found	
// str = {"dkeu84kf8k48kdj39kdj74945du942"};	
// std::string str2{"84"};	
// std::cout << str.find('8') << std::endl; // 4	
// std::cout << str.rfind('8') << std::endl; // 11	
// std::cout << str.find('8', 10) << std::endl; // 11	
// std::cout << str.find(str2) << std::endl; // 4	
// std::cout << str.rfind(str2) << std::endl; // 4	
// std::cout << str.find(str2, 10) << std::endl; // 18446744073709551615	
// str2 = "0123456789";	
// std::cout << str.find_first_of("678") << std::endl; // 4	
// std::cout << str.find_last_of("678") << std::endl; // 20	
// std::cout << str.find_first_of("678", 10) << std::endl; // 11	
// std::cout << str.find_first_of(str2) << std::endl; // 4	
// std::cout << str.find_last_of(str2) << std::endl; // 29	
// std::cout << str.find_first_of(str2, 10) << std::endl; // 10	
// std::cout << str.find_first_not_of("678") << std::endl; // 0	
// std::cout << str.find_last_not_of("678") << std::endl; // 29	
// std::cout << str.find_first_not_of("678", 10) << std::endl; // 10	
// std::cout << str.find_first_not_of(str2) << std::endl; // 0	
// std::cout << str.find_last_not_of(str2) << std::endl; // 26	
// std::cout << str.find_first_not_of(str2, 10) << std::endl; // 12	
// return 0;	
// }	
// =====	
/*	

The call std::find(str2, 10) returns std::string::npos. The value for this, on this platform, is 18446744073709551615.

```
/*
// =====
// =====
/*
Problem statement#
The find variations of std::string have a special interface. Write a program that uses a few of these variations.

*/
// =====
// #include <iostream>
// #include <string>

// int main()
//{
// std::cout << std::endl;

// std::string str;

// // std::string::size_type idx= str.find("no");
// auto idx = str.find("no");

// if (idx == std::string::npos)
// std::cout << "no not found " << std::endl;

// std::cout << std::endl;

// str = {"dkeu84kf8k48kdj39kdj74945du942"};
// std::string str2{"84"};

// std::cout << "str: " << str << std::endl;
// std::cout << "str2: " << str2 << std::endl;

// std::cout << "str.find('8'): " << str.find('8') << std::endl;
// std::cout << "str.rfind('8'): " << str.rfind('8') << std::endl;
// std::cout << "str.find('8', 10): " << str.find('8', 10) << std::endl;
// std::cout << "str.find(str2): " << str.find(str2) << std::endl;
// std::cout << "str.rfind(str2): " << str.rfind(str2) << std::endl;
// std::cout << "str.find(str2, 10): " << str.find(str2, 10) << std::endl;

// std::cout << std::endl;

// str2 = "0123456789";
// std::cout << "str: " << str << std::endl;
// std::cout << "str2: " << str2 << std::endl;

// std::cout << "str.find_first_of(678): " << str.find_first_of("678") << std::endl;
// std::cout << "str.find_last_of(678): " << str.find_last_of("678") << std::endl;
// std::cout << "str.find_first_of(678, 10): " << str.find_first_of("678", 10) << std::endl;
```

```

// std::cout << "str.find_first_of(str2): " << str.find_first_of(str2) << std::endl;
// std::cout << "str.find_last_of(str2): " << str.find_last_of(str2) << std::endl;
// std::cout << "str.find_first_of(str2, 10): " << str.find_first_of(str2, 10) << std::endl;

// std::cout << std::endl;

// std::cout << "str: " << str << std::endl;
// std::cout << "str2: " << str2 << std::endl;

// std::cout << "str.find_first_not_of(678): " << str.find_first_not_of("678") << std::endl;
// std::cout << "str.find_last_not_of(678): " << str.find_last_not_of("678") << std::endl;
// std::cout << "str.find_first_not_of(678, 10): " << str.find_first_not_of("678", 10) << std::endl;
// std::cout << "str.find_first_not_of(str2): " << str.find_first_not_of(str2) << std::endl;
// std::cout << "str.find_last_not_of(str2): " << str.find_last_not_of(str2) << std::endl;
// std::cout << "str.find_first_not_of(str2, 10): " << str.find_first_not_of(str2, 10) << std::endl;

// std::cout << std::endl;
// }
// =====
/*

```

Modifying Operations

C++ gives us a variety of tools to modify and manipulate strings.

Strings have many operations that can modify them. `str.assign` assigns a new string to the string `str`. With `str.swap` we can swap two strings. To remove a character from a string use `str.pop_back` or `str.erase`. On the contrary, `str.clear` or `str.erase` deletes the whole string. To append new characters to a string, use `+=`, `std.append` or `str.push_back`. We can use `str.insert` to insert new characters or `str.replace` to replace them.

Methods Description

<code>str= str2</code>	Assigns <code>str2</code> to <code>str</code> .
<code>str.assign(...)</code>	Assigns a new string to <code>str</code> .
<code>str.swap(str2)</code>	Swaps <code>str</code> and <code>str2</code> .
<code>str.pop_back()</code>	Removes the last character from <code>str</code> .
<code>str.erase(...)</code>	Removes characters from <code>str</code> .
<code>str.clear()</code>	Clears the <code>str</code> .
<code>str.append(...)</code>	Appends characters to <code>str</code> .
<code>str.push_back(s)</code>	Appends the character <code>s</code> to <code>str</code> .
<code>str.insert(pos, ...)</code>	Inserts characters in <code>str</code> starting at <code>pos</code> .
<code>str.replace(pos, len, ...)</code>	Replaces the <code>len</code> characters from <code>str</code> starting at <code>pos</code>

Methods for modifying a string

The operations have many overloaded versions. The methods `str.assign`, `str.append`, `str.insert`, and `str.replace` are very similar. All four can not only be invoked with C++ strings and substrings but also characters, C strings, C string arrays, ranges, and initializer lists. `str.erase` can not only erase a single character and whole ranges, but also many characters starting at a given position.

The following code snippet shows many of the variations. For the sake of simplicity, only the effects of the strings modifications are displayed:

```

*/
// =====
// #include <iostream>
```

```
// #include <string>

// int main(){
//   std::cout << std::endl;
//   std::cout << "ASSIGN: " << std::endl;
//   std::string str{"New String"};
//   std::string str2{"Other String"};
//   std::cout << "str: " << str << std::endl;
//
//   str.assign(str2, 4, std::string::npos);
//   std::cout << str << std::endl;
//
//   str.assign(5, '-');
//   std::cout << str << std::endl;
//   std::cout << std::endl;
//
//   std::cout << "DELETE" << std::endl;
//
//   str={"0123456789"};
//   std::cout << "str: " << str << std::endl;
//   str.erase(7, 2);
//   std::cout << str << std::endl;
//
//   str.erase(str.begin() + 2, str.end() - 2);
//   std::cout << str << std::endl;
//
//   str.erase(str.begin() + 2, str.end());
//   std::cout << str << std::endl;
//
//   str.pop_back();
//   std::cout << str << std::endl;
//
//   str.erase();
//   std::cout << str << std::endl;
//
//   std::cout << "APPEND" << std::endl;
//
//   str="01234";
//   std::cout << "str: " << str << std::endl;
//
//   str+="56";
//   std::cout << str << std::endl;
//
//   str+='7';
//   std::cout << str << std::endl;
//
//   str+={'8', '9'};
//   std::cout << str << std::endl;
```

```
// str.append(str);
// std::cout << str << std::endl;

// str.append(str, 2, 4);
// std::cout << str << std::endl;

// str.append(10, '0');
// std::cout << str << std::endl;

// str.append(str, 10, 10);
// std::cout << str << std::endl;

// str.push_back('9');
// std::cout << str << std::endl;

// std::cout << std::endl;

// std::cout << "INSERT" << std::endl;
// str={"345"};
// std::cout << "str: " << str << std::endl;

// str.insert(3, "6789");
// std::cout << str << std::endl;

// str.insert(0, "012");
// std::cout << str << std::endl;

// std::cout << std::endl;

// std::cout << "REPLACE" << std::endl;
// str={"only for testing purpose."};
// std::cout << "str: " << str << std::endl;

// str.replace(0, 0, "O");
// std::cout << str << std::endl;

// str.replace(0, 5, "Only", 0, 4);
// std::cout << str << std::endl;

// str.replace(16, 8, "");
// std::cout << str << std::endl;

// str.replace(4, 0, 5, 'y');
// std::cout << str << std::endl;

// str.replace(str.begin(), str.end(), "Only for testing purpose.");
// std::cout << str << std::endl;

// str.replace(str.begin()+4, str.end()-8, 10, '#');
// std::cout << str << std::endl;
```

```
// std::cout << std::endl;  
// }  
// ======  
/*
```

Numeric Conversions

Apart from conversions to C string, a string can also be converted to a float.

We can convert numbers or floating point numbers to the corresponding `std::string` or `std::wstring` with `std::to_string(val)` and `std::to_wstring(val)`.

The inverse of the aforementioned (for the numbers or floating point numbers) can be achieved through a function family of `sto*` functions. All functions need the header `<string>`.

 Read `sto *` as 'string to'

The seven ways to convert a string into a natural or floating point number follow a simple pattern. All functions start with `sto` and add further characters, denoting the type to which the strings should be converted. For example, `stol` stands for 'string to long' or `stod` for 'string to double'.

The `sto` functions all have the same interface. The example shows it for the type `long`.

```
std::stol(str, idx= nullptr, base= 10)
```

The function takes a string and determines the long representation of the base `base`. `stol` ignores leading spaces and optionally returns the index of the first invalid character in `idx`. By default, the base is 10. Valid values for the base are 0 and 2 through 36. If we use base 0, the compiler automatically determines the type based on the format of the string. If the base is bigger than 10, the compiler encodes them in the characters a to z. The representation is analogous to the representation of hexadecimal numbers.

The table gives an overview of all functions.

Method Description

<code>std::to_string(val)</code>	Converts <code>val</code> into a <code>std::string</code> .
<code>std::to_wstring(val)</code>	Converts <code>val</code> into a <code>std::wstring</code> .
<code>std::stoi(str)</code>	Returns an <code>int</code> value.
<code>std::stol(str)</code>	Returns a <code>long</code> value.
<code>std::stoll(str)</code>	Returns a <code>long long</code> value.
<code>std::stoul(str)</code>	Returns an <code>unsigned long</code> value.
<code>std::stoull(str)</code>	Returns an <code>unsigned long long</code> value.
<code>std::stof(str)</code>	Returns a <code>float</code> value.
<code>std::stod(str)</code>	Returns a <code>double</code> value.
<code>std::stold(str)</code>	Returns an <code>long double</code> value.

Numeric conversion of strings

 Where is the `stou` function?

In case we're curious, the C++ `sto` functions are thin wrappers around the C `strto*` functions, but there is no `strtou` function in C. Therefore C++ has no `stou` function.

The functions throw an `std::invalid_argument` exception if the conversion is not possible. If the determined value is too big for the destination type, we get an `std::out_of_range` exception.

```
*/  
// ======  
// #include <iostream>  
// #include <limits>  
// #include <string>  
  
// int main(){  
  
// //std::cout << std::endl;  
  
// std::cout << "to_string, to_wstring" << std::endl;  
  
// std::string maxLongLongString=std::to_string(std::numeric_limits<long long>::max());  
// std::wstring maxLongLongWstring=std::to_wstring(std::numeric_limits<long long>::max());  
  
// std::cout << std::numeric_limits<long long>::max() << std::endl;  
// std::cout << maxLongLongString << std::endl;  
// std::wcout << maxLongLongWstring << std::endl;  
  
// std::cout << std::endl;  
  
// std::cout << "atoi* " << std::endl;  
  
// std::string str("10010101");  
  
// std::cout << std::stoi(str) << std::endl;  
// std::cout << std::stoi(str, nullptr, 16) << std::endl;  
// std::cout << std::stoi(str, nullptr, 8) << std::endl;  
// std::cout << std::stoi(str, nullptr, 2) << std::endl;  
  
// std::cout << std::endl;  
  
// std::size_t idx;  
// std::cout << std::stod(" 3.5 km", &idx) << std::endl;  
// std::cout << "Not numeric char at position " << idx << "." << std::endl;  
  
// std::cout << std::endl;  
  
// try{  
//   std::cout << std::stoi(" 3.5 km") << std::endl;  
//   std::cout << std::stoi(" 3.5 km", nullptr, 2) << std::endl;  
  
// }  
// catch (const std::exception& e){  
//   std::cerr << e.what() << std::endl;  
// }  
  
// std::cout << std::endl;  
  
// }
```

```
// =====  
/*
```

Introduction

The string view class builds itself on the string class. It restricts the operations that can be performed on a string.

A string view is a non-owning reference to a string. It represents a view of a sequence of characters. This sequence of characters can be a C++ string or a C-string. A string view needs the header <string_view>.

 A string view is a for copying optimized string

From a birds-eye perspective the purpose of std::string_view is to avoid copying data that is already owned by someone else and to allow immutable access to a std::string like object. The string view is a kind of a restricted string that supports only immutable operations. Additionally, a string view sv has two additional mutating operations: sv.remove_prefix and sv.remove_suffix.

String views are class templates parameterized by their character and their character trait. The character trait has a default. In contrast to a string, a string view is non-owner and, therefore, needs no allocator.

```
template < class CharT, class Traits = std::char_traits<CharT> >  
class basic_string_view;
```

According to strings, there exist for string views four synonyms for the underlying character types char, wchar_t, char16_t, and char32_t.

```
typedef std::string_view std::basic_string_view<char>  
typedef std::wstring_view std::basic_string_view<wchar_t>  
typedef std::u16string_view std::basic_string_view<char16_t>  
typedef std::u32string_view std::basic_string_view<char32_t>
```

 std::string_view is the string view

If we speak in C++ about a string view, we refer with 99% probability to the specialization std::basic_string_view for the character type char. This statement is also true for this book.

```
*/
```

```
// =====  
// =====  
/*
```

Create and initialise

As always, let's start by creating a string view.

You can create an empty string view. You can also create a string view from an existing string, an existing character-array, or an existing string view.

The table below gives you an overview of the various ways of creating a string view.

Methods Example

Empty string view	std::string_view str_view
From a C-string	std::string_view str_view2("C-string")
From a string view	std::string_view str_view3(str_view2)
From a C array	std::string_view str_view4(arr, sizeof arr)
From a string_view	str_view4= str_view3.substring(2, 3)
From a string view	std::string_view str_view5 = str_view4

Methods to create and set a string view

```
*/  
// ======  
// ======  
/*
```

Non-modifying operations

In this lesson, we've listed down the non-modifying operations that can be performed on a string view.

To make this chapter concise and not repeat the detailed descriptions from the chapter on strings, I only mention the non-modifying operations of the string view. For further details, please use the link to the associated documentation in the string chapter.

Element access: operator[], at, front, back, data (see string: element access)

Capacity: size, length, max_size, empty (see string: size versus capacity)

Find: find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of (see string: search)

Copy: copy (see string: conversion between a C++ string and a C-String)

```
*/  
// ======  
// ======  
/*
```

Modifying operations

There are a few modifying operations that a string view can perform. Some of these are unique to string views only. Let's check them out.

The call `stringView.swap(stringView2)` swaps the content of the two string views. The methods `remove_prefix` and `remove_suffix` are unique to a string view because a string supports neither. `remove_prefix` shrinks its start forward; `remove_suffix` shrinks its end backward.

```
*/  
// ======  
// string_view.cpp  
  
// #include <iostream>  
// #include <string>  
// #include <experimental/string_view>  
  
// int main(){  
  
//   std::string str = " A lot of space";  
//   std::experimental::string_view strView = str;  
//   strView.remove_prefix(std::min(strView.find_first_not_of(" "), strView.size()));  
//   std::cout << "str : " << str << std::endl  
//       << "strView : " << strView << std::endl;  
  
//   std::cout << std::endl;  
  
//   char arr[] = {'A', ' ', 'l', 'o', 't', ' ', 'o', 'f', ' ', 's', ' ', 'p', ' ', 'a', ' ', 'c', ' ', 'e', '\0', '\0', '\0'};  
//   std::experimental::string_view strView2(arr, sizeof arr);
```

```
// auto trimPos = strView2.find('\0');
// if(trimPos != strView2npos) strView2.remove_suffix(strView2.size() - trimPos);
// std::cout << "arr : " << arr << ", size=" << sizeof arr << std::endl
// << "strView2: " << strView2 << ", size=" << strView2.size() << std::endl;
```

```
// }
```

```
//================================================================
```

```
/*
```

 No memory allocation with a string view

If you create a string view or copy a string view, there is no memory allocation necessary. This is in contrast to a string; creating a string or copying a string requires memory allocation.

```
*/
```

```
//================================================================
```

```
// stringView.cpp
```

```
// #include <cassert>
```

```
// #include <iostream>
```

```
// #include <string>
```

```
// #include <string_view>
```

```
// void* operator new(std::size_t count){
```

```
//   std::cout << " " << count << " bytes" << std::endl;
```

```
//   return malloc(count);
```

```
// }
```

```
// void getString(const std::string& str){}
```

```
// void getStringView(std::string_view strView){}
```

```
// int main() {
```

```
//   std::cout << std::endl;
```

```
//   std::cout << "std::string" << std::endl;
```

```
//   std::string large = "0123456789-123456789-123456789-123456789";
```

```
//   std::string substr = large.substr(10);
```

```
//   std::cout << std::endl;
```

```
//   std::cout << "std::string_view" << std::endl;
```

```
//   std::string_view largeStringView{large.c_str(), large.size()};
```

```
//   largeStringView.remove_prefix(10);
```

```
//   assert(substr == largeStringView);
```

```
//   std::cout << std::endl;
```

```
// std::cout << "getString" << std::endl;  
  
// getString(large);  
// getString("0123456789-123456789-123456789-123456789");  
// const char message []= "0123456789-123456789-123456789-123456789";  
// getString(message);  
  
// std::cout << std::endl;  
  
// std::cout << "getStringView" << std::endl;  
  
// getStringView(large);  
// getStringView("0123456789-123456789-123456789-123456789");  
// getStringView(message);  
  
// std::cout << std::endl;  
  
// }  
// ======  
/*  
💡 Memory allocation  
Thanks to the global overload operator new I can observe each memory allocation.  
*/  
// ======  
// ======  
/*
```

Introduction

Now we move on to regular expressions in C++. Our first step will be to understand the purpose of this library.

Regular expression is a language for describing text patterns. They need the header `<regex>`.

Regular expressions are a powerful tool for the following tasks:

Check if a text matches a text pattern: `std::regex_match`

Search for a text pattern in a text: `std::regex_search`

Replace a text pattern with a text: `std::regex_replace`

Iterate through all text patterns in a text: `std::regex_iterator` and `std::regex_token_iterator`

C++ supports six different grammars for regular expressions. By default, the ECMAScript grammar is used. This one is the most powerful grammar of the six grammars and is quite similar to the grammar used in Perl 5. The other five grammars are the basic, extended, awk, grep and egrep grammars.

💡 ** Use raw strings**

Use raw string literals in regular expressions. The regular expression for the text C++ is quite ugly: `C\\\" + \\\" +`. You have to use two backslashes for each + sign. First, the + sign is a special character in a regular expression. Second, the backslash is a special character in a string. Therefore one backslash escapes the + sign, the other

backslash escapes the backslash. By using a raw string literal the second backslash is not necessary anymore, because the backslash is not interpreted in the string.

```
#include <regex>
//...
std::string regExpr("C\\\\+\\\\+");
std::string regExprRaw(R"(C\+\+)");
Dealing with regular expressions is typically done in three steps:
```

I. Define the regular expression:

```
std::string text="C++ or c++.";
std::string regExpr(R"(C\+\+)");
std::regex rx(regExpr);
```

II. Store the result of the search:

```
std::smatch result;
std::regex_search(text, result, rx);
```

III. Process the result:

```
std::cout << result[0] << std::endl;
*/
// =====
// =====
/*
Character Types
```

This lesson will explain how regex identifies the different types of text in C++.

The type of the text determines the character type of the regular expression, the type of search result, and the type of action with the search result.

The following table shows the four different combinations of type of text, regular expression, search result, and action:

Text type	Regular expression type	Result type	Action type
const char*	std::regex	std::smatch	std::regex_search
std::string	std::regex	std::smatch	std::regex_search
const wchar_t*	std::wregex	std::wcmatch	std::wregex_search
std::wstring	std::wregex	std::wsmatch	std::wregex_search

Combinations of type of text, regular expression, search result and action

The program shown in the Search section of this chapter provides the four combinations in detail.

```
/*
// =====
// =====
/*
```

Regular Expression Objects

Let's take a look at the various types and grammars which C++ provides for regex objects.

Objects of type regular expression are instances of the class template template <class charT, class traits= regex_traits <charT>> class basic_regex parametrized by their character type and traits class. The traits class defines the interpretation of the properties of the regular grammar. There are two type synonyms in C++:

```
typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;
```

You can further customise the object of type regular expression. Therefore you can specify the used grammar or adapt the syntax. As said before, C++ supports the basic, extended, awk, grep and egrep grammars. A regular expression qualified by the std::regex_constants::icase flag is case insensitive. If you want to adopt the syntax, you have to specify the grammar explicitly.

```
/*
// =====
// #include <iostream>
// #include <regex>
// #include <string>

// int main()
//{
//    std::cout << std::endl;
//
//    std::string theQuestion = "C++ or c++, that's the question.";
//
//    // // regular expression for c++
//    std::string regExprStr(R"(c\+\+\+)");
//
//    // // regular expression object
//    std::regex rgx(regExprStr);
//
//    // // search result holder
//    std::smatch smatch;
//
//    std::cout << theQuestion << std::endl;
//
//    // // looking for a partial match (case sensitive)
//    if (std::regex_search(theQuestion, smatch, rgx))
//    {
//
//        std::cout << std::endl;
//        std::cout << "The answer is case sensitive: " << smatch[0] << std::endl;
//    }
//
//    // // regular expression object (case insensitive)
//    std::regex rgxIn(regExprStr, std::regex_constants::ECMAScript | std::regex_constants::icase);
//
//    // // looking for a partial match (case sensitive)
//    if (std::regex_search(theQuestion, smatch, rgxIn))
//    {
//
//        std::cout << std::endl;
//    }
//}
```

```

// std::cout << "The answer is case insensitive: " << smatch[0] << std::endl;
// }

// std::cout << std::endl;
// */
// =====
/*
If you use the case-sensitive regular expression rgx the result of the search in the text theQuestion is c++.
That's not the case if your case-insensitive regular expression rgxIn is applied. Now you get the match string
C++.
*/
// =====
// =====
/*

```

The Search Result

Whenever we verify whether a piece of text satisfies our regular expression, we have to store the results somewhere. `std::match_results` allows us to do just that.

We'll cover the following

`std::sub_match`

The object of type `std::match_results` is the result of a `std::regex_match` or `std::regex_search`.

`std::match_results` is a sequential container having at least one capture group of a `std::sub_match` object. The `std::sub_match` objects are sequences of characters.

i What is a capture group?

Capture groups allow it to further analyse the search result in a regular expression. They are defined by a pair of parentheses `()`. The regular expression `((a+)(b+)(c+))` has four capture groups: `((a+)(b+)(c+))`, `(a+)`, `(b+)` and `(c+)`. The total result is the 0th capture group.

C++ has four types of synonyms of type `std::match_results`:

```

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;

```

The search result `std::smatch` `smatch` has a powerful interface.

Some methods of the `std::smatch` interface:

Method	Description
--------	-------------

`smatch.size()` Returns the number of capture groups.

`smatch.empty()` Returns if the search result has a capture group.

`smatch[i]` Returns the *i*th capture group.

`smatch.length(i)` Returns the length of the *i*th capture group.

`smatch.position(i)` Returns the position of the *i*th capture group.

`smatch.str(i)` Returns the *i*th capture group as string.

`smatch.prefix()` and `smatch.suffix()` Returns the string before and after the capture group.

`smatch.begin()` and `smatch.end()` Returns the begin and end iterator for the capture groups.

`smatch.format(...)` Formats `std::smatch` objects for the output.

Interface of std::smatch

The following program shows the output of the first four capture groups for different regular expressions.

```
/*
// =====
// #include <regex>

// #include <iomanip>
// #include <iostream>
// #include <string>

// void showCaptureGroups(const std::string& regEx, const std::string& text){

//   // regular expression holder
//   std::regex rgx(regEx);

//   // result holder
//   std::smatch smatch;

//   // result evaluation
//   if (std::regex_search(text, smatch, rgx)){
//     std::cout << std::setw(10) << regEx << std::setw(30) << text << std::setw(30) << smatch[0] <<
//     std::setw(25) << smatch[1] << std::setw(28) << smatch[2] << std::setw(36) << smatch[3] << std::setw(30) <<
//     smatch[4] << std::endl;
//   }
// }

// int main(){

//   std::cout << std::endl;

//   std::cout << std::setw(10) << "reg Expr" << std::setw(30) << "text" << std::setw(30) << "smatch[0]" <<
//   std::setw(30) << "smatch[1]" << std::setw(30) << "smatch[2]" << std::setw(30) << "smatch[3]" << std::setw(30)
//   << "smatch[4]" << std::endl;

//   showCaptureGroups("abc+", "abcccc");

//   showCaptureGroups("(a+)(b+)(c+)", "aaabccc");

//   showCaptureGroups("((a+)(b+)(c+))", "aaabccc");

//   showCaptureGroups("(ab)(abc)+", "abababc");

//   std::cout << std::endl;

// }

// =====
/*  
std::sub_match#
```

The capture groups are of type std::sub_match. As with std::match_results C++ defines the following four type synonyms.

```
typedef sub_match<const char*> csub_match;
typedef sub_match<const wchar_t*> wcsub_match;
typedef sub_match<string::const_iterator> ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;
*/
// =====
// =====
/*
```

You can further analyze the capture group cap:

Method	Description
--------	-------------

cap.matched()	Indicates if this match was successful.
cap.first() and cap.end()	Returns the begin and end iterator of the character sequence.
cap.length()	Returns the length of the capture group.
cap.str()	Returns the capture group as string.
cap.compare(other)	Compares the current capture group with another capture group.

The `std::sub_match` object

Here is a code snippet showing the interplay between the search result std::match_results and its capture groups std::sub_match.

```
/*
// =====
// #include <iostream>
// #include <string>
// #include <regex>

// int main()
//{
//    // Simple regular expression matching
//    std::string fnames[] = {"foo.txt", "bar.txt", "baz.dat", "zoidberg"};
//    std::regex txt_regex("[a-z]+\\.txt");

//    for (const auto &fname : fnames) {
//        std::cout << fname << ":" << std::regex_match(fname, txt_regex) << '\n';
//    }

//    // Extraction of a sub-match
//    std::regex base_regex("([a-z]+)\\.txt");
//    std::smatch base_match;

//    for (const auto &fname : fnames) {
//        if (std::regex_match(fname, base_match, base_regex)) {
//            // The first sub_match is the whole string; the next
//            // sub_match is the first parenthesized expression.
//            if (base_match.size() == 2) {
//                std::ssub_match base_sub_match = base_match[1];
//                std::string base = base_sub_match.str();
//                std::cout << fname << " has a base of " << base << '\n';
//            }
//        }
//    }
}
```

```

//      }
//    }
// }

// // Extraction of several sub-matches
// std::regex pieces_regex("([a-z]+)\\([a-z]+)");
// std::smatch pieces_match;

// for (const auto &fname : fnames) {
//   if (std::regex_match(fname, pieces_match, pieces_regex)) {
//     std::cout << fname << '\n';
//     for (size_t i = 0; i < pieces_match.size(); ++i) {
//       std::smatch sub_match = pieces_match[i];
//       std::string piece = sub_match.str();
//       std::cout << " submatch " << i << ":" << piece << '\n';
//     }
//   }
// }
// =====
/*
Match

```

We learned about `std::match_results`. Now, we will look at one of the functions which allow us to send data to `match_results`.

`std::regex_match` determines if text matches a text pattern. You can further analyze the search result of type `std::match_results`.

The code snippet below shows three simple applications of `std::regex_match`: a C string, a C++ string and a range returning only a boolean. The three variants are available for `std::match_results` objects respectively.

```

*/
// =====
// #include <iostream>
// #include <regex>
// #include <string>
// #include <vector>

// int main()
//{
//   std::cout << std::endl;

//   // regular expression for a number, not including an exponent
//   std::string numberRegEx(R"([-+]?([0-9]*\.[0-9]+|[0-9]+))");

//   // regular expression holder
//   std::regex rgx(numberRegEx);

//   // using const char*
//   const char *numChar{"2011"};
//   if (std::regex_match(numChar, rgx))

```

```

// {
//   std::cout << numChar << " is a number." << std::endl;
// }

// // using std::string
// const std::string numStr{"3.14159265359"};
// if (std::regex_match(numStr, rgx))
// {
//   std::cout << numStr << " is a number." << std::endl;
// }

// // using bidirectional iterators
// const std::vector<char> numVec{'-', '2', '.', '7', '1', '8', '2', '8', '1', '8', '2', '8'};
// if (std::regex_match(numVec.begin(), numVec.end(), rgx))
// {
//   for (auto c : numVec)
//   {
//     std::cout << c;
//   }
//   std::cout << " is a number." << std::endl;
// }

// std::cout << std::endl;
// }
// =====
/*
Search

```

In this lesson, we'll see the implementation of the second look-up function for regex statements: `regex_search`.

`std::regex_search` checks if text contains a text pattern. You can use the function with and without a `std::match_results` object and apply it to a C string, a C++ string or a range.

The example below shows how to use `std::regex_search` with texts of type `const char*`, `std::string`, `const wchar_t*` and `std::wstring`.

```

*/
// =====
// #include <iostream>
// #include <regex>
// #include <string>

// int main()
//{
//   std::cout << std::endl;

//   // regular expression holder for time
//   std::regex crgx("[01]?[0-9]|2[0-3]):[0-5][0-9]");

//   // const char*
//   std::cout << "const char*" << std::endl;

```

```
// std::cmatch cmatch;

// const char *ctime{"Now it is 23:10."};

// if (std::regex_search(ctime, cmatch, crgx))
// {

//   std::cout << ctime << std::endl;
//   std::cout << "Time: " << cmatch[0] << std::endl;
// }

// std::cout << std::endl;

// // std::string
// std::cout << "std::string" << std::endl;
// std::smatch smatch;

// std::string stime{"Now it is 23:25."};
// if (std::regex_search(stime, smatch, crgx))
// {

//   std::cout << stime << std::endl;
//   std::cout << "Time: " << smatch[0] << std::endl;
// }

// std::cout << std::endl;

// // regular expression holder for time
// std::wregex wrgx(L"([01]?[0-9]|2[0-3]):[0-5][0-9]");

// // const wchar_t
// std::cout << "const wchar_t* " << std::endl;
// std::wcmatch wcmatch;

// const wchar_t *wctime{L"Now it is 23:47."};

// if (std::regex_search(wctime, wcmatch, wrgx))
// {

//   std::wcout << wctime << std::endl;
//   std::wcout << "Time: " << wcmatch[0] << std::endl;
// }

// std::cout << std::endl;

// // std::wstring
// std::cout << "std::wstring" << std::endl;
// std::wsmatch wsmatch;

// std::wstring wstime{L"Now it is 00:03."};
```

```

// if (std::regex_search(wstime, wsmatch, wrgx))
// {

// std::wcout << wstime << std::endl;
// std::wcout << "Time: " << wsmatch[0] << std::endl;
// }

// std::cout << std::endl;
//}

=====
/*
Replace
Along with searching, you can also alter the text if it matches your regex condition.

std::regex_replace replaces sequences in a text matching a text pattern. It returns in the simple form
std::regex_replace(text, regex, replString) its result as string. The function replaces an occurrence of regex in
text with replString.
*/
=====

// #include <iomanip>
// #include <iostream>
// #include <regex>
// #include <string>

// int main()
// {

// std::cout << std::endl;

// std::string future{"Future"};
// int len = sizeof(future);

// std::string unofficialStandardName{"The unofficial name of the new C++ standard is C++0x."};
// std::cout << std::setw(len) << std::left << "Past: " << unofficialStandardName << std::endl;

// // replace C++0x with C++11
// std::regex rgxCpp(R"(C\+\+0x)");
// std::string newCppName{"C++11"};

// std::string newStandardName{std::regex_replace(unofficialStandardName, rgxCpp, newCppName)};
// // replace unofficial with official
// std::regex rgxOff{"unofficial"};
// std::string makeOfficial{"official"};

// std::string officialName{std::regex_replace(newStandardName, rgxOff, makeOfficial)};
// std::cout << std::setw(len) << std::left << "Now: " << officialName << std::endl;

// std::cout << std::endl;
//}

=====
/*

```

In addition to the simple version, C++ has a version of std::regex_replace working on ranges. It enables you to push the modified string directly into another string:

```
typedef basic_regex<char> regex;
std::string str2;
std::regex_replace(std::back_inserter(str2), text.begin(), text.end(), regex,replString);
```

All variants of std::regex_replace have an additional optional parameter. If you set the parameter to std::regex_constants::format_no_copy you will get the part of the text matching the regular expression, the unmatched text is not copied. If you set the parameter to std::regex_constants::format_first_only std::regex_replace will only be applied once.

```
/*
// =====
// =====
/*
Format
```

Regular expressions can also specify the format of the target text. Find the implementation below.

std::regex_replace and std::match_results.format in combination with capture groups enables you to format text. You can use a format string together with a placeholder to insert the value.

Here are both possibilities:

```
/*
// =====
// #include <regex>

// #include <iomanip>
// #include <iostream>
// #include <string>

// int main()
// {

//   std::cout << std::endl;

//   std::string future{"Future"};
//   int len = sizeof(future);

//   const std::string unofficial{"unofficial, C++0x"};
//   const std::string official{"official, C++11"};

//   std::regex regValues{".*"), (.*)"};

//   std::string standardText{"The $1 name of the new C++ standard is $2."};

//   // using std::regex_replace
//   std::string textNow = std::regex_replace(unofficial, regValues, standardText);

//   std::cout << std::setw(len) << std::left << "Now: " << textNow << std::endl;

//   // using std::match_results
```

```

// // typedef match_results<string::const_iterator> smatch;
// std::smatch smatch;
// if (std::regex_match(official, smatch, regValues))
// {

//   std::string textFuture = smatch.format(standardText);
//   std::cout << std::setw(len) << std::left << "Future: " << textFuture << std::endl;
// }

// std::cout << std::endl;
//}
//=====
/*

```

In the function call `std::regex_replace(unofficial, regValues, standardText)` the text matching the first and second capture group of the regular expression `regValues` is extracted from the string `unofficial`. The placeholders `$1` and `$2` in the text `standardText` are then replaced by the extracted values. The strategy of `smatch.format(standardText)` is similar but there is a difference:

The creation of the search results `smatch` is separated from their usage when formatting the string.

In addition to capture groups, C++ supports additional format escape sequences. You can use them in format strings:

Format escape sequence	Description
<code>\$&</code>	Returns the total match (0th capture group).
<code>\$\$</code>	Returns <code>\$</code> .
<code>\$` (backward tic)</code>	Returns the text before the total match.
<code>\$` (forward tic)</code>	Returns the text after the total match.
<code>\$i</code>	Returns the <i>i</i> th capture group.

Format escape sequences

```

*/
//=====
//=====
/*
```

Repeated Search

We will now introduce the concept of iterators in regular expressions.

We'll cover the following

`std::regex_iterator`
`std::regex_token_iterator`

It's quite convenient to iterate with `std::regex_iterator` and `std::regex_token_iterator` over the matched texts. `std::regex_iterator` supports the matches and their capture groups. `std::regex_token_iterator` supports more. You can address the components of each capture and by using a negative index, you can access the text between the matches.

`std::regex_iterator#`
C++ defines the following four type synonyms for `std::regex_iterator`.

`typedef cregex_iterator regex_iterator<const char*>`
`typedef wcregex_iterator regex_iterator<const wchar_t*>`

```
typedef sregex_iterator regex_iterator<std::string::const_iterator>
typedef wsregex_iterator regex_iterator<std::wstring::const_iterator>
```

You can use std::regex_iterator to count the occurrences of the words in a text:

```
/*
// =====
// #include <regex>

// #include <iostream>
// #include <string>
// #include <unordered_map>

// int main(){

// std::cout << std::endl;

// // Bjarne Stroustrup about C++0x on http://www2.research.att.com/~bs/C++0xFAQ.html
// std::string text{"That's a (to me) amazingly frequent question. It may be the most frequently asked
question. Surprisingly, C++0x feels like a new language: The pieces just fit together better than they used to
and I find a higher-level style of programming more natural than before and as efficient as ever."};

// // regular expression for a word
// std::regex wordReg(R"(\w+");

// // get all words from text
// std::sregex_iterator wordItBegin(text.begin(), text.end(), wordReg);
// const std::sregex_iterator wordItEnd;

// // use unordered_map to count the words
// std::unordered_map<std::string, std::size_t> allWords;

// // count the words
// for (; wordItBegin != wordItEnd; ++wordItBegin){
//   ++allWords[wordItBegin->str()];
// }

// for ( auto wordIt: allWords) std::cout << wordIt.first << ":" << wordIt.second << "\n";

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

// }
// =====
/*
```

A word consists of at least one character (\w+). This regular expression is used to define the begin iterator wordItBegin and the end iterator wordItEnd. The iteration through the matches happens in the for loop. Each word increments the counter: ++allWords[wordItBegin->str()]. A word with counter equals to 1 is created if it is not already in allWords.

std::regex_token_iterator#

C++ defines the following four type synonyms for std::regex_token_iterator.

```
typedef cregex_iterator regex_iterator<const char*>
typedef wcregex_iterator regex_iterator<const wchar_t*>
typedef sregex_iterator regex_iterator<std::string::const_iterator>
typedef wsregex_iterator regex_iterator<std::wstring::const_iterator>
```

std::regex_token_iterator enables you by using indexes to specify which capture groups you are interested in explicitly. If you don't specify the index, you will get all capture groups, but you can also request specific capture groups using its respective index. The -1 index is special: You can use -1 to address the text between the matches:

```
/*
// =====
// #include <regex>

// #include <iostream>
// #include <string>
// #include <vector>

// int main(){

// std::cout << std::endl;

// // a few books
// std::string text{"Pete Becker, The C++ Standard Library Extensions, 2006:Nicolai Josuttis, The C++ Standard Library, 1999:Andrei Alexandrescu, Modern C++ Design, 2001"};

// // regular expression for a book
// std::regex regBook(R"((\w+)\s(\w+), ([\w\s]+)*, (\d{4}))");

// // get all books from text
// std::sregex_token_iterator bookItBegin(text.begin(), text.end(), regBook);
// const std::sregex_token_iterator bookItEnd;

// std::cout << "##### std::match_results #####" << "\n\n";
// while ( bookItBegin != bookItEnd){
//   std::cout << *bookItBegin++ << std::endl;
// }

// std::cout << "\n\n" << "##### last name, date of publication #####" << "\n\n";

// // get all last name and date of publication for the entries
// std::sregex_token_iterator bookItNameIssueBegin(text.begin(), text.end(), regBook, {2, 4});
// const std::sregex_token_iterator bookItNameIssueEnd;
// while ( bookItNameIssueBegin != bookItNameIssueEnd){
//   std::cout << *bookItNameIssueBegin++ << ", ";
//   std::cout << *bookItNameIssueBegin++ << std::endl;
// }

// // regular expression for a book, using negativ search
// std::regex regBookNeg(":");
```

```

// std::cout << "\n\n" << "##### get each entry, using negativ search #####" << "\n\n";
// // get all entries, only using ":" as regular expression
// std::sregex_token_iterator bookItNegBegin(text.begin(), text.end(), regBookNeg, -1);
// const std::sregex_token_iterator bookItNegEnd;
// while ( bookItNegBegin != bookItNegEnd){
//   std::cout << *bookItNegBegin++ << std::endl;
// }

// std::cout << std::endl;

// }
// =====
/*
bookItBegin using no indices and bookItNegbegin using the negative index returns both the total capture
group, but bookNameIssueBegin only the second and fourth capture group {{2,4}}.
*/
// =====
// =====
/*
input output stream:
Introduction
This chapter will cover all the input and output features in C++17.

```

The input and output streams enable you to communicate with the outside world. A stream is an infinite character stream on which you can push or pull data. Push is called writing, pull is called reading.

The input and output streams

were used long before the first C++ standard (C++98) in 1998,

are a for the extensibility designed framework,

are implemented according to the object-oriented and generic paradigms.

String Streams

The string stream family lets us store and manipulate strings.

We'll cover the following

Streams

String Streams

Streams#

A stream is an infinite data stream on which you can push or pull data. String streams and file streams enable strings and files to interact with the stream directly.

String Streams#

String streams need the header `<sstream>`. They are not connected to an input or output stream and store their data in a string.

Whether you use a string stream for input or output or with the character type `char` or `wchar_t` there are various string stream classes:

Class Use

`std::istringstream` and `std::wistringstream` String stream for the input of data of type `char` and `wchar_t`.
`std::ostringstream` and `std::wostringstream` String stream for the output of data of type `char` and `wchar_t`.
`std::stringstream` and `std::wstringstream` String stream for the input or output of data of type `char` and `wchar_t`

Typical operations on a string stream are:

Write data in a string stream:

```
std::stringstream os;
os << "New String";
os.str("Another new String");
```

Read data from a string stream:

```
std::stringstream os;
std::string str;
os >> str;
str= os.str();
```

Clear a string stream:

```
std::stringstream os;
os.str("");
```

String streams are often used for the type safe conversion between strings and numeric values:

```
/*
// =====
// #include <iomanip>
// #include <iostream>
// #include <sstream>
// #include <string>

// template < class T >
// T StringTo ( const std::string& source ){

//   std::istringstream iss(source);
//   T ret;
//   iss >> ret;

//   return ret;
}

// template< class T >
// std::string ToString(const T& n){

//   std::ostringstream tmp ;
//   tmp << n;
```

```

// return tmp.str();
//}

// int main(){
// std::cout << std::endl;
// std::cout << "5 = " << std::string("5") << std::endl;
// std::cout << "5 = " << StringTo<int>("5") << std::endl;
// std::cout << "5 + 6 = " << StringTo<int>("5") + 6 << std::endl;

// std::string erg(ToString(StringTo<int> ("5") + 6 ) );
// std::cout << "5 + 6 = " << erg << std::endl;

// std::cout << "5e10: " << std::fixed << StringTo<double>("5e10") << std::endl;
// std::cout << std::endl;

//}
//=====
/*

```

File Streams

Now, we shall learn how to communicate with files using C++.

We'll cover the following

Random Access

File streams enable you to work with files. They need the header `<fstream>`. The file streams automatically manage the lifetime of their file.

Whether you use a file stream for input or output or with the character type `char` or `wchar_t` there are various file stream classes:

Class Use

<code>std::ifstream</code> and <code>std::wifstream</code>	File stream for the input of data of type <code>char</code> and <code>wchar_t</code> .
<code>std::ofstream</code> and <code>std::wofstream</code>	File stream for the output of data of type <code>char</code> and <code>wchar_t</code>
<code>std::fstream</code> and <code>std::wfstream</code>	File stream for the input and output of data of type <code>char</code> and <code>wchar_t</code> .
<code>std::filebuf</code> and <code>std::wfilebuf</code>	Data buffer of type <code>char</code> and <code>wchar_t</code> .

Set the file position pointer

File streams used for reading and writing have to set the file position pointer after the contents change.

Flags enable you to set the opening mode of a file stream. Here are a few of those flags:

Flag Description

<code>std::ios::in</code>	Opens the file stream for reading (default for <code>std::ifstream</code> and <code>std::wifstream</code>).
<code>std::ios::out</code>	Opens the file stream for writing (default for <code>std::ofstream</code> and <code>std::wofstream</code>).
<code>std::ios::app</code>	Appends the character to the end of the file stream.
<code>std::ios::ate</code>	Sets the initial position of the file position pointer on the end of the file stream.
<code>std::ios::trunc</code>	Deletes the original file.
<code>std::ios::binary</code>	Suppresses the interpretation of an escape sequence in the file stream.

Flags for the opening of a file stream

It's quite easy to copy the file named in to the file named out with the file buffer in.rdbuf(). The error handling is missing in this short example.

```
#include <fstream>
...
std::ifstream in("inFile.txt");
std::ofstream out("outFile.txt");
out << in.rdbuf();
```

If you combine the C++ flags, you can compare the C++ and C modes to open a file:

C++ mode	Description	C mode
std::ios::in	Reads the file. "r"	
std::ios::out	Writes the file. "w"	
std::ios::out std::ios::app	Appends to the file. "a"	
std::ios::in std::ios::out	Reads and writes the file. "r+"	
std::ios::in std::ios::out std::ios::trunc	Writes and reads the file. "w+"	
Opening of a file with C++ and C		

The file has to exist with the mode "r" and "r+". On the contrary, the file is to be created with "a" and "w+". The file is overwritten with "w".

You can explicitly manage the lifetime of a file stream:

Flag	Description
infile.open(name)	Opens the file name for reading.
infile.open(name, flags)	Opens the file name with the flags flags for reading.
infile.close()	Closes the file name.
infile.is_open()	Checks if the file is open.
Managing the lifetime of a file stream	

Random Access#

Random access enables you to set the file position pointer arbitrarily.

When a file stream is constructed, the files position pointer points to the beginning of the file. You can adjust the position with the methods of the file stream file.

Here are the methods for navigating in a file stream:

Method	Description
file.tellg()	Returns the read position of file.
file.tellp()	Returns the write position of file.
file.seekg(pos)	Sets the read position of file to pos.
file.seekp(pos)	Sets the write position of file to pos.
file.seekg(off, rpos)	Sets the read position of file to the offset off relative to rpos.
file.seekp(off, rpos)	Sets the write position of file to the offset off relative to rpos.
Navigate in a file stream	

off has to be a number. rpos can have three values:

rpos value	Description
std::ios::beg	Position at the beginning of the file.
std::ios::cur	Position at the current position
std::ios::end	Position at the end of the file.

⚠ Respect the file boundaries

If you randomly access a file, the C++ runtime does not check the file boundaries. Reading or writing data outside the boundaries is undefined behaviour.

```
*/  
// ======  
// #include <fstream>  
// #include <iostream>  
// #include <string>  
  
// int writeFile(const std::string name){  
  
// std::ofstream outFile(name);  
  
// if (!outFile){  
//   std::cerr << "Could not open file " << name << std::endl;  
//   exit(1);  
// }  
  
// for ( unsigned int i=0; i < 10; ++i){  
//   outFile << i << "    0123456789" << std::endl;  
// }  
// }  
  
// int main(){  
  
// std::cout << std::endl;  
  
// std::string random{"random.txt"};  
  
// writeFile(random);  
  
// std::ifstream inFile(random);  
  
// if (!inFile){  
//   std::cerr << "Could not open file " << random << std::endl;  
//   exit(1);  
// }  
  
// std::string line;  
  
// std::cout << "The whole file : " << std::endl;  
// std::cout << inFile.rdbuf();  
// std::cout << "inFile.tellg(): " << inFile.tellg() << std::endl;  
  
// std::cout << std::endl;
```

```

// inFile.seekg(0);
// inFile.seekg(0, std::ios::beg); // redundant
// getline(inFile, line);
// std::cout << line << std::endl;

// inFile.seekg(20, std::ios::cur);
// getline(inFile, line);
// std::cout << line << std::endl;

// inFile.seekg(-20, std::ios::end);
// getline(inFile, line);
// std::cout << line << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

State of the Stream

Are there tools we can use in C++ which allow us to check the current condition of the stream? Flags answer this question.

Flags represent the state of the stream stream. The methods for dealing with these flags need the header `<iostream>`. Flags representing the state of a stream are:

Flag	Query of the flag	Description
<code>std::ios::goodbit</code>	<code>stream.good()</code>	No bit set
<code>std::ios::eofbit</code>	<code>stream.eof()</code>	end-of-file bit set
<code>std::ios::failbit</code>	<code>stream.fail()</code>	Error
<code>std::ios::badbit</code>	<code>stream.bad()</code>	Undefined behaviour

State of a stream

Here are examples of conditions causing the different states of a stream:

`std::ios::eofbit`:

Reading beyond the last valid character.

`std::ios::failbit`:

False formatted reading.

Reading beyond the last valid character.

Opening of a file went wrong.

`std::ios::badbit`:

Size of the stream buffer cannot be adjusted.

Code conversion of the stream buffer went wrong.

A part of the stream threw an exception.

`stream.fail()`:

returns whether `std::ios::failbit` or `std::ios::badbit` is set.

The state of a stream can be read and set.

stream.clear():

Initializes the flags and sets the stream in the goodbit state.

stream.clear(sta):

Initializes the flags and set the stream into sta state.

stream.rdstate():

Returns the current state.

stream.setstate(fla):

Sets the additional flag fla.

Operations on a stream only work if the stream is in the goodbit state. If the stream is in the badbit state you cannot set it to goodbit state.

```
/*
// =====
// #include <iostream>
// #include <iomanip>

// int main(){
//   std::cout << std::boolalpha << std::endl;
//   std::cout << "In failbit-state: " << std::cin.fail() << std::endl;
//   std::cout << std::endl;
//   int myInt;
//   while (std::cin >> myInt){
//     std::cout << "Output: " << myInt << std::endl;
//     std::cout << "In failbit-state: " << std::cin.fail() << std::endl;
//     std::cout << std::endl;
//   }
//   std::cout << "In failbit-state: " << std::endl;
//   std::cin.clear();
//   std::cout << "In failbit-state: " << std::cin.fail() << std::endl;
//   std::cout << std::endl;
// }
```

```
/*
// =====
/*
```

The input of the character a causes the stream std::cin to be in std::ios::failbit state. Therefore a and std::cin.fail() cannot be displayed. At first you have to initialize the stream std::cin.

```
/*
// =====
// =====
/*
```

User-defined Data Types

We can also set our own preferences for the input and output operators.

If you overload the input and output operators, your data type behaves like a built-in data type.

```
friend std::istream& operator>> (std::istream& in, Fraction& frac);
friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);
```

For overloading the input and output operators you have to keep a few rules in mind:

To support the chaining of input and output operations you have to get and return the input and output streams by non-constant reference.

To access the private members of the class, the input and output operators have to be friends of your data type.

The input operator >> takes its data type as a non-constant reference.

The output operator << takes its data type as a constant reference.

```
/*
// =====
// #include <iostream>

// class Fraction
//{
// public:
//   Fraction(int num = 0, int denom = 0) : numerator(num), denominator(denom) {}

//   friend std::istream &operator>>(std::istream &in, Fraction &frac);
//   friend std::ostream &operator<<(std::ostream &out, const Fraction &frac);

// private:
//   int numerator;
//   int denominator;
//};

// std::istream &operator>>(std::istream &in, Fraction &frac)
//{
//   in >> frac.numerator;
//   in >> frac.denominator;

//   return in;
//}

// std::ostream &operator<<(std::ostream &out, const Fraction &frac)
//{
//   out << frac.numerator << "/" << frac.denominator;
//   return out;
//}
```

```

// int main()
//{
// std::cout << std::endl;

// Fraction frac(3, 4);
// Fraction frac2(7, 8);
// std::cout << "frac(3, 4): " << frac << std::endl;
// std::cout << frac << " " << frac2 << std::endl;

// std::cout << std::endl;

// std::cout << "Enter two natural numbers for a Fraction: " << std::endl;
// Fraction fracDef;
// std::cin >> fracDef;
// std::cout << "fracDef: " << fracDef << std::endl;

// std::cout << std::endl;
//}
//=====
/*
Hierarchy

```

In this lesson, we will get insight into the structure of input/output streams.

Name Description

basic_streambuf<>	Reads and writes the data
ios_base	Properties of all stream classes independent on the character type
basic_ios<>	Properties of all stream classes dependent of the character type
basic_istream<>	Base for the stream classes for the reading of the data
basic_ostream<>	Base for the stream classes for the writing of the data
basic_iostream<>	Base for the stream classes for the reading and writing of the data

The class hierarchy has type synonyms for the character types `char` and `wchar_t`. Names not starting with `w` are type synonyms for `char`, names starting with `w` for `wchar_t`.

The base classes of the class `std::basic_iostream<>` are virtually derived from `std::basic_ios<>`, therefore `std::basic_iostream<>` has only one instance of `std::basic_ios`.

iostream

The most frequently used read/write stream is `iostream`. We'll look at the implementation shortly.

The stream classes `std::istream` and `std::ostream` are often used for the reading and writing of data. Use of `std::istream` classes requires the `<iostream>` header; use of `std::ostream` classes requires the `<ostream>` header. You can have both with the header `<iostream>`. `std::istream` is a `typedef` for the class `basic_istream` and the character type `char`, `std::ostream` for the class `basic_ostream` respectively:

```

typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;

```

C++ has four predefined stream objects for the convenience of dealing with the keyboard and the monitor.

Stream object C pendant	Device Buffered
std::cinstdin	keyboard yes
std::cout	stdout monitor yes
std::cerr	stderr monitor no
std::clog	monitor yes

The four predefined stream objects

i The stream objects are also available for wchar_t

The four stream objects for wchar_t std::wcin, std::wcout, std::wcerr and std::wclog are by far not so heavily used as their char pendants. Therefore I treat them only marginally.

The stream objects are sufficient to write a program that reads from the command line and returns the sum.

```
/*
// =====
// #include <iostream>

// int main()
// {

//   std::cout << std::endl;
//   std::cout << "Type in your numbers(Quit with an arbitrary character): " << std::endl;

//   int sum{0};
//   int val;

//   while (std::cin >> val)
//     sum += val;

//   std::cout << "Sum: " << sum << std::endl
//   << std::endl;
// }
// =====
/*
```

The small program above uses the stream operators << and >> and the stream manipulator std::endl.

The insert operator << pushes characters onto the output stream std::cout; the extract operator >> pulls the characters from the input stream std::cin. You can build chains of insert or extract operators because both operators return a reference to themselves.

std::endl is a stream manipulator because it puts a '\n' character onto std::cout and flushes the output buffer.

Here are the most frequently used stream manipulators.

Manipulator	Stream type	Description
std::endl	output	Inserts a new-line character and flushes the stream.
std::flush	output	Flushes the stream.
std::wsinput	Discards leading whitespace.	

The most frequently used stream manipulators

```
/*
// =====
// =====
```

```
/*
Input and Output Functions
Apart from 'cin' and 'cout', there are many other functions we can use to perform input/output operations.
```

We'll cover the following

Input

Formatted Input

Unformatted Input

Output

Input#

You can read in C++ in two way from the input stream: Formatted with the extractor `>>` and unformatted with explicit methods.

Formatted Input#

The extraction operator `>>`

is predefined for all built-in types and strings,
can be implemented for user-defined data types,
can be configured by format specifiers.

 std::cin ignores by default leading whitespace

```
#include <iostream>
//...
int a, b;
std::cout << "Two natural numbers: " << std::endl;
std::cin >> a >> b; // < 2000 11>
std::cout << "a: " << a << " b: " << b;
```

Unformatted Input#

There are many methods for the unformatted input from an input stream is:

Method	Description
--------	-------------

is.get(ch) Reads one character into ch.

is.get(buf, num) Reads at most num characters into the buffer buf.

is.getline(buf, num[, delim]) Reads at most num characters into the buffer buf. Uses optionally the line-delimiter delim (default `\n`).

is.gcount() Returns the number of characters that were last extracted from is by an unformatted operation.

is.ignore(streamsize sz= 1, int delim= end-of-file) Ignores sz characters until delim.

is.peek() Gets one characters from is without consuming it.

is.unget() Pushes the last read character back to is.

is.putback(ch) Pushes the character ch onto the stream is.

Unformatted input from an input stream

 std::string has a getline function

The getline function of std::string has a big advantage above the getline function of the istream. The std::string automatically takes care of its memory. On the contrary, you have to reserve the memory for the buffer buf in the is.get(buf, num) function.

```
*/
```

```
// =====
// #include <iostream>
```

```

// #include <iostream>
// int main()
//{
// std::cout << std::endl;

// std::string line;
// std::cout << "Write a line: " << std::endl;
// std::getline(std::cin, line);
// std::cout << line << std::endl;

// std::cout << std::endl;

// std::cout << "Write numbers, separated by;" << std::endl;
// while (std::getline(std::cin, line, ',')) {
//   std::cout << line << std::endl;
// }

// std::cout << std::endl;
// =====
/*
Output#

```

You can push characters with the insert operator `<<` onto the output stream.

The insert operator `<<`

is predefined for all built-in types and strings,
can be implemented for user-defined data types,
can be adjusted by format specifiers.

```

*/
// =====
// =====
/*

```

Format Specifier

Since we're tackling text, it would only be appropriate to have tools which can help us format our data.

Format specifiers enable you to adjust the input and output data explicitly.

i I use manipulators as format specifiers

The format specifiers are available as manipulators and flags. I only present manipulators in this book because their functionality is quite similar and manipulators are more comfortable to use.

```

*/
// =====
// #include <iostream>

// int main(){
// std::cout << std::endl;

```

```

// int num{2011};

// std::cout << num << "\n\n";

// std::cout.setf(std::ios::hex, std::ios::basefield);
// std::cout << num << std::endl;
// std::cout.setf(std::ios::dec, std::ios::basefield);
// std::cout << num << std::endl;

// std::cout << std::endl;

// std::cout << std::hex << num << std::endl;
// std::cout << std::dec << num << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

The followings tables present the important format specifiers. The format specifiers are sticky except for the field width, which is reset after each application.

The manipulators without any arguments require the header `<iostream>`, and the manipulators with arguments require the header `<iomanip>`.

Manipulator Stream type Description

`std::boolalpha` input and output Displays the boolean as a word.

`std::noboolalpha` input and output Displays the boolean as number (default).

Displaying of boolean values

Manipulator Stream type Description

`std::setw(val)` input and output Sets the field width to val.

`std::setfill(c)` output stream Sets the fill character to c (default: spaces).

Set the field width and the fill character

Manipulator Stream type Description

`std::left` output Aligns the output left.

`std::right` output Aligns the output right.

`std::internal` output Aligns the signs of numbers left, the values right.

Alignment of the text

Manipulator Stream type Description

`std::showpos` output Displays positive signs.

`std::noshowpos` output Doesn't display positive signs (default).

`std::uppercaseoutput` Uses upper case characters for numeric values (default).

`std::lowercaseoutput` Uses lower case characters for numeric values.

Positive signs and upper or lower case

Manipulator Stream type Description

`std::oct` input and output Uses natural numbers in octal format.

std::dec input and output Uses natural numbers in decimal format (default).
 std::hex input and output Uses natural numbers in hexadecimal format.
 std::showbase output Displays the numeric base.
 std::noshowbase output Doesn't display the numeric base (default).
 Display of the numeric base

There are special rules for floating point numbers:

The number of significant digits (digits after the comma) is by default 6.
 If the number of significant digits is not big enough the numbers are displayed in scientific notation.
 Leading and trailing zeros are not be displayed.
 If possible the decimal point is not be displayed.

Manipulator Stream type Description

std::setprecision(val) output Adjusts the precision of the output to val.
 std::showpoint output Displays the decimal point.
 std::noshowpoint output Doesn't display the decimal point (default).
 std::fixed output Displays the floating point number in decimal format.
 std::scientific output Displays the floating point number in scientific format.
 std::hexfloat output Displays the floating-point number in hexadecimal format.
 std::defaultfloat output Displays the floating-point number in default floating-point notation.

Floating point numbers

```

*/
// =====
// #include <iomanip>
// #include <iostream>

// int main(){

// std::cout << std::endl;

// std::cout << "std::setw, std::setfill and std::left, right and internal: " << std::endl;

// std::cout.fill('#');
// std::cout << -12345 << std::endl;
// std::cout << std::setw(10) << -12345 << std::endl;
// std::cout << std::setw(10) << std::left << -12345 << std::endl;
// std::cout << std::setw(10) << std::right << -12345 << std::endl;
// std::cout << std::setw(10) << std::internal << -12345 << std::endl;

// std::cout << std::endl;

// std::cout << "std::showpos:" << std::endl;

// std::cout << 2011 << std::endl;
// std::cout << std::showpos << 2011 << std::endl;

// std::cout << std::noshowpos << std::endl;

// std::cout << "std::uppercase: " << std::endl;
// std::cout << 12345678.9 << std::endl;
// std::cout << std::uppercase << 12345678.9 << std::endl;

```

```
// std::cout << std::nouppercase << std::endl;

// std::cout << "std::showbase and std::oct, dec and hex: " << std::endl;
// std::cout << 2011 << std::endl;
// std::cout << std::oct << 2011 << std::endl;
// std::cout << std::hex << 2011 << std::endl;

// std::cout << std::endl;

// std::cout << std::showbase;
// std::cout << std::dec << 2011 << std::endl;
// std::cout << std::oct << 2011 << std::endl;
// std::cout << std::hex << 2011 << std::endl;

// std::cout << std::dec << std::endl;

// std::cout << "std::setprecision, std::fixed and std::scientific: " << std::endl;

// std::cout << 123.456789 << std::endl;
// std::cout << std::fixed << std::endl;
// std::cout << std::setprecision(3) << 123.456789 << std::endl;
// std::cout << std::setprecision(4) << 123.456789 << std::endl;
// std::cout << std::setprecision(5) << 123.456789 << std::endl;
// std::cout << std::setprecision(6) << 123.456789 << std::endl;
// std::cout << std::setprecision(7) << 123.456789 << std::endl;
// std::cout << std::setprecision(8) << 123.456789 << std::endl;
// std::cout << std::setprecision(9) << 123.456789 << std::endl;

// std::cout << std::endl;
// std::cout << std::setprecision(6) << 123.456789 << std::endl;
// std::cout << std::scientific << std::endl;
// std::cout << std::setprecision(6) << 123.456789 << std::endl;
// std::cout << std::setprecision(3) << 123.456789 << std::endl;
// std::cout << std::setprecision(4) << 123.456789 << std::endl;
// std::cout << std::setprecision(5) << 123.456789 << std::endl;
// std::cout << std::setprecision(6) << 123.456789 << std::endl;
// std::cout << std::setprecision(7) << 123.456789 << std::endl;
// std::cout << std::setprecision(8) << 123.456789 << std::endl;
// std::cout << std::setprecision(9) << 123.456789 << std::endl;

// std::cout << std::endl;

// }
```

```
// =====
// =====
// =====
/*
```

About Templates

Let's learn about template basics and their importance in C++.

We'll cover the following

Templates

Templates#

Templates are one of the outstanding features of C++. They become more and more important with each new C++ standard. The reason is quite simple, templates provide abstraction without an abstraction performance penalty.

We have templates for classes (class templates) and functions (function templates) which are used to create concrete classes or functions:

Class and function templates are families of classes and functions respectively.

Templates play an important role in the development of generic libraries like the Standard Template Library (STL)

In the next lesson, we'll discuss who should take this course.

Function Templates

In this lesson, we'll explore function templates in detail.

We'll cover the following

Function Templates

Passing Arguments in Function Templates

Instantiation

Overloading

Function Templates#

A function template will be defined by placing the keyword template followed by type or non-type parameters in front of a concrete function. After that, you replace the concrete types or non-types with the type or non-type parameters in the function.

The keyword class or typename declares the parameters.

The name T is usually used for the first parameter.

The parameters can be used in the body of the function.

Passing Arguments in Function Templates#

In the given code snippet, we'll look at how we can call the initialized variables with our template. Look at line 2, the function arguments x and y in the function xchg must have the same type. By providing two type parameters like in line 5, the types of arguments can be different. In line 9, you see a non-type template parameter N.

```
/*
// =====
// template <typename T>
// void xchg(T& x , T& y){
// ...
```

```
// template <typename T, typename T1>
// void add(T& x, T1& y){
// ...
```

```
// template <int N>
// int nTimes(int n){
// ...
// =====
/*
Instantiation#
```

The process of substituting the template parameters for the template arguments is called template instantiation.

The compiler:

Automatically creates an instance of the function template.

Will automatically create a function template if the template parameters can be derived from the function arguments.

If the compiler cannot deduce the template arguments from the function arguments, you will have to specify them explicitly.

```
*/
// =====
// template <typename T>
// void xchg(T& x, T& y){ ...
```

```
// int a, b;
// xchg(a, b);
```

```
// template <int N>
// int nTimes(int n){ ...
```

```
// int n = 5;
// nTimes<10>(n);
// =====
/*
```

Overloading#

Function templates can be overloaded.

The following rules hold:

Templates do not support an automatic type conversion.

If a free function is better or equally as good as a function template that already exists, the free function can be used.

You can explicitly specify the type of the function template.

```
func<type>(...)
```

You can specify that you are only interested in a specific instantiation of a function template.

```
func<>(...)
```

To learn more about function templates, click [here](#).

```
*/
// =====
// =====
```

```
/*
Example 1: Templates in Functions#
*/
// =====
// templateFunctionsTemplates.cpp

// #include <iostream>
// #include <string>
// #include <vector>

// template <typename T>
// void xchg(T &x, T &y)
//{
//    T t = x;
//    x = y;
//    y = t;
//}

// template <int N>
// int nTimes(int n)
//{
//    return N * n;
//}

// int main()
//{
//    std::cout << std::endl;

//    bool t = true;
//    bool f = false;
//    std::cout << "(t, f): (" << t << ", " << f << ")" << std::endl;
//    xchg(t, f);
//    std::cout << "(t, f): (" << t << ", " << f << ")" << std::endl;

//    std::cout << std::endl;

//    int int2011 = 2011;
//    int int2014 = 2014;
//    std::cout << "(int2011, int2014): (" << int2011 << ", " << int2014 << ")" << std::endl;
//    xchg(int2011, int2014);
//    std::cout << "(int2011, int2014): (" << int2011 << ", " << int2014 << ")" << std::endl;

//    std::cout << std::endl;

//    std::string first{"first"};
//    std::string second{"second"};
//    std::cout << "(first, second): (" << first << ", " << second << ")" << std::endl;
//    xchg(first, second);
//    std::cout << "(first, second): (" << first << ", " << second << ")" << std::endl;
```

```

// std::cout << std::endl;
// std::vector<int> intVec1{1, 2, 3, 4, 5};
// std::vector<int> intVec2{5, 4, 3, 2, 1};

// std::cout << "vec1: ";
// for (auto v : intVec1)
//   std::cout << v << " ";
// std::cout << "\nvec2: ";
// for (auto v : intVec2)
//   std::cout << v << " ";
// std::cout << std::endl;
// xchg(intVec1, intVec2);

// std::cout << "vec1: ";
// for (auto v : intVec1)
//   std::cout << v << " ";
// std::cout << "\nvec2: ";
// for (auto v : intVec2)
//   std::cout << v << " ";
// std::cout << std::endl;

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

// std::cout << "nTimes<5>(10): " << nTimes<5>(10) << std::endl;
// std::cout << "nTimes<10>(5): " << nTimes<10>(5) << std::endl;

// std::cout << std::endl;
// }
// =====
/*
Explanation#
In the example above, we've declared two function templates: xchg and nTimes in lines 8 and 15. xchg swaps the values passed as arguments. The only non-type, we use is N in the function templates nTimes. nTimes returns the N times of the number passed n. We have initialized multiple instances to check for functions in lines 31 and 32, lines 39 and 40, and lines 46 and 47.
```

Example 2: Overloading Function Templates#

```

*/
// =====
// templateFunctionsTemplatesOverloading.cpp

// #include <iostream>

// void xchg(int &x, int &y)
// { // 1
//   int t = x;
//   x = y;
//   y = t;
// }

// template <typename T> // 2
```

```
// void xchg(T &x, T &y)
//{
//    T t = x;
//    x = y;
//    y = t;
//}

// template <typename T> // 3
// void xchg(T &x, T &y, T &z)
//{
//    xchg(x, y);
//    xchg(x, z);
//}

// int main()
//{
//    std::cout << std::endl;

//    int intA = 5;
//    int intB = 10;
//    int intC = 20;

//    double doubleA = 5.5;
//    double doubleB = 10.0;

//    std::cout << "Before: " << intA << ", " << intB << std::endl;
//    xchg(intA, intB); // 1
//    std::cout << "After: " << intA << ", " << intB << std::endl;

//    std::cout << std::endl;

//    std::cout << "Before: " << doubleA << ", " << doubleB << std::endl;
//    xchg(doubleA, doubleB); // 2
//    std::cout << "After: " << doubleA << ", " << doubleB << std::endl;

//    std::cout << std::endl;

//    xchg<>(intA, intB); // explicit 2
//    xchg<int>(intA, intB); // explicit 2: xchg<int>
//    // xchg<double>(intA, intB); // ERROR explicit xchg<double>

//    std::cout << "Before: " << intA << ", " << intB << ", " << intC << std::endl;
//    xchg(intA, intB, intC); // 3
//    std::cout << "After: " << intA << ", " << intB << ", " << intC << std::endl;

//    std::cout << std::endl;
//}

// =====
/*
```

Explanation#

In the above example, we used the concept of function overloading by calling xchg with different arguments passed to the function. We used the xchg function with different data types by passing two arguments and three arguments. In line 37, the non-template function is called, whereas, on all other calls to xchg(), the template function is used. The call xchg<double>(intA, intB) would be fine, when xchg would take its arguments by value.

In the next lesson, we'll solve an exercise related to function templates.

```
*/  
// ======  
// ======  
/*
```

Problem Statement#

You have to implement a function which calculates 210 in the program. Try using templates to implement the function.

```
*/  
// ======  
// power1.cpp
```

```
// #include <iostream>
```

```
// int power(int m, int n)  
// {  
//   int r = 1;  
//   for (int k = 1; k <= n; ++k)  
//     r *= m;  
//   return r;  
// }
```

```
// int main()  
// {  
//   std::cout << power(2, 10) << std::endl;  
// }  
// ======  
/*
```

Explanation#

We're using a for loop to compute the power. The loop runs a total of n times by multiplying the number m with r for every iteration of the loop in line 7.

To get a more in-depth insight into the above solution, click [here](#). It shows how things are handled at the assembler level.

The critical point of this example is that the function runs at runtime.

Solution 2: Using Template Arguments#

```
*/  
// ======  
// power2.cpp
```

```
// #include <iostream>
```

```

// template<int m, int n>
// struct Power{
//   static int const value = m * Power<m,n-1>::value;
// };

// template<int m>
// struct Power<m,0>{
//   static int const value = 1;
// };

// int main(){
//   std::cout << Power<2,10>::value << std::endl;
// }
// =====
/*
Explanation#

```

The call `Power<2, 10>::value` in line 16 triggers the recursive calculation. First, the primary template in line 5 is called, then the `Power<m, n-1>::value` in line 7 is executed. This expression instantiates recursively until the end condition is met; `n` is equal to 0. Now, the boundary condition in line 12 is applied, which returns 1. In the end, `Power<2, 10>::value` contains the result.

To view how things are happening at the assembler level, click [here](#).

The critical point is that the calculation is done at compile-time.

Solution 3: Using Template Arguments and Function Arguments#

```

*/
// =====
// power3 .cpp

// #include <iostream>

// template<int n>
// int power(int m){
//   return m * power<n-1>(m);
// }

// template<>
// int power<1>(int m){
//   return m;
// }

// template<>
// int power<0>(int m){
//   return 1;
// }

// int main(){
//   std::cout << power<10>(2) << std::endl;
// }
// =====

```

```
/*
Explanation#
```

In the above code, the power function template exists in three variations. First in the primary template in line 6. Second and third in the full specializations for 1 and 0 in lines 11, and 16. The call `power<10>(2)` triggers the recursive invocation of the primary template. The recursion ends with the full specialization for 1. When you study the example carefully, you'll see that the full specialization for 0 is not necessary in this case because the full specialization for 0 is also a valid boundary condition.

When we invoke the `power<10>(2)` function, the argument in () brackets is evaluated at runtime and the argument in <> brackets is evaluated at compile-time. Therefore, we can say that the round brackets are run time arguments and the angle brackets are compile-time arguments.

Let's have a look at the assembler code and how they are managing this. [Click here to view the code.](#)

```
/*
=====
=====
/*
```

Class Templates

In this lesson, we'll learn about the class templates.

We'll cover the following

Syntax

Explanation

Instantiation

Method Templates

Inheritance

3 Solutions:

Templates: Alias Templates

A class template defines a family of classes.

Syntax#

`template < parameter-list >`

class-declaration

Explanation#

parameter-list: A non-empty comma-separated list of the template parameters, each of which is either a non-type parameter, a type parameter, a template parameter, or a mixture of any of those.

class-declaration: A class declaration. The class name declared becomes a template name.

Instantiation#

The process of substituting the template parameters with the template arguments is called instantiation.

In contrast to a function template, a class template is not capable of automatically deriving the template parameters. Each template argument must be explicitly specified. This restriction no longer exists with C++17.

Let's have a look at the declaration of function and class templates:

Function Template Declaration

```
template <typename T>
```

```
void xchg(T& x, T&y){
```

```
...  
}
```

```
int a, b;  
xchg(a, b);  
Class Template Declaration  
template <typename T, int N>  
class Array{  
...  
};
```

```
Array<double, 10> doubleArray;  
Array<Account, 1000> accountArray;  
Method Templates#  
Method templates are function templates used in a class or class template.
```

Method templates can be defined inside or outside the class. When you define the method template outside the class, the syntax is quite complicated because you have to repeat the class template declaration and the method template declaration.

Let's have a look at the declaration of the method template inside the class and its definition outside the class:

```
template <class T, int N> class Array{  
public:  
    template <class T2>  
        Array<T, N>& operator = (const Array<T2, N>& a); ...  
};  
  
template<class T, int N>  
template<class T2>  
    Array<T, N>& Array<T, N>::operator = (const Array<T2, N>& a{  
...  
}  
*/  
// ======  
// ======  
/*  
The destructor and copy constructor cannot be templates.
```

Inheritance#

Class and class template can inherit from each other in arbitrary combinations.

If a class or a class template inherits from a class template, the methods of the base class or base class template are not automatically available in the derived class.

```
*/  
// ======  
// template <typename T>  
// struct Base{  
//     void func(){ ...  
// };
```

```

// template <typename T> struct Derived: Base<T>{
// void func2(){}
// func(); // ERROR
// }
// =====
/*

```

There are three ways to make a method from the derived class template available.

3 Solutions:#

Qualification via this pointer: this->func()

Introducing the name using Base<T>::func

Full qualified access Base<T>::func()

Templates: Alias Templates#

Alias templates, aka template typedefs, allow you to give a name to partially bound templates. An example of partial specialization from templates is given below:

```

*/
// =====
// template <typename T, int Line, int Col> class Matrix{
// ...
// };

// template <typename T, int Line>
// using Square = Matrix<T, Line, Line>;

// template <typename T, int Line>
// using Vector = Matrix<T, Line, 1>

// Matrix<int, 5, 3> ma;
// Square<double, 4> sq;
// Vector<char, 5> vec;
// =====
/*
Example 1: Templates in Class#
*/
// =====
// templateClassTemplate.cpp

```

```

// #include <iostream>

// class Account{
// public:
// explicit Account(double amount=0.0): balance(amount){}

// void deposit(double amount){
// balance+= amount;
// }

// void withdraw(double amount){
// balance-= amount;
// }

```

```

// double getBalance() const{
//   return balance;
// }

// private:
// double balance;
// };

// template <typename T, int N>
// class Array{

// public:
// Array()= default;
// int getSize() const;

// private:
// T elem[N];
// };

// template <typename T, int N>
// int Array<T,N>::getSize() const {
//   return N;
// }

// int main(){

// std::cout << std::endl;

// Array<double,10> doubleArray;
// std::cout << "doubleArray.getSize(): " << doubleArray.getSize() << std::endl;

// Array<Account,1000> accountArray;
// std::cout << "accountArray.getSize(): " << accountArray.getSize() << std::endl;

// std::cout << std::endl;
// }
// =====
/*
Explanation#

```

We have created two Array class objects, i.e., doubleArray and accountArray in lines 45 and 48. By calling generic function getSize() in line 37, we can access the size of different objects.

Example 2: Inheritance in Class Templates#

```

*/
// =====
// templateClassTemplateInheritance.cpp

// #include <iostream>

// template <typename T>
```

```

// class Base{
// public:
// void func1() const {
//   std::cout << "func1()" << std::endl;
// }
// void func2() const {
//   std::cout << "func2()" << std::endl;
// }
// void func3() const {
//   std::cout << "func3()" << std::endl;
// }
//};

// template <typename T>
// class Derived: public Base<T>{
// public:
// using Base<T>::func2;

// void callAllBaseFunctions(){
//   this->func1();
//   func2();
//   Base<T>::func3();
// }
//};

// int main(){
// std::cout << std::endl;

// Derived<int> derived;
// derived.callAllBaseFunctions();

// std::cout << std::endl;
// }
// =====
/*
Explanation#

```

We have implemented both a Base and a Derived class. Derived is publicly inherited from Base and may, therefore, use in its method callAllBaseFunctions in line 24, the methods func1, func2, and func3 from the Base class.

Make the name dependent: The call this->func1 in line 25 is dependent. The name lookup will consider in this case all base classes.

Introduce the name into the current scope: The expression using Base<T>::func2 (line 22) introduces func2 into the current scope.

Call the name fully qualified: Calling func3 fully qualified (line 27) will break a virtual dispatch and may cause new surprises.

We have created a Derived class object named derived. By using this object, we can access the base class functions by calling the method callAllBaseFunctions.

Example 3: Methods in Class Templates#

```
/*
// =====
// templateClassTemplateMethods.cpp

// #include <algorithm>
// #include <iostream>
// #include <vector>

// template <typename T, int N>
// class Array{

// public:
//     Array()= default;

//     template <typename T2>
//     Array<T, N>& operator=(const Array<T2, N>& arr){
//         elem.clear();
//         elem.insert(elem.begin(), arr.elem.begin(), arr.elem.end());
//         return *this;
//     }

//     int getSize() const;

//     std::vector<T> elem;
// };

// template <typename T, int N>
// int Array<T, N>::getSize() const {
//     return N;
// }

// int main(){

//     Array<double, 10> doubleArray{};
//     Array<int, 10> intArray{};

//     doubleArray= intArray;

//     Array<std::string, 10> strArray{};
//     Array<int, 100> bigIntArray{};

//     // // doubleArray= strArray;      // ERROR: cannot convert 'const std::basic_string<char>' to 'double'
//     // // doubleArray= bigIntArray;    // ERROR: no match for 'operator=' in 'doubleArray = bigIntArray'
// }

// /**
Explanation#
```

In the example above, we have initialized two instances of Array class namely doubleArray and intArray in lines 32 and 33. We're using the generic = operator to copy the intArray elements to doubleArray in line 35. When you look carefully, you see that the generic = is only applicable, when both arrays have the same length.

```
*/  
// ======  
// ======  
/*
```

Template Parameters

Let's familiarize ourselves with template parameters in this lesson.

We'll cover the following

Template Parameter

Types

Non-Types

Template Parameter#

Every template is parameterized by one or more template parameters, indicated in the parameter-list of the template.

C++ supports three different kinds of template parameters

Type parameter

```
std::vector<int> vec = {1, 2, 3, 4, 5};
```

Non-type parameter

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

Template-template parameter

```
template <typename T, template <typename, typename> class Cont> class Matrix{
```

...

```
Matrix<int, std::vector> myIntVec;
```

Types#

A type parameter is a typical case for template arguments.

Type parameters are class types and fundamental types

Non-Types#

Non-types are template parameters which can be evaluated at compile-time.

The following types are possible

Integers and enumerations

Pointers to objects, functions, and attributes of a class

References to objects and functions

std::nullptr_t constant

With C++17, floating-point numbers and strings cannot be used as non-type parameters.

To learn more about template parameters, click [here](#).

In the next lesson, we'll look at the examples of the three different types of template parameters.

```
/*
// =====
// =====
/*
Example 1: Type Parameter#
*/
// =====
// templateTypeParameter.cpp

// #include <iostream>
// #include <typeinfo>

// class Account{
// public:
// explicit Account(double amt): balance(amt){}
// private:
// double balance;

//};

// union WithString{
// std::string s;
// int i;
// WithString():s("hello"){}
// ~WithString(){}
//};

// template <typename T>
// class ClassTemplate{
// public:
// ClassTemplate(){
// std::cout << "typeid(T).name(): " << typeid(T).name() << std::endl;
// }
//};

// int main(){

// std::cout << std::endl;

// ClassTemplate<int> clTempInt;
// ClassTemplate<double> clTempDouble;
// ClassTemplate<std::string> clTempString;

// ClassTemplate<Account> clTempAccount;
// ClassTemplate<WithString> clTempWithString;

// std::cout << std::endl;

// }
// =====
/*
```

Explanation#

In the above code, we are identifying the type of different data types that we have passed in the parameter list. We can identify the type of variable passed to the function by using the keyword typeid in line 25. If we pass string or class type object in the parameter list, it will display the type of parameter passed along with the size of the object.

```
/*
// =====
// =====
/*
```

Example 2: Non-Type Template Parameter#

```
/*
// =====
// =====
// array.cpp
```

```
// #include <algorithm>
// #include <array>
// #include <iostream>

// int main(){

// std::cout << std::endl;

// // output the array
// std::array <int,8> array1{1,2,3,4,5,6,7,8};
// std::for_each( array1.begin(),array1.end(),[](int v){std::cout << v << " ";});

// std::cout << std::endl;

// // calculate the sum of the array by using a global variable
// int sum = 0;
// std::for_each(array1.begin(), array1.end(),[&sum](int v) { sum += v; });
// std::cout << "sum of array{1,2,3,4,5,6,7,8}: " << sum << std::endl;

// // change each array element to the second power
// std::for_each(array1.begin(), array1.end(),[](int& v) { v=v*v; });
// std::for_each( array1.begin(),array1.end(),[](int v){std::cout << v << " ";});
// std::cout << std::endl;

// std::cout << std::endl;
// }
```

```
/*
```

Explanation#

When you define an std::array in line 12, you have to specify its size. The size is a non-type template argument, which has to be specified at compile-time.

Therefore, you can output array1 in line 13 with a lambda-function [] and the range-based for-loop. By using the summation variable sum in line 19, you can sum up the elements of the std::array. The lambda-function in line 23 takes its arguments by reference and can, therefore, map each element to its square. There is nothing really special, but we are dealing with an std::array.

With C++11 we have the free function templates std::begin and std::end returning iterators for a C array. A C array is quite comfortable and safe to use with these function templates because we don't have to remember its size.

Example 3: Template-Template Parameter#

```
/*
// =====
// templateTemplateTemplatesParameter.cpp

// #include <initializer_list>
// #include <iostream>
// #include <list>
// #include <vector>

// template <typename T, template <typename, typename> class Cont >
// class Matrix{
// public:
// explicit Matrix(std::initializer_list<T> inList): data(inList){
//   for (auto d: data) std::cout << d << " ";
// }
// int getSize() const{
//   return data.size();
// }

// private:
// Cont<T, std::allocator<T>> data;

// };

// int main(){

// std::cout << std::endl;

// Matrix<int,std::vector> myIntVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// std::cout << std::endl;
// std::cout << "myIntVec.getSize(): " << myIntVec.getSize() << std::endl;

// std::cout << std::endl;

// Matrix<double,std::vector> myDoubleVec{1.1, 2.2, 3.3, 4.4, 5.5};
// std::cout << std::endl;
// std::cout << "myDoubleVec.getSize(): " << myDoubleVec.getSize() << std::endl;

// std::cout << std::endl;

// Matrix<std::string,std::list> myStringList{"one", "two", "three", "four"};
// std::cout << std::endl;
// std::cout << "myStringList.getSize(): " << myStringList.getSize() << std::endl;

// std::cout << std::endl;
// }
```

```
// =====
/*
Explanation#
We have declared a Matrix class which contains a function, i.e., getSize, and an explicit constructor that prints all entries of the passed parameter. Cont in line 8 is a template, which takes two arguments. There's no need for us to name the template parameters in the template declaration. We have to specify them in the instantiation of the template (line 19). The template used in the template parameter has exactly the signature of the sequence containers. The result is, that we can instantiate a matrix with an std::vector, or an std::list. Of course std::deque and std::forward_list would also be possible. In the end, you have a Matrix, which stores its elements in a vector or in a list.
```

If you want to study more examples for template-template parameter, you can check container adaptors.

We'll be solving a small exercise on template parameters in the next lesson.

```
/*
// =====
// =====
/*
Template Arguments
```

In this lesson, we'll learn about template arguments.

We'll cover the following

Template Arguments

Template Arguments (C++17)

Argument Deduction

Explicit Template Arguments

Default Template Arguments

Template Arguments#

Template arguments can, in general, automatically be deduced for function templates. The compiler deduces the template arguments for the function arguments. From the user's perspective, function templates feel like functions.

Conversion:

The compiler uses simple conversions for deducing the template arguments from the function arguments. The compiler removes const or volatile from the function arguments and converts C-arrays and functions to pointers.

Template argument deduction for function templates:

```
template <typename T>
void func(ParamType param);
Two datatypes were deduced:
```

T

ParamType

ParamType can be

Reference or pointer

Universal reference(&&)

Value (copy)

The parameter type is a reference or a pointer

```
template <typename T>
void func(T& param);
// void func(T* param);
func(expr);
```

T ignores reference or pointer

Pattern matching on expr for T& or T

The parameter type is a universal reference (&&)

```
template <typename T>
void func(T&& param);
func(expr);
```

expr is an lvalue: T and ParamType become lvalue references

expr is an rvalue: T is deduced such as the ParamType is a reference (case 1)

Parameter type is a value (copy)

```
template <typename T>
void func(T param);
func(expr);
```

expr is a reference: the reference (pointer) of the argument is ignored

expr is const or volatile: const or volatile is ignored

Template Arguments (C++17)

The constructor can deduce its template arguments from its function arguments.

Template Argument deduction for a constructor is available since C++17, but for function templates since C++98.

```
std::pair<int, double> myPair(2011, 1.23);
std::pair myPair(2011, 1.23);
```

Many of the make_ functions such as std::make_pair are not necessary any more:

```
auto myPair = std::make_pair(2011, 1.23);
```

Argument Deduction#

The types of function arguments have to be exact, otherwise, no conversion takes place.

```
template <typename T>
bool isSmaller(T fir, T sec){
    return fir < sec;
}
```

isSmaller(1, 5LL); // ERROR int != long long int

Providing a second template parameter makes this example work.

```
template <typename T, typename U>
bool isSmaller(T fir, U sec){
    return fir < sec;
}
isSmaller(1, 5LL); // OK
*/
// =====
```

```
// =====  
/*
```

Explicit Template Arguments#

Unlike in line 5 in the previous example, sometimes the template argument types need to be explicitly specified. This is necessary in the following cases:

Explicit Template Arguments

if the template argument cannot be deduced from the function argument.

if a specific instance of a function template is needed.

```
template <typename R, typename T, typename U>
```

```
R add(T fir, U sec){
```

```
    return fir * sec;
```

```
}
```

```
add<long long int>(1000000, 1000000LL);
```

Missing template arguments are automatically derived from the function arguments.

Default Template Arguments#

The default for template parameters can be specified for class templates and function templates. If a template parameter has a default parameter, all subsequent template parameters also need a default argument.

```
template <typename T, typename Pred = std::less<T>>
```

```
bool isSmaller(T fir, T sec, Pred pred = Pred()){
```

```
    return pred(fir, sec);
```

```
}
```

To learn more about template arguments, click [here](#).

In the next lesson, we'll take a look at the examples of template arguments.

Examples

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Example 1: Deduction of Template Arguments#

```
*/
```

```
// =====
```

```
// templateArgumentDeduction.cpp
```

```
// #include <iostream>
```

```
// template <typename T>
```

```
// bool isSmaller(T fir, T sec)
```

```
//{

```

```
//    return fir < sec;
```

```
//}

```

```
// template <typename T, typename U>
```

```
// bool isSmaller2(T fir, U sec)
```

```
//{

```

```

// return fir < sec;
//}

// template <typename R, typename T, typename U>
// R add(T fir, U sec)
//{
// return fir + sec;
//}

// int main()
//{
// std::cout << std::boolalpha << std::endl;

// std::cout << "isSmaller(1,2): " << isSmaller(1, 2) << std::endl;
// // std::cout << "isSmaller(1,5LL): " << isSmaller(1,5LL) << std::endl; // ERROR

// std::cout << "isSmaller<int>(1,5LL): " << isSmaller<int>(1, 5LL) << std::endl;
// std::cout << "isSmaller<double>(1,5LL): " << isSmaller<double>(1, 5LL) << std::endl;

// std::cout << std::endl;

// std::cout << "isSmaller2(1,5LL): " << isSmaller2(1, 5LL) << std::endl;

// std::cout << std::endl;

// std::cout << "add<long long int>(1000000,1000000): " << add<long long int>(1000000, 1000000) <<
std::endl;
// std::cout << "add<double,double>(1000000,1000000): " << add<double, double>(1000000, 1000000) <<
std::endl;
// std::cout << "add<double,double,float>(1000000,1000000): " << add<double, double, float>(1000000,
1000000) << std::endl;

// std::cout << std::endl;
//}
// =====
/*
Explanation#
In the above example, we have defined 3 function templates
```

isSmaller takes two arguments which have the same type and returns true if the first element is less than the second element (line 6). Invoking the function with arguments of different types would give a compile-time error (line 25).

isSmaller2 takes two arguments which can have a different type. The function returns true if the first element is less than the second element (line 11).

add takes two arguments which can have different types (line 16). The return type must be specified because it cannot be deduced from the function arguments.

Example 2: Template Default Arguments#

```
/*
// =====
// templateDefaultArgument.cpp
```

```

// #include <functional>
// #include <iostream>
// #include <string>

// class Account
//{
// public:
// explicit Account(double b) : balance(b){}
// double getBalance() const
// {
// return balance;
// }

// private:
// double balance;
//};

// template <typename T, typename Pred = std::less<T>>
// bool isSmaller(T fir, T sec, Pred pred = Pred())
//{
// return pred(fir, sec);
//}

// int main()
//{
// std::cout << std::boolalpha << std::endl;

// std::cout << "isSmaller(3,4): " << isSmaller(3, 4) << std::endl;
// std::cout << "isSmaller(2.14,3.14): " << isSmaller(2.14, 3.14) << std::endl;
// std::cout << "isSmaller(std::string(abc),std::string(def)): " << isSmaller(std::string("abc"), std::string("def"))
// << std::endl;

// bool resAcc = isSmaller(Account(100.0), Account(200.0), [] (const Account &fir, const Account &sec)
// {
// return fir.getBalance() < sec.getBalance(); });
// std::cout << "isSmaller(Account(100.0),Account(200.0)): " << resAcc << std::endl;

// bool acc = isSmaller(std::string("3.14"), std::string("2.14"), [] (const std::string &fir, const std::string &sec)
// {
// return std::stod(fir) < std::stod(sec); });
// std::cout << "isSmaller(std::string(3.14),std::string(2.14)): " << acc << std::endl;

// std::cout << std::endl;
//}
// =====
/*
Explanation#

```

In the first example, we have passed only the built-in data types. In this example, we have used the built-in types `int`, `double`, `std::string`, and an `Account` class in lines 26 – 28. The function template `isSmaller` is parametrized by a second template parameter, which defines the comparison criterion. The default for the comparison is the predefined function object `std::less`. A function object is a class for which the call operator

(operator ()) is overloaded. This means that instances of function objects behave similarly as a function. The Account class doesn't support the < operator. Thanks to the second template parameter, a lambda expression like in lines 30 and 33 can be used. This means Account can be compared by their balance and strings by their number. stod converts a string to a double.

Since C++17, the constructor of a class template can deduce its arguments. Study the first example of Class template argument deduction for a deeper understanding.

Example 3: Function Template Argument Deduction by Reference#

```
/*
// =====
// functionTemplateArgumentDeductionReference.cpp

// template <typename T>
// void func(T &param) {}

// template <typename T>
// void constFunc(const T &param) {}

// int main()
//{
//    int x = 2011;
//    const int cx = x;
//    const int &rx = x;

//    func(x);
//    func(cx);
//    func(rx);

//    constFunc(x);
//    constFunc(cx);
//    constFunc(rx);
//}

// =====
/*
Explanation#
In the above example, we have created two functions func and constFunc in lines 4 and 7. Both of these functions accept parameters by reference (19 – 21).
```

For better understanding, click [here](#) to analyze the process using C++ Insight.

Example 4: Function Template Argument Deduction by Universal Reference#

```
/*
// =====
// functionTemplateArgumentDeductionUniversalReference.cpp

// template <typename T>
// void funcUniversal(T&& param){}

// int main(){}
```

```

// int x = 2011;
// const int cx = x;
// const int& rx = x;

// funcUniversal(x);
// funcUniversal(cx);
// funcUniversal(rx);
// funcUniversal(2014);
// }
// =====
/*
Explanation#

```

In the above code, we have defined a function funcUniversal in line 4 which accepts its parameters with a universal reference.

For better understanding click [here](#) to analyze the process using C++ Insight.

Example 5: Function Template Argument Deduction by Value#

```

*/
// =====
// functionTemplateArgumentDeductionValue.cpp

```

```

// template <typename T>
// void funcValue(T param){}

```

```

// int main(){

```

```

// int x = 2011;
// const int cx = x;
// const int& rx = x;

```

```

// funcValue(x);
// funcValue(cx);
// funcValue(rx);
// }
// =====
/*

```

Explanation#

In the above example, we have implemented a function funcValue in line 4 which takes its parameter by value.

For better understanding click [here](#) to analyze the process using C++ Insight.

```

*/
// =====
// =====
/*

```

Problem Statement#

The class Matrix holds its values in the container Cont.

Cont should have a default argument std::vector.

Instantiate myIntVec and myDoubleVec without specifying the container explicitly.

```
/*
// =====
// templateClassTemplateMethods3.cpp

// #include <initializer_list>
// #include <iostream>
// #include <list>
// #include <vector>

// template <typename T, template <typename, typename> class Cont = std::vector>
// class Matrix{
// public:
// explicit Matrix(std::initializer_list<T> inList): data(inList){
// for (auto d: data) std::cout << d << " ";
// }
// int getSize() const{
// return data.size();
// }

// private:
// Cont<T, std::allocator<T>> data;

// };

// int main(){

// std::cout << std::endl;

// Matrix<int> myIntVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// std::cout << std::endl;
// std::cout << "myIntVec.getSize(): " << myIntVec.getSize() << std::endl;

// std::cout << std::endl;

// Matrix<double> myDoubleVec{1.1, 2.2, 3.3, 4.4, 5.5};
// std::cout << std::endl;
// std::cout << "myDoubleVec.getSize(): " << myDoubleVec.getSize() << std::endl;

// std::cout << std::endl;

// Matrix<std::string, std::list> myStringList{"one", "two", "three", "four"};
// std::cout << std::endl;
// std::cout << "myStringList.getSize(): " << myStringList.getSize() << std::endl;

// std::cout << std::endl;
// }
// =====
/*
```

Specialization

Let's learn about template specialization in this lesson.

We'll cover the following

Specialization

Primary Template

Partial Specialization

Rules for Partial Specializations:

Rules for Right Specialization:

Full Specialization

Specialization#

Template specialization addresses the need to have different code for different template argument types.

Templates define the behavior of families of classes and functions.

Often it is necessary that special types, non-types, or templates as arguments are treated as special.

You can fully specialize templates; class templates can even be partially specialized.

The methods and attributes of specialization don't have to be identical.

General or Primary templates can coexist with partially or fully specialized templates.

The compiler prefers fully specialized to partially specialized templates and partially specialized templates to primary templates.

Primary Template#

The primary template has to be declared before the partially or fully specialized templates.

If the primary template is not needed, just a declaration will suffice.

```
template <typename T, int Line, int Column> class Matrix;
```

```
template <typename T>
```

```
class Matrix<T, 3, 3>{};
```

```
template <>class Matrix<int, 3, 3>{};
```

Partial Specialization#

The partial specialization of a template is only supported for class templates and it has template arguments and template parameters.

```
template <typename T, int Line, int Column> class Matrix{};
```

```
template <typename T>
```

```
class Matrix<T, 3, 3>{};
```

```
template <int Line, int Column>
```

```
class Matrix<double, Line, Column>{};
```

```
Matrix<int, 3, 3> m1; // class Matrix<T, 3, 3>
```

```
Matrix<double, 10, 10> m2; // class Matrix<double, Line, Column>
```

```
Matrix<std::string, 4, 3> m3; // class Matrix<T, Line, Column>
```

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

```
template <typename T, int Line, int Column> class Matrix{};
```

```

template <typename T>
class Matrix<T, 3, 3>{};

template <int Line, int Column>
class Matrix<double, Line, Column>{};

Matrix<int, 3, 3> m1; // class Matrix<T, 3, 3>
Matrix<double, 10, 10> m2; // class Matrix<double, Line, Column>
Matrix<std::string, 4, 3> m3; // class Matrix<T, Line, Column>

```

```

template <typename T> struct Type{
    std::string getName() const {
        return "unknown";
    }
};

template <>
struct Type<Account>{
    std::string getName() const {
        return "Account";
    }
};

```

If you define the methods of a class template outside of the class, you have to specify the template arguments in angle brackets after the name of the class. Define the method of a fully specialized class template outside the class body without the empty template parameter list: template <>.

```

/*
// =====

// template <typename T, int Line, int Column>
// struct Matrix;

// template <>
// struct Matrix<int, 3, 3>
// {
//     int numberOfElements() const;
// };

// // template <>
// int Matrix<int, 3, 3>::numberOfElements() const
// {
//     return 3 * 3;
// }

// =====

/*
Example 1: Template Specialization#
*/
// =====
// TemplateSpecialization.cpp

// #include <iostream>

```

```
// class Account{
// public:
// explicit Account(double b): balance(b){}
// double getBalance() const {
//   return balance;
// }
// private:
// double balance;
//};

// template <typename T, int Line, int Column>
// class Matrix{
// std::string getName() const { return "Primary Template"; }
//};

// template <typename T>
// class Matrix<T,3,3>{
// std::string name{"Partial Specialization"};
//};

// template <>
// class Matrix<int,3,3>{};

// template<typename T>
// bool isSmaller(T fir, T sec){
//   return fir < sec;
// }

// template <>
// bool isSmaller<Account>(Account fir, Account sec){
//   return fir.getBalance() < sec.getBalance();
// }

// int main(){

// std::cout << std::boolalpha << std::endl;

// Matrix<double,3,4> primaryM;
// Matrix<double,3,3> partialM;
// Matrix<int,3,3> fullM;

// std::cout << "isSmaller(3,4): " << isSmaller(3,4) << std::endl;
// std::cout << "isSmaller(Account(100.0),Account(200.0)): "<< isSmaller(Account(100.0),Account(200.0)) << std::endl;

// std::cout << std::endl;

// }

// =====
/*
```

Explanation#

In the above example, we're modifying the codes that we have used in the previous lesson.

Primary template is called when we use values other than `Matrix<data_type, 3, 3>` (line 43).

Partial specialization is called when we instantiate `Matrix<data_type, 3, 3>` where `data_type` is not `int` (line 44).

Full specialization is called when we explicitly use `int` as a data type: `Matrix<int, 3, 3>` (line 45)

Full specialization of the function template `isSmaller` is only applicable for `Account` objects. This allows it to compare two `Account` objects based on their balance (line 48).

Example 2: Template Specialization External#

```
/*
// =====
// TemplateSpecializationExternal.cpp

// #include <iostream>

// template <typename T=std::string, int Line=10, int Column=Line>
// class Matrix{
// public:
// int numberOfElements() const;
// };

// template <typename T, int Line, int Column>
// int Matrix<T,Line,Column>::numberOfElements() const {
// return Line * Column;
// }

// template <typename T>
// class Matrix<T,3>{
// public:
// int numberOfElements() const;
// };

// template <typename T>
// int Matrix<T,3>::numberOfElements() const {
// return 3*3;
// }

// template <>
// class Matrix<int,3>{
// public:
// int numberOfElements() const;
// };

// int Matrix<int,3>::numberOfElements() const {
// return 3*3;
// }

// int main(){
// std::cout << std::endl;
```

```

// Matrix<double,10,5> matBigDouble;
// std::cout << "matBigDouble.numberOfElements(): " << matBigDouble.numberOfElements() << std::endl;

// // Matrix matString; // ERROR
// Matrix<> matString;
// std::cout << "matString.numberOfElements(): " << matString.numberOfElements() << std::endl;

// Matrix<float> matFloat;
// std::cout << "matFloat.numberOfElements(): " << matFloat.numberOfElements() << std::endl;

// Matrix<bool,20> matBool;
// std::cout << "matBool.numberOfElements(): " << matBool.numberOfElements() << std::endl;

// Matrix<double,3,3> matSmallDouble;
// std::cout << "matSmallDouble.numberOfElements(): " << matSmallDouble.numberOfElements() << std::endl;

// Matrix<int,3,3> matInt;
// std::cout << "matInt.numberOfElements(): " << matInt.numberOfElements() << std::endl;

// std::cout << std::endl;

// }
// =====
/*
Explanation#

```

In the above example, we have set the default value of line to 10 (line 6) and used the value for line as the default for column. The method `numberOfElements` returns the product of both numbers as a result. If we call the `Matrix` with arguments, then these passed arguments override the default. For float and string, it returns the 100 as no arguments are passed and the default arguments are used (lines 48 and 51).

Example 3: Template Specialization Full#

```

*/
// =====
// templateSpecializationFull.cpp

// #include <iostream>
// #include <string>

// template<typename T>
// T min(T fir, T sec){
//   return (fir < sec) ? fir : sec;
// }

// template<>
// bool min<bool>(bool fir, bool sec){
//   return fir & sec;
// }

// int main(){

```

```

// std::cout << std::boolalpha << std::endl;

// std::cout << "min(3.5, 4.5): " << min(3.5, 4.5) << std::endl;
// std::cout << "min<double>(3.5, 4.5): " << min<double>(3.5, 4.5) << std::endl;

// std::cout << "min(true, false): " << min(true, false) << std::endl;
// std::cout << "min<bool>(true, false): " << min<bool>(true, false) << std::endl;

// std::cout << std::endl;

// }
// =====
/*
xplanation#

```

In the above example, we have defined a full specialization for bool. The primary and the full specialization are implicitly invoked in the lines (20 and 23) and explicitly invoked in the lines (21 and 24).

Example 4: Template Specialization Type Traits#

```

*/
// =====
// templateSpecializationTypeTraits.cpp

// #include <iostream>
// #include <type_traits>

// using namespace std;

// template <typename T>
// void getPrimaryTypeCategory()
// {

// cout << boolalpha << endl;

// cout << "is_void<T>::value: " << is_void<T>::value << endl;
// cout << "is_integral<T>::value: " << is_integral<T>::value << endl;
// cout << "is_floating_point<T>::value: " << is_floating_point<T>::value << endl;
// cout << "is_array<T>::value: " << is_array<T>::value << endl;
// cout << "is_pointer<T>::value: " << is_pointer<T>::value << endl;
// cout << "is_reference<T>::value: " << is_reference<T>::value << endl;
// cout << "is_member_object_pointer<T>::value: " << is_member_object_pointer<T>::value << endl;
// cout << "is_member_function_pointer<T>::value: " << is_member_function_pointer<T>::value << endl;
// cout << "is_enum<T>::value: " << is_enum<T>::value << endl;
// cout << "is_union<T>::value: " << is_union<T>::value << endl;
// cout << "is_class<T>::value: " << is_class<T>::value << endl;
// cout << "is_function<T>::value: " << is_function<T>::value << endl;
// cout << "is_lvalue_reference<T>::value: " << is_lvalue_reference<T>::value << endl;
// cout << "is_rvalue_reference<T>::value: " << is_rvalue_reference<T>::value << endl;

// cout << endl;
// }

```

```
// int main()
//{
// getPrimaryTypeCategory<void>();
//}
//=====
/*
Explanation#
We have used the type_traits library which detects at compile-time to which primary type category void (line 13) belongs to. The primary type categories are complete and exclusive. This means each type belongs exactly to one primary type category. For example, void returns true for the type-trait std::is_void and false for all the other type categories.
```

Example 5: Template Types#

```
/*
// =====
// templateTypes.cpp

// #include <iostream>
// #include <string>

// template <typename T>
// struct Type
//{
// std::string getName() const
// {
//   return "unknown";
// }
//};

// int main()
//{
// std::cout << std::boolalpha << std::endl;

// Type<float> tFloat;
// std::cout << "tFloat.getName(): " << tFloat.getName() << std::endl;

// std::cout << std::endl;
//}
//=====
/*
Explanation#
In the above example, the method getName returns unknown for any type passed in the argument of type function (line 8). If we specialize the class template for further types, we will implement a type deduction system at compile-time. We'll look at it in the coming exercise.
```

```
/*
// =====
// =====
/*
Problem Statement#
```

The class template Type in the code below returns to each type the name unknown.

Use the class template Type as a starting point to write a type introspection system with the help of partial and full specialization.

You need to write code for int, double, an arbitrary class named Account, pointer, const, and string.

```
/*
// =====
// Template Types

// #include <iostream>
// #include <string>

// class Account{};

// template<typename T>
// struct Type{
//   std::string getName() const {
//     return "unknown";
//   }
// };

// template<typename T>
// struct Type<T*>{
//   std::string getName() const {
//     return "pointer";
//   }
// };

// template<typename T>
// struct Type<const T>{
//   std::string getName() const {
//     return "const";
//   }
// };

// template<>
// struct Type<int>{
//   std::string getName() const {
//     return "int";
//   }
// };

// template<>
// struct Type<double>{
//   std::string getName() const {
//     return "double";
//   }
// };

// template<>
// struct Type<std::string>{
```

```

// std::string getName() const {
//   return "std::string";
// }
//;

// template<>
// struct Type<Account>{
//   std::string getName() const {
//     return "Account";
//   }
// };

// int main(){
//   std::cout << std::boolalpha << std::endl;
//   Type<float> tFloat;
//   std::cout << "tFloat.getName(): " << tFloat.getName() << std::endl;

//   Type<const float> tConstFloat;
//   std::cout << "tConstFloat.getName(): " << tConstFloat.getName() << std::endl;

//   Type<float*> tFloatPointer;
//   std::cout << "tFloatPointer.getName(): " << tFloatPointer.getName() << std::endl;

//   Type<double> tDouble;
//   std::cout << "tDouble.getName(): " << tDouble.getName() << std::endl;

//   Type<std::string> tString;
//   std::cout << "tString.getName(): " << tString.getName() << std::endl;

//   Type<int> tInt;
//   std::cout << "tInt.getName(): " << tInt.getName() << std::endl;

//   Type<Account> tAccount;
//   std::cout << "tAccount.getName(): " << tAccount.getName() << std::endl;

//   std::cout << std::endl;
// }

// =====
/*
Explanation#
In the above code, we have separately defined the partial and full specialization of the class template Type. The partial and full specializations accept an int, double, Account, string, const, and Pointer. On calling each type, the relative type is returned. We have not defined the full specialization for float, so when instantiating the class template for float, it gives an unknown in response.

We have covered the basics of templates in this chapter. In the next chapter, we'll learn about the details of templates. Let's start with template instantiation in the next lesson.
*/

```

```
// =====
// =====
/*
```

Template Instantiation

In this lesson, we'll learn about template instantiation.

We'll cover the following

Template Instantiation

Implicit

Explicit

Lazy Evaluation

Template Instantiation#

Templates can be implicitly and explicitly instantiated. Implicit instantiation means automatically and explicit means manually.

Implicit#

```
std::vector<int> vec{};
bool isSmaller<double>(fir, sec);
bool isSmaller(fir, sec);
Explicit#
template class std::vector<int>;
template bool std::vector<double>::empty() const;
template bool isSmaller<double>(double, double);
template bool isSmaller(double, double);
```

Lazy Evaluation#

When a class is instantiated, only the method declarations are available.

The definition of a method is only instantiated when it is used.

It is not necessary that all methods of class templates are valid for the template arguments. You can only use the methods, which are valid for a given instantiation.

In the next lesson, we'll look at an example of template instantiation.

```
*/
// =====
// =====
/*
Example: Template Instantiation#
*/
// =====
// templateInstantiation.cpp

// #include <iostream>
// #include <vector>

// template <typename T, int N>
// class Array{

// public:
//   Array()= default;
```

```

// int getSize() const{
//   return N;
// }

// std::vector<T> elem;
// };

// template<typename T>
// bool isSmaller(T fir, T sec){
//   return fir < sec;
// }

// template class std::vector<int>;
// template bool std::vector<double>::empty() const;

// template class Array<int, 20>;
// template int Array<double, 5>::getSize() const;

// template bool isSmaller(double, double);
// template bool isSmaller<int>(int, int);

// int main(){

// std::cout << std::endl;

// std::cout << std::boolalpha << "implicit" << std::endl;

// std::cout << std::endl;

// std::vector<int> vec{};
// std::cout << "vec.size(): " << vec.size() << std::endl;

// Array<int, 10> arr;
// std::cout << "arr.getSize(): " << arr.getSize() << std::endl;

// std::cout << std::endl;

// std::cout << "isSmaller(5, 10): " << isSmaller(5,10) << std::endl;

// std::cout << "isSmaller<double>(5.5, 6.5): " << isSmaller<double>(5.5, 6.5) << std::endl;

// std::cout << std::endl;

// }

// =====
/*
Explanation#
In the above example, we have implemented a template class Array which includes a function getSize() that returns the size of the element N passed into the constructor. We have also defined a template function

```

isSmaller bool and its return type is declared explicitly which returns true if the first passed argument is less than the second argument.

Lines 24 – 31 contain explicit template instantiation. The main program contains implicit template instantiation. Line 24 is an explicit instantiation for int and line 25 is an explicit instantiation of the method getSize for double. The lines 27 and 28 are quite similar for Array. The compiler can automatically deduce the template argument for the function argument in line 30.

```
/*
// =====
// =====
/*
Problem Statement#
```

Define a class template with at least one method. This method should not be valid for all possible template arguments. Instantiate the class template for an invalid template argument. What happens, when you

don't instantiate?

implicitly instantiate?

or explicitly instantiate the method?

```
/*
// =====
// templateInstantiationInvalid.cpp
```

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

```
// template <int Nom, int Denom>
// class Rational{
// public:
//     int getFloor(){
//         return Nom / Denom;
//     }
// };
```

```
/// template int Rational<5, 0>::getFloor();
```

```
// int main(){
```

```
// std::cout << std::endl;
```

```
// Rational<5, 3> rat1;
// std::cout << "rat1.getFloor(): " << rat1.getFloor() << std::endl;
```

```
// Rational<5, 0> rat2;
// // std::cout << "rat2.getFloor(): " << rat2.getFloor() << std::endl;
```

```
// std::cout << std::endl;
```

```
// }
```

```
/*
Explanation#
```

In the above code, we have called the getFloor function for 5 and 3 in line 9, and it returns 1. To invoke the function for an invalid call, we can give the arguments 5 and 0 which gives an error.

```
/*
// =====
// =====
/*
Variadic Templates
```

Let's learn about variadic templates in detail in this lesson.

We'll cover the following

Variadic Templates

Parameter Pack

Variadic Templates#

A variadic template is a template that can has an arbitrary number of parameters.

```
template <typename ... Args>
void variadicTemplate(Args ... args){ .... }
```

Parameter Pack#

A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A function parameter pack is a function parameter that accepts zero or more function arguments.

By using the ellipse (...), Args- or args becomes a parameter pack.

Args is a template parameter pack; args is a function parameter pack.

Parameter packs can only be packed and unpacked.

If the ellipsis is left from Args, the parameter pack will be packed and if the ellipse is right from Args, the parameter pack will be unpacked.

The compiler can automatically deduce the template arguments in case of a function template.

For example, the following classes/functions in STL extensively use variadic templates. Variadic Templates are often used in the Standard Template Library:

```
sizeof-Operator, std::tuple, std::thread, std::make_unique, std::lock
The usage of parameter packs obeys a typical pattern for class templates.
```

Perform an operation on the first element of the parameter pack and recursively invoke the operation on the remaining elements.

The recursion ends after a finite number of steps.

The boundary condition is typically a fully specialized template.

```
template<>
struct Mult<>{ .... }
template<int i, int ... tail >
struct Mult<i, tail ...>{ ....
*/
// =====
// =====
/*
Example 1: Variadic Template#
*/
```

```

// =====
// templateVariadicTemplates.cpp

// #include <iostream>

// template <typename... Args>
// int printSize(Args... args){
//   return sizeof ... (args);
// }

// template<int ...>
// struct Mult;

// template<>
// struct Mult<>{
//   static const int value= 1;
// };

// template<int i, int ... tail>
// struct Mult<i, tail ...>{
//   static const int value= i * Mult<tail ...>::value;
// };

// int main(){

// std::cout << std::endl;

// std::cout << "printSize(): " << printSize() << std::endl;
// std::cout << "printSize(template,2011,true): " << printSize("template",2011,true) << std::endl;
// std::cout << "printSize(1, 2.5, 4, 5, 10): " << printSize(1, 2.5, 4, 5, 10) << std::endl;

// std::cout << std::endl;

// std::cout << "Mult<10>::value: " << Mult<10>::value << std::endl;
// std::cout << "Mult<10,10,10>::value: " << Mult<10,10,10>::value << std::endl;
// std::cout << "Mult<1,2,3,4,5>::value: " << Mult<1,2,3,4,5>::value << std::endl;

// std::cout << std::endl;

// }
// =====
/*
Explanation#
In the above example, we have used printSize function, which prints the number of elements (of any type) passed as arguments. It detects the number of elements on compile-time using the sizeof operator, and in case of an empty argument list, it returns 0.

There is a struct defined as Mult which takes arguments of integer type and return their product. If there is no argument passed, then it returns 1 which is the neutral element for multiplication. The result is stored in the value in the fully specialized template in lines 13 – 16. The partial specialization in lines 18 – 21 starts the

```

recursion, which ends with the aforementioned fully specialization for 0. The primary template in line 10 is never used and must, therefore, never be defined.

To visualize the template instantiation for the above-mentioned example click [here](#).

Example 2: Template Perfect Forwarding#

```
/*
// =====
// templatePerfectForwarding.cpp

// #include <iostream>
// #include <utility>

// template<typename T, typename ... Args>
// T createT(Args&& ... args){
//   return T(std::forward<Args>(args) ...);
// }

// struct MyStruct{
//   MyStruct(int&, double&, double&&){}
//   friend std::ostream& operator<< (std::ostream& out, const MyStruct&){
//     out << "MyStruct" << std::endl;
//     return out;
//   }
// };

// int main(){

//   std::cout << std::endl;

//   double myDouble= createT<double>();
//   std::cout << "myDouble: " << myDouble << std::endl;

//   int myInt= createT<int>(1);
//   std::cout << "myInt: " << myInt << std::endl;

//   std::string myString= createT<std::string>("My String");
//   std::cout << "myString: " << myString << std::endl;

//   MyStruct myStruct= createT<MyStruct>(myInt, myDouble, 3.14);
//   std::cout << "myStruct: " << myStruct << std::endl;

//   std::cout << std::endl;
// }
// =====
/*
```

Explanation#

In the above example, we have created a `createT` function which invokes the constructor `T` with the arguments `args`. If there is no value passed, it invokes the default constructor. The magic of the factory function `createT` is that it can invoke each constructor. Thanks to perfect forwarding, each value can be used

such as an lvalue or an rvalue; thanks to parameter packs, any number of arguments can be used. In the case of MyStruct, a constructor that requires three arguments is used.

The pattern of the function template createT is exactly the pattern, factory functions such as std::make_unique, std::make_shared, std::make_pair, or std::make_tuple use.

```
*/
// =====
// =====
/*
```

Fold Expressions

In this lesson, we'll study fold expressions.

We'll cover the following

Fold Expressions (C++17)

Two variations

Fold Expressions (C++17)

Fold expressions is a nice syntax to evaluate binary operators at compile-time. Fold expressions reduce parameter packs on binary operators.

C++11 provides support for parameter packs:

```
bool all_14(){
    return true;
}

template<typename T, typename ...Ts>
bool all_14(T t, Ts ... ts){
    return t && all_14(ts...);
}

template<typename ... Args>
bool all_17(Args ... args){
    return ( ... && args);
}

bool val == all_14(true, true, true, false)
    == all_17(true, true, true, false)
    == ((true && true)&& true)&& false
    == false;
*/
// =====
// =====
/*
```

Two variations#

The fold expression either has or does not have an initial value

The parameter pack will be processed from left or right

C++17 supports the following 32 operators in fold expressions:

```
+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && || .*= ->*
```

Operators with their default values:

Operator	Symbol	Default Value
Logical AND	&&	true
Logical OR		false
Comma operator	,	void()

For binary operators that have no default value, you have to provide an initial value. For binary operators that have a default value, you can specify an initial value.

If the ellipsis stands left of the parameter pack, the parameter pack will be processed from the left. The same holds for right. This is also true if you provide an initial value.

The following table shows the four variations and their Haskell pendants. The C++17 standard requires that fold expressions with initial value use the same binary operator op.

C++ templates support Haskells fold* variants at compile-time, i.e., foldl, foldl1, foldr and foldr1

Fold Expressions Haskell Description

... op pack	foldl op list	Processes from left with operator op
pack op ...	foldr1 op list	Processes from right with operator op
init op ... op pack	foldl op init list	Processes from left with operator op and initial value init
pack op ... op init	foldr op init list	Processes from right with operator op and initial value init

The C++ and Haskell variations differ in two points. The C++ version uses the default value as the initial value while the Haskell version uses the first element as the initial value. The C++ version processes the parameter pack at compile-time and the Haskell version, at run time.

The small code snippet shows once more the algorithm all – this time, we use true as the initial value.

```
template<typename... Args>
bool all(Args... args){
    return (true && ... && args);
}
```

To learn more about fold expressions, click here.

```
/*
// =====
// =====
/*
Example 1: Fold Expression#
```

```
// foldExpression.cpp
```

```
#include <iostream>
```

```
template<typename... Args>
bool all(Args... args) { return (... && args); }
```

```
template<typename... Args>
bool any(Args... args) { return (... || args); }
```

```

template<typename... Args>
bool none(Args... args) { return not(... || args); }

int main(){
    std::cout << std::endl;
    std::cout << std::boolalpha;

    std::cout << "all(true): " << all(true) << std::endl;
    std::cout << "any(true): " << any(true) << std::endl;
    std::cout << "none(true): " << none(true) << std::endl;

    std::cout << std::endl;
    std::cout << "all(true, true, true, false): " << all(true, true, true, false) << std::endl;
    std::cout << "any(true, true, true, false): " << any(true, true, true, false) << std::endl;
    std::cout << "none(true, true, true, false): " << none(true, true, true, false) << std::endl;

    std::cout << std::endl;
    std::cout << "all(false, false, false, false): " << all(false, false, false, false) << std::endl;
    std::cout << "any(false, false, false, false): " << any(false, false, false, false) << std::endl;
    std::cout << "none(false, false, false, false): " << none(false, false, false, false) << std::endl;

    std::cout << std::endl;
}
*/
// =====
// =====
/*
Explanation#
In the above example, we have three predicates.
```

all function returns true only if all the values passed to it are true, else false because we're using `&&` as an operator.

any function returns true if any passed value is true, else false because we're using `||` as an operator.

none function returns true only if all the passed parameters are false because we're using `||` operator with not and it will invert the result.

Example 2: String Concatenation#

```

*/
// =====
// #include <iostream>
// #include <string>

// template<typename ...Args>
// auto addLeft(Args ... args){
```

```

// return (std::string("0") + ... + args); // (((std::string("0")+"1")+"2")+"3")
//}

// template<typename ...Args>
// auto addRight(Args ... args){
//   return (args + ... + std::string("0")); // ("1"+("2"+("3" + std::string("0"))))
//}

// int main(){

//   std::cout << addLeft("1", "2", "3") << std::endl; // 0123
//   std::cout << addRight("1", "2", "3") << std::endl; // 1230
//}
// =====
/*
Explanation#
The above-mentioned example shows the difference between left and right fold. We had to start with a std::string("0") and not "0" because "0" + "1" gives an error. String concatenation requires at least one string.
*/
// =====
// =====
/*
Friends

```

Friends#

Friends of a class template have access to all members of the class template.

A class or a class template can have a friendship to class or class templates, function or function templates, and types.

Rules:

The declaration of friends can be made at an arbitrary place in the class declaration.

The access rights in the class have no influence.

Friendship will not be inherited.

Friendship is not transitive.

A friend has unrestricted access to the members of the class.

General Friends#

A class or a class template can grant friendship to each instance of a class template or a function template.

```
template <typename T> int myFriendFunction(T);
```

```
template <typename T> class MyFriend;
```

```
template <typename T>
class GrantingFriendshipAsClassTemplate{
    template <typename U> friend int myFriendFunction(U);
    template <typename U> friend class MyFriend;
    ...
}
```

```
/*When a class template grants friendship to a template, the typename of the class template should be  
different from the typename of the template. If both use the same name, the friendship is only granted for  
the same types.*/  
/*  
Special Friends#  
A special friendship is a friendship that depends on the type of the template parameter.  
*/  
=====// template <typename T> int myFriendFunction(T);  
=====// template <typename T> class MyFriend;  
  
=====// template <typename T>  
=====// class GrantingFriendshipAsClassTemplate{  
=====// friend int myFriendFunction<>(double);  
=====// friend class MyFriend<int>;  
=====// friend class MyFriend<T>;  
=====// ======/*  
If the name of the template parameter is identical to the name of the template parameter granting the  
friendship, the friendship will be between instances of the same type.
```

Friend to Types#

A class template can grant its friendship to a type parameter.

```
template <typename T>  
class Array{  
    friend T;  
    ...  
};  
Array<Account> myAccount;  
To know more about friends, click here.  
*/  
=====// ======/*  
- Examples  
In this lesson, we'll look at a few examples of using templates with friends.
```

Example 1: Class Template General Friendship#

```
/*  
=====// templateClassTemplateGeneralFriendship.cpp  
  
// #include <iostream>  
  
// template <typename T> void myFriendFunction(T);  
// template <typename U> class MyFriend;  
  
// class GrantingFriendshipAsClass{  
  
//     template <typename U> friend void myFriendFunction(U);
```

```
// template <typename U> friend class MyFriend;  
  
// private:  
// std::string secret{"My secret from GrantingFriendshipAsClass."};  
  
// };  
  
// template <typename T>  
// class GrantingFriendshipAsClassTemplate{  
  
// template <typename U> friend void myFriendFunction(U);  
// template <typename U> friend class MyFriend;  
  
// private:  
// std::string secret{"My secret from GrantingFriendshipAsClassTemplate."};  
  
// };  
  
// template <typename T>  
// void myFriendFunction(T){  
// GrantingFriendshipAsClass myFriend;  
// std::cout << myFriend.secret << std::endl;  
  
// GrantingFriendshipAsClassTemplate<double> myFriend1;  
// std::cout << myFriend1.secret << std::endl;  
// }  
  
// template <typename T>  
// class MyFriend{  
// public:  
// MyFriend(){  
// GrantingFriendshipAsClass myFriend;  
// std::cout << myFriend.secret << std::endl;  
  
// GrantingFriendshipAsClassTemplate<T> myFriend1;  
// std::cout << myFriend1.secret << std::endl;  
// }  
// };  
  
// int main(){  
  
// std::cout << std::endl;  
  
// int a{2011};  
// myFriendFunction(a);  
  
// MyFriend<double> myFriend;  
  
// std::cout << std::endl;  
  
// }
```

```
// =====
/*
Explanation#
In the above example, we have created a function myFriendFunction and a class MyFriend. We have defined two classes: GrantingFriendshipAsClass and GrantingFriendshipAsClassTemplate. As the name mentioned as well, we are using one class with template and one without a template. The class MyFriend and the function myFriendFunction have access to the private members of the other classes by using a friend keyword. We have defined a private variable secret which is of a string type and can be called with the object of myFriendFunction and MyFriend.
```

Example 2: Class Template Special Friendship#

```
/*
// =====
// templateClassTemplateSpecialFriendship.cpp

// #include <iostream>

// template <typename T> void myFriendFunction(T);
// template <typename U> class MyFriend;

// class GrantingFriendshipAsClass{

//   friend void myFriendFunction<>(int);
//   friend class MyFriend<int>;

// private:
//   std::string secret{"My secret from GrantingFriendshipAsClass."};

// };

// template <typename T>
// class GrantingFriendshipAsClassTemplate{

//   friend void myFriendFunction<>(int);
//   friend class MyFriend<int>;
//   friend class MyFriend<T>;

// private:
//   std::string secret{"My secret from GrantingFriendshipAsClassTemplate."};

// };

// template <typename T>
// void myFriendFunction(T){
//   GrantingFriendshipAsClass myFriend;
//   std::cout << myFriend.secret << std::endl;

//   GrantingFriendshipAsClassTemplate<T> myFriend1;
//   std::cout << myFriend1.secret << std::endl;
// }
```

```

// template <typename T>
// class MyFriend{
// public:
// MyFriend(){}
// GrantingFriendshipAsClass myFriend;
// std::cout << myFriend.secret << std::endl;

// GrantingFriendshipAsClassTemplate<int> myFriendInt;
// std::cout << myFriendInt.secret << std::endl;

// GrantingFriendshipAsClassTemplate<T> myFriendT;
// std::cout << myFriendT.secret << std::endl;
// }
//};

// int main(){

// std::cout << std::endl;

// int a{2011};
// myFriendFunction(a);

// MyFriend<int> myFriend;

// std::cout << std::endl;

// }
//=====
/*
Explanation#

```

This example is similar to example 1 with a small change; we have explicitly stated the type of class template to int. Now, the class template is called for int and also for any other type mentioned in the typename portion.

Example 3: Class Template Type Friendship#

```

*/
//=====
// templateClassTemplateTypeFriendship.cpp

// #include <iostream>

// template <typename T>
// class Bank
// {
// std::string secret{"Import secret from the bank."};
// friend T;
// };

// class Account
// {

```

```

// public:
// Account()
// {
//   Bank<Account> bank;
//   std::cout << bank.secret << std::endl;
// }
//};

// int main()
//{
// std::cout << std::endl;
// Account acc;
// std::cout << std::endl;
//}

// =====
/*
Explanation#

```

In the above code, we have created an Account class which contains the Bank class object. We can access the Bank class member secret with the help of friend. Now, the value stored in the secret is accessible in the Account class.

```

*/
// =====
// =====
/*

```

Dependent Names

In this lesson, we'll study dependent names.

We'll cover the following

Dependent Names

Two-phase name lookup

The Dependent Name is a Type typename

The Dependent Name is a Template .template

Dependent Names#

A dependent name is essentially a name that depends on a template parameter. A dependent name can be a type, a non-type, or a template-template parameter.

If you use a dependent name in a template declaration or template definition, the compiler has no idea, whether this name refers to a type, a non-type, or a template parameter. In this case, the compiler assumes that the dependent name refers to a non-type, which may be wrong.

Let's have a look at the example of dependent names:

```

*/
// =====
// template<typename T>
// struct X : B<T> // "B<T>" is dependent on T
//{
//   typename T::A* pa; // "T::A" is dependent on T
}
```

```

// void f(B<T>* pb) {
//     static int i = B<T>::i; // "B<T>::i" is dependent on T
//     pb->j++; // "pb->j" is dependent on T
// }
// =====
/*

```

T is the template parameter. The names B<T>, T::A, B<T>, B<T>::i, and pb->j are dependent.

Two-phase name lookup#

Dependent names are resolved during template instantiation.

Non-dependent names are resolved during template definition.

A from a template parameter T is dependent, qualified name T::A can be a

Type

Non-type

Template

The compiler assumes by default that T::A is a non-type.

The compiler has to be convinced that T::A is a type or a template.

The Dependent Name is a Type typename#

```

template <typename T> void test(){
    std::vector<T>::const_iterator* p1;      // ERROR
    typename std::vector<T>::const_iterator* p2; //OK
}

```

Without typename like in line 3, the expression in line 2 would be interpreted as multiplication.

The Dependent Name is a Template .template#

```

*/
// =====
// template<typename T>
// struct S{
//     template <typename U> void func(){}
// }

// template<typename T>
// void func2(){
//     S<T> s;
//     s.func<T>();      // ERROR
//     s.template func<T>(); // OK
// }
// =====
/*

```

Compare lines 9 and 10. When the compiler reads the name s.func (line 9), it decides to interpret it as non-type. This means, the < sign stands in this case for the comparison operator but not opening square bracket of

the template argument of the generic method func. To help the parser, you have to specify that s.func is a template like in line 10: s.template func.

To learn more about dependent names, click [here](#).

```
/*
// =====
// =====
/*
Example: Template Lookup#
*/
// =====
// templateLookup.cpp

// #include <iostream>

// void g(double) { std::cout << "g(double)\n"; }

// template <class T>
// struct S
//{
// void f() const
//{
// g(1); // non-dependent
//}
//}
//;

// void g(int) { std::cout << "g(int)\n"; }

// int main()
//{
// g(1); // calls g(int)

// S<int> s;
// s.f(); // calls g(double)
//}
// =====
/*
Explanation#
If we access the defined functions g with double or int type object, they work fine. We have created the struct object S of int type in line 19. When we try to access the g function then it follows the same order and calls, the g with a double type parameter is defined first. The call to g() on line 17 calls the g(int) version and the call to g() through the call to f() on line 20 calls g(double).
```

In this chapter, we have learned about the details of templates. In the next chapter, we'll familiarize ourselves with the techniques used in templates. Let's start with Automatic Return type in the next

```
/*
// =====
// =====
/*
utomatic Return Type
In this lesson, we'll look at the technique that deduces return type automatically.
```

We'll cover the following

Automatic Return Type

Automatic Return Type: C++14

Automatic Return Type#

A function template is automatically able to deduce their return type.

```
template <typename T1, typename T2>
auto add(T1 fir, T2 sec) -> decltype( fir + sec ) {
    return fir + sec;
}
```

The automatic return type deduction is typically used for function templates but can also be applied to non-template functions.

Rules:

auto: introduces the syntax for the delayed return type

auto: auto type deduction is based on the function template argument deduction. Function template argument deduction (decays). So it means auto does not return the exact type but a decayed type such as for template argument deduction

decltype: declares the return type

The alternative function syntax is obligatory

The C++11 syntax for automatically deducing the return type breaks the crucial principle of software development: DRY. DRY stands for Don't Repeat Yourself.

Automatic Return Type: C++14#

A function template is automatically able to deduce their return type.

```
template <typename T1, typename T2>
auto add(T1 fir, T2 sec){
    return fir + sec;
}
```

Rules

auto: introduces the syntax for the delayed return type

decltype: declares the return type

The alternative function syntax is obligatory.

With the expression decltype(auto), auto uses the same rules to determine the type as decltype. This means, in particular, no decay takes place.

Both declarations are identical.

```
decltype(expr) v= expr;
decltype(auto) v= expr;
```

The syntax also applies for the automatic return type of a function template.

```
template <typename T1, typename T2>
decltype(auto) add(T1 fir, T2 sec){
```

```
    return fir + sec;
```

```
}
```

When a function template has more than one return statements, all return statements must have the same type.

In the next lesson, we'll study an example of automatic return type deduction.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Example: Automatic Template Return Type#

```
*/
```

```
// =====
```

```
// templateAutomaticReturnType.cpp
```

```
// #include <iostream>
```

```
// #include <typeinfo>
```

```
// template<typename T1, typename T2>
```

```
// auto add(T1 first, T2 second) -> decltype(first + second){
```

```
//   return first + second;
```

```
// }
```

```
// int main(){
```

```
//   std::cout << std::endl;
```

```
//   std::cout << "add(1, 1)= " << add(1,1) << std::endl;
```

```
//   std::cout << "typeid(add(1, 1)).name()= " << typeid(add(1, 1)).name() << std::endl;
```

```
//   std::cout << std::endl;
```

```
//   std::cout << "add(1, 2.1)= " << add(1,2.1) << std::endl;
```

```
//   std::cout << "typeid(add(1, 2.1)).name()= " << typeid(add(1, 2.1)).name() << std::endl;
```

```
//   std::cout << std::endl;
```

```
//   std::cout << "add(1000LL, 5)= " << add(1000LL,5) << std::endl;
```

```
//   std::cout << "typeid(add(1000LL, 5)).name()= " << typeid(add(1000LL, 5)).name() << std::endl;
```

```
//   std::cout << std::endl;
```

```
// }
```

```
// =====
```

```
/*
```

Explanation#

The example has a function add which takes two arguments and returns their sum. The return type of the function is deduced by the compiler by applying the decltype operator on the sum of the arguments. The expression typeid(add(1, 2.1)).name() such as in line 21 returns a string representation of the type of result.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

```
Problem Statement#
```

```
The two algorithms gcdConditional and gcdCommon, in the given code, use the type-trait library.
```

Study both algorithms.

Both algorithms determine their return type at compile-time.

Why is it not possible to use auto or decltype(auto) to automatically deduce the return type?

```
*/
```

```
// =====
```

```
// gcdVariation.cpp
```

```
// #include <iostream>
// #include <type_traits>
// #include <typeinfo>

// template<typename T1, typename T2,
//           typename R = typename std::conditional <(sizeof(T1) < sizeof(T2)), T1, T2>::type>
// R gcdConditional(T1 a, T2 b){
//   static_assert(std::is_integral<T1>::value, "T1 should be an integral type!");
//   static_assert(std::is_integral<T2>::value, "T2 should be an integral type!");
//   if( b == 0 ){ return a; }
//   else{
//     return gcdConditional(b, a % b);
//   }
// }

// template<typename T1, typename T2,
//           typename R = typename std::common_type<T1, T2>::type>
// R gcdCommon(T1 a, T2 b){
//   static_assert(std::is_integral<T1>::value, "T1 should be an integral type!");
//   static_assert(std::is_integral<T2>::value, "T2 should be an integral type!");
//   if( b == 0 ){ return a; }
//   else{
//     return gcdCommon(b, a % b);
//   }
// }

// int main(){

// std::cout << std::endl;

// std::cout << "gcdConditional(100, 10LL) = " << gcdConditional(100, 10LL) << std::endl;
// std::cout << "gcdCommon(100, 10LL) = " << gcdCommon(100, 10LL) << std::endl;

// std::conditional <(sizeof(int) < sizeof(long long)), int, long long>::type gcd1 = gcdConditional(100, 10LL);
// auto gcd2 = gcdCommon(100, 10LL);

// std::cout << std::endl;

// std::cout << "typeid(gcd1).name() = " << typeid(gcd1).name() << std::endl;
// std::cout << "typeid(gcd2).name() = " << typeid(gcd2).name() << std::endl;
```

```
// std::cout << std::endl;  
//  
// ======  
/*  
Explanation#
```

In the above code, we have defined an automatic return type R which returns data based on the data type passed in the function gcdCommon and gcdConditional.

Automatic return type deduction can't be used in both algorithms, because both algorithms solve their job by recursion (lines 24 and 26). The critical observation is that this recursion swaps their arguments. For example, gcdCommon(a, b) invokes gcdCommon(b, a % b). This recursion, therefore, creates a function with different return types. Having a function with different return types is not valid.

```
*/  
// ======  
// ======  
/*
```

Template Metaprogramming

In this lesson, we'll learn about template metaprogramming.

We'll cover the following

Template Metaprogramming

How this all started:

Calculating at Compile-Time

Type Manipulations

Explanation

Metadata and Metafunctions

Functions vs Meta Functions

Pure Functional Sublanguage

Template Metaprogramming#

How this all started:#

1994 Erwin Unruh discovered template metaprogramming by accident.

His program failed to compile but calculated the first 30 prime numbers at compile-time.

To prove his point, he used the error messages to display the first 30 prime numbers.

Let's have a look at the screenshot of the error:

We have highlighted the important parts in red. We hope you can see the pattern. The program calculates at compile-time the first 30 prime numbers. This means template instantiation can be used to do math at compile-time. It gets even better. Template metaprogramming is Turing-complete and can, therefore, be used to solve any computational problem. Of course, Turing-completeness holds only in theory for template metaprogramming because the recursion depth (at least 1024 with C++11) and the length of the names which are generated during template instantiation provide some limitations.

Calculating at Compile-Time#

The Factorial program is the Hello World of template metaprogramming.

```
template <int N>  
struct Factorial{  
    static int const value= N * Factorial<N-1>::value;
```

```
};

template <> struct Factorial<1>{
    static int const value = 1;
};

std::cout << Factorial<5>::value << std::endl;
std::cout << 120 << std::endl;
```

The call Factorial<5>::value in line 10 causes the instantiation of the primary or general template in line 3. During this instantiation, Factorial<4>::value will be instantiated. This recursion will end if the fully specialized class template Factorial<1> (line 6) kicks in as the boundary condition. Maybe, you like it more pictorial.

The following picture shows this process.

```
*/
// =====
// =====
/*
Assembler Instructions
```

From the assemblers point of view, the Factorial<5>::value boils down to the constant 0x78, which is 120.

```
mov 0x78, %esi
mov 0x601060, %edi
...
mov 0x78, %esi
mov 0x601060, %edi
...
Type Manipulations#
Manipulating types at compile-time is typically for template metaprogramming.
*/
// =====
// template <typename T>
// struct RemoveConst{
//   typedef T type;
// };

// template <typename T>
// struct RemoveConst<const T>{
//   typedef T type;
// };

// int main(){
//   std::is_same<int, RemoveConst<int>::type>::value;    // true
//   std::is_same<int, RemoveConst<const int>::type>::value; // true
// }
```

Explanation#

In the code, we have defined the class template removeConst in two versions. We have implemented removeConst the way std::remove_const is probably implemented in the type-trait library.

std::is_same from the type-trait library helps us to decide at compile-time if both types are the same. In case of removeConst<int>, the first or general class template kicks in; in case of removeConst<const int>, the partial specialization for const T applies. The key observation is that both class templates return the underlying type in lines 3 and 8, therefore, the constness is removed.

This kind of technique, which is heavily used in the type-trait library, is a compile-time if on types.

To jump into more details of type traits click [here](#).

Metadata and Metafunctions#

At compile-time, we speak about metadata and metafunctions instead of data and functions.

Metadata: Types and integral types that are used in metafunctions.

Metafunctions: Functions that are executed at compile-time. Class templates are used to implement metafunctions.

Return their value by ::value.

```
template <>
struct Factorial<1>{
    static int const value = 1;
};
```

Return their type by ::type.

```
template <typename T>
struct RemoveConst<const T>{
    typedef T type;
};
```

Functions vs Meta Functions#

From the conceptual view, it helps a lot to compare functions and metafunctions.

Characteristics	Functions	Metafunctions
Call	power(2,10)	Power<2,10>::value
Execution Time	Runtime	Compile-time
Arguments	Function arguments	Template arguments
Arguments and return value	Arbitrary values	Types, non-types and templates
Implementation	Callable	Class template
Data	Mutable	Immutable
Modification	Data can be modified	New data are created
State	Has state	Has no state

What does the table above mean for a concrete function and a concrete metafunction?

Function

```
int power(int m, int n){
    int r = 1;
    for(int k=1; k<=n; ++k){
        r *= m;
    }
```

```
    return r;  
}
```

Metaprogramming

```
template<int m, int n>  
struct Power{  
    static int const value = m * Power<m, n-1>::value;  
};
```

```
template<int m>  
struct Power<m, 0>{  
    static int const value = 1;  
};
```

Function arguments go into round () braces and template arguments go into sharp <> braces.

```
int main(){  
    std::cout << power(2, 10) << std::endl;      // 1024  
    std::cout << Power<2, 10>::value << std::endl; // 1024  
}
```

Pure Functional Sublanguage#

Template metaprogramming is

an embedded pure functional language in the imperative language C++.

Turing-complete. Turing-complete means, that all can be calculated what is calculable.

an intellectual playground for C++ experts.

the foundation for many boost libraries.

The template recursion depth is limited.

C++03: 17

C++11: 1024

In the next lesson, we'll look at a few examples of template metaprogramming.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Example 1: Template Prime Number#

```
*/
```

```
// =====
```

```
// templatePrimeNumber.cpp
```

```
// Prime number computation by Erwin Unruh
```

```
// template <int i>
```

```
// struct D
```

```
// {
```

```
// D(void *);
```

```
// operator int();
```

```
// };
```

```
// template <int p, int i>
```

```
// struct is_prime
```

```
// {
```

```
// enum
// {
//   prim = (p == 2) || (p % i) && is_prime<(i > 2 ? p : 0), i - 1>::prim
// };
// };

// template <int i>
// struct Prime_print
//{
//   Prime_print<i - 1> a;
//   enum
//   {
//     prim = is_prime<i, i - 1>::prim
//   };
//   void f()
//   {
//     D<i> d = prim ? 1 : 0;
//     a.f();
//   }
//};

// template <>
// struct is_prime<0, 0>
//{
//   enum
//   {
//     prim = 1
//   };
//};

// template <>
// struct is_prime<0, 1>
//{
//   enum
//   {
//     prim = 1
//   };
//};

// template <>
// struct Prime_print<1>
//{
//   enum
//   {
//     prim = 0
//   };
//   void f() { D<1> d = prim ? 1 : 0; }
//};

// #ifndef LAST
// #define LAST 18
// #endif
```

```
// int main()
//{
// Prime_print<LAST> a;
// a.f();
//}
//=====================================================================
/*
```

Explanation#

This is the original prime number program by Erwin Unruh, which was the starting point of template metaprogramming. Current compilers will not produce the same output as the ancient compiler, which Erwin Unruh used more than 20 years ago.

Example 2: Template Type Manipulation#

```
/*
//=====================================================================
// templateTypeManipulation.cpp

// #include <iostream>
// #include <type_traits>

// template <typename T>
// struct RemoveConst{
//   typedef T type;
// };

// template <typename T>
// struct RemoveConst<const T>{
//   typedef T type;
// };

// int main(){

//   std::cout << std::boolalpha << std::endl;

//   std::cout << "std::is_same<int, RemoveConst<int>::type>::value: " << std::is_same<int,
// RemoveConst<int>::type>::value << std::endl;
//   std::cout << "std::is_same<int, RemoveConst<const int>::type>::value: " << std::is_same<int,
// RemoveConst<const int>::type>::value << std::endl;

//   std::cout << std::endl;

// }
```

```
//=====================================================================
/*
```

Explanation#

The code uses the function `std::is_same` from the type-trait library. `std::is_same` compares the type passed and returns at compile time if they are the same. Thanks to the type-trait function, we can verify the `RemoveConst` class template from the previous subsection.

Example 3: Template Power#

```

*/
// =====
// templatePower.cpp

// #include <iostream>

// int power(int m, int n){
//     int r = 1;
//     for(int k=1; k<=n; ++k) r*= m;
//     return r;
// }

// template<int m, int n>
// struct Power{
//     static const int value = Power<m,n-1>::value * m;
// };

// template<int m>
// struct Power<m,0>{
//     static const int value = 1;
// };

// template<int n>
// int power2(const int& m){
//     return power2<n-1>(m) * m;
// }

// template<>
// int power2<1>(const int& m){
//     return m;
// }

// template<>
// int power2<0>(const int&){
//     return 1;
// }

// int main(){

//     std::cout << std::endl;

//     std::cout << "power(2,10):    " << power(2,10) << std::endl;
//     std::cout << "power2<10>(2):    " << power2<10>(2) << std::endl;
//     std::cout << "Power<2,10>::value: " << Power<2,10>::value << std::endl;

//     std::cout << std::endl;
// }
// =====
/*
Explanation#
The program calculates 210 in three different variants.

```

power is a function in line 5
Power is a class template in line 12
power2 is a function template in line 22
The key question is: When is the function executed?

power runs at runtime
Power runs at compile-time
power2 runs at runtime and at compile-time too
the template argument is evaluated at compile-time
the function argument is evaluated at runtime
*/
// ======
// ======
/*

Type-Traits Overview

In this lesson, we'll study the type traits library and its goals along with type-checks. This section could only provide an overview of the many functions of the type-trait library.

We'll cover the following

Type-Traits Library

Type-Traits: Goals

Optimization

Correctness

Type Checks

Type-Traits Library#

Type-trait enable type checks, type comparisons, and type modifications at compile-time.

Below are some applications of template metaprogramming:

Programming at compile-time

Programming with types and values

Compiler translates the templates and transforms it in C++ source code

We need to add a type_traits library in the header to enable all the functions present in the library.

#include <type_traits>

Type-Traits: Goals#

If you look carefully, you'll see that type-trait have a significant optimization potential. In the first step, type-trait help to analyze the code at compile-time and in the second step, to optimize the code based on that analysis. How is that possible? Depending on the type of variable, a faster variant of an algorithm will be chosen.

Optimization#

Code that optimizes itself. Depending on the type of a variable another code will be chosen.

Optimized version of std::copy, std::fill, or std::equal is used so that algorithms can work on memory blocks. The optimized version of operations happens on all the elements in a container in one step and not on each element individually.

Correctness#

Type checks will be performed at compile-time.

Type information, together with static_assert, defines the requirements for the code.

With the concepts in C++20, the correctness aspect of the type-trait becomes less important.

Type Checks#

C++ has 14 primary type categories. They are complete and orthogonal. This means, that each type is a member of exactly one type category. The check for the type categories is independent of the type qualifiers const or volatile.

Let's have a look at these categories syntactically:

```
/*
// =====
// template <class T> struct is_void;
// template <class T> struct is_integral;
// template <class T> struct is_floating_point;
// template <class T> struct is_array;
// template <class T> struct is_pointer;
// template <class T> struct is_reference;
// template <class T> struct is_member_object_pointer;
// template <class T> struct is_member_function_pointer;
// template <class T> struct is_enum;
// template <class T> struct is_union;
// template <class T> struct is_class;
// template <class T> struct is_function;
// template <class T> struct is_lvalue_reference;
// template <class T> struct is_rvalue_reference;
// =====
/*
```

We can divide the type-trait into smaller sets for simplicity.

Primary type category (::value)

```
std::is_pointer<T>,
std::is_integral<T>,
std::is_floating_point<T>
```

Composed type category (::value)

```
std::is_arithmetic<T>,
std::is_object<T>
```

Type comparisons (::value)

```
std::is_same<T,U>,
std::is_base_of<Base,Derived>,
std::is_convertible<From,To>
```

Type transformation (::type)

```
std::add_const<T>,
std::remove_reference<T>
std::make_signed<T>,
std::make_unsigned<T>
```

Others (::type)

```
std::enable_if<bool,T>
std::conditional<bool,T,F>
std::common_type<T1, T2, T3, ... >
```

The above-mentioned functions, from the type-trait, give only a rough idea of their power. To learn more about type checks, click [here](#). The above-mentioned functions are available on the given link with more detail.

```
*/
```

```

// =====
// removeConst.cpp

// #include <iostream>
// #include <string>
// #include <type_traits>

// namespace rgr{

// template<class T, class U>
// struct is_same : std::false_type {};

// template<class T>
// struct is_same<T, T> : std::true_type {};

// template< class T >
// struct remove_const{
//   typedef T type;
// };

// template< class T >
// struct remove_const<const T> {
//   typedef T type;
// };

// int main(){

// std::cout << std::boolalpha << std::endl;

// std::cout << std::is_same<int,std::remove_const<const int>::type>::value << std::endl;
// std::cout << rgr::is_same<int,rgr::remove_const<const int>::type>::value << std::endl;

// typedef rgr::remove_const<double>::type myDouble;
// std::cout << rgr::is_same<double,myDouble>::value << std::endl;

// typedef rgr::remove_const<const std::string>::type myString;
// std::cout << rgr::is_same<std::string,myString>::value << std::endl;

// typedef rgr::remove_const<std::add_const<int>::type>::type myInt;
// std::cout << rgr::is_same<int,myInt>::value << std::endl;

// std::cout << std::endl;

//}

// =====
/*

```

We have implemented `is_same` and `remove_const` in the namespace `rgr`. This corresponds to the type-trait library. For simplicity reason, we use the static constants `std::false_type` and `std::true_type` (lines 10 and 13). Thanks to the base class `std::false_type`, the class template has a member `value`. Respectively for `std::true_type`. The key observation of the class template `is_same` is to distinguish the general template (lines

9 and 10) from the partially specialized template (line 12 and 13). The compiler will use the partially specialized template if both template arguments have the same type. The partially specialized template, as opposed to the general template, has only one type parameter.

Our reasoning for the class template remove_const is similar. The general template returns, via its member type, exactly the same type; the partially specialized template returns the new type after removing the const property (line 22). The compiler will choose the partially specialized template if its template argument is const.

The rest is quickly explained. In lines 31 and 32, we used the functions of the type-trait library and our own versions. We declared a typedef mydouble (line 34), a type myString (line 37), and a type myInt (line 40). All types are non-constant.

```
*/  
// ======  
// ======  
/*
```

Type-Traits (Correctness and Optimization)

In this lesson, we'll study type-trait correctness and their optimization using a gcd (greatest common divisor) algorithm along with fill and equal (type-trait features).

We'll cover the following

Correctness

gcd - The First

gcd - The Second

gcd - The Third

The Smaller Type

The Common Type

gcd - The Fourth

Type-Traits: Performance

Type-Traits fill

Type-Traits std::equal

Correctness#

The reason we use type-trait is correctness and optimization. Let's start with correctness. The idea is to implement generic gcd algorithms, step by step, to make it more type-safe with the help of the type-trait library.

gcd - The First#

Our starting point is the euclid algorithm to calculate the greatest common divisor of two numbers.

It's quite easy to implement the algorithm as a function template and feed it with various arguments. Let's start!

```
*/  
// ======  
// #include <iostream>  
  
// template<typename T>  
// T gcd(T a, T b){  
// if( b == 0 ) return a;  
// else return gcd(b, a % b);  
// }
```

```

// int main(){
// std::cout << gcd(100, 10) << std::endl; // 10
// std::cout << gcd(100, 33) << std::endl; // 1
// std::cout << gcd(100, 0) << std::endl; // 100
// std::cout << gcd(3.5, 4.0) << std::endl; // ERROR
// std::cout << gcd("100", "10") << std::endl; // ERROR
// std::cout << gcd(100, 10L) << std::endl; // ERROR
//}
//=====
/*

```

The function template has two serious issues.

First, it is too generic. The function template accepts doubles (line 13) and C strings (line 14). But it makes no sense to determine the greatest common divisor of both data types. The modulo operation (%) for the double and the C string values fails in line 6. But that's not the only issue.

Second, gcds depend on one type parameter, T. This shows the function template signature gcd(T a, T b)). a and b have to be of the same type T. There is no conversion for type parameters. Therefore, the instantiation of gcd with an int type and a long type (line 15) fails.

gcd - The Second#

We can ignore the rest of the examples below where both arguments have to be positive numbers. The static_assert operator and the predicate std::is_integral<T>::value will help us to check at compile-time whether T is an integral type. A predicate always returns a boolean value.

```

*/
// =====
// #include <iostream>
// #include <type_traits>

// template<typename T>
// T gcd(T a, T b){
//     static_assert(std::is_integral<T>::value, "T should be integral type!");
//     if( b == 0 ) return a;
//     else return gcd(b, a % b);
//}
// int main(){
//     std::cout << gcd(3.5, 4.0) << std::endl;
//     std::cout << gcd("100", "10") << std::endl;
//}
// =====
/*

```

Great. We have solved the first issue of the gcd algorithm. The compilation will not fail by accident because the modulo operator is not defined for a double value and a C string. The compilation fails because the assertion in line 6 will not hold true. The subtle difference is that we now get an exact error message and not a cryptic output of a failed template instantiation as in the first example.

But what about the second issue. The gcd algorithm should accept arguments of a different type.

gcd - The Third#

That's no big deal. But wait, what should the type of result be?

```

*/
// =====

```

```

// #include <iostream>
// #include <type_traits>

// template<typename T1, typename T2>
// ??? gcd(T1 a, T2 b){
//   static_assert(std::is_integral<T1>::value, "T1 should be integral!");
//   static_assert(std::is_integral<T2>::value, "T2 should be integral!");
//   if( b == 0 )
//     return a;
//   else
//     return gcd(b, a % b);
// }

// int main(){
//   std::cout << gcd(100, 10L) << std::endl;
// }
// =====
/*

```

The three question marks in line 5 show the core of the issue. Should the first type or the second type be the return type of the algorithm? Or should the algorithm derive a new type from the two arguments? The type-trait library comes to the rescue. We will present two variations.

The Smaller Type#

A good choice for the return type is to use the smaller of both types. Therefore, we need a ternary operator at compile-time. Thanks to the type-trait library, we have it. The ternary function `std::conditional` operates on types and not on values. That's because we apply the function at compile-time. So, we have to feed `std::conditional` with the right constant expression and we are done. `std::conditional<(sizeof(T1) < sizeof(T2)), T1, T2>::type` will return, at compile-time, `T1` if `T1` is smaller than `T2`; it will return `T2` if `T2` is not smaller than `T1`.

Let's apply the logic.

```

*/
// =====
// #include <iostream>
// #include <type_traits>
// #include <typeinfo>
// template <typename T1, typename T2>
// typename std::conditional<(sizeof(T1) < sizeof(T2)), T1, T2>::type gcd(T1 a, T2 b)
// {
//   static_assert(std::is_integral<T1>::value, "T1 should be integral!");
//   static_assert(std::is_integral<T2>::value, "T2 should be integral!");
//   if (b == 0)
//     return a;
//   else
//     return gcd(b, a % b);
// }

// int main()
//{
//   std::cout << gcd(100, 10LL) << std::endl;
//   auto res = gcd(100, 10LL);

```

```

// std::conditional<(sizeof(long long) < sizeof(long)), long long, long>::type res2 = gcd(100LL, 10L);
// std::cout << typeid(res).name() << std::endl; // i
// std::cout << typeid(res2).name() << std::endl; // l
// std::cout << std::endl;
// }
// =====
/*

```

The critical line of the program is in line 5 with the return type of the gcd algorithm. Of course, the algorithm can also deal with template arguments of the same type. What about line 15? We used the number 100 of type int and the number 10 of type long long int. The result for the greatest common divisor is 10. Line 17 is extremely ugly. We have to repeat the expression std::conditional <(sizeof(100) < sizeof(10LL)), long long, long>::type to determine the right type of the variable res2. Automatic type deduction with auto comes to my rescue (line 16). The typeid operator in line 18 and 19 shows that the result type of the arguments of type int and long long int is int; that the result type of the types long long int and long int is long int.

The Common Type#

Now to the second variation. Often it is not necessary to determine the smaller type at compile-time but to determine the type to which all types can implicitly be converted to. std::common_type can handle an arbitrary number of template arguments. To say it more formally. std::common_type is a variadic template.

```

*/
// =====
// #include <iostream>
// #include <type_traits>
// #include <typeinfo>

```

```

// template<typename T1, typename T2>
// typename std::common_type<T1, T2>::type gcd(T1 a, T2 b){
//   static_assert(std::is_integral<T1>::value, "T1 should be an integral type!");
//   static_assert(std::is_integral<T2>::value, "T2 should be an integral type!");
//   if( b == 0 ){
//     return a;
//   }
//   else{
//     return gcd(b, a % b);
//   }
// }
// int main(){
//   std::cout << typeid(gcd(100, 10)).name() << std::endl; // i
//   std::cout << typeid(gcd(100, 10L)).name() << std::endl; // l
//   std::cout << typeid(gcd(100, 10LL)).name() << std::endl; // x
// }
// =====
/*

```

The only difference to the last implementation is that std::common_type in line 6 determines the return type. We ignored the results of the gcd algorithm in this example because we're more interested in the types of results. With the argument types int and int we get int; with the argument types int and long int we get long int, and with int and long long int we get long long int.

gcd - The Fourth#

But that's not all. std::enable_if from the type-trait library provides a very interesting variation. What the previous implementations have in common is that they will check in the function body if the arguments are of

integral types or not. The key observation is that the compiler always tries to instantiate the function templates but sometimes fails. We know the result. If the expression std::is_integral returns false, the instantiation will fail. That is not the best way. It would be better if the function template is only available for the valid types. Therefore, we put the check of the function template from the template body to the template signature.

```
/*
// =====
// #include <iostream>
// #include <type_traits>

// template<typename T1, typename T2,
//   typename std::enable_if<std::is_integral<T1>::value,T1 >::type= 0,
//   typename std::enable_if<std::is_integral<T2>::value,T2 >::type= 0,
//   typename R = typename std::conditional<(sizeof(T1) < sizeof(T2)),T1,T2>::type>
// R gcd(T1 a, T2 b){
//   if( b == 0 ){
//     return a;
//   }
//   else{
//     return gcd(b, a % b);
//   }
// }

// int main(){
//   std::cout << "gcd(100, 10)= " << gcd(100, 10) << std::endl;
//   std::cout << "gcd(100, 33)= " << gcd(100, 33) << std::endl;
//   std::cout << "gcd(3.5, 4.0)= " << gcd(3.5, 4.0) << std::endl;
// }
// =====
/*
```

Lines 5 and 6 are the key lines of the new program. The expression std::is_integral determines whether the type parameter T1 and T2 are integrals. If T1 and T2 are not integrals, and therefore they return false, we will not get a template instantiation. This is the decisive observation.

If std::enable_if returns true as the first parameter, std::enable_if will have a public member typedef type. This type is used in lines 5 and 6. If std::enable_if returns false as first parameter, std::enable_if will have no member type. Therefore, lines 5 and 6 are not valid. This is not an error but a common technique in C++: SFINAЕ. SFINAЕ stands for Substitution Failure Is Not An Error. Only the template for exactly this type will not be instantiated and the compiler tries to instantiate the template in another way.

Type-Traits: Performance#

The idea is quite straightforward and is used in current implementations of the Standard Template Library (STL). If the elements of a container are simple enough, the algorithm of the STL like std::copy, std::fill, or std::equal will directly be applied on the memory area. Instead of using std::copy to copy the elements one by one, all is done in one step. Internally, C functions like memcmp, memset, memcpy, or memmove are used. The small difference between memcpy and memmove is that memmove can deal with overlapping memory areas.

The implementations of the algorithm std::copy, std::fill, or std::equal use a simple strategy. std::copy is like a wrapper. This wrapper checks if the element is simple enough. If so, the wrapper will delegate the work to the optimized copy function. If not, the general copy algorithm will be used. This one copies each element after

one another. To make the right decision, the functions of the type-trait library will be used if the elements are simple enough.

The graphic shows this strategy once more:

```
/*
// =====
// =====
/*
Type-Traits fill#
std::fill assigns each element, in the range, a value. The listing shows a simple implementation which is based
on the GCC implementation.
*/
// =====
// fill.cpp

// #include <cstring>
// #include <chrono>
// #include <iostream>
// #include <type_traits>

// namespace my{

// template <typename I, typename T, bool b>
// void fillImpl(I first, I last, const T& val, const std::integral_constant<bool, b>&){
//   while(first != last){
//     *first = val;
//     ++first;
//   }
// }

// template <typename T>
// void fillImpl(T* first, T* last, const T& val, const std::true_type&){
//   std::memset(first, val, last-first);
// }

// template <class I, class T>
// inline void fill(I first, I last, const T& val){
//   // typedef std::integral_constant<bool, std::has_trivial_copy_assign<T>::value && (sizeof(T) == 1)>
//   boolType;
//   // typedef std::integral_constant<bool, std::is_trivially_copy_assignable<T>::value && (sizeof(T) == 1)>
//   boolType;
//   fillImpl(first, last, val, boolType());
// }
// }

// const int arraySize = 100000000;
// char charArray1[arraySize] = {0,};
// char charArray2[arraySize] = {0,};

// int main(){
```

```

// std::cout << std::endl;

// auto begin= std::chrono::system_clock::now();
// my::fill(charArray1, charArray1 + arraySize,1);
// auto last= std::chrono::system_clock::now() - begin;
// std::cout << "charArray1: " << std::chrono::duration<double>(last).count() << " seconds" << std::endl;

// begin= std::chrono::system_clock::now();
// my::fill(charArray2, charArray2 + arraySize, static_cast<char>(1));
// last= std::chrono::system_clock::now() - begin;
// std::cout << "charArray2: " << std::chrono::duration<double>(last).count() << " seconds" << std::endl;

// std::cout << std::endl;

//}

//=====
/*
my::fill make in line 27 the decision which implementation of my::fill_Impl is applied. To use the optimized
variant, the elements should have a compiler generated copy assignment operator
std::is_trivially_copy_assignable<T> and should be 1 byte large: sizeof(T) == 1. The function
std::is_trivially_copy_assignable is part of the type-trait library. The first call my::fill(charArray1, charArray1 +
arraySize,1); has the last parameter 1, which is an int that has a size of 4 bytes. That is why, the test on line 26
evaluates to false.

```

Our GCC calls the function std::is_trivially_copy_assignable instead of std::has_trivial_copy_assign. If we request with the keyword default from the compiler the copy assignment operator, the operator will be trivial.

Type-Traits std::equal#

The following code snippet shows a part of the implementation of std::equal in the GCC:

```

*/
// =====
// template<typename _I1, typename _I2>
// inline bool __equal_aux(_I1 __first1, _I1 __last1, _I2 __first2){
//   typedef typename iterator_traits<_I1>::value_type _ValueType1;
//   typedef typename iterator_traits<_I2>::value_type _ValueType2;
//   const bool __simple = ((__is_integer<_ValueType1>::__value
//                         || __is_pointer<_ValueType1>::__value )
//                         && __is_pointer<_I1>::__value
//                         && __is_pointer<_I2>::__value
//                         && __are_same<_ValueType1, _ValueType2>::__value
//                         );
//   return std::__equal<__simple>::equal(__first1, __last1, __first2);
// }
// =====
/*

```

Example 1: Template fill#

```

*/
// =====
// templatefill.cpp

```

```

// #include <cstring>
// #include <chrono>
// #include <iostream>
// #include <type_traits>

// namespace my{

// template <typename I, typename T, bool b>
// void fillImpl(I first, I last, const T& val, const std::integral_constant<bool, b>&){
//   while(first != last){
//     *first = val;
//     ++first;
//   }
// }

// template <typename T>
// void fillImpl(T* first, T* last, const T& val, const std::true_type&){
//   std::memset(first, val, last-first);
// }

// template <class I, class T>
// inline void fill(I first, I last, const T& val){
//   typedef std::integral_constant<bool, std::is_trivially_copy_assignable<T> ::value && (sizeof(T) == 1)> boolType;
//   fillImpl(first, last, val, boolType());
// }
// }

// const int arraySize = 100000000;
// char charArray1[arraySize] = {0,};
// char charArray2[arraySize] = {0,};

// int main(){

// std::cout << std::endl;

// auto begin= std::chrono::system_clock::now();
// my::fill(charArray1, charArray1 + arraySize, 1);
// auto last= std::chrono::system_clock::now() - begin;
// std::cout << "charArray1: " << std::chrono::duration<double>(last).count() << " seconds" << std::endl;

// begin= std::chrono::system_clock::now();
// my::fill(charArray2, charArray2 + arraySize, static_cast<char>(1));
// last= std::chrono::system_clock::now() - begin;
// std::cout << "charArray2: " << std::chrono::duration<double>(last).count() << " seconds" << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

Explanation#

In line 26, my::fill make the decision as to which implementation of my::fillImpl is applied. To use the optimized variant, the elements should have a compiler generated copy assignment operator std::is_trivially_copy_assignable<T> and should be 1 byte large: sizeof(T) == 1. The function std::is_trivially_copy_assignable is part of the type-trait library.

If the expression boolType() in line 26 is true, the optimized version of my::fillImpl in the lines 17 - 20 will be used. This variant fills in opposite of the generic variant my::fillImpl (line 10-16) the entire memory area - consisting of 100 million entries - with the value 1. sizeof(char) is 1.

What's about the performance of the program? We compiled the program with full optimization. The execution of the optimized variant is about 3 times faster on windows; about 20 times faster on Linux.

Example 2: Template Type Manipulation#

```
/*
// =====
// templateTypeManipulation.cpp

// #include <iostream>
// #include <type_traits>

// template <typename T>
// struct RemoveConst{
//     typedef T type;
// };

// template <typename T>
// struct RemoveConst<const T>{
//     typedef T type;
// };

// int main(){
//     std::cout << std::boolalpha << std::endl;
//
//     std::cout << "std::is_same<int, RemoveConst<int>::type>::value: " << std::is_same<int,
// RemoveConst<int>::type>::value << std::endl;
//     std::cout << "std::is_same<int, RemoveConst<const int>::type>::value: " << std::is_same<int,
// RemoveConst<const int>::type>::value << std::endl;
//
//     std::cout << std::endl;
//
// }
// =====
/*
```

Explanation#

The code above uses the function std::is_same from the type-trait library. std::is_same compares the type passed in the function and the type given in the function defined by us, and it returns true only when both types are the same.

```
/*
// =====
```

```
// =====  
/*
```

constexpr

In this lesson, we'll study `constexpr`.

Constant Expressions#

You can define, with the keyword `constexpr`, an expression that can be evaluated at compile-time. `constexpr` can be used for variables, functions, and user-defined types. An expression that is evaluated at compile-time has a lot of advantages. A constant expression

can be evaluated at compile-time.

gives the compiler deep insight into the code.

is implicitly thread-safe.

constexpr - Variables and Objects#

If you declare a variable as `constexpr`, the compiler will evaluate them at compile-time. This holds not only true for built-in types but also for instantiations of user-defined types. There are a few serious restrictions for objects in order to evaluate them at compile-time.

To make life easier for us, we will call the C types like `bool`, `char`, `int`, and `double` primitive data types. We will call the remaining data types as user-defined data types. These are for example, `std::string`, types from the C++ library and non-primitive data types. Non-primitive data types typically hold primitive data types.

Variables#

By using the keyword `constexpr`, the variable becomes a constant expression.

```
constexpr double pi= 3.14;
```

Therefore, we can use the variable in contexts that require a constant expression. For example, if we want to define the size of an array. This has to be done at compile-time.

For the declaration of a `constexpr` variable, you have to keep a few rules in mind.

The variable:

is implicitly `const`.

has to be initialized.

requires a constant expression for initialization.

The above rules make sense because if we evaluate a variable at compile-time, the variable can only depend on values that can be evaluated at compile time.

The objects are created by the invocation of the constructor and the constructor has a few special rules as well.

User-Defined Types#

A `constexpr` constructor

can only be invoked with constant expressions.

cannot use exception handling.

has to be declared as default or delete or the function body must be empty (C++11).

The `constexpr` user-defined type

cannot have virtual base classes.

requires that each base object and each non-static member has to be initialized in the initialization list of the constructor or directly in the class body. Consequently, it holds that each used constructor (e.g of a base class) has to be a `constexpr` constructor and that the applied initializers have to be constant expressions.

Example#

```
struct MyDouble{
    double myVal;
    constexpr MyDouble(double v): myVal(v){}
    constexpr double getVal(){return myVal;}
};
```

The constructor has to be empty and a constant expression.

The user-defined type can have methods which are constant expressions and cannot be virtual.

Instances of `MyDouble` can be instantiated at compile-time.

Functions#

`constexpr` functions are functions that have the potential to run at compile-time. With `constexpr` functions, you can perform a lot of calculations at compile-time. Therefore, the result of the calculation is available at runtime and stored as a constant in the ROM available. In addition, `constexpr` functions are implicitly inline.

A `constexpr` function can be invoked with a non-`constexpr` value. In this case, the function runs at runtime. A `constexpr` function is executed at compile-time when it is used in an expression which is evaluated at compile-time. Some examples would be when using a `static_assert` or the definition of a C-array. A `constexpr` function is also executed at compile-time, when the result is requested at compile-time, for example: `constexpr auto res = constexprFunction();`.

For `constexpr` functions there are a few restrictions:

The function

has to be non-virtual.

has to have arguments and a return value of a literal type. Literal types are the types of `constexpr` variables.

can only have one return statement.

must return a value.

will be executed at compile-time if invoked within a constant expression.

can only have a function body consisting of a return statement.

must have a constant return value

is implicitly inline.

Examples#

```
constexpr int fac(int n){
    return n > 0 ? n * fac(n-1): 1;
}
```

```
constexpr int gcd(int a, int b){
    return (b==0) ? a : gcd(b, a % b);
}
```

Functions with C++14#

The syntax of `constexpr` functions was massively improved with the change from C++11 to C++14. In C++11, you had to keep in mind which feature you can use in a `constexpr` functions. With C++14, you only have to keep in mind which feature you can't use in a `constexpr` function.

`constexpr` Functions in C++14#

can have variables that have to be initialized by a constant expression.

cannot have static or `thread_local` data.

can have conditional jump instructions or loop instructions.

can have more than one instruction.

Example#

```
/*
// =====
// constexpr auto gcd(int a, int b)
//{
//   while (b != 0)
//   {
//     auto t = b;
//     b = a % b;
//     a = t;
//   }
//   return a;
//}
// =====
/*
```

Template Metaprogramming vs `constexpr` Functions#

CharacteristicsTemplate Metaprogramming Constant Expressions

Execution time Compile-time Compile-time and runtime

Arguments Types and values Values

Programming paradigm Functional Imperative

Modification No Yes

Control structure Recursion Conditions and Loops

Conditional execution Template specialization Conditional statement

There are a few remarks about the above-mentioned table.

A template metaprogram runs at compile-time, but a `constexpr` functions can run at compile-time or runtime.

Arguments of a template (template metaprogram) can be types and values. To be more specific, a template can take types, `std::vector<int>`, values, `std::array<int, 5>`, and even templates `std::stack<int, std::vector<int>>`. `constexpr` functions are just functions which have the potential to run at compile time. Therefore, they can only accept values.

To learn more about `constexpr`, click [here](#).

```
/*
// =====
// =====
/*
```

- Examples

In this lesson, we'll get into examples of `constexpr`.

We'll cover the following

Example 1: `constexpr` using C++ 11

Explanation

Example 2: `constexpr` function in C++ 14

Explanation

Example 1: `constexpr` using C++ 11#

*/

// =====

// constExpression.cpp

// #include <iostream>

// constexpr int square(int x) { return x * x; }

// constexpr int squareToSquare(int x){ return square(square(x));}

// int main() {

// std::cout << std::endl;

// static_assert(square(10) == 100, "you calculated it wrong");

// static_assert(squareToSquare(10) == 10000 , "you calculated it wrong");

// std::cout << "square(10)= " << square(10) << std::endl;

// std::cout << "squareToSquare(10)= " << squareToSquare(10) << std::endl;

// constexpr int constExpr= square(10);

// int arrayClassic[100];

// int arrayNewWithConstExpression[constExpr];

// int arrayNewWithConstExpressioFunction[square(10)];

// std::cout << std::endl;

// }

// =====

/*

Explanation#

In the example above, we have implemented two `constexpr` functions: `constexpr int square(int x)` and `constexpr int squareToSquare(int x)`. As you can see, both the functions follow the conventions for `constexpr` functions definition in C++11.

The assertion in lines 12 and 13 succeed since 10 is a literal type. Making a `constexpr` variable will allow the code compilation to pass the assertions.

In line 17, we have initialized a `constexpr` variable `constExpr` using the `sqaure` function.

In lines 19-21, we have initialized three arrays

by using a constant 100

by using a constexpr variable constExpr

by calling the function square(10). Notice that the input argument for this function call is constant.

Example 2: constexpr function in C++ 14#

```
/*
// =====
// constExpressionCpp14.cpp

// #include <iostream>

// constexpr int gcd(int a, int b){
//   while (b != 0){
//     auto t= b;
//     b= a % b;
//     a= t;
//   }
//   return a;
// }

// int main(){
//   std::cout << std::endl;
//   constexpr auto res= gcd(100, 10);
//   std::cout << "gcd(100, 10) " << res << std::endl;
//
//   auto val= 100;
//   auto res2= gcd(val, 10);
//   std::cout << "gcd(val, 10) " << res2 << std::endl;
//
// }
```

/*

Explanation#

Line 18 calculates the result res at compile-time, and line 22 res2 at runtime.

The difference between ordinary functions and constexpr functions in C++14 is minimal. Therefore, it's quite easy to implement the gcd algorithm in C++14 as a constexpr function.

We have defined res as constexpr variable and its type is automatically determined by auto.

```
/*
// =====
// =====
/*
constexpr if
```

Let's study constexpr if in this lesson.

We'll cover the following

constexpr if (C++17)

constexpr if (C++17) #

`constexpr` if enables us to compile source code conditionally.

```
if constexpr (cond) statement1;  
else statement2;
```

The expression `cond` has to be a constant expression.

The unused code has to be valid.

Thanks to `constexpr` if, functions can have different return types.

The following code snippet shows a function, which returns an `int` or a `double`.

```
template <typename T>  
auto getAnswer(T t){  
    static_assert(std::is_arithmetic_v<T>); // arithmetic  
    if constexpr (std::is_integral_v<T>) // integral  
        return 42;  
    else  
        return 42.0; // floating point  
}
```

`constexpr` if is a replacement for tag dispatching and SFINAE. SFINAE stands for Substitution Failure Is Not An Error.

To study further about tag dispatching, click [here](#).

In the next lesson, we'll look at an example of `constexpr` if.

```
/*  
// ======  
// ======  
/*  
Example: constexpr if#  
*/  
// ======  
// constexprIf.cpp  
  
// #include <iostream>  
// #include <type_traits>  
  
// // SFINAE  
  
// template <typename T, std::enable_if_t<std::is_arithmetic<T>{}> * = nullptr>  
// auto get_value_SFINAE(T)  
// {  
//     std::cout << "get_Value_SFINAE(5)" << std::endl;  
// }  
  
// template <typename T, std::enable_if_t<!std::is_arithmetic<T>{}> * = nullptr>  
// auto get_value_SFINAE(T)  
// {  
//     std::cout << "get_Value_SFINAE(five)" << std::endl;  
// }  
  
// // Tag dispatch
```

```
// template <typename T>
// auto get_value_TAG_DISPATCH(T, std::true_type)
//{
// std::cout << "get_Value_TAG_DISPATCH(5)" << std::endl;
//}

// template <typename T>
// auto get_value_TAG_DISPATCH(T, std::false_type)
//{
// std::cout << "get_Value_TAG_DISPATCH(five)" << std::endl;
//}

// template <typename T>
// auto get_value_TAG_DISPATCH(T t)
//{
// return get_value_TAG_DISPATCH(t, std::is_arithmetic<T>{});
//}

/// constexpr if

// template <typename T>
// auto get_value_CONSTEXPR_IF(T)
//{
// if constexpr (std::is_arithmetic_v<T>)
// {
// std::cout << "get_Value_CONSTEXPR_IF(5)" << std::endl;
// }
// else
// {
// std::cout << "get_Value_CONSTEXPR_IF(five)" << std::endl;
// }
//}

// int main()
//{
// std::cout << std::endl;

// get_value_SFINAE(5);
// get_value_SFINAE("five");

// std::cout << std::endl;

// get_value_TAG_DISPATCH(5);
// get_value_TAG_DISPATCH("five");

// std::cout << std::endl;

// get_value_CONSTEXPR_IF(5);
// get_value_CONSTEXPR_IF("five");
```

```
// std::cout << std::endl;
// }
// =====
/*
```

Explanation#

We have created get_value functions which use SFINAE, TAG_DISPATCH, and CONSTEXPR_IF. These functions use the std::is_arithmetic function from the type-trait library. std::is_arithmetic returns true only if std::is_integral or std::is_floating_point is true for a given type. All the calls from main verify that the passed argument falls in their required category.

get_value_SFINAE uses the function std::enable_if from the type-trait library. std::enable_if is only true, if the given type is arithmetic.

get_value_TAG_DISPATCH in line 33 dispatches on the result of std::is_arithmetic.

get_value_CONSTEXPR_IF creates another branch of the if-statement depending on the result of the expression std::is_arithmetic_v<T> which is a shorthand for std::is_arithmetic<T>::value.

We have learned about the techniques which we used in templates. In the next chapter, we'll see different design techniques used in C++ Templates.

```
/*
// =====
// =====
/*
```

Static Versus Dynamic Polymorphism

Let's dive deep into polymorphism in this lesson.

We'll cover the following

Polymorphism

Polymorphism#

Polymorphism means that an object can have different behaviors.

Dynamic Polymorphism

Polymorphism happens at runtime.

A key feature of object-orientation.

Based on interfaces and virtual methods.

Needs one indirection such as a reference or a pointer in C++.

Static Polymorphism

Polymorphism happens at compile-time.

Is not bound to interfaces or derivation hierarchies => Duck Typing

No indirection such as pointers or references required.

Static polymorphism is typically faster than dynamic polymorphism.

```
/*
// =====
// =====
/*
```

Examples

Let's have a look at a couple of examples of polymorphism.

We'll cover the following

Example 1: Dispatch with Dynamic Polymorphism

Explanation

Example 2: Dispatch with Static Polymorphism

Explanation

Example 1: Dispatch with Dynamic Polymorphism#

*/

```
// =====
```

```
// dispatchDynamicPolymorphism.cpp
```

```
// #include <chrono>
```

```
// #include <iostream>
```

```
// auto start = std::chrono::steady_clock::now();
```

```
// void writeElapsedTime(){
```

```
//   auto now = std::chrono::steady_clock::now();
```

```
//   std::chrono::duration<double> diff = now - start;
```

```
//   std::cerr << diff.count() << " sec. elapsed: ";
```

```
// }
```

```
// struct MessageSeverity{
```

```
//   virtual void writeMessage() const {
```

```
//     std::cerr << "unexpected" << std::endl;
```

```
//   }
```

```
//};
```

```
// struct MessageInformation: MessageSeverity{
```

```
//   void writeMessage() const override {
```

```
//     std::cerr << "information" << std::endl;
```

```
//   }
```

```
//};
```

```
// struct MessageWarning: MessageSeverity{
```

```
//   void writeMessage() const override {
```

```
//     std::cerr << "warning" << std::endl;
```

```
//   }
```

```
//};
```

```
// struct MessageFatal: MessageSeverity{};
```

```
// void writeMessageReference(const MessageSeverity& messServer){
```

```
//   writeElapsedTime();
```

```
//   messServer.writeMessage();
```

```
// }
```

```
// void writeMessagePointer(const MessageSeverity* messServer){
```

```
//   writeElapsedTime();
```

```

//     messServer->writeMessage();

// }

// int main(){

//   std::cout << std::endl;

//   MessageInformation messInfo;
//   MessageWarning messWarn;
//   MessageFatal messFatal;

//   MessageSeverity& messRef1 = messInfo;
//   MessageSeverity& messRef2 = messWarn;
//   MessageSeverity& messRef3 = messFatal;

//   writeMessageReference(messRef1);
//   writeMessageReference(messRef2);
//   writeMessageReference(messRef3);

//   std::cerr << std::endl;

//   MessageSeverity* messPoin1 = new MessageInformation;
//   MessageSeverity* messPoin2 = new MessageWarning;
//   MessageSeverity* messPoin3 = new MessageFatal;

//   writeMessagePointer(messPoin1);
//   writeMessagePointer(messPoin2);
//   writeMessagePointer(messPoin3);

//   std::cout << std::endl;

// }
// =====
/*

```

Note: std::cerr of the class std::ostream represents the standard error stream. This is not a runtime error.

Explanation#

The structs in lines 15, 21, and 27 know what they should display if used. The key idea is that the static type MessageSeverity differs from the dynamic type such as MessageInformation (line 61); therefore, the late binding will kick in and the writeMessage methods in lines 71, 72, and 73 are of the dynamic types. Dynamic polymorphism requires a kind of indirection. We can use references (57-59) or pointers (67-69).

From a performance perspective, we can do better and make the dispatch at compile time.

Example 2: Dispatch with Static Polymorphism#

```

*/
// =====
// DispatchStaticPolymorphism.cpp

// #include <chrono>

```

```
// #include <iostream>

// auto start = std::chrono::steady_clock::now();

// void writeElapsedTime(){
//   auto now = std::chrono::steady_clock::now();
//   std::chrono::duration<double> diff = now - start;

//   std::cerr << diff.count() << " sec. elapsed: ";
// }

// template <typename ConcreteMessage>
// struct MessageSeverity{
//   void writeMessage(){
//     static_cast<ConcreteMessage*>(this)->writeMessageImplementation();
//   }
//   void writeMessageImplementation() const {
//     std::cerr << "unexpected" << std::endl;
//   }
// };

// struct MessageInformation: MessageSeverity<MessageInformation>{
//   void writeMessageImplementation() const {
//     std::cerr << "information" << std::endl;
//   }
// };

// struct MessageWarning:
// MessageSeverity<MessageWarning>{
//   void writeMessageImplementation() const {
//     std::cerr << "warning" << std::endl;
//   }
// };

// struct MessageFatal:
// MessageSeverity<MessageFatal>{};

// template <typename T>
// void writeMessage(T& messServer){

//   writeElapsedTime();
//   messServer.writeMessage();

// }

// int main(){

//   std::cout << std::endl;

//   MessageInformation messInfo;
//   writeMessage(messInfo);
```

```

// MessageWarning messWarn;
// writeMessage(messWarn);

// MessageFatal messFatal;
// writeMessage(messFatal);

// std::cout << std::endl;

//}
// =====
/*
Explanation#

```

In this case, all concrete structs in lines 25, 31, and 38 are derived from the base class `MessageSeverity`. The method `writeMessage` serves as an interface that dispatches to the concrete implementations `writeMessageImplementation`. To make that happen, the object will be upcasted to the `ConcreteMessage`:

```

static_cast<ConcreteMessage*>(this)->writeMessageImplementation();
This is the dispatch at compile time; therefore, this technique is called static polymorphism.

```

To be honest, it took me a bit of time to get used to it but applying static polymorphism like that on line 42 is actually quite easy.

```

*/
// =====
// =====
/*
Problem Statement#
Polymorphism can be simulated in various ways. Implement the polymorphism with if, with switch, and with a dispatch table and compare the three implementations using static and dynamic polymorphism.
*/
// =====
// #include <iostream>

/// // Write your code here
/// // After writing your code uncomment the lines in main

// int main() {

// std::cout << std::endl;
// /*
// writeMessage(MessageSeverity::information);
// writeMessage(MessageSeverity::warning);
// writeMessage(MessageSeverity::fatal);
// */
// std::cout << std::endl;

//}
// =====
/*

```

Solution 1: Using the if Statement#

```
/*
// =====
// dispatchIf.cpp

// #include <chrono>
// #include <iostream>

// enum class MessageSeverity
//{
//   information,
//   warning,
//   fatal,
//};

// auto start = std::chrono::steady_clock::now();

// void writeElapsedTime()
//{
//   auto now = std::chrono::steady_clock::now();
//   std::chrono::duration<double> diff = now - start;

//   std::cerr << diff.count() << " sec. elapsed: ";
//}

// void writeInformation() { std::cerr << "information" << std::endl; }
// void writeWarning() { std::cerr << "warning" << std::endl; }
// void writeUnexpected() { std::cerr << "unexpected" << std::endl; }

// void writeMessage(MessageSeverity messServer)
//{
//   writeElapsedTime();

//   if (MessageSeverity::information == messServer)
//   {
//     writeInformation();
//   }
//   else if (MessageSeverity::warning == messServer)
//   {
//     writeWarning();
//   }
//   else
//   {
//     writeUnexpected();
//   }
//}

// int main()
//{
//}
```

```

// std::cout << std::endl;
// writeMessage(MessageSeverity::information);
// writeMessage(MessageSeverity::warning);
// writeMessage(MessageSeverity::fatal);

// std::cout << std::endl;
//}
// =====
/*
Explanation#

```

Note: std::cerr of the class std::ostream represents the standard error stream. This is not a runtime error.

The function writeMessage in line 25 displays the elapsed time in seconds in line 27 since the start of the program and a log message. It uses an enumeration in line 6 for the message severity. We used the start time in line 12 and the current time in line 15 to calculate the elapsed time. As the name suggests, the std::steady_clock cannot be adjusted; therefore, it is the right choice for this measurement. The key part of the program is the part of the function writeMessage in line 25, in which we made the decision which message should be displayed. In this case, we used if-else statements.

Solution 2: Using the switch statement#

```

*/
// =====
// dispatchSwitch

// #include <chrono>
// #include <iostream>

// enum class MessageSeverity{
//   information,
//   warning,
//   fatal,
// };

// auto start = std::chrono::steady_clock::now();

// void writeElapsedTime(){
//   auto now = std::chrono::steady_clock::now();
//   std::chrono::duration<double> diff = now - start;

//   std::cerr << diff.count() << " sec. elapsed: ";
// }

// void writeInformation(){ std::cerr << "information" << std::endl; }
// void writeWarning(){ std::cerr << "warning" << std::endl; }
// void writeUnexpected(){ std::cerr << "unexpected" << std::endl; }

// void writeMessage(MessageSeverity messSever){

//   writeElapsedTime();

```

```

// switch(messSever){
//   case MessageSeverity::information:
//     writeInformation();
//     break;
//   case MessageSeverity::warning:
//     writeWarning();
//     break;
//   default:
//     writeUnexpected();
//     break;
// }

//}

// int main(){

// std::cout << std::endl;

// writeMessage(MessageSeverity::information);
// writeMessage(MessageSeverity::warning);
// writeMessage(MessageSeverity::fatal);

// std::cout << std::endl;

// }
//=====
/*
Explanation#

```

Note: std::cerr of the class std::ostream represents the standard error stream. This is not a runtime error.

The following program is quite similar to the previous one. Only the implementation of the function writeMessage changed. The function writeMessage in line 25 displays the elapsed time in seconds (line 27) since the start of the program and a log message. It uses an enumeration (line 6) for the message severity. We used the start time (line 12) and the current time (line 15) to calculate the elapsed time. As the name suggested, the std::steady_clock cannot be adjusted; therefore, it is the right choice for this measurement. The key part of the program is the part of the function writeMessage (line 25), in which we made the decision which message should be displayed. In this case, we used the switch statements.

To be honest, I had to look up the syntax for the switch statements to make it right.

Solution 3: Using a Dispatch Table#

```

*/
//=====
// dispatchHasttables

// #include <chrono>
// #include <functional>
// #include <iostream>
// #include <unordered_map>

// enum class MessageSeverity{

```

```

// information,
// warning,
// fatal,
//};

// auto start = std::chrono::steady_clock::now();

// void writeElapsedTime(){
//   auto now = std::chrono::steady_clock::now();
//   std::chrono::duration<double> diff = now - start;
//
//   std::cerr << diff.count() << " sec. elapsed: ";
//}

// void writeInformation(){ std::cerr << "information" << std::endl; }
// void writeWarning(){ std::cerr << "warning" << std::endl; }
// void writeUnexpected(){ std::cerr << "unexpected" << std::endl; }

// std::unordered_map<MessageSeverity, std::function<void()>> mess2Func{
//   {MessageSeverity::information, writeInformation},
//   {MessageSeverity::warning, writeWarning},
//   {MessageSeverity::fatal, writeUnexpected}
// };

// void writeMessage(MessageSeverity messServer){

//   writeElapsedTime();

//   mess2Func[messServer]();

//}

// int main(){

//   std::cout << std::endl;

//   writeMessage(MessageSeverity::information);
//   writeMessage(MessageSeverity::warning);
//   writeMessage(MessageSeverity::fatal);

//   std::cout << std::endl;

//}

// =====
/*
Explanation#

```

Note: std::cerr of the class std::ostream represents the standard error stream. This is not a runtime error.

With the if-else or the switch statement, we used enumerator for dispatching to the right case. The key to our dispatch table behaves in a similar way.

Dynamic or static polymorphism is totally different. Instead of an enumerator or a key for dispatching to the right action, we used objects which decide autonomously at runtime (dynamic polymorphism) or compile-time (static polymorphism) what should be done.

Let's move on to CRTP in the next lesson.

```
*/  
// ======  
// ======  
/*  
CRTP
```

CRTP#

The acronym CRTP stands for the C++ idiom Curiously Recurring Template Pattern and is a technique in C++ in which a Derived class derives from a class template Base. The key is that Base has Derived as a template argument.

Let's have a look at an example:

```
template<class T>  
class Base{  
    ...  
};  
  
class Derived: public Base<Derived>{  
    ...  
};
```

CRTP enables static polymorphism.

Typical use-case#

There are two typical use-cases for CRTP: Mixins and static polymorphism.

Mixins#

Mixins is a popular concept in the design of classes to mix in new code. Therefore, it's an often-used technique in Python to change the behavior of a class by using multiple inheritances. In contrast to C++, in Python, it is legal to have more than one definition of a method in a class hierarchy. Python simply uses the method that is first in the Method Resolution Order (MRO).

You can implement mixins in C++ by using CRTP. A prominent example is the class `std::enable_shared_from_this`. By using this class, you can create objects that return an `std::shared_ptr` to themselves. We have to derive your class `MySharedClass` public from `std::enable_shared_from_this`. Now, our class `MySharedClass` has a method `shared_from_this`.

An additional typical use-case for mixins is a class that you want to extend with the capability that their instances support the comparison for equality and inequality.

Static Polymorphism#

Static polymorphism is quite similar to dynamic polymorphism. But contrary to dynamic polymorphism with virtual methods, the dispatch of the method calls will take place at compile-time. Now, we are at the center of the CRTP idiom.

```
class ShareMe: public std::enable_shared_from_this<ShareMe>{
```

```
std::shared_ptr<ShareMe> getShared(){}
    return shared_from_this();
}
};

std::enable_shared_from_this creates a shared _ptr for an object.
std::enable_shared_from_this: base class of the object.
shared_from_this: returns the shared object
To learn more about CRTP, click here.
```

In the next lesson, we'll look at a couple of examples of CRTP.

```
/*
// =====
// =====
/*
Example 1: Mixins with CRTP#
*/
// =====
// templateCRTPRelational.cpp

// #include <iostream>
// #include <string>

// template<class Derived>
// class Relational{};

// // Relational Operators

// template <class Derived>
// bool operator > (Relational<Derived> const& op1, Relational<Derived> const & op2){
//     Derived const& d1 = static_cast<Derived const&>(op1);
//     Derived const& d2 = static_cast<Derived const&>(op2);
//     return d2 < d1;
// }

// template <class Derived>
// bool operator == (Relational<Derived> const& op1, Relational<Derived> const & op2){
//     Derived const& d1 = static_cast<Derived const&>(op1);
//     Derived const& d2 = static_cast<Derived const&>(op2);
//     return !(d1 < d2) && !(d2 < d1);
// }

// template <class Derived>
// bool operator != (Relational<Derived> const& op1, Relational<Derived> const & op2){
//     Derived const& d1 = static_cast<Derived const&>(op1);
//     Derived const& d2 = static_cast<Derived const&>(op2);
//     return (d1 < d2) || (d2 < d1);
// }

// template <class Derived>
// bool operator <= (Relational<Derived> const& op1, Relational<Derived> const & op2){
//     Derived const& d1 = static_cast<Derived const&>(op1);
```

```

// Derived const& d2 = static_cast<Derived const&>(op2);
// return (d1 < d2) || (d1 == d2);
// }

// template <class Derived>
// bool operator >= (Relational<Derived> const& op1, Relational<Derived> const & op2){
//   Derived const& d1 = static_cast<Derived const&>(op1);
//   Derived const& d2 = static_cast<Derived const&>(op2);
//   return (d1 > d2) || (d1 == d2);
// }

/// / Apple

// class Apple:public Relational<Apple>{
// public:
//   explicit Apple(int s): size{s} {};
//   friend bool operator < (Apple const& a1, Apple const& a2){
//     return a1.size < a2.size;
//   }
// private:
//   int size;
// };

/// / Man

// class Man:public Relational<Man>{
// public:
//   explicit Man(const std::string& n): name{n} {}
//   friend bool operator < (Man const& m1, Man const& m2){
//     return m1.name < m2.name;
//   }
// private:
//   std::string name;
// };

int main(){

// std::cout << std::boolalpha << std::endl;

// Apple apple1{5};
// Apple apple2{10};
// std::cout << "apple1 < apple2: " << (apple1 < apple2) << std::endl;
// std::cout << "apple1 > apple2: " << (apple1 > apple2) << std::endl;
// std::cout << "apple1 == apple2: " << (apple1 == apple2) << std::endl;
// std::cout << "apple1 != apple2: " << (apple1 != apple2) << std::endl;
// std::cout << "apple1 <= apple2: " << (apple1 <= apple2) << std::endl;
// std::cout << "apple1 >= apple2: " << (apple1 >= apple2) << std::endl;

// std::cout << std::endl;

// Man man1{"grimm"};

```

```

// Man man2{"jaud"};
// std::cout << "man1 < man2: " << (man1 < man2) << std::endl;
// std::cout << "man1 > man2: " << (man1 > man2) << std::endl;
// std::cout << "man1 == man2: " << (man1 == man2) << std::endl;
// std::cout << "man1 != man2: " << (man1 != man2) << std::endl;
// std::cout << "man1 <= man2: " << (man1 <= man2) << std::endl;
// std::cout << "man1 >= man2: " << (man1 >= man2) << std::endl;

// std::cout << std::endl;

//}
// =====
/*
Explanation#

```

We have implemented, for the classes Apple and Man, the smaller operator separately (lines 51-52 and 63-64). The classes Man and Apple are publicly derived (line 48 and 60) from the class Relational<Man> and Relational<Apple>. We have implemented for classes of the kind Relational the greater than operator > (lines 11 – 16), the equality operator == (lines 18 – 23), the not equal operator != (lines 25 – 30), the less than or equal operator <= (line 32 – 37) and the greater than or equal operator >= (lines 39 – 44). The less than or equal (<=) and greater than or equal (>=) operators used the equality operator == (line 36 and 43). All these operators convert their operands: Derived const&: Derived const& d1 = static_cast<Derived const&>(op1).

In the main program, we have compared Apple and Man classes for all the above-mentioned operators.

Example 2: Static Polymorphism with CRTP#

```

*/
// =====
// templateCRTP.cpp

// #include <iostream>

// template <typename Derived>
// struct Base{
//   void interface(){
//     static_cast<Derived*>(this)->implementation();
//   }
//
//   void implementation(){
//     std::cout << "Implementation Base" << std::endl;
//   }
// };

// struct Derived1: Base<Derived1>{
//   void implementation(){
//     std::cout << "Implementation Derived1" << std::endl;
//   }
// };

// struct Derived2: Base<Derived2>{
//   void implementation(){
//     std::cout << "Implementation Derived2" << std::endl;
//   }
// };

```

```

// }
//};

// struct Derived3: Base<Derived3>{};

// template <typename T>
// void execute(T& base){
//   base.interface();
// }

// int main(){

// std::cout << std::endl;

// Derived1 d1;
// execute(d1);

// Derived2 d2;
// execute(d2);

// Derived3 d3;
// execute(d3);

// std::cout << std::endl;

// }
// =====
/*

```

Explanation#

We have used static polymorphism in the function template execute (lines 30-33). We invoked the method base.interface on each base argument. The method Base::interface, in lines (7-9), is the key point of the CRTP idiom. The methods dispatch to the implementation of the derived class: static_cast<Derived*>(this)->implementation(). That is possible because the method will be instantiated when called. At this point in time, the derived classes Derived1, Derived2, and Derived3 are fully defined. Therefore, the method Base::interface can use the details of its derived classes. Especially interesting is the method Base::implementation (lines 11-13). This method plays the role of a default implementation for the static polymorphism for the class Derived3 (line 28).

We'll solve a few exercises around CRTP in the next lesson.

```

*/
// =====
// =====
/*

```

Expression Templates#

Expression templates are “structures representing a computation at compile-time, which are evaluated only as needed to produce efficient code for the entire computation.”

Now we are at the center of lazy evaluation.

Lazy Evaluation#

The story about lazy evaluation in C++ is quite short. That will change in C++20, with the ranges library from Eric Niebler. Lazy evaluation is the default in Haskell. Lazy evaluation means that an expression is only evaluated when needed.

This strategy has two benefits.

Lazy evaluation helps you to save time and memory.

You can define an algorithm on infinite data structures. Of course, you can only ask for a finite number of values at runtime.

Advantages:

Creates a domain-specific language (DSL)

Avoidance of temporary data structures

Disadvantages:

Longer compile-times

Advanced programming technique (template metaprogramming)

What problem do expression templates solve? Thanks to expression templates, we can get rid of superfluous temporary objects in expressions. What do we mean by superfluous temporary objects? To demonstrate that, let's look at the implementation of the class MyVector below.

A First Naive Approach#

MyVector is a simple wrapper for an std::vector<T>. The wrapper class has two constructors (line 12 and 15), we have a size function which returns its size (line 18 - 20), and the reading index operator (line 23 - 25) and writing index access (line 27 - 29).

Given below is the naive vector implementation:

```
/*
// =====

// vectorArithmeticOperatorOverloading.cpp
// vectorArithmeticOperatorOverloading.cpp

// #include <iostream>
// #include <vector>

// template<typename T>
// class MyVector{
//   std::vector<T> cont;

// public:
//   // MyVector with initial size
//   MyVector(const std::size_t n) : cont(n){}

//   // MyVector with initial size and value
//   MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue){}

//   // size of underlying container
//   std::size_t size() const{
//     return cont.size();
//   }
```

```

// // index operators
// T operator[](const std::size_t i) const{
//   return cont[i];
// }

// T& operator[](const std::size_t i){
//   return cont[i];
// }

// };

/// function template for the + operator
// template<typename T>
// MyVector<T> operator+ (const MyVector<T>& a, const MyVector<T>& b){
//   MyVector<T> result(a.size());
//   for (std::size_t s= 0; s <= a.size(); ++s){
//     result[s]= a[s]+b[s];
//   }
//   return result;
// }

/// function template for the * operator
// template<typename T>
// MyVector<T> operator* (const MyVector<T>& a, const MyVector<T>& b){
//   MyVector<T> result(a.size());
//   for (std::size_t s= 0; s <= a.size(); ++s){
//     result[s]= a[s]*b[s];
//   }
//   return result;
// }

/// function template for << operator
// template<typename T>
// std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont){
//   std::cout << std::endl;
//   for (int i=0; i<cont.size(); ++i) {
//     os << cont[i] << ' ';
//   }
//   os << std::endl;
//   return os;
// }

// int main(){

//   MyVector<double> x(10,5.4);
//   MyVector<double> y(10,10.3);

//   MyVector<double> result(10);

//   result= x+x + y*y;

```

```
// std::cout << result << std::endl;
```

```
// }
```

```
// =====
```

```
/*
```

Thanks to the overloaded `+` operator (line 34 - 41), the overloaded `*` operator (line 44 - 51), and the overloaded output operator (line 54 - 62), the objects `x`, `y` and `result` feel like numbers.

Why is this implementation naive? The answer is in the expression `result = x+x + y*y`. In order to evaluate the expression, three temporary objects are needed to hold the result of each arithmetic subexpression.

The Issue#

```
result = x+x + y*y;
```

Arithmetic expressions create many temporary objects

svg viewer

The Solution#

svg viewer

The Idea#

The overloaded operators return proxy objects

The final assignment `result[i] = x[i] + x[i] + y[i] * y[i]` triggers the calculation

No temporary objects are necessary.

The Entire Magic#

```
result[i] = x[i] + x[i] + y[i] * y[i];
```

Thanks to the compiler explorer you can see that no temporary objects are created. The expression is lazily evaluated in place.

widget

A sharp view at the previous screenshot helps to unwrap the magic. Let's have a look at the sequence of calls:

widget

In the next lesson, we'll look at some examples of expression templates.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Example 1: Vector Arithmetic Based on Object-Orientation#

```
*/
```

```
// =====
```

```
// vectorArithmetricOperatorOverloading.cpp
```

```
// #include <iostream>
```

```
// #include <vector>
```

```
// template <typename T>
```

```
// class MyVector
```

```
// {
// std::vector<T> cont;

// public:
// // MyVector with initial size
// explicit MyVector(const std::size_t n) : cont(n) {}

// // MyVector with initial size and value
// MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue) {}

// // size of underlying container
// std::size_t size() const
// {
//   return cont.size();
// }

// // index operators
// T operator[](const std::size_t i) const
// {
//   return cont[i];
// }

// T &operator[](const std::size_t i)
// {
//   return cont[i];
// };

/// // function template for the + operator
// template <typename T>
// MyVector<T> operator+(const MyVector<T> &a, const MyVector<T> &b)
// {
//   MyVector<T> result(a.size());
//   for (std::size_t s = 0; s <= a.size(); ++s)
//   {
//     result[s] = a[s] + b[s];
//   }
//   return result;
// }

/// // function template for the * operator
// template <typename T>
// MyVector<T> operator*(const MyVector<T> &a, const MyVector<T> &b)
// {
//   MyVector<T> result(a.size());
//   for (std::size_t s = 0; s <= a.size(); ++s)
//   {
//     result[s] = a[s] * b[s];
//   }
//   return result;
// }
```

```

/// // function template for << operator
// template <typename T>
// std::ostream &operator<<(std::ostream &os, const MyVector<T> &cont)
//{
// os << std::endl;
// for (std::size_t i = 0; i < cont.size(); ++i)
// {
// os << cont[i] << ' ';
// }
// return os;
//}

// int main()
//{
// MyVector<double> x(10, 5.4);
// MyVector<double> y(10, 10.3);

// MyVector<double> result(10);

// result = x + x + y * y;

// std::cout << result << std::endl;
//}
// =====
/*
Explanation#
MyVector is a simple wrapper for an std::vector<T>. The wrapper class has two constructors (line 12 and 15), we have a size function which returns its size (line 18 - 20), and the reading index operator (line 23 - 25) and writing index access (line 27 - 29).

```

Thanks to the overloaded + operator (line 34 - 41), the overloaded * operator (line 44 - 51) and the overloaded output operator (line 54 - 61), the objects x, y, and result feel like numbers on lines 70 and 72.

Example 2: Vector Arithmetic Based on Expression Templates#

```

*/
// =====
// vectorArithmeticExpressionTemplates.cpp

// #include <cassert>
// #include <iostream>
// #include <vector>

// template<typename T, typename Cont = std::vector<T>>
// class MyVector{
// Cont cont;

// public:
// // MyVector with initial size
// MyVector(const std::size_t n) : cont(n){}

```

```

// // MyVector with initial size and value
// MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue){}

// // Constructor for underlying container
// MyVector(const Cont& other) : cont(other){}

// // assignment operator for MyVector of different type
// template<typename T2, typename R2>
// MyVector& operator=(const MyVector<T2, R2>& other){
//   assert(size() == other.size());
//   for (std::size_t i = 0; i < cont.size(); ++i) cont[i] = other[i];
//   return *this;
// }

// // size of underlying container
// std::size_t size() const{
//   return cont.size();
// }

// // index operators
// T operator[](const std::size_t i) const{
//   return cont[i];
// }

// T& operator[](const std::size_t i){
//   return cont[i];
// }

// // returns the underlying data
// const Cont& data() const{
//   return cont;
// }

// Cont& data(){
//   return cont;
// }

/// MyVector + MyVector
// template<typename T, typename Op1 , typename Op2>
// class MyVectorAdd{
//   const Op1& op1;
//   const Op2& op2;

// public:
//   MyVectorAdd(const Op1& a, const Op2& b): op1(a), op2(b){}

//   T operator[](const std::size_t i) const{
//     return op1[i] + op2[i];
//   }

```

```

// std::size_t size() const{
//   return op1.size();
// }
//};

/// // elementwise MyVector * MyVector
// template< typename T, typename Op1 , typename Op2 >
// class MyVectorMul {
//   const Op1& op1;
//   const Op2& op2;

// public:
//   MyVectorMul(const Op1& a, const Op2& b ): op1(a), op2(b) {}

//   T operator[](const std::size_t i) const{
//     return op1[i] * op2[i];
//   }

//   std::size_t size() const{
//     return op1.size();
//   }
// };

/// // function template for the + operator
// template<typename T, typename R1, typename R2>
// MyVector<T, MyVectorAdd<T, R1, R2> >
// operator+ (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
//   return MyVector<T, MyVectorAdd<T, R1, R2> >(MyVectorAdd<T, R1, R2 >(a.data(), b.data()));
// }

/// // function template for the * operator
// template<typename T, typename R1, typename R2>
// MyVector<T, MyVectorMul< T, R1, R2> >
// operator* (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
//   return MyVector<T, MyVectorMul<T, R1, R2> >(MyVectorMul<T, R1, R2 >(a.data(), b.data()));
// }

/// // function template for << operator
// template<typename T>
// std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont){
//   std::cout << std::endl;
//   for (int i=0; i<cont.size(); ++i) {
//     os << cont[i] << ' ';
//   }
//   os << std::endl;
//   return os;
// }

// int main(){

```

```

// MyVector<double> x(10,5.4);
// MyVector<double> y(10,10.3);

// MyVector<double> result(10);

// result= x+x + y*y;

// std::cout << result << std::endl;

// }
// =====
/*
Explanation#

```

The key difference between the first naive implementation and this implementation with expression templates is that the overloaded + and * operators return in case of the expression tree proxy objects. These proxies represent the expression tree (lines 91 and 98). The expression tree is only created but not evaluated. Lazy, of course. The assignment operator (lines 22 - 27) triggers the evaluation of the expression tree that needs no temporary objects.

```

*/
// =====
// =====
/*

```

Problem Statement 1#

Compare both programs containing + and * operators and study, in particular, the implementation based on expression templates.

Extend the example to expression templates by adding support subtraction and the division operators for the MyVector class.

```

*/
// =====
// #include <iostream>
// #include <vector>

// template<typename T>
// class MyVector{
//   std::vector<T> cont;

// public:
//   // MyVector with initial size
//   explicit MyVector(const std::size_t n) : cont(n){}

//   // MyVector with initial size and value
//   MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue){}

//   // size of underlying container
//   std::size_t size() const{
//     return cont.size();
//   }

//   // index operators
//   T operator[](const std::size_t i) const{

```

```

// return cont[i];
// }

// T& operator[](const std::size_t i){
//   return cont[i];
// }

// };

/// function template for the + operator
// template<typename T>
// MyVector<T> operator+ (const MyVector<T>& a, const MyVector<T>& b){
//   MyVector<T> result(a.size());
//   for (std::size_t s= 0; s <= a.size(); ++s){
//     result[s]= a[s]+b[s];
//   }
//   return result;
// }

/// function template for the * operator
// template<typename T>
// MyVector<T> operator* (const MyVector<T>& a, const MyVector<T>& b){
//   MyVector<T> result(a.size());
//   for (std::size_t s= 0; s <= a.size(); ++s){
//     result[s]= a[s]*b[s];
//   }
//   return result;
// }

/// function template for << operator
// template<typename T>
// std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont){
//   std::cout << std::endl;
//   for (int i=0; i<cont.size(); ++i) {
//     os << cont[i] << ' ';
//   }
//   os << std::endl;
//   return os;
// }

/// Implement subtraction and division operators here

```

```

// int main(){
//   // call these subtraction and division operators here
// }
// =====
/*

```

Problem Statement 2#

The solution of the previous example Vector Arithmetic Expression Templates is too laborious. In particular, the classes MyVectorAdd, MyVectorSub, MyVectorMul, and MyVectorDiv have a lot in common. Try to simplify the program.

```
*/  
// ======  
// #include <cassert>  
// #include <functional>  
// #include <iostream>  
// #include <vector>  
  
// template<typename T, typename Cont= std::vector<T> >  
// class MyVector{  
//   Cont cont;  
  
// public:  
//   // MyVector with initial size  
//   explicit MyVector(const std::size_t n) : cont(n){}  
  
//   // MyVector with initial size and value  
//   MyVector(const std::size_t n, const double initialValue) : cont(n, initialValue){}  
  
//   // Constructor for underlying container  
//   explicit MyVector(const Cont& other) : cont(other){}  
  
//   // assignment operator for MyVector of different type  
//   template<typename T2, typename R2>  
//   MyVector& operator=(const MyVector<T2, R2>& other){  
//     assert(size() == other.size());  
//     for (std::size_t i = 0; i < cont.size(); ++i) cont[i] = other[i];  
//     return *this;  
//   }  
  
//   // size of underlying container  
//   std::size_t size() const{  
//     return cont.size();  
//   }  
  
//   // index operators  
//   T operator[](const std::size_t i) const{  
//     return cont[i];  
//   }  
  
//   T& operator[](const std::size_t i){  
//     return cont[i];  
//   }  
  
//   // returns the underlying data  
//   const Cont& data() const{  
//     return cont;  
//   }  
  
//   Cont& data(){  
//     return cont;  
//   }
```

```

// };

// template<typename class Oper, typename T, typename Op1 , typename Op2>
// class MyVectorCalc{
//   const Op1& op1;
//   const Op2& op2;
//   Oper<T> oper;

// public:
//   MyVectorCalc(const Op1& a, const Op2& b): op1(a), op2(b) {}

//   T operator[](const std::size_t i) const{

//     return oper(op1[i], op2[i]);
//   }

//   std::size_t size() const{
//     return op1.size();
//   }
// };

/// // function template for the + operator
// template<typename T, typename R1, typename R2>
// MyVector<T, MyVectorCalc<std::plus, T, R1, R2> >
// operator+ (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
//   return MyVector<T, MyVectorCalc<std::plus, T, R1, R2> >(MyVectorCalc<std::plus, T, R1, R2 >(a.data(),
// b.data()));
// }

/// // function template for the - operator
// template<typename T, typename R1, typename R2>
// MyVector<T, MyVectorCalc<std::minus, T, R1, R2> >
// operator- (const MyVector<T, R1>& a, const MyVector<T, R2>& b) {
//   return MyVector<T, MyVectorCalc<std::minus, T, R1, R2> >(MyVectorCalc<std::minus, T, R1, R2
// >(a.data(), b.data()));
// }

/// // function template for the * operator
// template<typename T, typename R1, typename R2>
// MyVector<T, MyVectorCalc<std::multiplies, T, R1, R2> >
// operator* (const MyVector<T, R1>& a, const MyVector<T, R2>& b){
//   return MyVector<T, MyVectorCalc<std::multiplies, T, R1, R2> >(MyVectorCalc<std::multiplies, T, R1, R2
// >(a.data(), b.data()));
// }

/// // function template for the / operator
// template<typename T, typename R1, typename R2>
// MyVector<T, MyVectorCalc<std::divides, T, R1, R2> >
// operator/ (const MyVector<T, R1>& a, const MyVector<T, R2>& b) {
//   return MyVector<T, MyVectorCalc<std::divides, T, R1, R2> >(MyVectorCalc<std::divides, T, R1, R2
// >(a.data(), b.data()));

```

```

//}

/// function template for << operator
// template<typename T>
// std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont){
// std::cout << std::endl;
// for (int i=0; i<cont.size(); ++i) {
// os << cont[i] << ' ';
// }
// os << std::endl;
// return os;
//}

// int main(){

// MyVector<double> x(10,5.4);
// MyVector<double> y(10,10.3);

// MyVector<double> result(10);

// result = x + x + y * y - x + x + y / y;

// std::cout << result << std::endl;

// }
// =====
/*
Explanation#

```

In contrast to the previous example, in which each arithmetic operation such as addition (MyVectorAdd), subtraction (MyVectorSub), multiplication (MyVectorMult), or division (MyVectorDiv) is represented in a type, this improved version uses a generic binary operator (MyVectorCalc) in lines 52 – 69. This generic binary operator requires the concrete binary operator such as std::plus to become the concrete binary operator. The predefined function objects std::plus, std::minus, std::multiplies, and std::divides are part of the standard template library. You can think of them as a lambda-function representing the requested operation.

```

*/
// =====
// =====
/*

```

Idioms and Patterns: Policy and Traits

In this lesson, we will study about policy and traits in idioms and patterns.

We'll cover the following

Policy and Traits

Policy

Traits

Policy and Traits#

Policy#

A Policy is a generic function or class with adaptable behavior.

Policy parameters have typically default values.

This adaptable behavior is expressed in several type parameters, the so-called policy parameters. Due to different policy parameters, the concrete generic function or class behaves differently.

Typical examples for policies are the containers of the Standard Template Library such as std::vector, or std::unordered_map:

std::vector has a default policy for allocating memory, which is based on the type of the element:

std::allocator<T>

std::unordered_map has a default policy for generating the hash value (std::hash<Key>), comparing two keys (std::equal_to<Key>), and allocating memory (std::allocator<std::pair<const Key, T>>). The hash function and the comparison function are based on the key.

template<class T, class Allocator = std::allocator<T>>

class vector;

template<

 class Key,
 class T,
 class Hash = std::hash<Key>,
 class KeyEqual = std::equal_to<Key>,
 class Allocator = std::allocator<std::pair<const Key, T>>>

class unordered_map;

Traits#

Traits are class templates, which provide characteristics of a generic type.

template< class T >

struct is_integral;

template<T>

struct iterator_traits<T*> {
 using difference_type = std::ptrdiff_t;
 using value_type = T;
 using pointer = T*;
 using reference = T&;
 using iterator_category = std::random_access_iterator_tag;
};

Traits can extract one or more characteristics of a class template.

The function std::is_integral<T> from the type-trait library determines, if T is an integral type.

In the next lesson, we'll look at a few examples of policy and traits in idioms and patterns.

*/

// =====

// =====

/*

Example 1: Templates Policy#

*/

// =====

// PolicytemplatesPolicy.cpp

// #include <iostream>

// #include <unordered_map>

```

// struct MyInt{
//   explicit MyInt(int v):val(v){}
//   int val;
// };

// struct MyHash{
//   std::size_t operator()(MyInt m) const {
//     std::hash<int> hashVal;
//     return hashVal(m.val);
//   }
// };

// struct MyEqual{
//   bool operator () (const MyInt& fir, const MyInt& sec) const {
//     return fir.val == sec.val;
//   }
// };

// std::ostream& operator << (std::ostream& strm, const MyInt& myIn){
//   strm << "MyInt(" << myIn.val << ")";
//   return strm;
// }

// int main(){

//   std::cout << std::endl;

//   typedef std::unordered_map<MyInt, int, MyHash, MyEqual> MyIntMap;

//   std::cout << "MyIntMap: ";
//   MyIntMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

//   for(auto m : myMap) std::cout << '{' << m.first << ", " << m.second << "}";

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

// }
// =====
/*
Explanation#

```

The example uses the user-defined type `MyInt` as key for an `std::unordered_map`. To use `MyInt` as a key, `MyInt` must support the hash value and must be equal comparable. The classes `MyHash` (lines 11 – 16) and `MyEqual` (lines 18 – 22) provides the functionality for `MyInt` by delegating the job to the underlying `int`. The `typedef` in line 33 brings all together. In line 36, `MyInt` is used as a key in a `std::unordered_map`. The class `std::unoredered_map` in line 33 is an example of the policy class. The adaptable behavior is, in this case, the hash function and the equal function. Both policy parameters have default but can also be specified.

Example 2: Templates Traits#

```

*/
// =====
// TemplatesTraits.cpp

```

```

// #include <iostream>
// #include <type_traits>

// using namespace std;

// template <typename T>
// void getPrimaryTypeCategory(){

// cout << boolalpha << endl;

// cout << "is_void<T>::value: " << is_void<T>::value << endl;
// cout << "is_integral<T>::value: " << is_integral<T>::value << endl;
// cout << "is_floating_point<T>::value: " << is_floating_point<T>::value << endl;
// cout << "is_array<T>::value: " << is_array<T>::value << endl;
// cout << "is_pointer<T>::value: " << is_pointer<T>::value << endl;
// cout << "is_reference<T>::value: " << is_reference<T>::value << endl;
// cout << "is_member_object_pointer<T>::value: " << is_member_object_pointer<T>::value << endl;
// cout << "is_member_function_pointer<T>::value: " << is_member_function_pointer<T>::value << endl;
// cout << "is_enum<T>::value: " << is_enum<T>::value << endl;
// cout << "is_union<T>::value: " << is_union<T>::value << endl;
// cout << "is_class<T>::value: " << is_class<T>::value << endl;
// cout << "is_function<T>::value: " << is_function<T>::value << endl;
// cout << "is_lvalue_reference<T>::value: " << is_lvalue_reference<T>::value << endl;
// cout << "is_rvalue_reference<T>::value: " << is_rvalue_reference<T>::value << endl;

// cout << endl;

//}

// int main(){
// getPrimaryTypeCategory<void>();
// }

=====
/*
Explanation#
In the example above, we have defined the function getPrimaryTypeCategory in line 9 which takes a type T and determines which type category T belongs to. This example uses all 14 primary type categories of the type-trait library.

```

```

*/
=====
=====
/*

```

Idioms and Patterns: Tag Dispatching

In this lesson, we'll learn about tag dispatching in idioms and patterns.

We'll cover the following

Tag Dispatching

Tag Dispatching#

Tag Dispatching enables us to choose a function based on type characteristics.

The decision takes place at compile-time.

Traits make the decision based on specific properties of an argument.

The main benefit is performance.

The iterator categories from the Standard Template Library are a typical use-case:

```
struct input_iterator_tag{};  
struct output_iterator_tag{};  
struct forward_iterator_tag: public input_iterator_tag{};  
struct bidirectional_iterator_tag: public forward_iterator_tag{};  
struct random_access_iterator_tag: public bidirectional_iterator_tag{};  
In the concrete case, std::random_access_iterator is a refinement of std::bidirectional_iterator,  
std::bidirectional_iterator is a refinement of std::forward_iterator, and std::forward_iterator is a refinement of  
std::input_iterator.
```

In the next lesson, we'll look at an example of tag dispatching in idioms and patterns.

```
*/  
// ======  
// ======  
/*  
Example: Templates Tag Dispatching#  
*/  
// ======  
// TemplatesTagDispatching.cpp  
  
// #include <iterator>  
// #include <forward_list>  
// #include <list>  
// #include <vector>  
// #include <iostream>  
  
// template <typename InputIterator, typename Distance>  
// void advance_impl(InputIterator& i, Distance n, std::input_iterator_tag) {  
//     std::cout << "InputIterator used" << std::endl;  
//     while (n--) ++i;  
// }  
  
// template <typename BidirectionalIterator, typename Distance>  
// void advance_impl(BidirectionalIterator& i, Distance n, std::bidirectional_iterator_tag) {  
//     std::cout << "BidirectionalIterator used" << std::endl;  
//     if (n >= 0)  
//         while (n--) ++i;  
//     else  
//         while (n++) --i;  
// }  
  
// template <typename RandomAccessIterator, typename Distance>  
// void advance_impl(RandomAccessIterator& i, Distance n, std::random_access_iterator_tag) {
```

```

//     std::cout << "RandomAccessIterator used" << std::endl;
//     i += n;
// }

// template <typename InputIterator, typename Distance>
// void advance_(InputIterator& i, Distance n) {
//   typename std::iterator_traits<InputIterator>::iterator_category category;
//   advance_impl(i, n, category);
// }

// int main(){
// }
// =====
/*
Explanation#

```

The expression `std::iterator_traits::iterator_category category` in line 32 determines the iterator category at compile-time. Based on the iterator category, the most specific variant of the function template `advance_impl(i, n, category)` is used in line 33. Each container returns an iterator of the iterator category which corresponds to its structure.

```

*/
// =====
// =====
/*
Type Erasure#

```

Type Erasure enables you to use various concrete types through a single generic interface.

Type erasure is duck typing applied in C++

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
(James Whitcomb Riley)

Of course, you've already used type erasure in C++ or C. The C-ish way of type erasure is a void pointer; the C++ish way of type erasure is object-orientation.

Typical Use Case#

Type erasure can be performed with void pointers, object-orientation, or templates.

Case 1: Void Pointers#

`void qsort(void *ptr, std::size_t count, std::size_t size, cmp);`

Before the `qsort()`, consider using:

`int cmp(const void *a, const void *b);`

The comparison function `cmp` should return a

negative integer: the first argument is less than the second

zero: both arguments are equal

positive integer: the first argument is greater than the second

Thanks to the void pointer, `std::qsort` is generally applicable but also quite error-prone.

Maybe you want to sort a `std::vector<int>`, but you used a comparator for C-strings. The compiler cannot catch this error because the type of information was removed and we end up with undefined behavior.

In C++ we can do better.

Case 2: Object-Orientation#

Having a class hierarchy and using the `BaseClass` pointer instead of concrete types is one way to enable type erasure

```
std::vector<const BaseClass*> vec;
```

The `vec` object has a pointer to a constant `BaseClasses`.

Case 3: std::function#

`std::function` as a polymorphic function wrapper is a nice example of type erasure in C++. `std::function` can accept everything, which behaves like a function. To be more precise, this can be any callable such as a function, a function object, a function object created by `std::bind`, or just a lambda function.

Let's have a look at the difference between the three types listed above in the following table:

Technique	Each Datatype	Type-safe Interface	Easy to implment	Common Base Class
void*	Yes	No	Yes	No
Object-Orientation		No	Yes	Yes
Templates	Yes	Yes	No	No
*/				
// =====				
// class Object {				
// public:				
// ...				
// struct Concept {				
// virtual ~Concept() {}				
// };				
// template< typename T >				
// struct Model : Concept {				
// ...				
// };				
// std::shared_ptr<const Concept> object;				
//};				
// =====				
/*				

The small code snippet shows the structure of type erasure with templates.

The components are:

Object: Wrapper for a concrete type

object: Pointer to the concept

Concept: Generic interface

Model: Concrete class

Don't be irritated by the names Object, object, Concept, and Model. They are typically used for type erasure in the literature, so we stick to them. We will see a more explained version of this in the example in the next lesson.

```
/*
// =====
// =====
/*
Example 1: Type Erasure Using Object-Oriented Programming#
*/
// =====
// typeErasureOO.cpp

// #include <iostream>
// #include <string>
// #include <vector>

// struct BaseClass{
//     virtual std::string getName() const = 0;
// };

// struct Bar: BaseClass{
//     std::string getName() const override {
//         return "Bar";
//     }
// };

// struct Foo: BaseClass{
//     std::string getName() const override{
//         return "Foo";
//     }
// };

// void printName(std::vector<BaseClass*> vec){
//     for (auto v: vec) std::cout << v->getName() << std::endl;
// }

// int main(){
//     std::cout << std::endl;
//     Foo foo;
//     Bar bar;
//
//     std::vector<BaseClass*> vec{&foo, &bar};
//
//     printName(vec);
//
//     std::cout << std::endl;
// }
```

```
// =====
/*
Explanation#
The key point is that you can use instances of Foo or Bar instead of an instance for BaseClass.
std::vector<BaseClass*> (line 35) has a pointer to BaseClass. Well, actually it has two pointers to BaseClass
(derived) objects and it is a vector of such pointers. BaseClass is an abstract base class, which is used in line
23. Foo and Bar (lines 11 and 17) are the concrete classes.
```

What are the pros and cons of this implementation with object-orientation?

Pros:#

Type-safe

Easy to implement

Cons:#

Virtual dispatch

Intrusive because the derived class must know about its base class

Example 2: Type Erasure with Templates#

*/

```
// =====
// TypeErasure.cpp
```

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

// class Object {

// public:
//     template <typename T>
//     explicit Object(T&& obj): object(std::make_shared<Model<T>>(std::forward<T>(obj))) {}

//     std::string getName() const {
//         return object->getName();
//     }

//     struct Concept {
//         virtual ~Concept() {}
//         virtual std::string getName() const = 0;
//     };

//     template< typename T >
//     struct Model : Concept {
//         explicit Model(const T& t) : object(t) {}
//         std::string getName() const override {
//             return object.getName();
//         }
//     private:
//         T object;
//     };
}
```

```

// std::shared_ptr<const Concept> object;
// };

// void printName(std::vector<Object> vec){
//   for (auto v: vec) std::cout << v.getName() << std::endl;
// }

// struct Bar{
//   std::string getName() const {
//     return "Bar";
//   }
// };

// struct Foo{
//   std::string getName() const {
//     return "Foo";
//   }
// };

// int main(){

//   std::cout << std::endl;

//   std::vector<Object> vec{Object(Foo()), Object(Bar())};

//   printName(vec);

//   std::cout << std::endl;
// }
// =====
/*

```

Explanation#

First of all, the `std::vector` uses instances (line 57) of type `Object` and not pointers like in the previous object-oriented example. These instances can be created with arbitrary types because it has a generic constructor (line 12). The `Object` has the `getName` method (line 14) which is directly forwarded to the `getName` of the object. `object` is of type `std::shared_ptr<const Concept>`. The emphasis should not lay on the empty but on the virtual destructor in line 19. When the `std::shared_ptr<const Concept>` goes out of scope, the destructor is called. The static type of `object` is `std::shared_ptr<const Concept>` but the dynamic type `std::shared_ptr<Model<T>>`. The virtual destructor guarantees that the correct destructor is called. This means, in particular, the destructor of the dynamic type. The `getName` method of `Concept` is pure virtual (line 18), therefore, due to virtual dispatch, the `getName` method of `Model` (line 24) is used. In the end, the `getName` methods of `Bar` and `Foo` (lines 42 and 48) are applied in the `printName` function (line 37).

```

*/
// =====
// =====
/*

```

Overview

In this lesson, we'll look at an overview of what concepts are needed in C++20.

We'll cover the following

Too Specific Versus Too Generic

Too Specific

Narrowing Conversion

Integral Promotion

Too Generic

Concepts to Our Rescue

Advantages

Until C++20, we have two diametral ways to think about functions or user-defined types (classes). Functions or classes can be defined on specific types or on generic types. In the second case, we call them to function or class templates.

There exist two extremes while talking about functions in C++. Let's take a closer look at two function implementations:

Too Specific Versus Too Generic#

Too Specific

```
#include <iostream>
```

```
void needInt(int i){ // 1
    std::cout << i << std::endl;
}
```

```
int main(){
    double d{1.234};
    needInt(d);
}
```

Too Generic

```
#include <iostream>
```

```
template<typename T>
T gcd(T a, T b){
    if( b == 0 ){
        return a;
    }else{
        return gcd(b, a % b);
    }
}

int main(){
    std::cout << gcd(100, 10) << std::endl;
    std::cout << gcd(3.5, 4.0) << std::endl;
}
```

Too Specific#

It's quite a job to define for each specific type a function or a class. To avoid that burden, type conversion comes often to our rescue but it is also part of the problem.

Narrowing Conversion#

We have a function needInt(int a) in line 3 which we can be invoked with a double. Now narrowing conversion takes place.

```
/*
// =====
// #include <iostream>

// void needInt(int i){
//   std::cout << i << std::endl;
// }

// int main(){
//   double d{1.234};
//   needInt(d);
// }
// =====
/*
```

We assume that this is not the behavior we wanted. We started with a double and ended with an int.

But conversion also works the other way around.

Integral Promotion#

We have a user-defined type MyHouse. An instance of MyHouse can be constructed in two ways. When invoked without any argument in line 5, its attribute family is set to an empty string. This means the house is still empty. To quickly check if the house is empty or full, we implemented a conversion operator to bool in line 8.

```
/*
// =====
// #include <iostream>
// #include <string>

// struct MyHouse{
//   MyHouse() = default;
//   MyHouse(const std::string& fam): family(fam){}
//
//   operator bool(){ return !family.empty(); }
//
//   std::string family = "";
// };

// void needInt(int i){
//   std::cout << "int: " << i << std::endl;
// }

// int main(){
//   std::cout << std::boolalpha << std::endl;
```

```

// MyHouse firstHouse;
// if (!firstHouse){
//   std::cout << "The firstHouse is still empty." << std::endl;
// }

// MyHouse secondHouse("grimm");
// if (secondHouse){
//   std::cout << "Family grimm lives in secondHouse." << std::endl;
// }

// std::cout << std::endl;

// needInt(firstHouse);
// needInt(secondHouse);

// std::cout << std::endl;

// }
// =====
/*

```

Now, instances of MyHouse can be used, when an int is required. Strange!

Because of the overloaded operator bool in line 8, instances of MyHouse can be used as an int and can, therefore, be used in arithmetic expressions: auto res = MyHouse() + 5. This was not our intention! But we have noted it just for completeness. With C++11, we should declare those conversion operators as explicit. Therefore, implicit conversions will not take place.

Too Generic#

Generic functions or classes can be invoked with arbitrary values. If the values do not satisfy the requirements of the function or class, it is not a problem because you will get a compile-time error.

```

*/
// =====
// #include <iostream>

// template<typename T>
// T gcd(T a, T b){
//   if( b == 0 ){ return a; }
//   else{
//     return gcd(b, a % b);
//   }
// }

// int main(){

// std::cout << std::endl;

// std::cout << gcd(100, 10) << std::endl;

// std::cout << gcd(3.5, 4.0)<< std::endl;
// std::cout << gcd("100", "10") << std::endl;

```

```
// std::cout << std::endl;  
// }  
// ======  
/*  
What is the problem with this error message?
```

Of course, it is quite long and quite difficult to understand but our crucial concern is a different one. The compilation fails because neither double nor the C-string supports the % operator. This means the error is due to the failed template instantiation for double and C-string. This is too late and, therefore, really bad. No template instantiation for type double or C-strings should even be possible. The requirements for the arguments should be part of the function declaration and not a side-effect of an erroneous template instantiation.

Now concepts come to our rescue.

Concepts to Our Rescue#

With concepts, we get something in between. We can define functions or classes which act on semantic categories. Meaning, the arguments of functions or classes are neither too specific nor too generic but named sets of requirements such as Integral.

Advantages#

Express the template parameter requirements as part of the interface

Support the overloading of functions and the specialization of class templates

Generate drastically improved error messages by comparing the requirements of the template parameter with the template arguments

Use them as placeholders for generic programming

Empowers you to define your concepts

Can be used for all kinds of templates

In the next lesson, we'll learn about the history of C++ and talk about future concepts.

```
*/  
// ======  
// ======  
/*  
History
```

In this lesson, we'll learn about the history of C++ and talk about future concepts.

We'll cover the following

The Inspiration from Haskell

Classical Concepts

Concepts

The Advantages of Concepts

The Inspiration from Haskell#

Type classes are interfaces for similar types. If a type is a member of a type class, it has to have specific properties. Type classes play a similar role in generic programming as interfaces play in object-oriented programming. Here you can see a part of Haskell's type classes hierarchy.

Haskell Type Class Hierarchy

Haskell Type Class Hierarchy

Haskell's type classes build a hierarchy. The type class Ord is a subclass of the type class Eq. Therefore, instances of the type class Ord have to be members of the type class Eq and in addition support the comparison operators.

Classical Concepts#

The key idea of generic programming with templates is, to define functions and classes that can be used with different types. But it will often happen that you instantiate a template with the wrong type. The result may be a cryptic error message that is many pages long. It is sad but true: templates in C++ are known for this. Therefore, classical concepts were planned as one of the great features of C++11. They should allow you to specify constraints for templates that can be verified by the compiler. Due to their complexity, they were removed in July 2009 from the standard: "The C++0x concept design evolved into a monster of complexity." (Bjarne Stroustrup)

Concepts#

With C++20, we will get concepts. Although concepts are in the first implementations simplified classical concepts, they have a lot to offer.

The Advantages of Concepts#

Concepts

Empower the programmer to express their requirements as part of the interface directly.

Support the overloading of functions and the specialization of class templates based on the requirements of the template parameters.

Produce drastically improved error messages by comparing the requirements of the template parameter with the applied template arguments.

Can be used as placeholders for generic programming.

Empower you to define your own concepts.

We will get the benefits without additional compile-time or runtime time costs. Concepts are similar to Haskell's type classes. Concepts describe semantic categories and not syntactic restrictions. For types of the standard library, we get library concepts such as DefaultConstructible, MoveConstructible, CopyConstructible, MoveAssignable, CopyAssignable, or Destructible. For the containers, we get concepts such as ReversibleContainer, AllocatorAwareContainer, SequenceContainer, ContiguousContainer, AssociativeContainer, or UnorderedAssociativeContainer. You can read more about concepts and their constraints here: cppreference.com.

In the next lesson, we'll learn about the details of functions and classes with reference to concepts.

```
*/  
// ======  
// ======  
/*
```

Functions#

Using the concept Sortable.

Implicit

```
template<Sortable Cont>
```

```
void sort(Cont& container){  
...  
}
```

The container has to be Sortable.

The implicit version from the left is syntactic sugar to the explicit version:

Explicit

```
template<typename Cont>  
    requires Sortable<Cont>()  
void sort(Cont& container){  
    ...  
}
```

Sortable has to be a constant expression that is a predicate. That means that the expression has to be evaluable at compile-time and has to return a boolean.

If you invoke the sort algorithm with a container `lst` that is not sortable, you will get a unique error message from the compiler.

Usage:

```
std::list<int> lst = {1998, 2014, 2003, 2011};  
sort(lst); // ERROR: lst is no random-access container with <  
You can use concepts for all kind of templates.
```

Classes#

We can define a class template `MyVector` that will only accept objects as template arguments:

```
template<Object T>  
class MyVector{};
```

`MyVector<int> v1; // OK`

`MyVector<int&> v2 // ERROR: int& does not satisfy the constraint Object`

Now, the compiler complains that the reference (`int&`) is not an object. `MyVector` can be further adjusted.

A reference is not an object.

Methods of a Class#

```
template<Object T>  
class MyVector{  
    ...  
    requires Copyable<T>()  
    void push_back(const T& e);  
    ...  
};
```

Now the method `push_back` from `MyVector` requires that the template argument has to be copy-able. The concepts have to be placed before the method declaration.

More Requirements#

A template can have more than one requirement for its template parameters.

```
template <SequenceContainer S,
```

```
EqualityComparable<value_type<S>> T>
Iterator_type<S> find(S&& seq, const T& val){
...
}
```

The function template `find` has two requirements. On one hand, the container has to store its elements in a linear arrangement (`SequenceContainer`), on the other hand, the elements of the container have to be equally comparable: `EqualityComparable<value_type<S>>`.

Overloading#

Concepts support the overloading of functions.

```
template<InputIterator I>
void advance(I& iter, int n){...}
```

```
template<BidirectionalIterator I>
void advance(I& iter, int n){...}
```

```
template<RandomAccessIterator I>
void advance(I& iter, int n){...}
```

```
std::list<int> lst{1,2,3,4,5,6,7,8,9};
std::list<int>::iterator i = lst.begin();
std::advance(i, 2); // BidirectionalIterator
```

The function template `std::advance` puts its iterator `n` positions further. Depending if the iterator is a forward, a bidirectional, or a random access iterator, different function templates will be used. In case of a `std::list`, the `BidirectionalIterator` will be chosen.

Concepts also support the specialization of class templates.

Specialization#

```
template<typename T>
class MyVector{};
```

```
template<Object T>
class MyVector{};
```

```
MyVector<int> v1; // Object T
```

```
MyVector<int&> v2; // typename T
```

For `MyVector<int&> v2`, the compiler uses the general template in the first line; on the contrary, the compiler uses for `MyVector<int>` the specialization `template<Object T> class MyVector{}`.

`MyVector<int&>` goes to the unconstrained template parameter.

`MyVector<int>` goes to the constrained template parameter.

In the next lesson, we'll study the placeholder syntax.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Placeholder Syntax

Let's learn about placeholder syntax in this lesson.

We'll cover the following

Placeholder Syntax: auto

Inconsistency in C++14

Constrained and Unconstrained Placeholders

Example:

With auto, C++11 has unconstrained placeholders. We can use concepts in C++20 as constrained placeholders. Decisive quantum leap does not look so thrilling at the first glimpse. C++ templates will become easy to use C++ features.

According to our definition, C++98 is not a consistent language. By consistent, we mean that you have to apply a few rules to derived C++ syntax from it. C++11 is something in between. For example, we have consistent rules like initializing all with curly braces (see { } - Initialization). Of course, even C++14 has a lot of features where we miss a consistent principle. One of the favorites is the generalized lambda function.

Placeholder Syntax: auto#

```
auto genLambdaFunction= [](auto a, auto b) {
    return a < b;
};
```

```
template <typename T, typename T2> // 3
auto genFunction(T a, T2 b){ // 4
    return a < b;
}
```

Inconsistency in C++14#

By using the placeholder auto for the parameter a and b, the generalized lambda function becomes - in a magic way - a function template. We know, genLambdaFunction is a function object that has an overloaded call operator which accepts two type parameters. The genFunction is also a function template. But wouldn't it be nice to define a function template by just using auto in a function definition? This would be consistent but is not possible. Hence, we have to use a lot more syntax (line 3 and 4). That syntax is often too difficult for a lot of C++ programmer.

Exactly that inconsistency will be removed with the placeholder syntax. Therefore, we have a new simple principle and C++ will become - according to my definition - a lot easier to use.

Generic Lambdas introduced a new way to define templates.

Constrained and Unconstrained Placeholders#

We will get unconstrained and constrained placeholders. auto is an unconstrained placeholder because a with auto defined variable can be of any type. A concept is a constrained placeholder because it can only be used to define a variable that satisfies the concept.

General Rule: Constrained Concepts can be used where unconstrained templates (auto) are usable.

Let's define and use a simple concept before we dig into the details.

Example:#

```
*/
// =====
// conceptsPlaceholder.cpp
```

```

// #include <iostream>
// #include <type_traits>
// #include <vector>

// template<typename T>
// concept bool Integral(){
//     return std::is_integral<T>::value;
// }

// Integral getIntegral(auto val){
//     return val;
// }

// int main(){

//     std::cout << std::boolalpha << std::endl;

//     std::vector<int> myVec{1, 2, 3, 4, 5};
//     for (Integral& i: myVec) std::cout << i << " ";
//     std::cout << std::endl;

//     Integral b= true;
//     std::cout << b << std::endl;

//     Integral integ= getIntegral(10);
//     std::cout << integ << std::endl;

//     auto integ1= getIntegral(10);
//     std::cout << integ1 << std::endl;

//     std::cout << std::endl;

// }
// =====
/*

```

We have defined in line 8 the concept `Integral`. The concept `Integral` will evaluate to true if the predicate `std::is_integral<T>::value` returns true for `T`. `std::is_integral<T>` is a function of the type-trait library. The functions of the type-trait library enable, amongst other things, that we can check types at compile-time. Hence, we iterate over `Integral`'s in the range-based for-loop in line 21 and the variable `b` in line 24 has to be `Integral`. Our usage of concepts goes on in lines 27 and 30. We required in line 27 that the return type of `getIntegral` (line 12) has to fulfill the concept `Integral`. We're not so strict in line 30. Here we're fine with an unconstrained placeholder.

```

*/
// =====
// =====
/*

```

Predefined Concepts

Let's dive deep into predefined concepts of C++20 in this lesson.

We'll cover the following

Predefined Concept

Concepts Definition: Variable Concepts

Concepts Definition: Function Concepts

Concepts TS

Concepts Draft

The Concept Equal

Before moving on to predefined concepts, let's get to know about Syntactic Sugar.

Syntactic Sugar: This is from Wikipedia: In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language sweeter for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

Predefined Concept#

We should use the predefined concepts. cppreference.com gives a great introduction to predefined concepts. Here are a few of them:

Core language concepts

Same

DerivedFrom

ConvertibleTo

Common

Integral

Signed Integral

Unsigned Integral

Assignable

Swappable

Comparison concepts

Boolean

EqualityComparable

StrictTotallyOrdered

Object concepts

Destructible

Constructible

DefaultConstructible

MoveConstructible

Copy Constructible

Movable

Copyable

Semi-regular

Regular

Callable concepts

Callable

RegularCallable

Predicate

Relation

StrictWeakOrder

There are two ways to define concepts: variable concepts and function concepts. If we use a variable template for our concept, it's called a variable concept; in the second case a function concept.

Concepts Definition: Variable Concepts#

```
template<typename T>
concept bool Integral =
    std::is_integral<T>::value;
}
```

We have defined the concept `Integral` by using a variable template. Variable templates are new with C++14 and declare a family of variables. The concept `Integral` will evaluate to true if the predicate `std::is_integral<T>::value` returns true for `T`. `std::is_integral<T>` is a function of the type-trait library. The functions of the type-trait library enable, amongst other things, that we can check types at compile time.

Concepts Definition: Function Concepts#

The original syntax of Concepts Technical Specification (Concepts TS) was a bit adjusted to the proposed Draft C++20 Standard. Here is the original syntax from the Concepts TS, which is used in this course.

Concepts TS#

```
template<typename T>
concept bool Equal(){
    return requires(T a, T b) {
        { a == b } -> bool;
        { a != b } -> bool;
    };
}
```

`Integral` is a variable concept and `Equal` is a function concept. Both return a boolean.

The type parameter `T` fulfills the variable concept `Integral` if `std::is_integral<T>::value` returns true.

The type parameter `T` fulfills the function concept `Equal` if there are overloaded operators `==` and `!=` for `T` that returns a boolean.

Concepts Draft#

The proposed syntax for C++20 is even more concise.

```
template<typename T>
concept Equal =
    requires(T a, T b) {
        { a == b } -> bool;
        { a != b } -> bool;
    };
}
```

`T` fulfills the function concept if `==` and `!=` are overloaded and return a boolean.

The Concept Equal#

```
*/
// =====
// conceptsDefintionEqual.cpp
```

```
// #include <iostream>
```

```
// template<typename T>
// concept bool Equal(){
//     return requires(T a, T b) {
```

```

// { a == b } -> bool;
// { a != b } -> bool;
// };
// }

// bool areEqual(Equal a, Equal b){
// return a == b;
// }

// struct WithoutEqual{
// bool operator==(const WithoutEqual& other) = delete;
// };

// struct WithoutUnequal{
// bool operator!=(const WithoutUnequal& other) = delete;
// };

// int main(){

// std::cout << std::boolalpha << std::endl;

// std::cout << "areEqual(1, 5): " << areEqual(1, 5) << std::endl;

// bool res = areEqual(WithoutEqual(), WithoutEqual());

// bool res2 = areEqual(WithoutUnequal(), WithoutUnequal());

// std::cout << std::endl;

// }
// =====
/*

```

We used the concept Equal in the (generic) function areEqual (line 13 to 15) and that's not so exciting.

What's more interesting is if we use the class WithoutEqual and WithoutUnequal. We set for both the == or respectively the != operator to delete. The compiler complains immediately that both types do not fulfill the concept.

Let's have a look at the screenshot of the error taken from the machine:

```

*/
// =====
// =====
/*

```

Define your Concepts: Equal and Ord

In this lesson, we'll define the concepts Equal and Ord for C++.

We'll cover the following

Eq versus Equal

Haskell's Type Class Ord

The Concept Equal and the Concept Ord

Eq versus Equal#

The Type Class Eq (Haskell)

```
class Eq a where
```

```
  (==) :: a -> a -> Bool
```

```
  (/=) :: a -> a -> Bool
```

The Concept Equal (C++)

```
template <typename T>
concept bool Equal(){
    return requires(T a, T b){
        { a == b } -> bool;
        { a != b } -> bool;
    };
}
```

Let's have a closer look at Haskell's type class Eq. Eq requires from its instances, that

they have equal == and unequal /= operation that returns a Bool.

both take two arguments (a -> a) of the same type.

If you compare Haskell's type class with C++'s concept, you see the similarity.

Of course, the instances are the concrete types such as int.

Haskell Type Class

Haskell Type Class

Now we have two questions in mind if we look at Haskell's type hierarchy above. How is the definition of the type class Ord in Haskell and can we model the inheritance relation in C++?

Haskell's Type Class Ord#

```
class Eq a => Ord a where
```

```
    compare :: a -> a -> Ordering
```

```
    (<) :: a -> a -> Bool
```

```
    (≤) :: a -> a -> Bool
```

```
    (>) :: a -> a -> Bool
```

```
    (≥) :: a -> a -> Bool
```

```
    max :: a -> a -> a
```

Each type supporting Ord must support Eq.

The most interesting point about the typeclass Ord is the first line of its definition. An instance of the typeclass Ord has to be already an instance of the typeclass Eq. Ordering is an enumeration having the values EQ, LT, and GT.

The Concept Equal and the Concept Ord#

Let's define the corresponding concepts in C++.

The concept Equal

```
template <typename T>
```

```
concept bool Equal(){
```

```
    return requires(T a, T b){
```

```

{ a == b } -> bool;
{ a != b } -> bool;
};

}

```

The Concept Ord

```

template <typename T>
concept bool Ord(){
    return requires(T a, T b){
        requires Equal<T>();
        { a <= b } -> bool;
        { a < b } -> bool;
        { a > b } -> bool;
        { a >= b } -> bool;
    };
}

```

To make the job a little bit easier, we ignored the requirements compare and max from Haskell's type class in the concept Ord. The key point about the concept is that the line requires Equal<T>(). Here we required that the type parameter T has to fulfill the requirement Equal. If we use more requirements such as in the definition of the concept Equal, each requirement from top to bottom will be checked. That will be done in a short-circuiting evaluation. So, the first requirement returning false will end the process.

```

*/
// =====
// conceptsDefintionOrd.cpp
// #include <iostream>
// #include <unordered_set>

// template<typename T>
// concept bool Equal(){
//     return requires(T a, T b){
//         { a == b } -> bool;
//         { a != b } -> bool;
//     };
// }

// template <typename T>
// concept bool Ord(){
//     return requires(T a, T b){
//         requires Equal<T>();
//         { a <= b } -> bool;
//         { a < b } -> bool;
//         { a > b } -> bool;
//         { a >= b } -> bool;
//     };
// }

// bool areEqual(Equal a, Equal b){
//     return a == b;
// }

```

```

// Ord getSmaller(Ord a, Ord b){
//   return (a < b) ? a : b;
// }

// int main(){

// std::cout << std::boolalpha << std::endl;

// std::cout << "areEqual(1, 5): " << areEqual(1, 5) << std::endl;

// std::cout << "getSmaller(1, 5): " << getSmaller(1, 5) << std::endl;

// std::unordered_set<int> firSet{1, 2, 3, 4, 5};
// std::unordered_set<int> secSet{5, 4, 3, 2, 1};

// std::cout << "areEqual(firSet, secSet): " << areEqual(firSet, secSet) << std::endl;

// auto smallerSet= getSmaller(firSet, secSet);

// std::cout << std::endl;

// }
// =====
/*
Equality and inequality are defined for the data types int and std::unordered_set.

```

What would happen, when we uncomment line 45 and compare firSet and secSet. To remind you, the type of both variables is std::unordered_set. This says very explicitly that they don't support an ordering.

Let's check what happens when we run this code:

```

*/
// =====
// =====
/*

```

Define your Concept: Regular and SemiRegular

In this lesson, we'll gain an understanding of the important predefined concepts: Regular and SemiRegular.

We'll cover the following

References are not Regular

The first question we have to answer is quite obvious. What is a Regular or a SemiRegular type? Our answer is based on the proposal p0898. We assume you may have already guessed it that Regular and SemiRegular are concepts, which are defined by other concepts. Given is the list of all concepts.

Regular

DefaultConstructible

CopyConstructible, CopyAssignable

MoveConstructible, MoveAssignable

Destructible

Swappable

EqualityComparable

SemiRegular

Regular without EqualityComparable

The term Regular goes back to the father of the Standard Template Library Alexander Stepanov. He introduced the term in his book Fundamentals of Generic Programming. Here is a short excerpt. It's quite easy to remember the eight concepts used to define a regular type. First, there is the well-known rule of six:

Default constructor: X()

Copy constructor: X(const X&)

Copy assignment: operator=(const X&)

Move constructor: X(X&&)

Move assignment: operator=(X&&)

Destructor: ~X()

Second, add the Swappable and EqualityComparable concepts to it. There is a more informal way to say that a type T is regular: T behaves like an int.

To get SemiRegular, we have to subtract EqualityComparable from Regular.

References are not Regular#

Thanks to the type-trait library the following program checks at compile-time if int& is a SemiRegular type.

Let's have a look at the example:

```
/*
// =====
// #include <iostream>
// #include <type_traits>

// int main(){
//   std::cout << std::boolalpha << std::endl;

//   std::cout << "std::is_default_constructible<int&>::value: " << std::is_default_constructible<int&>::value
//   << std::endl;
//   std::cout << "std::is_copy_constructible<int&>::value: " << std::is_copy_constructible<int&>::value <<
//   std::endl;
//   std::cout << "std::is_copy_assignable<int&>::value: " << std::is_copy_assignable<int&>::value <<
//   std::endl;
//   std::cout << "std::is_move_constructible<int&>::value: " << std::is_move_constructible<int&>::value <<
//   std::endl;
//   std::cout << "std::is_move_assignable<int&>::value: " << std::is_move_assignable<int&>::value <<
//   std::endl;
//   std::cout << "std::is_destructible<int&>::value: " << std::is_destructible<int&>::value << std::endl;
//   std::cout << std::endl;
//   //std::cout << "std::is_swappable<int&>::value: " << std::is_swappable<int&>::value << std::endl;    //
// requires C++17

//   std::cout << std::endl;
//}
// =====
/*
```

First of all, the function `std::is_swappable` requires C++17 that's why we have commented it out otherwise it will give an error. We see that the reference such as `int&` is not default-constructible. The output shows that a reference is not `SemiRegular` and, therefore, not `Regular`. To check, if a type is `Regular` at compile-time, we need a function `isEqualityComparable` which is not part of the type-trait library. Let's define it.

```
/*
// =====
// #include <experimental/type_traits>
// #include <iostream>

// template<typename T>
// using equal_comparable_t = decltype(std::declval<T&>() == std::declval<T&>());

// template<typename T>
// struct isEqualityComparable:
//     std::experimental::is_detected<equal_comparable_t, T>{};

// struct EqualityComparable { };
// bool operator == (EqualityComparable const&, EqualityComparable const&) { return true; }

// struct NotEqualityComparable { };

// int main() {
//     std::cout << std::boolalpha << std::endl;
//
//     std::cout << "isEqualityComparable<EqualityComparable>::value: " <<
//             isEqualityComparable<EqualityComparable>::value << std::endl;
//
//     std::cout << "isEqualityComparable<NotEqualityComparable>::value: " <<
//             isEqualityComparable<NotEqualityComparable>::value << std::endl;
//
//     std::cout << std::endl;
//
// }
// =====
/*
```

The new feature is in the experimental namespace in line 1. Line 9 is a crucial one. It detects at compile-time if the expression in line 5 is valid for the type `T`. The type-trait `isEqualityComparable` works for an `EqualityComparable` (line 11) and a `NotEqualityComparable` (line 14) type. Only `EqualityComparable` returns true because we overloaded the Equal-Comparison operator.

Now, we have all the ingredients to define `Regular` and `SemiRegular`. Here are our new type-trait.

```
/*
// =====
// #include <experimental/type_traits>
// #include <iostream>

// template<typename T>
// using equal_comparable_t = decltype(std::declval<T&>() == std::declval<T&>());

// template<typename T>
```

```

// struct isEqualityComparable:
//     std::experimental::is_detected<equal_comparable_t, T>
//     {};

// template<typename T>
// struct isSemiRegular: std::integral_constant<bool,
//     std::is_default_constructible<T>::value &&
//     std::is_copy_constructible<T>::value &&
//     std::is_copy_assignable<T>::value &&
//     std::is_move_constructible<T>::value &&
//     std::is_move_assignable<T>::value &&
//     std::is_destructible<T>::value &&
//     std::is_swappable<T>::value >{};

// template<typename T>
// struct isRegular: std::integral_constant<bool,
//     isSemiRegular<T>::value &&
//     isEqualityComparable<T>::value >{};

// int main(){

//     std::cout << std::boolalpha << std::endl;

//     std::cout << "isSemiRegular<int>::value: " << isSemiRegular<int>::value << std::endl;
//     std::cout << "isRegular<int>::value: " << isRegular<int>::value << std::endl;

//     std::cout << std::endl;

//     std::cout << "isSemiRegular<int&>::value: " << isSemiRegular<int&>::value << std::endl;
//     std::cout << "isRegular<int&>::value: " << isRegular<int&>::value << std::endl;

//     std::cout << std::endl;

// }

// =====
/*
The usage of the new type-trait isSemiRegular and isRegular makes the main program quite readable.

```

In this chapter, we have learned about the future concept of C++20. Let's move on to the next lesson for the conclusion of this course.

```

*/
// =====
// =====
/*

```

Concurrency in modern c++

Introduction

Learn what is concurrency and why it is important!

We'll cover the following

Concurrency

C++ concurrency in action: real-world applications

Example 1: Email server

Example 2: Web crawler

We started this path from the basics of C++ language and went on to explore the power of OOP and templates in C++. Further, we learned the new features which have been introduced in the C++ Standard Library. Let's learn how we can fully utilize the resources on our machine using concurrency and multithreading.

In the modern tech ecosystem, concurrency has become an essential skill for all C++ programmers. As programs continue to get more complex, computers are designed with more CPU cores to match.

The best way for you to make use of these multicore machines is the coding technique of concurrency.

Concurrency#

Concurrency can be termed as the process of executing several tasks at the same time. We can achieve this by either executing tasks on a time shared manner or by executing tasks in parallel on multiple cores.

Different approaches for concurrency

C++ concurrency in action: real-world applications#

Multithreaded programs are common in modern business systems, in fact, you likely use some more complex versions of the above programs in your everyday life.

Example 1: Email server#

One example could be an email server, returning mailbox contents when requested by a user. With this program, we have no way of knowing how many people will be requesting their mail at any given time.

By using a thread pool, the program can process as many user requests as possible without risking an overload.

As above, each thread would execute a defined function, such as receiving the mailbox of the identifier passed in, void request_mail (string user_name).

Example 2: Web crawler#

Another example could be a web crawler, which downloads pages across the web. By using multithreading, the developer would ensure that the web crawler is using as much of the hardware's capability as possible to download and index multiple pages at once.

Based on just these two examples, we can see the breadth of functions in which concurrency can be advantageous. With the number of CPU cores in each computer increasing by the year, concurrency is certain to remain an invaluable asset in the arsenal of the modern developer.

*/

// =====

```
// =====  
/*
```

C++11 and C++14: The Foundation

This lesson demonstrates the foundation and overview of concurrency in C++11 and C++14.

We'll cover the following

C++11 and C++14: The foundation

Memory Model

Atomics

Memory Model#

The foundation of multithreading is a well-defined memory model. This memory model has to deal with the following aspects:

Atomic operations: operations that can be performed without interruption.

Partial ordering of operations: the sequence of operations that must not be reordered.

Visible effects of operations: guarantees when operations on shared variables are visible in other threads.

The C++ memory model was inspired by its predecessor: the Java memory model. Unlike the Java memory model, however, C++ allows us to break the constraints of sequential consistency, which is the default behavior of atomic operations. Sequential consistency provides two guarantees.

The instructions of a program are executed in source code order

There is a global order for all operations on all threads

The memory model is based on atomic operations on atomic data types (short atomics).

Atomics#

C++ has a set of simple atomic data types: booleans, characters, numbers, and pointers in many variants. You can define your own atomic data type with the class template `std::atomic`. Atomics establish synchronization and ordering constraints that can also hold for non-atomic types. The standardized threading interface is the core of concurrency in C++.

Multithreading in C++

Illustrating the fundamentals of multithreading in C++.

We'll cover the following

Threads

Shared Data

Mutexes

Locks

Thread-safe Initialization of Data

Thread Local Data

Condition Variables

Tasks

Multithreading in C++ consists of threads, synchronization primitives for shared data, thread-local data, and tasks.

Threads#

A `std::thread` represents an independent unit of program execution. The executable unit, which is started immediately, receives its work package as a callable unit. A callable unit can be a named function, a function object, or a lambda function.

The creator of a thread is responsible for its lifecycle. The executable unit of the new thread ends with the end of the callable. Either the creator waits until the created thread `t` is done (`t.join()`), or the creator detaches itself from the created thread: (`t.detach()`). A thread `t` is joinable if no operation `t.join()` or `t.detach()` was performed on it. A joinable thread calls `std::terminate` in its destructor and the program terminates.

A thread that is detached from its creator is typically called a daemon thread because it runs in the background. A `std::thread` is a variadic template. This means that it can receive an arbitrary number of arguments by copy or reference; either the callable or the thread can get the arguments.

Shared Data#

You have to coordinate access to a shared variable if more than one thread is using it at the same time and the variable is mutable (non-const). Reading and writing a shared variable at the same time is a data race, and therefore, undefined behavior. Coordinating access to a shared variable is achieved with mutexes and locks in C++.

Mutexes#

A mutex (mutual exclusion) guarantees that only one thread can access a shared variable at any given time. A mutex locks and unlocks the critical section that the shared variable belongs to. C++ has five different mutexes; they can lock recursively, tentatively, and with or without time constraints. Even mutexes can share a lock at the same time.

Locks#

You should encapsulate a mutex in a lock to release the mutex automatically. A lock implements the RAII idiom by binding a mutex's lifetime to its own. C++ has a `std::lock_guard` for the simple cases, and a `std::unique_lock` / `std::shared_lock` for the advanced use-cases, such as the explicit locking or unlocking of the mutex respectively.

Thread-safe Initialization of Data#

If shared data is read-only, it's sufficient to initialize it in a thread-safe way. C++ offers various ways to achieve this including using constant expression, a static variable with block scope, or using the function `std::call_once` in combination with the flag `std::once_flag`.

Thread Local Data#

Declaring a variable as thread-local ensures that each thread gets its own copy; therefore, there is no shared variable. The lifetime of a thread local data is bound to the lifetime of its thread.

Condition Variables#

Condition variables enable threads to be synchronized via messages. One thread acts as the sender while another one acts as the receiver of the message, where the receiver blocks wait for the message from the sender. Typical use cases for condition variables are producer-consumer workflows. A condition variable can

be either the sender or the receiver of the message. Using condition variables correctly is quite challenging; therefore, tasks are often the easier solution.

Tasks#

Tasks have a lot in common with threads. While you explicitly create a thread, a task is simply a job you start. The C++ runtime will automatically handle, in the simple case of std::async, the lifetime of the task.

Tasks are like data channels between two communication endpoints. They enable thread-safe communication between threads: the promise at one endpoint puts data into the data channel, and the future at the other endpoint picks the value up. The data can be a value, an exception, or simply a notification. In addition to std::async, C++ has the class templates std::promise and std::future that give you more control over the task.

```
*/  
// ======  
// ======  
/*
```

Case Studies

A short introduction to some pertinent case studies used in this course to apply the theory portions.

We'll cover the following

Calculating the Sum of a Vector

Thread-Safe Initialization of a Singleton

Ongoing Optimization with CppMem

After presenting the theory of the memory model and the multithreading interface, I will apply the theory in a few case studies.

Calculating the Sum of a Vector#

Calculating the sum of a vector can be done in various ways. You can do it sequentially, or concurrently with maximum and minimum sharing of data. The performance numbers differ drastically.

Thread-Safe Initialization of a Singleton#

Thread-safe initialization of a singleton is the classical use-case for thread-safe initialization of a shared variable. There are many ways to do it, with varying performance characteristics.

Ongoing Optimization with CppMem#

I will start with a small program and successively improve it, and verify each step of my process of ongoing optimization with CppMem. CppMem is an interactive tool for exploring the behavior of small code snippets using the C++ memory model.

```
*/  
// ======  
// ======  
/*
```

With C++17, concurrency in C++ has drastically changed - particularly for the parallel algorithms of the Standard Template Library (STL). C++11 and C++14 only provide the basic building blocks for concurrency.

These tools are suitable for a library or framework developer, but not for the application developer. Multithreading in C++11 and C++14 will become an assembly language for concurrency in C++17!

Execution Policy#

With C++17, most of the STL algorithms will be available in a parallel implementation. This makes it possible for you to invoke an algorithm with a so-called execution policy. This policy specifies whether the algorithm runs sequentially std::seq, in parallel std::par, or in parallel with additional vectorization std::par_unseq.

New Algorithms#

In addition to the 69 algorithms that are available for parallel or vectorized executions in overloaded versions, we get eight new algorithms. These new ones are well suited for parallel reducing, scanning, or transforming.

```
*/  
// ======  
// ======  
/*
```

C++20: The Concurrent Future

A short introduction to concurrent future and all the new libraries and techniques which are predicted to be launched with C++20.

```
*/  
// ======  
// ======  
/*
```

It is difficult to make predictions, especially about the future (Niels Bohr). Therefore, I will make statements about the concurrency features of C++20.

Atomic Smart Pointers#

The smart pointers std::shared_ptr and std::weak_ptr have a conceptional issue in concurrent programs: they share an intrinsically mutable state. Therefore, they are prone to data races and, thus, lead to undefined behavior. std::shared_ptr and std::weak_ptr guarantee that the incrementing or decrementing of the reference counter is an atomic operation and the resource will be deleted exactly once, but neither of them can guarantee that the access to its resource is atomic. The new atomic smart pointers std::atomic_shared_ptr and std::atomic_weak_ptr solve this issue.

Extended Futures#

Tasks called promises and futures, introduced in C++11, have a lot to offer. However, they also have a drawback: tasks are not composable into powerful workflows. That limitation will not hold for the extended futures in C++20. Therefore, an extended future becomes ready when its predecessor (then) becomes ready, when_any one of its predecessors becomes ready, or when_all of its predecessors become ready.

Latches and Barriers#

C++14 has no semaphores, i.e. the variables used to control access to a limited number of resources. Worry no longer, because C++20 proposes latches and barriers. You can use latches and barriers for waiting at a synchronization point until the counter becomes zero. The difference between latches and barriers is that an std::latch can only be used once, while an std::barrier and std::flex_barrier can be used more than once. In contrast to a std::barrier, a std::flex_barrier can adjust its counter after each iteration.

Coroutines#

Coroutines are functions that can suspend and resume their execution while maintaining their state.

Coroutines are often the preferred approach to implement cooperative multitasking in operating systems, event loops, infinite lists, or pipelines.

Transactional Memory#

The transactional memory is based on the ideas underlying transactions in database theory. A transaction is an action that provides the first three properties of ACID database transactions: Atomicity, Consistency, and Isolation. The durability that is characteristic for databases will not hold for the proposed transactional memory in C++. The new standard will have transactional memory in two flavors: synchronized blocks and atomic blocks. Both will be executed in total order and behave as if they were protected by a global lock. In contrast to synchronized blocks, atomic blocks cannot execute transaction-unsafe code.

Task Blocks#

Task Blocks implement the fork-join paradigm in C++. The following graph illustrates the key idea of a task block: you have a fork phase in which you launch tasks and a join phase in which you synchronize them.

```
*/  
// ======  
// ======  
/*
```

The Contract

This lesson briefs the start of the C++ memory model with an introduction to the contract between the programmer and the system.

We'll cover the following

The Contract

First Level

Second Level

Third Level

The foundation of multithreading is a well-defined memory model. From the reader's perspective, it consists of two aspects. On the one hand, there is the enormous complexity of it, which often contradicts our intuition. On the other hand, it helps a lot to get a deeper insight into the multithreading challenges. In the first approach, I want to give you a mental model. That being said, the C++ memory model defines a contract.

The Contract#

This contract is between the programmer and the system. The system consists of the compiler that generates machine code and the processor that executes the machine code, and it includes the different caches that store the state of the program. The result is - in the good case - a well-defined executable that is fully optimized for the hardware platform. To be precise, there is not only a single contract, but a fine-grained set of contracts; i.e. the weaker the rules are that the programmer has to follow, the more potential there is for the system to generate a highly optimized executable.

There is a rule of thumb: the stronger the contract, the fewer liberties for the system to generate an optimized executable. Sadly, the other way around will not work. When the programmer uses an extremely

weak contract or memory model, there are a lot of optimization choices. The consequences are that the program is only manageable by a handful of worldwide recognized experts worldwide, and neither you nor I am likely to belong to that group. Roughly speaking, there are three contract levels in C++11.

widget

First Level#

Before C++11, there was only one contract. The C++ language specification did not include multithreading or atomics. The system only knew about one control flow and, therefore, there were only restricted opportunities to optimize the executable. The key point of the system was to guarantee—for the programmer—that the observed behavior of the program corresponded to the sequence of the instructions in the source code. Of course, this means that there was no memory model. Instead, there was the concept of a sequence point. Sequence points are points in the program, at which the effects of all instructions preceding it must be observable. The start or the end of the execution of a function are sequence points. When you invoke a function with two arguments, the C++ standard makes no guarantee about which arguments will be evaluated first, so the behavior is unspecified. The reason is straightforward: the comma operator is not a sequence point and this will not change in C++.

Second Level#

With C++11 everything has changed. C++11 is the first standard aware of multiple threads. The C++ memory model that was heavily inspired by the Java memory model is the reason for the well-defined behavior of threads. However, the C++ memory model goes – as always – a few steps further. The programmer has to obey a few rules in dealing with shared variables to get a well-defined program. The program is undefined if there exists at least one data race. As I already mentioned, you have to be aware of data races if your threads share mutable data. Tasks are a lot easier to use than threads or condition variables.

Third Level#

With atomics, we enter the domain of the experts. This will become more evident, the more we weaken the C++ memory model. We often talk about lock-free programming when we use atomics. I spoke in this subsection about the weak and strong rules; indeed, the sequential consistency is called the strong memory model, and the relaxed semantic is called the weak memory model.

```
*/  
// ======  
// ======  
/*
```

The Foundation & Challenges

This lesson briefs the foundation and challenges in the C++ memory model.

We'll cover the following

The Foundation

The Challenges

The Foundation#

The C++ memory model has to deal with the following points:

Atomic operations: operations that can be performed without interruption.

Partial ordering of operations: sequences of operations that must not be reordered.

Visible effects of operations: guarantees when operations on shared variables are visible to other threads.

The foundation of the contract are operations on atomics that have two characteristics: They are by definition atomic or indivisible, and they create synchronization and order constraints on the program execution. These

synchronization and order constraints will also hold for operations on non-atomics. On one hand, an operation on an atomic is always atomic; on the other hand, you can tailor the synchronizations and order constraints to your needs.

The Challenges#

The more we weaken the memory model, the more we will change our focus towards other things:

More optimization potential for the system

The possible number of control flows of the program increases exponentially

Domain for the experts

Breaks our intuition of the control flow

Areas for micro-optimization

To deal with multithreading, we should be an expert. In case we want to deal with atomics (sequential consistency), we should open the door to the next level of expertise. What will happen when we talk about the acquire-release semantic or relaxed semantic? We'll advance one step higher to (or deeper into) the next expertise level.

widget

Now, we dive deeper into the C++ memory model and start with lock-free programming. On our journey, I will write about atomics and their operations. Once we are done with the basics, the different levels of the memory model will follow. The starting point will be the straightforward sequential consistency, the acquire-release semantic will follow, and the not-so-intuitive relaxed semantic will be the end point of our journey.

Let's start with atomics in the next lesson. See ya!

```
*/  
// ======  
// ======  
/*
```

Strong Memory Model

This lesson gives a brief overview of the strong memory model regarding concurrency in C++.

We'll cover the following

Strong Memory Model

Atomics are the base of the C++ memory model. By default, the strong version of the memory model is applied to the atomics; therefore, it makes a lot of sense to understand the features of the strong memory model. You can see from the subsection on Contract: The Challenges, with the strong memory model I refer to sequential consistency, and with the weak memory model I refer to relaxed semantic.

Strong Memory Model#

Java 5.0 got its current memory model in 2004, and C++ got its model in 2011. Before that, Java had an erroneous memory model and C++ had no memory model. Those who think this is the endpoint of a long process are completely wrong. The foundations of multithreaded programming are 40 to 50 years old; Leslie Lamport defined the concept of sequential consistency in 1979.

Sequential consistency provides two guarantees:

The instructions of a program are executed in source code order.

There is a global order of all operations on all threads.

Before I dive deeper into these two guarantees, I want to explicitly emphasize that these statements only hold for atomics, but still influence non-atomics. This graphic shows two threads: each thread stores its variable x or y respectively, loads the other variable y or x, and stores them in the variable res1 or res2.

Because the variables are atomic, the operations are executed atomically; by default, sequential consistency applies. The question is, in which order can the statements be executed?

The first guarantee of the sequential consistency is that the instructions will be executed in the order defined in the source code. This is easy; no store operation can overtake a load operation.

The second guarantee of the sequential consistency is that all instructions of all threads have to follow a global order. In the case listed above, it means that thread 2 sees the operations of thread 1 in the same order in which thread 1 executes them. This is the key observation: thread 2 sees all operations of thread 1 in the source code order of thread 1. The same holds from the perspective of thread 1. You can think about characteristic number 2 as a global clock which all threads have to obey. The global clock is the global order. Each time the clock makes a tick, one atomic operation takes place, but you never know which one.

We are not yet done with our riddle! We still need to look at the different interleaving executions of the two threads. So, the following six interleavings of the two threads are possible.

That was easy, right? That was sequential consistency, also known as the Strong Memory Model.

```
*/  
// ======  
// ======  
/*
```

Weak Memory Model

This lesson gives a brief overview of the weak memory model regarding concurrency in C++.

We'll cover the following

With Relaxed Semantic

With Acquire-Release Semantic

Let's refer to the contract between the programmer and the system.

The programmer uses atomics in this particular example; He obeys his part of the contract. The system guarantees well-defined program behavior without data races. In addition to that, the system can execute the four operations in each combination. If the programmer uses the relaxed semantic, the pillars of the contract dramatically change. On one hand, it is a lot more difficult for the programmer to understand possible interleavings of the two threads. On the other hand, the system has a lot more optimization possibilities.

With Relaxed Semantic#

With the relaxed semantic - also called Weak Memory Model - many more combinations of the four operations are possible. The counter-intuitive behavior is that thread 1 can see the operations of thread 2 in a different order, so there is no view of a global clock. From the perspective of thread 1, it is possible that the operation `res2= x.load()` overtakes `y.store(1)`. It is even possible that thread 1 or thread 2 do not perform their operations in the order defined in the source code. For example, thread 2 can first execute `res2= x.load()` and then `y.store(1)`.

With Acquire-Release Semantic#

There are a few models between the sequential consistency and the relaxed-semantic. The most important one is the acquire-release semantic. With the acquire-release semantic, the programmer has to obey weaker rules than with sequential consistency. In contrast, the system has more optimization possibilities. The acquire-release semantic is the key to a deeper understanding of synchronization and partial ordering in multithreading programming because the threads will be synchronized at specific synchronization points in the code. Without these synchronization points, it's not possible to have well-defined behavior of threads, tasks, or condition variables possible.

In the last section, I introduced sequential consistency as the default behavior of atomic operations. But what does that mean? You can specify the memory order for each atomic operation. If no memory order is specified, sequential consistency is applied - meaning that the flag `std::memory_order_seq_cst` is implicitly applied to each operation on an atomic. So, the following piece of code is equivalent to the latter piece of code:

```
x.store(1);
res = x.load();
```

is equivalent to the following piece of code:

```
x.store(1, std::memory_order_seq_cst);
res = x.load(std::memory_order_seq_cst);
*/
// =====
// =====
/*
```

The Atomic Flag

This lesson gives an overview of the atomic flag, which is used from the perspective of concurrency in C++.

The atomic flag, i.e. `std::atomic_flag`, has a very simple interface. Its `clear` method enables you to set its value to false; with the `test_and_set` method you can set the value back to true. There is no method to exclusively ask for the current value. To use `std::atomic_flag` it must be initialized to false with the constant `ATOMIC_FLAG_INIT`. `std::atomic_flag` has two outstanding properties.

`std::atomic_flag` is:

the only lock-free atomic. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress. the building block for higher level thread abstractions.

The only lock-free atomic? The remaining more powerful atomics can provide their functionality by using a mutex internally according to the C++ standard. These remaining atomics have a method called `is_lock_free` to check if the atomic uses a mutex internally. On the popular microprocessor architectures, I always get the answer true. That being said, my implementation internally uses no mutex; you should be aware of this and check it on your target system if you want to program lock-free.

The interface of `std::atomic_flag` is powerful enough to build a spinlock. With a spinlock, you can protect a critical section as you would with a mutex. The spinlock will not passively wait, in contrast to a mutex, until it gets it to lock. It will eagerly ask for the lock to get access to the critical section. It fully utilizes the CPU and does waste CPU cycles.

The example shows the implementation of a spinlock with the help of std::atomic_flag.

```
/*
// =====
// spinLock.cpp
// #include <iostream>
// #include <atomic>
// #include <thread>

// class Spinlock{
//   std::atomic_flag flag;
// public:
//   Spinlock(): flag(ATOMIC_FLAG_INIT){}

//   void lock(){
//     while( flag.test_and_set() );
//   }

//   void unlock(){
//     flag.clear();
//   }
// };

// Spinlock spin;

// void workOnResource(){
//   spin.lock();
//   // shared resource
//   spin.unlock();
//   std::cout << "Work done" << std::endl;
// }

// int main(){
//   std::thread t(workOnResource);
//   std::thread t2(workOnResource);

//   t.join();
//   t2.join();

// }
```

Both threads t and t2 (lines 31 and 32) are competing for the critical section. For simplicity, the critical section in line 24 consists only of a comment. How does it work? The class Spinlock has the methods lock and unlock—similar to a mutex. In addition to this, the constructor of Spinlock initializes the std::atomic_flag to false (line 9).

If thread t is going to execute the function workOnResource, the following scenarios can happen:

Thread t gets the lock because the lock invocation was successful. The lock invocation is successful if the initial value of the flag in line 12 is false. In this case, thread t sets it in an atomic operation to true. The value true is the value the while loop returns to thread t2 if it tries to get the lock. So thread t2 is caught in the rat race.

Thread t2 has no possibility to set the value of the flag to false, so t2 must wait until thread t1 executes the unlock method and sets the flag to false (lines 15 - 17).

Thread t doesn't get the lock, so we are in scenario 1 with swapped roles.

I want you to focus your attention on the method `test_and_set` of `std::atomic_flag`. The method `test_and_set` consists of two operations: reading and writing. It's key that both operations are performed in one atomic operation. If not, we would have a read and a write on the shared resource (line 24). That is-- by definition-- a data race, and the program has undefined behavior.

```
/*
// =====
// =====
/*
```

Spinlock vs. Mutex

It's very interesting to compare the active waiting of a spinlock with the passive waiting of a mutex. Let's continue our discussing from the previous lesson and make a comparison between these two.

What will happen to the CPU load if the function `workOnResource` locks the spinlock for 2 seconds (lines 24 - 26)?

```
/*
// =====
// spinLockSleep.cpp

// #include <iostream>
// #include <atomic>
// #include <thread>

// class Spinlock{
//   std::atomic_flag flag;
// public:
//   Spinlock(): flag(ATOMIC_FLAG_INIT){}

//   void lock(){
//     while( flag.test_and_set() );
//   }

//   void unlock(){
//     flag.clear();
//   }
// };

// Spinlock spin;

// void workOnResource(){
//   spin.lock();
//   std::this_thread::sleep_for(std::chrono::milliseconds(2000));
//   spin.unlock();
//   std::cout << "Work done" << std::endl;
// }

// int main(){
//   std::thread t(workOnResource);
```

```
// std::thread t2(workOnResource);

// t.join();
// t2.join();

// }
```

```
/*
According to the theory, one of the four cores of PC will be fully utilized, and that's exactly what happened as the load of one core reaches 100% on my PC. Each time, a different core performs busy waiting.
```

Now, I will use a mutex instead of a spinlock. Let's see what happens.

```
/*
// =====
// mutex.cpp
// #include <iostream>
// #include <mutex>
// #include <thread>

// std::mutex mut;

// void workOnResource(){
//   mut.lock();
//   std::this_thread::sleep_for(std::chrono::milliseconds(5000));
//   mut.unlock();
//   std::cout << "Work done" << std::endl;
// }

// int main(){

//   std::thread t(workOnResource);
//   std::thread t2(workOnResource);

//   t.join();
//   t2.join();

// }
```

```
/*
Although I executed the program several times, I did not observe a significant load on any of the cores.
```

In the next lesson, let's go one step further from the basic building block `std::atomic_flag` to the more advanced atomics: the class template `std::atomic`. The various partial and full specializations for bools, integral types, and pointers provide a more powerful interface than `std::atomic_flag`. The downside is that you do not have the guarantee that these specializations are lock-free.

```
/*
// =====
// =====
/*
```

```
std::atomic<bool>
```

This lesson gives an overview of `std::atomic<bool>` which is used from the perspective of concurrency in C++.

We'll cover the following

std::atomic<bool>

Let's start with the full specializations for bool: std::atomic<bool>

std::atomic<bool>#

std::atomic<bool> has a lot more to offer than std::atomic_flag. It can explicitly be set to true or false.

atomic is not volatile

What does the keyword volatile in C# and Java have in common with the keyword volatile in C++? Nothing! It's so easy in C++. That is the difference between volatile and std::atomic.

volatile: is for special objects, on which optimized read or write operations are not allowed

std::atomic: defines atomic variables, which are meant for a thread-safe reading and writing

It's so easy, but the confusion starts exactly here. The keyword volatile in Java and C# has the meaning of std::atomic in C++, i.e. volatile has no multithreading semantic in C++.

volatile is typically used in embedded programming to denote objects which can change independently of the regular program flow. One example is an object which represents an external device (memory-mapped I/O). Because these objects can change independently of the regular program flow and their value will directly be written into main memory, no optimized storing in caches takes place.

This is sufficient to synchronize two threads, so I can implement a kind of condition variable with an std::atomic<bool>. Therefore, let's first use a condition variable.

```
/*
// =====
// conditionVariable.cpp

// #include <condition_variable>
// #include <iostream>
// #include <thread>
// #include <vector>

// std::vector<int> mySharedWork;
// std::mutex mutex_;
// std::condition_variable condVar;

// bool dataReady{false};

// void waitingForWork(){
//     std::cout << "Waiting " << std::endl;
//     std::unique_lock<std::mutex> lck(mutex_);
//     condVar.wait(lck, []{ return dataReady; });
//     mySharedWork[1] = 2;
//     std::cout << "Work done " << std::endl;
// }
```

```

// void setDataReady(){
//   mySharedWork = {1, 0, 3};
//   {
//     std::lock_guard<std::mutex> lck(mutex_);
//     dataReady = true;
//   }
//   std::cout << "Data prepared" << std::endl;
//   condVar.notify_one();
// }

// int main(){

// std::cout << std::endl;

// std::thread t1(waitingForWork);
// std::thread t2(setDataReady);

// t1.join();
// t2.join();

// for (auto v: mySharedWork){
//   std::cout << v << " ";
// }

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

// }
// =====
/*

```

Let me say a few words about the program. For an in-depth discussion of condition variables, read the chapter condition variables in this course.

Thread t1 waits in line 17 for the notification of thread t2. Both threads use the same condition variable condVar and synchronize on the same mutex mutex_. How does the workflow run?

thread t2

prepares the work package mySharedWork = {1, 0, 3}

set the non-atomic boolean dataReady to true

send its notification condVar.notify_one

thread t1

waits for the notification condVar.wait(lck, []{ return dataReady; }) while holding the lock lck

continues its work mySharedWork[1] = 2 after getting the notification

The boolean dataReady, which thread t2 sets to true and thread t1 checks in the lambda-function []{ return dataReady; }, is a kind of memory for the stateless condition variable. Condition variables may be victim to two phenomena:

spurious wakeup: the receiver of the message wakes up, although no notification happened

lost wakeup: the sender sends its notification before the receiver gets to a wait state.

Now, here's the pendant with std::atomic<bool>

*/

```

// =====

```

```

// atomicCondition.cpp

// #include <atomic>
// #include <chrono>
// #include <iostream>
// #include <thread>
// #include <vector>

// std::vector<int> mySharedWork;
// std::atomic<bool> dataReady(false);

// void waitingForWork(){
//     std::cout << "Waiting " << std::endl;
//     while (!dataReady.load()){
//         std::this_thread::sleep_for(std::chrono::milliseconds(5));
//     }
//     mySharedWork[1] = 2;
//     std::cout << "Work done " << std::endl;
// }

// void setDataReady(){
//     mySharedWork = {1, 0, 3};
//     dataReady = true;
//     std::cout << "Data prepared" << std::endl;
// }

// int main(){

//     std::cout << std::endl;

//     std::thread t1(waitingForWork);
//     std::thread t2(setDataReady);

//     t1.join();
//     t2.join();

//     for (auto v: mySharedWork){
//         std::cout << v << " ";
//     }

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

// }
// =====
/*
Push versus Pull Principle

```

Obviously I cheated a little. There is one key difference between the synchronization of the threads with a condition variable and `std::atomic<bool>`. The condition variable notifies the waiting thread (`condVar.notify()`) that it should proceed with its work. The waiting thread with `std::atomic<bool>` checks if the sender is done with its work (`dataRead = true`).

The condition variable notifies the waiting thread (push principle) while the atomic boolean repeatedly asks for the value (pull principle).

`std::atomic<bool>` and the other full or partial specializations of `std::atomic` support the bread and butter of all atomic operations: `compare_exchange_strong` and `compare_exchange_weak`.

`compare_exchange_strong` and `compare_exchange_weak`

`compare_exchange_strong` has the syntax: `bool compare_exchange_strong(T& expected, T& desired)`. Because this operation compares and exchanges its values in one atomic operation, it is often called compare and swap (CAS). This kind of operation is available in many programming languages and is the foundation of non-blocking algorithms. Of course, the behavior may vary a little.

`atomicValue.compare_exchange_strong(expected, desired)` has the following behavior:

If the atomic comparison of `atomicValue` with `expected` returns true, `atomicValue` will be set in the same atomic operation to `desired`.

If the comparison returns false, `expected` will be set to `atomicValue`.

The reason the operation `compare_exchange_strong` is called strong is apparent. There is also a method `compare_exchange_weak`, although the weak version can spuriously fail. This means that although `*atomicValue == expected` holds, the weak variant returns false; so, you have to check the condition in a loop: `while (!atomicValue.compare_exchange_weak(expected, desired))`. The weak form exists because of performance, i.e. when called in a loop it can run faster on some platforms.

CAS operations are open for the so-called ABA problem. This means you read a value twice and each time it returns the same value A; therefore you conclude that nothing changed in between. But you overlooked that the value may have changed to B in between readings.

```
/*
// =====
// =====
/*
```

User Defined Atomics

This lesson gives an overview of user-defined atomics used from the perspective of concurrency in C++.

There are a lot of deep restrictions on a user-defined type `MyType` if you use it for an atomic type `std::atomic<MyType>`. These restrictions are on the type `MyType`, but also on the operations that `std::atomic<MyType>` can perform.

Here are the restrictions for `MyType` to become an atomic type:

The copy assignment operator for `MyType` (all base classes of `MyType` and all non-static members of `MyType`) must be trivial. This means that you must not define the copy assignment operator but request it by default from the compiler.

`MyType` must not have virtual methods or virtual base classes.

`MyType` must be bitwise comparable so that the C functions `memcpy` or `memcmp` can be applied.

Check the type properties at compile time

The type properties on `MyType` can be checked at compile time, by using the following functions:

`std::is_trivially_copy_constructible`, `std::is_polymorphic` and `std::is_trivial`. All these functions are part of the very powerful type-trait library.

The user-defined atomic type `std::atomic<MyType>` supports only a limited interface.

```
/*
// =====
// =====
/*
All Atomic Operations
```

This lesson lists all the useful atomic operations which are quite handy while working with concurrency in C++.

To get the full picture, here is a list of all atomic operations depending on the atomic type.

Method	<code>atomic_flag</code>	<code>atomic<bool></code>	<code>atomic<T*></code>	<code>atomic<integral></code>	<code>atomic<user defined></code>
<code>test_and_set</code>	yes	no	no	no	no
<code>clear</code>	yes	no	no	no	no
<code>is_lock_free</code>	yes	no	no	no	no
<code>load</code>	no	yes	yes	yes	yes
<code>store</code>	no	yes	yes	yes	yes
<code>exchange</code>	no	yes	yes	yes	yes
<code>compare_exchange_strong</code>			<code>compare_exchange_weak</code>		no yes yes yes yes
<code>fetch_add, +=</code>	<code>fetch_sub, -=</code>	no	no	yes yes	no
<code>fetch_or, =</code>	<code>fetch_and, &=</code>	<code>fetch_xor, ^=</code>	no no	no yes	no
<code>++, --</code>	no	yes	yes	no	

```
/*
// =====
// =====
/*
Atomic Operations on std::shared_ptr
```

We'll cover the following

Atomic Smart Pointers

There are specializations for the atomic operations load, store, compare, and exchange for an `std::shared_ptr`. By using the explicit variant, you can even specify the memory model. Here are the free atomic operations for `std::shared_ptr`:

```
std::atomic_is_lock_free(std::shared_ptr)
std::atomic_load(std::shared_ptr)
std::atomic_load_explicit(std::shared_ptr)
std::atomic_store(std::shared_ptr)
std::atomic_store_explicit(std::shared_ptr)
std::atomic_exchange(std::shared_ptr)
std::atomic_exchange_explicit(std::shared_ptr)
std::atomic_compare_exchange_weak(std::shared_ptr)
std::atomic_compare_exchange_strong(std::shared_ptr)
std::atomic_compare_exchange_weak_explicit(std::shared_ptr)
std::atomic_compare_exchange_strong_explicit(std::shared_ptr)
```

For the details, have a look at cppreference.com. Now it is quite easy to modify a shared pointer that is bound by reference in a thread-safe way.

```
/*
// =====
// std::shared_ptr<int> ptr = std::make_shared<int>(2011);
```

```

// for (auto i =0;i<10;i++){
//   std::thread([&ptr]{
//     auto localPtr= std::make_shared<int>(2014);
//     std::atomic_store(&ptr, localPtr);
//   }).detach();
//}
// =====
/*

```

The update of the `std::shared_ptr` `ptr` in the expression `auto localPtr= std::make_shared<int>(2014)` is thread-safe. All is well? NO! Finally, we need atomic smart pointers.

Atomic Smart Pointers#

That is not the end of the story for atomic smart pointers. With C++20, there is a high probability that we can expect two new smart pointers: `std::atomic_shared_ptr` and `std::atomic_weak_ptr`. For the impatient reader, here are the details of the upcoming atomic smart pointers.

Atomics and their atomic operations are the basic building blocks for the memory model. They establish synchronization and ordering constraints that hold for both atomics and non-atomics. Let's have a deeper look into the synchronization and ordering constraints.

```

*/
// =====
// =====
/*

```

Introduction

This lesson introduces the concepts of synchronization and ordering constraints in C++.

We'll cover the following

Variants of the Memory Model

Kind of Atomic Operation

You cannot configure the atomicity of an atomic data type, but you can accurately adjust the synchronization and ordering constraints of atomic operations. This possibility is unique to C++, as it's not possible in C#'s or Java's memory model.

There are six different variants of the memory model in C++. The key question is what are their characteristics?

Variants of the Memory Model#

We already know C++ has six variants of the memory models. The default for atomic operations is `std::memory_order_seq_cst`; this expression stands for sequential consistency. In addition, you can explicitly specify one of the other five. So what does C++ have to offer?

```

*/
// =====
// enum memory_order{
//   memory_order_relaxed,
//   memory_order_consume,
//   memory_order_acquire,
//   memory_order_release,
//   memory_order_acq_rel,

```

```
// memory_order_seq_cst
// }
// =====
/*
```

To classify these six memory models, it helps to answer two questions:

Which kind of atomic operations should use which memory model?

Which synchronization and ordering constraints are defined by the six variants?

My plan is quite simple: I will answer both questions.

Kind of Atomic Operation#

There are three different kinds of operations:

Read operation: `memory_order_acquire` and `memory_order_consume`

Write operation: `memory_order_release`

Read-modify-write operation: `memory_order_acq_rel` and `memory_order_seq_cst`

`memory_order_relaxed` defines no synchronization and ordering constraints; therefore, it does not fit in this taxonomy. The following table orders the atomic operations based on their reading and/or writing characteristics.

Operation	read	write	read-modify-write
-----------	------	-------	-------------------

<code>test_and_set</code>		yes	
---------------------------	--	-----	--

<code>clear</code>	yes		
--------------------	-----	--	--

<code>is_lock_free</code>	yes		
---------------------------	-----	--	--

<code>load</code>	yes		
-------------------	-----	--	--

<code>store</code>	yes		
--------------------	-----	--	--

<code>exchange</code>		yes	
-----------------------	--	-----	--

<code>compare_exchange_strong</code>	<code>compare_exchange_weak</code>		yes
--------------------------------------	------------------------------------	--	-----

<code>fetch_add, +=</code>	<code>fetch_sub, -=</code>	yes	
----------------------------	----------------------------	-----	--

<code>fetch_or, =</code>	<code>fetch_and, &=</code>	<code>fetch_xor, ^=</code>	yes
---------------------------	--------------------------------	----------------------------	-----

<code>++, --</code>	yes		
---------------------	-----	--	--

If you use an atomic operation `atomVar.load()` with a memory model that is designed for a write or read-modify-write operation, the write part has no effect. The result is that operation `atomVar.load(std::memory_order_acq_rel)` is equivalent to operation `atomVar.load(std::memory_order_acquire);`

operation `atomVar.load(std::memory_order_release)` is equivalent to

`atomVar.load(std::memory_order_relaxed).`

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Types of Synchronization & Ordering Constraints

This lesson introduces the types of synchronization and ordering constraints in C++.

We'll cover the following

Modification Order Consistency

There are, roughly speaking, three different types of synchronization and ordering constraints in C++:

Sequential consistency: memory_order_seq_cst

Acquire-release: memory_order_consume, memory_order_acquire, memory_order_release and
memory_order_acq_rel

Relaxed: memory_order_relaxed

Modification Order Consistency#

While the sequential consistency establishes a global order between threads, the acquire-release semantic establishes an ordering between read and write operations on the same atomic variable with different threads. The relaxed semantic only guarantees that operations on one specific data type in the same thread cannot be reordered. This guarantee is called modification order consistency, but other threads can see this operation in a different order.

The different memory models and their effects on atomic and non-atomic operations make the C++ memory model an interesting and challenging topic. In the next lessons, let us discuss the synchronization and ordering constraints of the sequential consistency, the acquire-release semantic, and the relaxed semantics.

```
*/  
// ======  
// ======  
/*  
Sequential Consistency  
Familiarize yourself with sequential consistency in C++.
```

We'll cover the following

Explanation:

Let us dive deeper into sequential consistency. The key for sequential consistency is that all operations on all threads obey a universal clock. This universal clock makes it quite intuitive to think about it.

The intuitiveness of the sequential consistency comes with a price. The downside is that the system has to do a lot of work to keep the threads in sync. The following program synchronizes the producer and the consumer thread with the help of sequential consistency.

```
*/  
// ======  
// producerConsumer.cpp  
  
// #include <atomic>  
// #include <iostream>  
// #include <string>  
// #include <thread>  
  
// std::string work;  
// std::atomic<bool> ready(false);  
  
// void consumer(){  
//   while(!ready.load()){}  
//   std::cout << work << std::endl;  
// }  
  
// void producer(){
```

```

// work= "done";
// ready=true;
// }

// int main(){
// std::thread prod(producer);
// std::thread con(consumer);
// prod.join();
// con.join();
// }
// =====
/*

```

The output of the program is not very exciting. Because of sequential consistency, the program execution is totally deterministic; its output is always “done”. The graphic depicts the sequence of operations. The consumer thread waits in the while-loop until the atomic variable ready is set to true. When this happens, the consumer thread will continue its work.

```

*/
// =====
// =====
/*

```

It is quite easy to understand that the program will always return “done”, as we only have to use the two characteristics of sequential consistency. On one hand, both threads execute their instructions in source code order; On the other hand, each thread sees the operations of the other thread in the same order. Both threads follow the same global timing. This timing will also hold - with the help of the while(!ready.load()){} loop - for the synchronization of the producer and the consumer thread.

Explanation:#

I can explain the reasoning a lot more formally by using the terminology of the memory model. Here is the formal version:

work= "done" is sequenced-before ready = true
 \Rightarrow work= "done" happens-before ready = true

while(!ready.load()){} is sequenced-before std::cout << work << std::endl
 \Rightarrow while(!ready.load()){} happens-before std::cout << work << std::endl

ready= true synchronizes-with while(!ready.load()){}
 \Rightarrow ready= true inter-thread happens-before while (!ready.load()){}
 \Rightarrow ready= true happens-before while (!ready.load()){}

The final conclusion: Because the happens-before relation is transitive, it follows work = "done" happens-before ready= true happens-before while(!ready.load()){} happens-before std::cout<< work << std::endl

In sequential consistency, a thread sees the operations of another thread and, therefore, of all other threads in the same order. The key characteristic of sequential consistency will not hold if we use the acquire-release semantic for atomic operations. This is an area where C# and Java will not follow. That's also an area where our intuition begins to wane. Let's look at acquire-release semantic in the next lesson.

```

*/
// =====

```

```
// =====  
/*
```

Acquire Release Semantic

This lesson introduces the concept of the acquire-release semantic used in C++.

We'll cover the following

Aquire-Release Operations

There is no global synchronization between threads in the acquire-release semantic; there is only a synchronization between atomic operations on the same atomic variable. A write operation on one thread synchronizes with a read operation on another thread on the same atomic variable.

The acquire-release semantic is based on one key idea: a release operation synchronizes with an acquire operation on the same atomic and establishes an ordering constraint. This means all subsequent read and write operations cannot be moved before an acquire operation, and all read and write operations cannot be moved after a release operation.

Aquire-Release Operations#

What is an acquire or release operation? The reading of an atomic variable with load or test_and_set is an acquire operation. That being said, there is more: the acquiring of a lock, the creation of a thread, or waiting on a condition variable. Of course, the opposite is also true: releasing a lock, the join call on a thread or the notification of a condition variable are release operations. Accordingly, a store or clear operation on an atomic variable is a release operation. Acquire and release operations usually come in pairs.

It is worthwhile to think about the last few sentences from a different perspective. The lock of a mutex is an acquire operation, and the unlock of a mutex is a release operation. Figuratively speaking, this implies that an operation var += 1 cannot be moved outside of a critical section. On the other hand, a variable can be moved inside of a critical section because the variable moves from the non-protected to the protected area.

It helps a lot to keep that picture in mind.

This is the main reason you should keep the memory model in mind. In particular, the acquire-release semantic helps you to get a better understanding of the high-level synchronization primitives such as a mutex. The same reasoning holds for the starting of a thread and the join-call on a thread: both are acquire-release operations. The story goes on with the wait and notify_one call on a condition variable; wait is the acquire and notify_one the release operation. What about notify_all? That is a release operation as well.

Now, let us look once more at the spinlock in the subsection std::atomic_flag**. We can write it more efficiently because the synchronization is done with the atomic_flag flag, therefore the acquire-release semantic applies.

```
*/  
// =====  
// spinlockAcquireRelease.cpp  
// #include<iostream>  
// #include <atomic>  
// #include <thread>  
  
// class Spinlock{  
// std::atomic_flag flag;
```

```

// public:
// Spinlock(): flag(ATOMIC_FLAG_INIT) {}

// void lock(){
//   while(flag.test_and_set(std::memory_order_acquire));
// }

// void unlock(){
//   flag.clear(std::memory_order_release);
// }
//};

// Spinlock spin;

// void workOnResource(){
//   spin.lock();
//   // shared resource
//   spin.unlock();
//   std::cout << "Work done" << std::endl;
// }

// int main(){

// std::thread t(workOnResource);
// std::thread t2(workOnResource);

// t.join();
// t2.join();

// }
// =====
/*

```

The `flag.clear` call in line 16 is a release, the `flag.test_and_set` call in line 12 is an acquire operation, and the acquire synchronizes with the release operation. The heavyweight synchronization of two threads with sequential consistency (`std::memory_order_seq_cst`) is replaced by the lightweight and more performant acquire-release semantic (`std::memory_order_acquire` and `std::memory_order_release`). The behavior is not affected.

Although the `flag.test_and_set(std::memory_order_acquire)` call is a read-modify-write operation, the acquire semantic is sufficient. In summary, `flag` is an atomic.

```

*/
// =====
// =====
/*

```

Is the Acquire-Release Semantic Transitive?

This lesson introduces the concept of the acquire-release semantic being transitive.

We'll cover the following

Transitivity

The acquire-release semantic is transitive. That means if you have an acquire-release semantic between threads (a,b) and an acquire-release semantic between threads (b,c), you get an acquire-release semantic between (a,c).

Transitivity#

A release operation synchronizes with an acquire operation on the same atomic variable and additionally establishes ordering constraint. These are the components to synchronize threads in a performant way if they act on the same atomic. How can that work if two threads share no atomic variable? We do not want any sequential consistency because that is too expensive, but we want the light-weight acquire-release semantic.

The answer to this question is straightforward. Applying the transitivity of the acquire-release semantic, we are able to synchronize threads that are independent.

In the following example, thread t2 with its work package deliveryBoy is the connection between two independent threads t1 and t3.

```
*/
// =====
// transitivity.cpp

// #include <atomic>
// #include <iostream>
// #include <thread>
// #include <vector>

// std::vector<int> mySharedWork;
// std::atomic<bool> dataProduced(false);
// std::atomic<bool> dataConsumed(false);

// void dataProducer(){
//     mySharedWork = {1,0,3};
//     dataProduced.store(true, std::memory_order_release);
// }

// void deliveryBoy(){
//     while(!dataProduced.load(std::memory_order_acquire));
//     dataConsumed.store(true, std::memory_order_release);
// }

// void dataConsumer(){
//     while(!dataConsumed.load(std::memory_order_acquire));
//     mySharedWork[1] = 2;
// }

// int main(){

//     std::cout << std::endl;

//     std::thread t1(dataConsumer);
//     std::thread t2(deliveryBoy);
//     std::thread t3(dataProducer);
```

```

// t1.join();
// t2.join();
// t3.join();

// for (auto v: mySharedWork){
//   std::cout << v << " ";
// }

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

//}
// =====
/*

```

The output of the program is totally deterministic. `mySharedWork` will have the values 1,2 and 3.

There are two important observations:

Thread t2 waits in line 18, until thread t3 sets `dataProduced` to true (line 14).

Thread t1 waits in line 23, until thread t2 sets `dataConsumed` to true (line 19).

The rest is better explained with a graphic.

```

*/
// =====
// =====
/*

```

The important parts of the picture are the arrows.

The blue arrows are the sequenced-before relations. This means that all operations in one thread will be executed in source code order.

The red arrows are the synchronizes-with relations; the reason for this is the acquire-release semantic of the atomic operations on the same atomic. Subsequently, the synchronization between the atomics, and therefore between the threads at specific points, takes place.

Both sequenced-before and synchronizes-with establishes a happens-before relation.

The rest is pretty simple. The happens-before order of the instructions corresponds to the direction of the arrows from top to bottom. Finally, we have the guarantee that `mySharedWork[1] == 2` will be executed last.

A release operation synchronizes-with an acquire operation on the same atomic variable, so we can easily synchronize threads, if... The typical misunderstanding is about the if.

```

*/
// =====
// =====
/*

```

Acquire Release: The Typical Misunderstanding

This lesson highlights a typical misunderstanding while using acquire-release in C++.

We'll cover the following

The Solution

What is my motivation for writing about the typical misunderstanding of the acquire-release semantic? Many of my readers and students have already fallen into this trap. Let's look at the straightforward case. Here is a simple program as a starting point.

```

*/
// =====
// acquireReleaseWithWaiting.cpp

// #include <atomic>
// #include <iostream>
// #include <thread>
// #include <vector>

// std::vector<int> mySharedWork;
// std::atomic<bool> dataProduced(false);

// void dataProducer(){
//   mySharedWork = {1, 0, 3};
//   dataProduced.store(true, std::memory_order_release);
// }

// void dataConsumer(){
//   while( !dataProduced.load(std::memory_order_acquire) );
//   mySharedWork[1] = 2;
// }

// int main(){

// std::cout << std::endl;

// std::thread t1(dataConsumer);
// std::thread t2(dataProducer);

// t1.join();
// t2.join();

// for (auto v: mySharedWork){
//   std::cout << v << " ";
// }

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

// }
// =====
/*

```

The consumer thread t1 in line 17 waits until the consumer thread t2 in line 13 sets dataProduced to true. dataProduced is the guard and it guarantees that access to the non-atomic variable mySharedWork is synchronized. This means that the producer thread t2 initializes mySharedWork, then the consumer thread t1 finishes the work by setting mySharedWork[1] to 2. Therefore, the program is well-defined.

The graph below shows the happens-before relation within the threads and the synchronizes-with relation between the threads; synchronizes-with establishes a happens-before relation. The rest of the reasoning is the transitivity of the happens-before relation: mySharedWork = {1, 0, 3} happens-before mySharedWork[1] = 2.

What aspect is often missing in this reasoning? The if. What happens if the consumer thread t1 in line 17 doesn't wait for the producer thread?

```
/*
// =====
// acquireReleaseWithoutWaiting.cpp

// #include <atomic>
// #include <iostream>
// #include <thread>
// #include <vector>

// std::vector<int> mySharedWork;
// std::atomic<bool> dataProduced(false);

// void dataProducer(){
//     mySharedWork = {1, 0, 3};
//     dataProduced.store(true, std::memory_order_release);
// }

// void dataConsumer(){
//     dataProduced.load(std::memory_order_acquire);
//     mySharedWork[1] = 2;
// }

// int main(){

//     std::cout << std::endl;

//     std::thread t1(dataConsumer);
//     std::thread t2(dataProducer);

//     t1.join();
//     t2.join();

//     for (auto v: mySharedWork){
//         std::cout << v << " ";
//     }

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

// }
// =====
/*
```

The program has undefined behavior because there is a data race on the variable mySharedWork. When we let the program run, we will get the following non-deterministic behavior.

What's the issue? It holds that dataProduced.store(true, std::memory_order_release) synchronizes-with dataProduced.load(std::memory_order_acquire). However, that doesn't mean the acquire operation waits for the release operation, and that is exactly what is displayed in the graphic. In the graphic, the dataProduced.load(std::memory_order_acquire) instruction is performed before the instruction dataProduced.store(true, std::memory_order_release). We have no synchronizes-with relation.

```
*/  
// ======  
// ======  
/*
```

The Solution#

synchronizes-with: If `dataProduced.store(true, std::memory_order_release)` happens before `dataProduced.load(std::memory_order_acquire)`, then all visible effects of the operations before `dataProduced.store(true, std::memory_order_release)` are visible after `dataProduced.load(std::memory_order_acquire)`. The key is the word if. That if will be guaranteed in the first program with the predicate (`while(!dataProduced.load(std::memory_order_acquire))`).

Now once again, but more formally:

All operations before `dataProduced.store(true, std::memory_order_release)` happens-before all operations after `dataProduced.load(std::memory_order_acquire)`, if the following holds : `dataProduced.store(true, std::memory_order_release)` happens-before `dataProduced.load(std::memory_order_acquire)`.

If you carefully follow my explanation like in the subsection Challenges, you probably expect Relaxed Semantic to come next. However, in the next lesson, I'll look first at the memory model `std::memory_order_consume` which is quite similar to `std::memory_order_acquire`.

```
*/  
// ======  
// ======  
/*
```

`std::mem_order_consume`

This lesson introduces `std::mem_order_consume` which is used for concurrency in C++.

We'll cover the following

Introduction

Release-acquire Ordering

Release-acquire vs. Release-consume ordering

Introduction#

That is for two reasons that `std::memory_order_consume` is the most legendary of the six memory models: first, `std::memory_order_consume` is extremely hard to understand, and second - which may change in the future - no compiler currently supports it. With C++17 the situation gets even worse. Here is the official wording: "The specification of release-consume ordering is being revised, and the use of `memory_order_consume` is temporarily discouraged."

How can it be that a compiler that implements the C++11 standard doesn't support the memory model `std::memory_order_consume`? The answer is that the compiler maps `std::memory_order_consume` to `std::memory_order_acquire`. This is acceptable because both are load or acquire operations; `std::memory_order_consume` requires weaker synchronization and ordering constraints than `std::memory_order_acquire`. Therefore, the release-acquire ordering is potentially slower than the release-consume ordering, but - and this is the key point - it's well-defined.

To get an understanding of the release-consume ordering, it is a good idea to compare it with the release-acquire ordering. I speak in the following subsection explicitly about the release-acquire ordering (not about the acquire-release semantic) to emphasize the strong relationship of `std::memory_order_consume` and `std::memory_order_acquire`.

Release-acquire Ordering#

Let's use the following program with two threads t1 and t2 as a starting point. t1 plays the role of the producer, t2 the role of the consumer. The atomic variable ptr helps to synchronize the producer and consumer.

```
/*
// =====
// acquireRelease.cpp

// acquireRelease.cpp

// #include <atomic>
// #include <thread>
// #include <iostream>
// #include <string>

// using namespace std;

// atomic<string*> ptr;
// int data;
// atomic<int> atoData;

// void producer(){
//     string* p = new string("C++11");
//     data = 2011;
//     atoData.store(2014, memory_order_relaxed);
//     ptr.store(p, memory_order_release);
// }

// void consumer(){
//     string* p2;
//     while (!(p2 = ptr.load(memory_order_acquire)));
//     cout << "*p2: " << *p2 << endl;
//     cout << "data: " << data << endl;
//     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
// }

// int main(){

//     cout << endl;

//     thread t1(producer);
//     thread t2(consumer);

//     t1.join();
//     t2.join();

//     cout << endl;

// }
```

```

// =====
/*
Release-acquire vs. Release-consume ordering#
Before analysing the program, I want to introduce a small variation. Replace the memory model
std::memory_order_acquire in line 23 with std::memory_order_consume.
*/
// =====
// acquireConsume.cpp

// #include <atomic>
// #include <thread>
// #include <iostream>
// #include <string>

// using namespace std;

// atomic<string*> ptr;
// int data;
// atomic<int> atoData;

// void producer(){
//   string* p = new string("C++11");
//   data = 2011;
//   atoData.store(2014,memory_order_relaxed);
//   ptr.store(p, memory_order_release);
// }

// void consumer(){
//   string* p2;
//   while (!(p2 = ptr.load(memory_order_consume)));
//   cout << "*p2: " << *p2 << endl;
//   cout << "data: " << data << endl;
//   cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
// }

// int main(){
//   cout << endl;
//   thread t1(producer);
//   thread t2(consumer);

//   t1.join();
//   t2.join();

//   cout << endl;
// }

// =====
/*

```

Now the program has undefined behavior. This statement is very hypothetical because my GCC 5.4 compiler implements std::memory_order_consume using std::memory_order_acquire. Under the hood, both programs actually do the same thing.

The outputs of the programs are identical. At the risk of repeating myself, I want to add a few words explaining why the first program acquireRelease.cpp is well-defined.

The store operation in line 17 synchronizes-with the load operation in line 23. The reason is that the store operation uses std::memory_order_release and the load operation uses std::memory_order_acquire. This is the synchronization. What are the ordering constraints for the release-acquire operations? The release-acquire ordering guarantees that the results of all operations before the store operation (line 17) are available after the load operation (line 23). So, in addition, the release-acquire operation orders the access on the non-atomic variable data (line 11) and the atomic variable atoData (line 12). That holds, although atoData uses the std::memory_order_relaxed memory model.

Here's the key question: what happens if I replace std::memory_order_acquire with std::memory_order_consume?

```
/*
// =====
// =====
/*
Data dependencies with std::memory_order_consume
```

This lesson explains data dependencies with std::mem_order_consume in C++.

std::memory_order_consume deals with data dependencies on atomics; these data dependencies exist in two ways. First, let us look at carries-a-dependency-to in a thread and dependency-ordered-before between two threads. Both dependencies introduce a happens-before relation. These are the kind of relations we are looking for. What does carries-a-dependency-to and dependency-order-before mean?

carries-a-dependency-to: If the result of operation A is used as an operand in operation B, then A carries-a-dependency-to B.

dependency-ordered-before: A store operation (with std::memory_order_release, std::memory_order_acq_rel or std::memory_order_seq_cst) is dependency-ordered-before load operation B (with std::memory_order_consume) if the result of load operation B is used in a further operation C in the same thread. It is important to note that operations B and C have to be in the same thread.

I know from personal experience that both definitions might not be easy to digest. Here is a graphic to visualize them.

```
/*
// =====
// =====
/*
```

The expression ptr.store(p, std::memory_order_release) is dependency-ordered-before the expression while !(p2 = ptr.load(std::memory_order_consume)), because the following line std::cout << *p2: " << *p2 << std::endl will be read as the result of the load operation. Furthermore it holds that while !(p2 = ptr.load(std::memory_order_consume)) carries-a-dependency-to std::cout << *p2: " << *p2 << std::endl, because the output of *p2 uses the result of the ptr.load operation.

We have no guarantee regarding the output of data and atoData. That's because neither has a carries-a-dependency relation to the ptr.load operation. That being said, it gets even worse. Since data is a non-atomic variable, there is a race condition on the variable data; this is because both threads can access data at the same time, and thread t1 wants to modify data. Therefore, the program has undefined behavior.

Finally, we'll cover our relaxed semantic topic in the next lesson!

```
*/  
// ======  
// ======  
/*
```

Relaxed Semantic

This lesson gives an overview of relaxed semantic which is used in C++ for concurrency.

We'll cover the following

No Synchronization & Ordering constraints?

Conclusion

The relaxed semantic is the other end of the spectrum. It's the weakest of all memory models and only guarantees that the operations on the same atomic data type in the same thread won't be reordered. That guarantee is called modification order consistency. Other threads can see these operations in a different order.

No Synchronization & Ordering constraints?#

This is quite easy; if there are no rules, we cannot violate them. But that is too easy, as the program should have well-defined behavior. In particular, this means that data races are not allowed. To guarantee this you typically use synchronization and ordering constraints of stronger memory models to control operations with relaxed semantic. How does this work? A thread can see the effects of another thread in arbitrary order, so you have to make sure there are points in your program where all operations on all threads get synchronized.

A typical example of an atomic operation, in which the sequence of operations doesn't matter, is a counter. The key observation for a counter is not in which order the different threads increment the counter; it's that all increments are atomic and all threads' tasks are done at the end. Have a look at the following example.

```
*/  
// ======  
// relaxed.cpp  
  
// #include <vector>  
// #include <iostream>  
// #include <thread>  
// #include <atomic>  
  
// std::atomic<int> count = {0};  
  
// void add()  
//{
//   for (int n = 0; n < 1000; ++n) {
//     count.fetch_add(1, std::memory_order_relaxed);
//   }
// }  
  
// int main()
//{
//   std::vector<std::thread> v;
//   for (int n = 0; n < 10; ++n) {
//     v.emplace_back(add);
```

```

// }
// for (auto& t : v) {
//   t.join();
// }
// std::cout << "Final counter value is " << count << '\n';
//}
//=====================================================================
/*

```

The three most interesting lines are 13, 24, and 26. In line 13, the atomic number count is incremented using the relaxed semantic, so we have a guarantee that the operation is atomic. The `fetch_add` operation establishes an ordering on `count`. The function `add` (lines 10 - 15) is the work package of the threads. Each thread gets its work package on line 21. Thread creation is one synchronization point. The other one being `t.join()` on line 24.

The creator thread synchronizes with all its children in line 24. It waits with the `t.join()` call until all its children are done. `t.join()` is the reason that the results of the atomic operations are published. To say it more formally, `t.join()` is a release operation.

Conclusion#

In conclusion, there is a happens-before relation between the increment operation in line 13 and the reading of the counter count in line 26. The result is that the program always returns 10000. Boring? No, it's reassuring!

A typical example of an atomic counter which uses the relaxed semantic is the reference counter of `std::shared_ptr`. This will only hold for the increment operation. The key property for incrementing the reference counter is that the operation is atomic; the order of the increment operations does not matter. This will not hold for the decrement of the reference counter. These operations need an acquire-release semantic for the destructor.

The add algorithm is wait_free

Have a closer look at function `add` in line 10. There is no synchronisation involved in the increment operation (line 13). The value 1 is just added to the atomic count. Therefore, the algorithm is not only lock-free but it is also wait-free.

The key idea of `std::atomic_thread_fence` is to establish synchronization and ordering constraints between threads without any atomic operations.

```

*/
//=====================================================================
//=====================================================================
/*

```

Fences as Memory Barriers

This lesson introduces a concept of fences as memory barriers in C++.

An `std::atomic_thread_fence` prevents specific operations from crossing a fence, and it doesn't need an atomic variable; they are frequently just referred to as fences or memory barriers. You quickly get an idea of what an `std::atomic_thread_fence` is all about.

What does it mean by Fences as Memory Barriers? Specific operations cannot cross a memory barrier. What kind of operations? From a bird's-eye view, we have two kinds of operations: read and write or load and store

operations. The expression `if(resultRead) return result` is a load, followed by a store operation. There are four different ways to combine load and store operations:

Combination	Meaning
LoadLoad	A load followed by a load
LoadStore	A load followed by a store
StoreLoad	A store followed by a load
StoreStore	A store followed by a store

Of course, there are more complex operations consisting of multiple load and stores (`count++`), and these operations fall into my general classification.

What about memory barriers? If you place memory barriers between two operations like LoadLoad, LoadStore, StoreLoad or StoreStore, you have the guarantee that specific LoadLoad, LoadStore, StoreLoad or StoreStore operations will not be reordered. The risk of reordering is always present if non-atomics or atomic operations with relaxed semantic are used.

```
*/  
// ======  
// ======  
/*
```

The Three Fences

This lesson gives an overview of the acquire, release, and full fences used in C++ as memory barriers.

We'll cover the following

Full fence

Acquire fence

Release fence

Typically, three kinds of fences are used: full fence, acquire fence and release fence. As a reminder, acquire is a load, and release is a store operation. What happens if I place one of the three memory barriers between the four combinations of load and store operations?

Full fence: A full fence `std::atomic_thread_fence()` between two arbitrary operations prevents the reordering of these operations, but guarantees that it won't hold for StoreLoad operations. Also, they can be reordered.

Acquire fence: An acquire fence `std::atomic_thread_fence(std::memory_order_acquire)` prevents a read operation before an acquire fence from being reordered with a read or write operation after the acquire fence.

Release fence: A release fence `std::atomic_thread_fence(std::memory_order_release)` prevents a read or write operation before a release fence from being reordered with a write operation after a release fence.

A lot of energy goes into accurately forming the definitions of the acquire and release fence and their consequences for lock-free programming. The subtle differences between the acquire-release semantic of atomic operations are especially challenging to understand. Before I get to that point, I will illustrate the definitions with graphics.

Which kind of operations can cross a memory barrier? Have a look at the following three graphics. If the arrow is crossed with a red bar, the fence prevents this type of operation.

Full fence#

Of course, instead of writing `std::atomic_thread_fence()` you can explicitly write `std::atomic_thread_fence(std::memory_order_seq_cst)`. Sequential consistency is applied to fences by default. If you use sequential consistency for a full fence, the `std::atomic_thread_fence` follows a global order.

```
*/  
// ======  
// ======  
/*
```

Acquire and Release Fences

This lesson gives an overview of acquire and release fences used in C++ as memory barriers.

We'll cover the following

Atomic Operations vs Fences

Acquire Operation:

Release Operation:

The most obvious difference between acquire and release fences and atomics with acquire-release semantics is that fences need no atomics. There is also a more subtle difference: the acquire and release fences are more heavyweight.

Atomic Operations vs Fences#

For the sake of simplicity, I will now refer to acquire operations when I use fences or atomic operations with acquire semantic. The same will hold for release operations.

The main idea of an acquire and a release operation is that it establishes synchronization and ordering constraints between threads. These synchronization and ordering constraints also hold for atomic operations with relaxed semantic or non-atomic operations. Note that acquire and release operations come in pairs. In addition, operations on atomic variables with acquire-release semantic must act on the same atomic variable. Having said that, I will now look at these operations in isolation. Let's start with the acquire operation.

Acquire Operation:#

A load (read) operation on an atomic variable with the memory model set to `std::memory_order_acquire` is an acquire operation.

```
*/  
// ======  
// ======  
/*
```

`std::atomic_thread_fence` with the memory order set to `std::memory_order_acquire` imposes stricter constraints on memory access reordering:

```
*/  
// ======  
// ======  
/*
```

This comparison emphasizes two points:

A fence with acquire-semantic establishes stronger ordering constraints. Although the acquire operation on an atomic and a fence requires that no read or write operation can be moved before the acquire operation, there is an additional guarantee with the acquire fence. No read operation can be moved after the acquire fence.

The relaxed-semantic is sufficient for the reading of the atomic variable var. Thanks to `std::atomic_thread_fence(std::memory_order_acquire)`, this operation cannot be moved after the acquire fence.

Similar observations can be made for the release fence.

Release Operation:#

The store (write) operation on an atomic variable attached with the memory model set to `std::memory_order_release` is a release operation.

Here is the corresponding release fence.

In addition to the constraints imposed by the release operation on an atomic variable var, the release fence guarantees two properties:

Store operations can't be moved before the fence.

It's sufficient for the variable var to have relaxed semantic.

But now, it's time to go one step further and build a program in the next lesson that will use fences.

```
*/  
// ======  
// ======  
/*
```

Synchronization with Atomic Variables

This lesson gives an overview of synchronization with atomic variables in C++.

We'll cover the following

Atomic Operations

As a starting point, I've implemented a typical consumer-producer workflow with the acquire-release semantic. Initially, I will use atomics and then will switch to fences. Let's start with atomics because most of us are comfortable with them. That will not hold for fences; they are almost completely ignored in the literature on the C++ memory model.

Atomic Operations#

```
*/  
// ======  
// acquireRelease.cpp  
  
// #include <atomic>  
// #include <thread>  
// #include <iostream>  
// #include <string>
```

```

// using namespace std;

// atomic<string*> ptr;
// int data;
// atomic<int> atoData;

// void producer(){
//   string* p = new string("C++11");
//   data = 2011;
//   atoData.store(2014, memory_order_relaxed);
//   ptr.store(p, memory_order_release);
//}

// void consumer(){
//   string* p2;
//   while (!(p2 = ptr.load(memory_order_acquire)));
//   cout << "*p2: " << *p2 << endl;
//   cout << "data: " << data << endl;
//   cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
//}

// int main(){

//   cout << endl;

//   thread t1(producer);
//   thread t2(consumer);

//   t1.join();
//   t2.join();

//   cout << endl;

//}

// =====
/*

```

This program should be quite familiar to you; it is the classic example that I used in the subsection about `std::memory_order_consume`. The graphic emphasizes exactly that the consumer thread `t2` sees all values from the producer thread `t1`.

```

*/
// =====
// =====
/*

```

The program is well-defined because the happens-before relation is transitive. I only have to combine the three happens-before relations:

Lines 14 - 16 happens-before line 17 `ptr.store(p, std::memory_order_release)`.
Line 23 `while(!(p2 = ptr.load(std::memory_order_acquire)))` happens-before the lines 24 - 26.
Line 17 synchronizes-with line 23. => Line 17 happens-before line 23.

But now the story becomes more interesting. How can I adjust the workflow to fences? We'll discuss this in the next lesson.

```
*/  
// ======  
// ======  
/*
```

Synchronization with Fences

This lesson gives an overview of synchronization with fences in C++.

It's quite straightforward to port the program to use fences.

```
*/  
// ======  
// ======  
// acquireReleaseFences.cpp  
  
// #include <atomic>  
// #include <thread>  
// #include <iostream>  
// #include <string>  
  
// using namespace std;  
  
// atomic<string *> ptr;  
// int data;  
// atomic<int> atoData;  
  
// void producer()  
// {  
//   string *p = new string("C++11");  
//   data = 2011;  
//   atoData.store(2014, memory_order_relaxed);  
//   atomic_thread_fence(memory_order_release);  
//   ptr.store(p, memory_order_relaxed);  
// }  
  
// void consumer()  
// {  
//   string *p2;  
//   while (!(p2 = ptr.load(memory_order_relaxed)))  
//     ;  
//   atomic_thread_fence(memory_order_acquire);  
//   cout << "*p2: " << *p2 << endl;  
//   cout << "data: " << data << endl;  
//   cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;  
// }  
  
// int main()  
// {  
  
//   cout << endl;  
  
//   thread t1(producer);
```

```
// thread t2(consumer);  
  
// t1.join();  
// t2.join();  
  
// delete ptr;
```

```
// cout << endl;  
//}  
//=====
```

/*
The first step was to add fences with the acquire and release semantic (lines 18 and 25). Next, I changed the atomic operations with acquire or release semantic to relaxed semantic (lines 17 and 24) - which was straightforward. Of course, I can only replace an acquire or release operation with the corresponding fence. The key point is that the release operation with the acquire operation establishes a synchronizes-with relation and, therefore, a happens-before relation. For a more visual reader, here's the entire relation graphically.

```
*/  
//=====
```

```
//=====
```

/*
This is the key question: Why do the operations after the acquire fence see the effects of the operations before the release fence? This is interesting because data is a non-atomic variable and `atoData` is used with relaxed semantic, which would suggest they can be reordered. However, thanks to the `std::atomic_thread_fence(std::memory_order_release)` as a release operation in combination with the `std::atomic_thread_fence(std::memory_order_acquire)`, neither can be reordered.

For clarity, here's the whole reasoning in a more concise form:

The acquire and release fences prevent the reordering of the atomic and non-atomic operations across the fences.

The consumer thread `t2` is waiting in the while (`!(p2 = ptr.load(std::memory_order_relaxed))`) loop, until the pointer `ptr.store(p, std::memory_order_relaxed)` is set in the producer thread `t1`.

The release fence synchronizes-with the acquire fence.

```
*/  
//=====
```

```
//=====
```

```
/*
```

Introduction to Threads

This lesson gives an introduction to threads in C++.

C++ has had a multithreading interface since C++11. This interface has all the basic building blocks for creating multithreaded programs: threads, synchronization primitives for shared data (e.g. mutexes and locks), thread-local data, synchronization mechanism for threads (e.g. condition variables), and tasks. Tasks are usually called promises, and they provide a higher level of abstraction than native threads. It is okay if you do not understand the terms discussed here, as all of them will be discussed in depth in the following lessons.

```
*/  
//=====
```

```
//=====
```

```
/*
```

Creation of Threads

This lesson gives an introduction on how to create threads in C++ using callable units such as functions and lambda functions.

We'll cover the following

Output

To launch a thread in C++, you have to include the <thread> header.

A thread std::thread represents an executable unit. This executable unit, which the thread immediately starts, gets its work package as a callable unit.

A callable unit is an entity that behaves like a function. Of course, it can be a function, but also a function object or a lambda function.

For example,

```
/*
// =====
// createThread.cpp

// #include <iostream>
// #include <thread>

// void helloFunction(){
//   std::cout << "Hello from a function." << std::endl;
// }

// class HelloFunctionObject{
//   public:
//     void operator()() const {
//       std::cout << "Hello from a function object." << std::endl;
//     }
// };

// int main(){

//   std::cout << std::endl;

//   std::thread t1(helloFunction);

//   HelloFunctionObject helloFunctionObject;
//   std::thread t2(helloFunctionObject);

//   std::thread t3([]{std::cout << "Hello from a lambda." << std::endl;});

//   t1.join();
//   t2.join();
//   t3.join();

//   std::cout << std::endl;
```

```
// };
// =====
/*
All three threads (t1, t2, and t3) write their messages to the console. The work package of thread t2 is a function object (lines 10 - 15), and the work package of thread t3 is a lambda function (line 26). In lines 28 - 30 the main thread is waiting until its children are done.
```

Output#

The three threads are executed in an arbitrary order; even the three output operations can interleave. The creator of the child - the main thread in our case - is responsible for the lifetime of the child.

```
/*
// =====
// =====
/*

```

Managing Thread Lifetime

This lesson gives an overview of how to use the join and detach functions to properly end thread execution in C++.

We'll cover the following

join & detach functions

Solution

The parent has to take care of its children - a simple principle that has significant consequences for the lifetime of a thread. This small program starts a thread that displays its ID:

```
/*
// =====
// threadWithoutJoin.cpp
```

```
// #include <iostream>
// #include <thread>

// int main(){
//   std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});

// }
```

But the program will not print the ID. What's the reason for this exception? Let's figure it out!

join & detach functions#

The lifetime of a created thread t ends with its callable unit. Therefore, the creator has two choices:

It can wait until its child is done: t.join().

It can detach itself from its child: t.detach().

A t.join() call is useful when the subsequent code relies on the result of the calculation performed in the thread. t.detach() permits the thread to execute independently from the thread handle t; therefore, the detached thread will run for the lifetime of the executable. Typically, you use a detached thread for a long-running background service such as a server.

A thread t with a callable unit (you can create threads without a callable unit) is called joinable if neither a t.join() nor a t.detach() call happened. The destructor of a joinable thread throws the std::terminate exception; this was the reason the program execution of threadWithoutJoin.cpp terminated with an exception. If you invoke t.join() or t.detach() more than once on a thread t, you get a std::system_error exception.

Solution#

The solution to this problem is quite simple: call t.join()

```
/*
// =====
// threadWithJoin.cpp

// #include <iostream>
// #include <thread>

// int main(){
//   std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});
//   t.join();
// }
// =====
/*
```

Thread Lifetime Management: Warnings and Tips

Some caveats and tips on the lifetime of threads in C++ coming your way...

We'll cover the following

Warnings

Tips

Warnings#

The Challenge of detach: Of course you can use t.detach() instead of t.join() in the last program. The thread t is not joinable any more; therefore, its destructor didn't call std::terminate. But now you have another issue. The program behaviour is undefined because the main program may complete before the thread t has time to complete its workpackage; therefore, its lifetime is too short to display the id. For more details, see lifetime issues of variables.

Tips#

 scoped_thread by Anthony Williams

If it's too bothersome for you to manually take care of the lifetime of your threads, you can encapsulate a std::thread in your own wrapper class. This class should automatically call join in its destructor. Of course you can go the other way and call detach, but there is an issue with detach.

Anthony Williams created such a useful class and presented it in his excellent book Concurrency in Action. He called the wrapper scoped_thread. scoped_thread gets a thread t in its constructor and checks if t is still joinable. If the thread t passed into the constructor is not joinable anymore, there is no need for the

scoped_thread. If t is joinable, the destructor calls t.join(). Because the copy constructor and copy assignment operator are declared as delete, instances of scoped_thread can not be copied to or assigned from.

```
/*
// =====
// scoped_thread.cpp

// #include <iostream>
// #include <thread>
// #include <utility>

// class scoped_thread{
// std::thread t;
// public:
// explicit scoped_thread(std::thread t_): t(std::move(t_)){
// if (!t.joinable()) throw std::logic_error("No thread");
// }
// ~scoped_thread(){
// t.join();
// }
// scoped_thread(scoped_thread&)= delete;
// scoped_thread& operator=(scoped_thread const &)= delete;
// };
// =====
/*
```

Passing Arguments to Threads

This lesson gives an overview of how to pass arguments to threads in C++ -- by copy and by reference.

We'll cover the following

Copy or Reference

A thread, like any arbitrary function, can get its arguments by copy, by move, or by reference. std::thread is a variadic template which means that it takes an arbitrary number of arguments.

In the case where your thread gets its data by reference, you have to be extremely careful about the lifetime of the arguments; not doing so may result in undefined behavior.

Copy or Reference#

Let's have a look at a small code snippet:

```
/*
// =====
// #include <thread>
// #include <iostream>

// int main()
//{
// std::string s{"C++11"};

// std::thread t1([=]
// { std::cout << s << std::endl; });
// t1.join();
```

```
// std::thread t2(&
//     { std::cout << s << std::endl; });
// t2.detach();
// }
// =====
/*
Thread t1 gets its argument by copy, thread t2 by reference.
```

Thread arguments by reference

To be honest, I cheated a little. Note that thread t2 gets its argument by reference, and the lambda function captures its argument by reference as well. If you need to pass the argument to a thread by reference, it must be wrapped in a reference wrapper. This is quite straightforward with the helper function `std::ref`.

```
void transferMoney(int amount, Account& from, Account& to){
    ...
}
...
std::thread thr1(transferMoney, 50, std::ref(account1), std::ref(account2));
```

Thread `thr1` executes the function `transferMoney` and `transferMoney` gets its arguments by reference; therefore, thread `thr1` gets its `account1` and `account2` by reference.

Not convinced? Let's take a closer look at what undefined behavior may look like in the next lesson.

```
/*
// =====
// =====
/*
Arguments of Threads: Undefined behavior
```

This lesson gives an example of undefined behavior caused by passing arguments improperly to threads in C++

As discussed in the previous lesson, here is another example of undefined behavior caused by improper handling of thread arguments.

```
/*
// =====
// threadArguments.cpp
```

```
// #include <chrono>
// #include <iostream>
// #include <thread>

// class Sleeper{
// public:
//     Sleeper(int& i_):i{i_}{};
//     void operator() (int k){
//         for (unsigned int j= 0; j <= 5; ++j){
//             std::this_thread::sleep_for(std::chrono::milliseconds(100));
//             i += k;
//         }
//         std::cout << std::this_thread::get_id() << std::endl;
//     }
```

```

// private:
// int& i;
// };

// int main(){

// std::cout << std::endl;

// int valSleeper = 1000;
// std::thread t(Sleeper(valSleeper), 5);
// t.detach();
// std::cout << "valSleeper = " << valSleeper << std::endl;

// std::cout << std::endl;

//}
// =====
/*

```

The question is, what value does valSleeper have in line 29? valSleeper is a global variable. Also, thread t gets a function object with the variable valSleeper and the number 5 (line 27) as its work package. The crucial observation is that the thread gets valSleeper by reference (line 9) and will be detached from the main thread (line 28). Next, it will execute the call operator of the function object (lines 10 - 16). In this method it counts from 0 to 5, sleeps in each iteration 1/10 of a second, and increments i by k. At the end, it displays its id on the screen. Nach Adam Riese (a German proverb), the result should be $1000 + 6 * 5 = 1030$.

But what happened? Something is going very wrong. In the next lesson, learn how to fix this issue.

```

*/
// =====
// =====
/*

```

Arguments of Threads - Race Conditions and Locks

This lesson defines race conditions, and explains how to address them in concurrent programming with C++

Both issues from the previous lesson are actually race conditions because the result of the program depends on the interleaving of the operations. The race condition is the cause of the data race.

Fixing the data race is quite easy; valSleeper should be protected using either a lock or an atomic. To overcome the lifetime issues of valSleeper and std::cout, you have to join the thread instead of detaching it.

Here is the modified main function.

```

*/
// =====
// #include <chrono>
// #include <iostream>
// #include <thread>

// class Sleeper{
// public:
// Sleeper(int& i_):i{i_}{};
// void operator() (int k){
// for (unsigned int j= 0; j <= 5; ++j){


```

```

//     std::this_thread::sleep_for(std::chrono::milliseconds(100));
//     i += k;
// }
// std::cout << std::this_thread::get_id() << std::endl;
// }
// private:
// int& i;
// };

// int main(){

// std::cout << std::endl;

// int valSleeper= 1000;
// std::thread t(Sleeper(valSleeper),5);
// t.join();
// std::cout << "valSleeper = " << valSleeper << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

Methods of Threads

This lesson lists and explains the commonly used methods of threads in C++.

We'll cover the following

More on Swap

Here is the interface of `std::thread t` in a concise table. For additional details, please refer to cppreference.com.

Method	Description
<code>t.join()</code>	Waits until thread t has finished its executable unit.
<code>t.detach()</code>	Executes the created thread t independently of the creator.
<code>t.joinable()</code>	Returns true if thread t is still joinable.
<code>t.get_id()</code> and <code>std::this_thread::get_id()</code>	Returns the identity of the thread.
<code>std::thread::hardware_concurrency()</code>	Returns the number of cores, or 0 if the runtime can not determine the number. Indicates the number of threads that can be run concurrently. This is according to the C++ standard.
<code>std::this_thread::sleep_until(absTime)</code>	Puts thread t to sleep until the time point absTime. Needs a time point or a time duration as an argument.
<code>std::this_thread::sleep_for(relTime)</code>	Puts thread t to sleep for the time duration relTime. Needs a time point or a time duration as an argument.
<code>std::this_thread::yield()</code>	Enables the system to run another thread.
<code>t.swap(t2)</code> and <code>std::swap(t1, t2)</code>	Swaps the threads.

More on Swap#

Also, note that threads cannot be copied, but they can be moved; the swap method performs a move when possible.

In the next lesson, we will demonstrate how a few of the methods are used in practice.

Access to the system-specific implementation

The C++11 threading interface is a wrapper around the underlying implementation. You can use the method `native_handle` to get access to the system-specific implementation. This holds true for threads, mutexes, and condition variables.

```
*/
// =====
// =====
/*
```

Methods of Threads in Practice

This lesson shows the application of commonly used thread methods such as `get_id`, `hardware_concurrency`, and `joinable` in C++.

Some of the most commonly used thread methods are mentioned in the following code widget:

```
/*
// =====
// threadMethods.cpp

// #include <iostream>
// #include <thread>

// using namespace std;

// int main(){

// cout << boolalpha << endl;

// cout << "hardwareConcurrency()= "<< thread::hardwareConcurrency() << endl;

// thread t1([]{cout << "t1 with id= " << this_thread::get_id() << endl;});
// thread t2([]{cout << "t2 with id= " << this_thread::get_id() << endl;});

// cout << endl;

// cout << "FROM MAIN: id of t1 " << t1.get_id() << endl;
// cout << "FROM MAIN: id of t2 " << t2.get_id() << endl;

// cout << endl;
// swap(t1,t2);

// cout << "FROM MAIN: id of t1 " << t1.get_id() << endl;
// cout << "FROM MAIN: id of t2 " << t2.get_id() << endl;

// cout << endl;

// cout << "FROM MAIN: id of main= " << this_thread::get_id() << endl;

// cout << endl;

// cout << "t1.joinable(): " << t1.joinable() << endl;
```

```

// cout << endl;

// t1.join();
// t2.join();

// cout << endl;

// cout << "t1.joinable(): " << t1.joinable() << endl;

// cout << endl;

//}

//=====
/*

```

In combination with the output, the program should be quite easy to follow.

Maybe it looks a little weird that threads t1 and t2 (lines 14 and 15) run at different points in time during the program execution. However, you have no guarantee when each thread runs; you only have the guarantee that both threads will run before t1.join() and t2.join() in lines 38 and 39.

The more mutable (non-const) variables threads share, the more challenging multithreading becomes.

```

*/
//=====
//=====
/*

```

Introduction to Shared Data

This lesson gives an introduction to how data can be shared between threads in C++.

We'll cover the following

Boss-Worker Model

You only need to think about synchronization if you have shared, mutable data because such data is prone to data races. If you have concurrent non-synchronised read and write access to data, your program will have undefined behavior. The easiest way to visualize concurrent, unsynchronized read and write operations is to write something to std::cout. Let's have a look:

```

*/
//=====
// coutUnsyncronised.cpp

// #include <chrono>
// #include <iostream>
// #include <thread>

// class Worker{
// public:
//   Worker(std::string n):name(n){}
//   void operator() (){
//     for (int i = 1; i <= 3; ++i){
//       // begin work
//       std::this_thread::sleep_for(std::chrono::milliseconds(200));
//     }
//   }
// }
```

```

//      // end work
//      std::cout << name << ":" << "Work " << i << " done !!!" << std::endl;
//    }
//  }
// private:
//  std::string name;
//};

// int main(){

// std::cout << std::endl;

// std::cout << "Boss: Let's start working.\n\n";

// std::thread herb= std::thread(Worker("Herb"));
// std::thread andrei= std::thread(Worker(" Andrei"));
// std::thread scott= std::thread(Worker(" Scott"));
// std::thread bjarne= std::thread(Worker(" Bjarne"));
// std::thread bart= std::thread(Worker(" Bart"));
// std::thread jenne= std::thread(Worker(" Jenne"));

// herb.join();
// andrei.join();
// scott.join();
// bjarne.join();
// bart.join();
// jenne.join();

// std::cout << "\n" << "Boss: Let's go home." << std::endl;

// std::cout << std::endl;

// }

//=====
/*

```

Boss-Worker Model#

In the boss-worker model, a single thread - the boss - accepts input for the entire program. Based on that input, the boss passes off specific tasks to one or more worker threads. In the code above, the boss has six workers (lines 29 - 34). Each worker has to take care of 3 work packages. The work package takes 1/5 second (line 13). After the worker finishes the work package, it screams out loudly to the boss (line 15). Once the boss receives notifications from all workers, it sends them home (line 43).

What a mess for such a simple workflow! The most straightforward solution is to use a mutex, which we will see in the next lesson.

```

*/
//=====
//=====
/*

```

Introduction to Mutexes

This lesson gives an introduction to mutexes which are used in C++ for concurrency.

Mutex stands for mutual exclusion. It ensures that only one thread can access a critical section at any one time. By using a mutex, the mess of the workflow turns into a harmony.

```
/*
// =====
// coutSynchronised.cpp

// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <thread>

// std::mutex coutMutex;

// class Worker{
// public:
//   Worker(std::string n):name(n){};

//   void operator() (){
//     for (int i = 1; i <= 3; ++i){
//       // begin work
//       std::this_thread::sleep_for(std::chrono::milliseconds(200));
//       // end work
//       coutMutex.lock();
//       std::cout << name << ":" << "Work " << i << " done !!!" << std::endl;
//       coutMutex.unlock();
//     }
//   }
// private:
//   std::string name;
// };

// int main(){

// std::cout << std::endl;

// std::cout << "Boss: Let's start working." << "\n\n";

// std::thread herb= std::thread(Worker("Herb"));
// std::thread andrei= std::thread(Worker(" Andrei"));
// std::thread scott= std::thread(Worker(" Scott"));
// std::thread bjarne= std::thread(Worker(" Bjarne"));
// std::thread bart= std::thread(Worker(" Bart"));
// std::thread jenne= std::thread(Worker(" Jenne"));

// herb.join();
// andrei.join();
// scott.join();
// bjarne.join();
// bart.join();
// jenne.join();
```

```

// std::cout << "\n" << "Boss: Let's go home." << std::endl;
// std::cout << std::endl;

// }
// =====
/*

```

Essentially, when the lock is set on a mutex, no other thread can access the locked region of code. In other words, lines between lock() and unlock() can only be accessed by one thread at a time. std::cout is protected by the coutMutex in line 8. A simple lock() in line 19 and the corresponding unlock() call in line 21 ensure that the workers won't scream all at once.

std::cout is thread-safe

The C++11 standard guarantees that you must not protect std::cout. Each character will be written atomically. It is possible that more output statements like those in the example will interleave. This is only a visual issue; the program is well-defined. This remark is valid for all global stream objects. Insertion to and extraction from global stream objects (std::cout, std::cin, std::cerr, and, std::clog) is thread-safe.

Let's put it more formally: writing to std::cout is not a data race, but it's a race condition which means that the result depends on the interleaving of threads.

Different locking methods will be discussed in the next lesson.

```

*/
// =====
// =====
/*

```

Mutex Types and Locking Methods

This lesson discusses different types of mutexes and their locking methods

We'll cover the following

std::shared_timed_mutex

Mutex try_lock methods

C++ has five different mutexes that can lock recursively (i.e., multiple layers of locking), tentative with and without time constraints.

Method	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex	shared_timed_mutex
m.lock	yes	yes	yes	yes	
m.unlock	yes	yes	yes	yes	yes
m.try_lock	yes	yes	yes	yes	yes
m.try_lock_for	no	yes	yes	yes	
m.try_lock_until	no	no	yes	yes	yes
m.try_lock_shared	yes	no	no	no	yes
m.try_lock_shared_for	no	no	no	no	yes
m.try_lock_shared_until	no	no	no	no	yes
std::shared_timed_mutex#					

With C++14 we have an std::shared_timed_mutex that is the base for reader-writer locks. It solves the infamous reader-writer problem.

The `std::shared_timed_mutex` enables you to implement reader-writer locks which means that you can use it for exclusive or shared locking. You will get an exclusive lock if you put the `std::shared_timed_mutex` into a `std::lock_guard`; you will get a shared lock if you put it into an `std::shared_lock`.

`std::shared_mutex` with C++17

With C++17 we get a new mutex: `std::shared_mutex`. `std::shared_mutex` is similar to `std::shared_timed_mutex`. Like the `std::shared_timed_mutex`, you can use it for exclusive or shared locking, but you can not specify a time point or a time duration.

Mutex `try_lock` methods#

The `m.try_lock_for(relTime)` (`m.try_lock_shared_for(relTime)`) method needs a relative time duration; the `m.try_lock_until(absTime)` (`m.try_lock_shared_until(absTime)`) method needs an absolute time point.

`m.try_lock` (`m.try_lock_shared`) tries to lock the mutex and returns immediately. On success, it returns true; otherwise, it's false. In contrast, the methods `try_lock_for` (`try_lock_shared_for`) and `try_lock_until` (`try_lock_shared_until`) try to lock until the specified timeout occurs or the lock is acquired, whichever comes first. You should use a steady clock for your time constraint. A steady clock cannot be adjusted.

Tip: You should not use mutexes directly; you should put mutexes into locks.

```
*/  
// ======  
// ======  
/*
```

Issues of Mutexes: Deadlocks

This lesson gives an overview of deadlocks caused by improper mutex locking in C++.

We'll cover the following

Deadlock:

Deadlock caused by Mutex locking order

Deadlock Example

Explanation:

The issues with mutexes boil down to one main concern: deadlocks.

Deadlock:#

A deadlock is a state where two or more threads are blocked because each thread waits for the release of a resource before it releases its own resource.

The result of a deadlock is a total standstill. The thread that tries to acquire the resource - and usually the whole program - is blocked forever. It is easy to produce a deadlock. Curious?

Deadlock caused by Mutex locking order#

Here is a typical scenario of a deadlock resulting from locking in a certain order.

Thread 1 and thread 2 need access to two resources to finish their work. The problem arises when the requested resources are protected by two separate mutexes and are locked in different orders (Thread 1: Lock 1, Lock 2; Thread 2: Lock 2, Lock 1). In this case, the thread executions will interleave in such a way that thread 1 gets mutex 1, then thread 2 gets mutex 2, and then we reach a standstill. Each thread wants to get the

other's mutex but to get the other's mutex the first thread has to release its mutex first. The expression "deadly embrace" describes this kind of deadlock very well.

Deadlock Example#

Translating this picture into code is easy.

```
/*
// =====
// deadlock.cpp

// #include <iostream>
// #include <chrono>
// #include <mutex>
// #include <thread>

// struct CriticalData{
//   std::mutex mut;
// };

// void deadLock(CriticalData& a, CriticalData& b){

//   a.mut.lock();
//   std::cout << "get the first mutex" << std::endl;
//   std::this_thread::sleep_for(std::chrono::milliseconds(1));
//   b.mut.lock();
//   std::cout << "get the second mutex" << std::endl;
//   // do something with a and b
//   a.mut.unlock();
//   b.mut.unlock();

// }

// int main(){

//   CriticalData c1;
//   CriticalData c2;

//   std::thread t1([&]{deadLock(c1,c2);});
//   std::thread t2([&]{deadLock(c2,c1);});

//   t1.join();
//   t2.join();

// }
// =====
/*
```

Explanation:#

Threads t1 and t2 call deadlock (lines 12 - 23). The function deadlock needs variables CriticalData c1 and c2 (lines 27 and 28). Because objects c1 and c2 have to be protected from shared access, they internally hold a mutex; (To keep this example short and simple, CriticalData doesn't have any other methods or members apart from a mutex).

A short sleep of about 1 millisecond in line 16 is sufficient to produce the deadlock. The only choice left is to press CTRL+C and kill the process.

Locks will not solve all the issues with mutexes, but they'll come to the rescue in many cases. We will cover locks in detail in the next lesson.

```
*/  
// ======  
// ======  
/*
```

Issues of Mutexes: Avoiding Exceptions

This lesson lists some caveats pertaining to mutexes and how to avoid them in C++.

We'll cover the following

Exceptions and Unknown Behavior

Issues & Best practices

Exceptions and Unknown Behavior#

The small code snippet has a lot of issues to look at, including a few exceptions and unknown behaviors in the program:

```
*/  
// ======  
// std::mutex m;  
// m.lock();  
// sharedVariable = getVar();  
// m.unlock();  
// ======  
/*
```

Issues & Best practices#

If the function `getVar()` throws an exception, the mutex `m` will not be released.

Never ever call an unknown function while holding a lock. If the function `getVar` tries to lock the mutex `m`, the program has undefined behavior because `m` is not a recursive mutex. Most of the time, the undefined behavior will result in a deadlock.

Avoid calling a function while holding a lock. Maybe the function is from a library and you get a new version of the library, or the function is rewritten, but there is always the danger of a deadlock.

The more locks your program needs, the more challenging it becomes. The dependency is very non-linear.

```
*/  
// ======  
// ======  
/*
```

Types of Locks: `std::lock_guard`

This lesson gives an introduction to locks and explains `std::lock_guard` used in C++.

We'll cover the following

`std::lock_guard`

Locks take care of their resource following the RAIID idiom. A lock automatically binds its mutex in the constructor and releases it in the destructor; this considerably reduces the risk of a deadlock because the runtime takes care of the mutex.

Locks are available in three different flavors: std::lock_guard for the simple use-cases; std::unique_lock for the advanced use-cases; std::shared_lock is available (with C++14) and can be used to implement reader-writer locks.

First, the simple use-case:

```
std::mutex m;
m.lock();
sharedVariable = getVar();
m.unlock();
*/
// =====
// =====
/*
```

The mutex m ensures that access to the critical section sharedVariable= getVar() is sequential. Sequential means in this special case that each thread gains access to the critical section after the other. This maintains a kind of total order in the system. The code is simple but prone to deadlocks. A deadlock appears if the critical section throws an exception or if the programmer simply forgets to unlock the mutex.

std::lock_guard#

With std::lock_guard we can do this in a more elegant way:

```
{
    std::mutex m,
    std::lock_guard<std::mutex> lockGuard(m);
    sharedVariable= getVar();
}
*/
// =====
// =====
/*
```

That was easy, but what's the story with the opening and closing brackets? The lifetime of std::lock_guard is limited by the curly brackets, which means that its lifetime ends when it passes the closing curly brackets. At exactly that point, the std::lock_guard destructor is called and - as you may have guessed - the mutex is released automatically. It is also released if getVar() in sharedVariable = getVar() throws an exception. The function scope and loop scope also limit the lifetime of an object.

std::scoped_lock with C++17

With C++17 we get a std::scoped_lock. It's very similar to std::unique_lock, but std::scoped_lock can lock an arbitrary number of mutexes atomically. That being said, you have to keep two facts in mind.

If one of the current threads already owns the corresponding mutex and the mutex is not recursive, the behaviour is undefined.

You can only take the ownership of the mutex without locking them. In this case, you have to provide the std::adopt_lock_t flag to the constructor: std::scoped_lock(std::adopt_lock_t, MutexTypes& ... m).

You can quite elegantly solve the previous deadlock by using a std::scoped_lock. I will discuss the resolution of the deadlock in the following section.

In the next lesson, we will see how std::unique_lock is stronger but more expensive than its little brother std::lock_guard.

```
*/  
// ======  
// ======  
/*
```

Types of Locks: std::unique_lock

This lesson gives an overview of std::unique_lock which is a type of lock used in C++.

We'll cover the following

Features

Methods

More on lk.try_lock and lk.release methods

How to solve deadlock with std::unique_lock

Features#

In addition to what's offered by a std::lock_guard, a std::unique_lock enables you to

create it without an associated mutex.

create it without locking the associated mutex.

explicitly and repeatedly set or release the lock of the associated mutex.

move the mutex.

try to lock the mutex.

delay the lock on the associated mutex.

Methods#

The following table shows the methods of a std::unique_lock lk.

Method	Description
lk.lock()	Locks the associated mutex.
std::lock(lk1, lk2, ...)	Locks atomically the arbitrary number of associated mutexes.
lk.try_lock() and lk.try_lock_for(relTime) and lk.try_lock_until(absTime)	Tries to lock the associated mutex.
lk.release()	Release the mutex. The mutex remains locked.
lk.swap(lk2) and std::swap(lk, lk2)	Swaps the locks.
lk.mutex()	Returns a pointer to the associated mutex.
lk.owns_lock()	Checks if the lock has a mutex.
More on lk.try_lock and lk.release methods#	
lk.try_lock_for(relTime)	needs a relative time duration; lk.try_lock_until(absTime)
point.	needs an absolute time

lk.try_lock tries to lock the mutex and returns immediately. On success, it returns true, but otherwise, it's false. In contrast the methods lk.try_lock_for and lk.try_lock_until block until the specified timeout occurs or the lock is acquired, whichever comes first. You should use a steady clock for your time constraint. A steady clock cannot be adjusted.

The method lk.release() returns the mutex; therefore, you have to unlock it manually.

How to solve deadlock with std::unique_lock#

Thanks to std::unique_lock, it is quite easy to lock many mutexes in one atomic step; therefore you can overcome deadlocks by locking mutexes in a different order. Remember the deadlock from the subsection Issues of Mutexes?

```

*/
// =====
// deadlock.cpp

// #include <iostream>
// #include <chrono>
// #include <mutex>
// #include <thread>

// struct CriticalData{
//   std::mutex mut;
// };

// void deadLock(CriticalData& a, CriticalData& b){

//   a.mut.lock();
//   std::cout << "get the first mutex" << std::endl;
//   std::this_thread::sleep_for(std::chrono::milliseconds(1));
//   b.mut.lock();
//   std::cout << "get the second mutex" << std::endl;
//   // do something with a and b
//   a.mut.unlock();
//   b.mut.unlock();

// }

// int main(){

//   CriticalData c1;
//   CriticalData c2;

//   std::thread t1([&]{deadLock(c1,c2);});
//   std::thread t2([&]{deadLock(c2,c1);});

//   t1.join();
//   t2.join();

// }
// =====
/*

```

Let's solve the issue. The function `deadLock` has to lock its mutexes atomically and that's exactly what happens in the following example.

```

*/
// =====
// deadlockResolved.cpp

// #include <iostream>
// #include <chrono>
// #include <mutex>
// #include <thread>
```

```

// using namespace std;

// struct CriticalData{
//   mutex mut;
// };

// void deadLock(CriticalData& a, CriticalData& b){

//   unique_lock<mutex> guard1(a.mut,defer_lock);
//   cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;

//   this_thread::sleep_for(chrono::milliseconds(1));

//   unique_lock<mutex> guard2(b.mut,defer_lock);
//   cout << " Thread: " << this_thread::get_id() << " second mutex" << endl;

//   cout << " Thread: " << this_thread::get_id() << " get both mutex" << endl;
//   lock(guard1,guard2);
//   // do something with a and b
// }

// int main(){

//   cout << endl;

//   CriticalData c1;
//   CriticalData c2;

//   thread t1([&]{deadLock(c1,c2)} );
//   thread t2([&]{deadLock(c2,c1)} );

//   t1.join();
//   t2.join();

//   cout << endl;

// }
// =====
/*

```

If you call the constructor of `std::unique_lock` with `std::defer_lock`, the underlying mutex will not be locked automatically. At this point (lines 16 and 21), the `std::unique_lock` is just the owner of the mutex. Thanks to the variadic template `std::lock`, the lock operation is performed in an atomic step (line 25). A variadic template is a template which can accept an arbitrary number of arguments. `std::lock` tries to get all locks in one atomic step, so it either gets all of them or none of them and retries until it succeeds.

In this example, `std::unique_lock` manages the lifetime of the resources and `std::lock` locks the associated mutex; you can also do it the other way around. In the first step the mutexes are locked, in the second `std::unique_lock` manages the lifetime of resources. Here is an example of the second approach.

```

*/
// =====
// std::lock(a.mut, b.mut);

```

```
// std::lock_guard<std::mutex> guard1(a.mut, std::adopt_lock);
// std::lock_guard<std::mutex> guard2(b.mut, std::adopt_lock);
// =====
/*
Let us see this approach in action:
*/
// =====
// deadlockResolved.cpp

// #include <iostream>
// #include <chrono>
// #include <mutex>
// #include <thread>

// using namespace std;

// struct CriticalData{
//   mutex mut;
// };

// void deadLock(CriticalData& a, CriticalData& b){

//   lock_guard<std::mutex> guard1(a.mut, std::adopt_lock);
//   cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;

//   this_thread::sleep_for(chrono::milliseconds(1));

//   lock_guard<std::mutex> guard2(b.mut, std::adopt_lock);
//   cout << " Thread: " << this_thread::get_id() << " second mutex" << endl;

//   cout << " Thread: " << this_thread::get_id() << " get both mutex" << endl;
//   lock(a.mut, b.mut);
//   // do something with a and b
// }

// int main(){

//   cout << endl;

//   CriticalData c1;
//   CriticalData c2;

//   thread t1([&]{deadLock(c1,c2);});
//   thread t2([&]{deadLock(c2,c1);});

//   t1.join();
//   t2.join();

//   cout << endl;

// }
```

```
// =====
/*
Resolving the deadlock with a std::scoped_lock
```

With C++17, the resolution of the deadlock becomes quite easy. We get the std::scoped_lock that can lock an arbitrary number of mutexes atomically - so long as you only have to use a std::lock_guard instead of the std::lock call. That's all. Here is the modified function deadlock.

```
/*
// =====
// deadlockResolvedScopedLock.cpp
// void deadLock(CriticalData& a, CriticalData& b){

// cout << "Thread: " << this_thread::get_id() << " first mutex" << endl;
// this_thread::sleep_for(chrono::milliseconds(1));
// cout << " Thread: " << this_thread::get_id() << " second mutex" << endl;
// cout << " Thread: " << this_thread::get_id() << " get both mutex" << endl;

// std::scoped_lock(a.mut, b.mut);
// // do something with a and b
// }
```

```
// =====
```

```
/*
```

Types of Locks: std::shared_lock

This lesson gives an overview of the std::shared_lock which is a type of lock used in C++.

We'll cover the following

Telephone book example for reader-writer locks

Undefined Behaviour

A std::shared_lock has the same interface as a std::unique_lock but behaves differently when used with a std::shared_timed_mutex. Many threads can share one std::shared_timed_mutex and, therefore, implement a reader-writer lock. The idea of reader-writer locks is straightforward and extremely useful. An arbitrary number of threads executing read operations can access the critical region at the same time, but only one thread is allowed to write.

Reader-writer locks do not solve the fundamental problem - threads competing for access to a critical region, but they do help to minimize the bottleneck.

Telephone book example for reader-writer locks#

A telephone book is a typical example using a reader-writer lock. Usually, a lot of people want to look up a telephone number, but only a few want to change them. Let's look at an example.

```
/*
// =====
// readerWriterLock.cpp

// #include <iostream>
// #include <map>
// #include <shared_mutex>
// #include <string>
// #include <thread>
```

```

// std::map<std::string,int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976},
// {"Ritchie", 1983}};

// std::shared_timed_mutex teleBookMutex;

// void addToTeleBook(const std::string& na, int tele){
//   std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
//   std::cout << "\nSTARTING UPDATE " << na;
//   std::this_thread::sleep_for(std::chrono::milliseconds(500));
//   teleBook[na]= tele;
//   std::cout << "... ENDING UPDATE " << na << std::endl;
// }

// void printNumber(const std::string& na){
//   std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
//   std::cout << na << ":" << teleBook[na];
// }

// int main(){

//   std::cout << std::endl;

//   std::thread reader1([]{ printNumber("Scott"); });
//   std::thread reader2([]{ printNumber("Ritchie"); });
//   std::thread w1([]{ addToTeleBook("Scott",1968); });
//   std::thread reader3([]{ printNumber("Dijkstra"); });
//   std::thread reader4([]{ printNumber("Scott"); });
//   std::thread w2([]{ addToTeleBook("Bjarne",1965); });
//   std::thread reader5([]{ printNumber("Scott"); });
//   std::thread reader6([]{ printNumber("Ritchie"); });
//   std::thread reader7([]{ printNumber("Scott"); });
//   std::thread reader8([]{ printNumber("Bjarne"); });

//   reader1.join();
//   reader2.join();
//   reader3.join();
//   reader4.join();
//   reader5.join();
//   reader6.join();
//   reader7.join();
//   reader8.join();

//   w1.join();
//   w2.join();

//   std::cout << std::endl;

//   std::cout << "\nThe new telephone book" << std::endl;
//   for (auto teleIt: teleBook){
//     std::cout << teleIt.first << ":" << teleIt.second << std::endl;
//   }
}

```

```

// std::cout << std::endl;

//}
//=====
/*
The telephone book in line 9 is the shared variable, which has to be protected. Eight threads want to read the telephone book, two threads want to modify it (lines 31 - 40). To access the telephone book at the same time, the reading threads use the std::shared_lock<std::shared_timed_mutex> in line 23. This is in contrast to the writing threads, which need exclusive access to the critical section. The exclusivity is given by the std::lock_guard<std::shared_timed_mutex> in line 15. In the end, the program displays the updated telephone book (lines 55 - 58). The output of the reading threads overlaps, while the writing threads are executed one after the other. This means that the reading operations are performed at the same time. That was easy. Too easy. The telephone book has undefined behavior

```

Undefined Behaviour#

The program has undefined behavior. To be more precise it has a data race. What? Before you continue, stop for a few seconds and think. By the way the concurrent access to std::cout is not the issue.

The characteristic of a data race is that at least two threads access the shared variable at the same time and at least one of them is a writer. This exact scenario may occur during program execution. One of the features of the ordered associative container is that reading of the container can modify it. This happens if the element is not available in the container. If "Bjarne" is not found in the telephone book, a pair ("Bjarne",0) will be created from the read access. You can simply force the data race by putting the printing of Bjarne in line 40 in front of all the threads (lines 31 - 40). Let's have a look.

You can see it right at the top, Bjarne has the value 0.

```

*/
//=====
// readerWriterLock.cpp

// #include <iostream>
// #include <map>
// #include <shared_mutex>
// #include <string>
// #include <thread>

// std::map<std::string, int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976}, {"Ritchie", 1983}};

// std::shared_timed_mutex teleBookMutex;

// void addToTeleBook(const std::string &na, int tele)
//{
// std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
// std::cout << "\nSTARTING UPDATE " << na;
// std::this_thread::sleep_for(std::chrono::milliseconds(500));
// teleBook[na] = tele;
// std::cout << "... ENDING UPDATE " << na << std::endl;
//}

```

```
// void printNumber(const std::string &na)
//{
//    std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
//    std::cout << na << ":" << teleBook[na];
//}

// int main()
//{
//    std::cout << std::endl;

//    std::thread reader8([])
//        { printNumber("Bjarne"); });
//    std::thread reader1([])
//        { printNumber("Scott"); });
//    std::thread reader2([])
//        { printNumber("Ritchie"); });
//    std::thread w1([])
//        { addToTeleBook("Scott", 1968); });
//    std::thread reader3([])
//        { printNumber("Dijkstra"); });
//    std::thread reader4([])
//        { printNumber("Scott"); });
//    std::thread w2([])
//        { addToTeleBook("Bjarne", 1965); });
//    std::thread reader5([])
//        { printNumber("Scott"); });
//    std::thread reader6([])
//        { printNumber("Ritchie"); });
//    std::thread reader7([])
//        { printNumber("Scott"); });

//    reader1.join();
//    reader2.join();
//    reader3.join();
//    reader4.join();
//    reader5.join();
//    reader6.join();
//    reader7.join();
//    reader8.join();
//    w1.join();
//    w2.join();

//    std::cout << std::endl;

//    std::cout << "\nThe new telephone book" << std::endl;
//    for (auto teleIt : teleBook)
//    {
//        std::cout << teleIt.first << ":" << teleIt.second << std::endl;
//    }
}
```

```

// std::cout << std::endl;
// }
// =====
/*
An obvious way to fix this issue is to use only reading operations in the function printNumber:
*/
// =====
// readerWriterLock.cpp

// #include <iostream>
// #include <map>
// #include <shared_mutex>
// #include <string>
// #include <thread>

// std::map<std::string,int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976},
//                                     {"Ritchie", 1983}};

// std::shared_timed_mutex teleBookMutex;

// void addToTeleBook(const std::string& na, int tele){
//   std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
//   std::cout << "\nSTARTING UPDATE " << na;
//   std::this_thread::sleep_for(std::chrono::milliseconds(500));
//   teleBook[na]= tele;
//   std::cout << "... ENDING UPDATE " << na << std::endl;
// }

// void printNumber(const std::string& na){
//   std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
//   auto searchEntry = teleBook.find(na);
//   if(searchEntry != teleBook.end()){
//     std::cout << searchEntry->first << ":" << searchEntry->second << std::endl;
//   }
//   else {
//     std::cout << na << " not found!" << std::endl;
//   }
// }

// int main(){

//   std::cout << std::endl;

//   std::thread reader8([]{ printNumber("Bjarne"); });
//   std::thread reader1([]{ printNumber("Scott"); });
//   std::thread reader2([]{ printNumber("Ritchie"); });
//   std::thread w1([]{ addToTeleBook("Scott",1968); });
//   std::thread reader3([]{ printNumber("Dijkstra"); });
//   std::thread reader4([]{ printNumber("Scott"); });
//   std::thread w2([]{ addToTeleBook("Bjarne",1965); });
//   std::thread reader5([]{ printNumber("Scott"); });


```

```

// std::thread reader6([]{ printNumber("Ritchie"); });
// std::thread reader7([]{ printNumber("Scott"); });

// reader1.join();
// reader2.join();
// reader3.join();
// reader4.join();
// reader5.join();
// reader6.join();
// reader7.join();
// reader8.join();
// w1.join();
// w2.join();

// std::cout << std::endl;

// std::cout << "\nThe new telephone book" << std::endl;
// for (auto teleIt: teleBook){
//   std::cout << teleIt.first << ":" << teleIt.second << std::endl;
// }

// std::cout << std::endl;

//}

//=====
/*
If a key is not in the telephone book, I will simply write not found to the console.

```

You can see the message Bjarne not found! in the output of the second program execution. In the first program execution, addToTeleBook will be executed first; therefore, Bjarne will be found.

```

*/
//=====
//=====
/*

```

Thread-Safe Initialization

This lesson gives a brief introduction to thread safe initialization of variables in concurrent programming with C++.

If the variable is never modified there is no need for synchronization by using an expensive lock or an atomic. You only have to ensure that it is initialized in a thread-safe way.

There are three ways in C++ to initialize variables in a thread-safe way.

Constant expressions.

The function `std::call_once` in combination with the flag `std::once_flag`.

A static variable with block scope.

Thread-safe initialisation in the main-thread

The easiest and fourth way to initialise a variable in a thread-safe way: initialise the variable in the main-thread before you create any child threads.

We will explain each thread-safe initialization method in the next 3 lessons.

```
/*
// =====
// =====
/*
```

Thread-Safe Initialization: Constant Expressions

This lesson gives an overview of thread-safe initialization in the perspective of concurrency in C++ with Constant Expressions

Constant expressions are expressions that the compiler can evaluate at compile time; they are implicitly thread-safe. Placing the keyword `constexpr` in front of a variable makes the variable a constant expression. The constant expression must be initialized immediately.

1

```
constexpr double pi = 3.14;
```

Additionally, user-defined types can also be constant expressions. For those types, there are a few restrictions that must be met in order to initialize it at compile time.

They must not have virtual methods or a virtual base class.

Their constructor must be empty and itself be a constant expression.

Their methods, which should be callable at compile time, must be constant expressions.

Instances of `MyDouble` satisfy all these requirements, so it is possible to instantiate them at compile time. This instantiation is thread-safe.

```
/*
// =====
// constexpr.cpp

// #include <iostream>

// class MyDouble{
// private:
//   double myVal1;
//   double myVal2;
// public:
//   constexpr MyDouble(double v1,double v2):myVal1(v1),myVal2(v2){}
//   constexpr double getSum() const { return myVal1 + myVal2; }
// };

// int main() {

//   constexpr double myStatVal = 2.0;
//   constexpr MyDouble myStatic(10.5, myStatVal);
//   constexpr double sumStat= myStatic.getSum();
//   std::cout << "SumStat: "<<sumStat << std::endl;
// }

/*
```

Thread-Safe Initialization: `call_once` and `once_flag`

This lesson gives an overview of thread-safe initialization in the perspective of concurrency in C++.

By using the std::call_once function you can register a callable. The std::once_flag ensures that only one registered function will be invoked, but you can register additional functions via the same std::once_flag. That being said, only one function from that group is called.

std::call_once obeys the following rules:

Exactly one execution of exactly one of the functions is performed. It is undefined which function will be selected for execution. The selected function runs in the same thread as the std::call_once invocation it was passed to.

No invocation in the group returns before the above-mentioned execution of the selected function completes successfully.

If the selected function exits via an exception, it is propagated to the caller. Another function is then selected and executed.

The short example demonstrates the application of std::call_once and the std::once_flag. Both of them are declared in the header <mutex>.

```
/*
// =====
// callOnce.cpp

// #include <iostream>
// #include <thread>
// #include <mutex>

// std::once_flag onceFlag;

// void do_once(){
//   std::call_once(onceFlag, [](){ std::cout << "Only once." << std::endl; });
// }

// int main(){
//   std::cout << std::endl;

//   std::thread t1(do_once);
//   std::thread t2(do_once);
//   std::thread t3(do_once);
//   std::thread t4(do_once);

//   t1.join();
//   t2.join();
//   t3.join();
//   t4.join();

//   std::cout << std::endl;

// }
// =====
/*
```

The program starts four threads (lines 17 - 20); each of them invokes do_once. The expected result is that the string “only once” is displayed only once.

The famous singleton pattern guarantees that only one instance of an object will be created. This is a challenging task in multithreading environments, but std::call_once and std::once_flag make the job a piece of cake. Now the singleton is initialized in a thread-safe way.

```
*/
// =====
// singletonCallOnce.cpp

// #include <iostream>
// #include <mutex>

// using namespace std;

// class MySingleton{

// private:
//     static once_flag initInstanceFlag;
//     static MySingleton* instance;
//     MySingleton() = default;
//     ~MySingleton() = default;

// public:
//     MySingleton(const MySingleton&) = delete;
//     MySingleton& operator=(const MySingleton&) = delete;

//     static MySingleton* getInstance(){
//         call_once(initInstanceFlag,MySingleton::initSingleton);
//         return instance;
//     }

//     static void initSingleton(){
//         instance= new MySingleton();
//     }
// };

// MySingleton* MySingleton::instance = nullptr;
// once_flag MySingleton::initInstanceFlag;

// int main(){

//     cout << endl;

//     cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << endl;
//     cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << endl;

//     cout << endl;

// }
// =====
/*
```

Let's first review the static std::once_flag. It is declared in line 11 and initialized in line 31. The static method getInstance (lines 20 - 23) uses the flag initInstanceFlag to ensure that the static method initSingleton (line 25 - 27) is executed exactly once. The singleton is created in the body of the method.

default and delete

You can request special methods from the compiler by using the keyword default. These methods are special because the compiler can create them for us. The result of annotating a method with delete is that the compiler generated methods will not be available and, therefore, cannot be called. If you try to use them, you'll get a compile-time error. Here are the details for the keywords default and delete.

The MySingleton::getInstance() method displays the address of the singleton.

In the next lesson, we will look at how variables can be initialized in a thread-safe way using static variables

```
*/  
// ======  
// ======  
/*
```

Thread-Safe Initialization - Static Variables with Block Scope

This lesson gives an overview of thread-safe initialisation with static variables in the perspective of concurrency in C++.

We'll cover the following

Thread-safe example:

Static variables with block scope will be created exactly once and lazily (i.e. created just at the moment of the usage). This characteristic is the basis of the so-called Meyers Singleton, named after Scott Meyers. This is by far the most elegant implementation of the singleton pattern. With C++11, static variables with block scope have an additional guarantee; they will be initialized in a thread-safe way.

Thread-safe example:#

Here is the thread-safe Meyers Singleton pattern.

```
*/  
// ======  
// meyersSingleton.cpp
```

```
// class MySingleton{  
// public:  
//   static MySingleton& getInstance(){  
//     static MySingleton instance;  
//     return instance;  
//   }  
// private:  
//   MySingleton();  
//   ~MySingleton();  
//   MySingleton(const MySingleton&)= delete;  
//   MySingleton& operator=(const MySingleton&)= delete;  
  
// };  
  
// MySingleton::MySingleton()= default;
```

```
// MySingleton::~MySingleton()= default;
```

```
// int main(){
```

```
//  MySingleton::getInstance();
```

```
// }
```

```
// =====
```

```
/*
```

Know your Compiler support for static

If you use the Meyers Singleton pattern in a concurrent environment, be sure that your compiler implements static variables with the C++11 thread-safe semantic. It happens quite often that programmers rely on the C++11 semantic of static variables, but their compiler does not support it. The result may be that more than one instance of a singleton is created.

In the next chapter, we will study `thread_local` data which has no sharing issues.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

Thread Local Data

This lesson explains how data that is local to a thread is created

Thread-local data, also known as thread-local storage, will be created for each thread separately. It behaves like static data because it's bound for the lifetime of the thread and it will be created at its first usage. Also, thread-local data belongs exclusively to the thread.

```
*/
```

```
// =====
```

```
// threadLocal.cpp
```

```
// #include <iostream>
```

```
// #include <string>
```

```
// #include <mutex>
```

```
// #include <thread>
```

```
// std::mutex coutMutex;
```

```
// thread_local std::string s("hello from ");
```

```
// void addThreadLocal(std::string const& s2){
```

```
//  s += s2;
```

```
//  // protect std::cout
```

```
//  std::lock_guard<std::mutex> guard(coutMutex);
```

```
//  std::cout << s << std::endl;
```

```
//  std::cout << "&s: " << &s << std::endl;
```

```
//  std::cout << std::endl;
```

```
// }
```

```

// int main(){
// std::cout << std::endl;

// std::thread t1(addThreadLocal,"t1");
// std::thread t2(addThreadLocal,"t2");
// std::thread t3(addThreadLocal,"t3");
// std::thread t4(addThreadLocal,"t4");

// t1.join();
// t2.join();
// t3.join();
// t4.join();

//}
// =====
/*

```

By using the keyword `thread_local` in line 10, the thread-local string `s` is created. Threads `t1` - `t4` (lines 27 - 30) use the function `addThreadLocal` (lines 12 - 21) as their work package. Likewise, those threads get the strings `t1` to `t4` respectively as their argument, and add them to the thread-local string `s`. In addition, `addThreadLocal` displays the address of `s` in line 18.

The output of the program shows it implicitly in line 17 and explicitly in line 18. The thread local string is created for each string `s`: First, each output shows a new thread-local string; second, each string `s` has a different address.

From a Single-Threaded to Multithreaded Program

Thread-local data helps to port a single-threaded program to a multithreaded environment. If the global variables are thread-local, there is the guarantee that each thread will get its own copy of the data. Due to this fact, there is no shared mutable state which may cause a data race resulting in undefined behavior.

In contrast to thread-local data, condition variables are not easy to use.

```

*/
// =====
// =====
/*

```

Condition Variables

This lesson explains condition variable such as `wait` s and their usage in C++ for multithreading purposes.

We'll cover the following

The Wait Workflow

Condition variables enable threads to be synchronized via messages. They need the `<condition_variable>` header, one thread to act as a sender, and the other as the receiver of the message; the receiver waits for the notification from the sender. Typical use cases for condition variables are sender-receiver or producer-consumer workflows.

A condition variable can be the sender but also the receiver of the message.

Method	Description
--------	-------------

cv.notify_one() Notifies a waiting thread.
cv.notify_all() Notifies all waiting threads.
cv.wait(lock, ...) Waits for the notification while holding a std::unique_lock.
cv.wait_for(lock, relTime, ...) Waits for a time duration for the notification while holding a std::unique_lock.
cv.wait_until(lock, absTime, ...) Waits until a time point for the notification while holding a std::unique_lock.

The subtle difference between cv.notify_one and cv.notify_all is that cv.notify_all will notify all waiting threads. In contrast, cv.notify_one will notify only one of the waiting threads while the other threads remain in the wait state. Before we cover the gory details of condition variables - which are the three dots in the wait operations - here is an example.

```
/*
// =====
// conditionVariable.cpp

// #include <iostream>
// #include <condition_variable>
// #include <mutex>
// #include <thread>

// std::mutex mutex_;
// std::condition_variable condVar;

// bool dataReady{false};

// void doTheWork(){
//   std::cout << "Processing shared data." << std::endl;
// }

// void waitingForWork(){
//   std::cout << "Worker: Waiting for work." << std::endl;
//   std::unique_lock<std::mutex> lck(mutex_);
//   condVar.wait(lck, []{ return dataReady; });
//   doTheWork();
//   std::cout << "Work done." << std::endl;
// }

// void setDataReady(){
//   {
//     std::lock_guard<std::mutex> lck(mutex_);
//     dataReady = true;
//   }
//   std::cout << "Sender: Data is ready." << std::endl;
//   condVar.notify_one();
// }

// int main(){
//   std::cout << std::endl;
//   std::thread t1(waitingForWork);
//   std::thread t2(setDataReady);
// }
```

```

// t1.join();
// t2.join();

// std::cout << std::endl;

//}
// =====
/*

```

The program has two child threads: t1 and t2. They get their work package waitingForWork and setDataRead in lines 38 and 39. setDataReady notifies - using the condition variable condVar - that it is done with the preparation of the work: condVar.notify_one(). While holding the lock, thread t1 waits for its notification: condVar.wait(lck,[){ return dataReady; }). Meanwhile, the sender and receiver need a lock. In the case of the sender a std::lock_guard is sufficient, because it calls to lock and unlock only once. In the case of the receiver, a std::unique_lock is necessary because it frequently locks and unlocks its mutex. The waiting thread has quite a complicated workflow.

The Wait Workflow#

If it is the first time wait is invoked, the following steps will happen.

The call to wait locks the mutex and checks if the predicate []{ return dataReady; } evaluates to true.

If true, the condition variable unlocks the mutex and continues.

If false, the condition variable unlocks the mutex and puts itself back in the wait state.

Subsequent wait calls behave differently:

The waiting thread gets a notification. It locks the mutex and checks if the predicate []{ return dataReady; } evaluates to true.

If true, the condition variable unlocks the mutex and continues.

If false, the condition variable unlocks the mutex and puts itself back in the wait state.

Maybe you are wondering why you need a predicate for the wait call when you can invoke wait without a predicate? Let's try it out.

```

*/
// =====
// conditionVariableBlock.cpp

```

```

// #include <iostream>
// #include <condition_variable>
// #include <mutex>
// #include <thread>

// std::mutex mutex_;
// std::condition_variable condVar;

// bool dataReady{false};

// void waitingForWork(){

//   std::cout << "Worker: Waiting for work." << std::endl;
//   std::unique_lock<std::mutex> lck(mutex_);

```

```

// condVar.wait(lck);
// // do the work
// std::cout << "Work done." << std::endl;

//}

// void setDataReady(){

// std::cout << "Sender: Data is ready." << std::endl;
// condVar.notify_one();

//}

// int main(){

// std::cout << std::endl;

// std::thread t1(setDataReady);
// std::thread t2(waitingForWork);

// t1.join();
// t2.join();

// std::cout << std::endl;

//}

// =====
/*

```

The first invocation of the program seems to work fine. The second invocation locks because the notification call (line 28) happens before thread t2 (line 37) enters the waiting state (line 19).

Now it is clear. The predicate is a kind of memory for the stateless condition variable; therefore, the wait call always checks the predicate at first. Condition variables are victim to two known phenomena: lost wakeup and spurious wakeup. We will discuss these phenomena in the next lesson.

```

*/
// =====
// =====
/*

```

The Caveats of Condition Variables

In this lesson, we discuss the lost wakeup and spurious wakeup pitfall of condition variables with concurrency in C++

We'll cover the following

Lost Wakeup

Spurious Wakeup

Lost Wakeup#

The phenomenon of the lost wakeup is that the sender sends its notification before the receiver gets to a wait state. The consequence is that the notification is lost. The C++ standard describes condition variables as a simultaneous synchronization mechanism: “The condition_variable class is a synchronization primitive that

can be used to block a thread, or multiple threads at the same time, ...". So, the notification gets lost and the receiver is waiting, and waiting, and...

Spurious Wakeup#

It can happen that the receiver wakes up, although no notification happened. At a minimum, POSIX Threads and the Windows API can be victims of these phenomena.

In most of the use-cases, tasks are the less error-prone way to synchronize threads.

```
*/  
// ======  
// ======  
/*
```

Introduction to Tasks

This lesson gives an introduction to tasks which consist of promises and futures, and are an important part of multithreading in C++.

In addition to threads, C++ has tasks to perform work asynchronously. These tasks need the <future> header, and they will be parameterized with a work package. Additionally, they consist of two associated components: a promise and a future; Both are connected via a data channel. The promise executes the work packages and puts the result in the data channel; the associated future picks up the result. Both communication endpoints can run in separate threads. What is special is that the future can pick up the result at a later time; therefore, the calculation of the result by the promise is independent of the query of the result by the associated future.

 Regard tasks as data channels between communication endpoints

Tasks behave like data channels between communication endpoints. One endpoint of the data channel is called the promise, the other endpoint of the data channel is called the future. These endpoints can exist in the same or in different threads. The promise puts its result in the data channel. The future waits for it and picks it up.

```
*/  
// ======  
// ======  
/*
```

Threads vs Tasks

This lesson highlights the differences between threads and tasks used in C++ for multithreading.

Threads are very different from tasks. Let's see how by looking at this piece of code first:

```
*/  
// ======  
// asyncVersusThread.cpp  
  
// #include <future>  
// #include <thread>  
// #include <iostream>  
  
// int main(){  
  
// std::cout << std::endl;
```

```

// int res;
// std::thread t([&]{ res = 2000 + 11; });
// t.join();
// std::cout << "res: " << res << std::endl;

// auto fut= std::async([]{ return 2000 + 11; });
// std::cout << "fut.get(): " << fut.get() << std::endl;

// std::cout << std::endl;

//}
//=====
/*

```

The child thread `t` and the asynchronous function call `std::async` to calculate both the sum of 2000 and 11. The creator thread gets the result from its child thread `t` via the shared variable `res` and displays it in line 14. The call `std::async` in line 16 creates the data channel between the sender (promise) and the receiver (future). Following that, the future asks the data channel with `fut.get()` (line 17) for the result of the calculation; this `fut.get` call is blocking.

Based on this program, I want to explicitly emphasize the differences between threads and tasks

```

*/
//=====
//=====
/*
Criteria      Threads      Tasks
Participants   creator and child thread    promise and future
Communication  shared variable        communication channel
Thread creation obligatory      optional
Synchronisation via join() (waits)    get call blocks
Exception in child thread    child and creator threads terminates return value of the promise
Kinds of communication    values values, notifications, and exceptions
Threads need the <thread> header; tasks the <future> header.

```

Communication between the creator thread and the created thread requires the use of a shared variable. The task communicates via its data channel which is implicitly protected; therefore, a task must not use a protection mechanism like a mutex.

While you can misuse a global mutable variable to communicate between the child and its creator, the communication of a task is more explicit. The future can request the result of the task only once (by calling `fut.get()`). Calling it more than once results in undefined behavior. This is not true for a `std::shared_future`, which can be queried multiple times.

The creator thread waits for its child with the call to `join`. The future `fut` uses the `fut.get()` call which blocks until the result is available. If an exception is thrown in the created thread, the created thread will terminate and so will the creator and the whole process. In contrast, the promise can send the exception to the future, which has to handle the exception.

A promise can serve one or many futures, and it can send a value, an exception, or just a notification. In addition, you can use a safe replacement for a condition variable. `std::async` is the easiest way to create a future and we'll see why in the next lesson.

```
*/
```

```
// =====
// =====
/*
```

Introduction to std::async

This lesson gives an introduction to std::async which is used in C++ for multithreading.

std::async behaves like an asynchronous function call. This function call takes a callable together with its arguments. std::async is a variadic template and can, therefore, take an arbitrary number of arguments. The call to std::async returns a future object fut. That's your handle for getting the result via fut.get().

std::async should be your first choice

The C++ runtime decides if std::async is executed in a separate thread. The decision of the C++ runtime may depend on the number of CPU cores available, the utilization of your system, or the size of your work package. By using std::async, you only specify the task that should run; the C++ runtime automatically manages the creation and also the lifetime of the thread.

Optionally, you can specify a start policy for std::async.

```
*/
// =====
// =====
/*
```

Introduction to std::async

This lesson gives an introduction to std::async which is used in C++ for multithreading.

std::async behaves like an asynchronous function call. This function call takes a callable together with its arguments. std::async is a variadic template and can, therefore, take an arbitrary number of arguments. The call to std::async returns a future object fut. That's your handle for getting the result via fut.get().

std::async should be your first choice

The C++ runtime decides if std::async is executed in a separate thread. The decision of the C++ runtime may depend on the number of CPU cores available, the utilization of your system, or the size of your work package. By using std::async, you only specify the task that should run; the C++ runtime automatically manages the creation and also the lifetime of the thread.

Optionally, you can specify a start policy for std::async.

```
/*
// =====
// asyncLazy.cpp
```

```
// #include <chrono>
// #include <future>
// #include <iostream>
```

```
// int main(){
//   std::cout << std::endl;
```

```
//   auto begin= std::chrono::system_clock::now();
```

```

// auto asyncLazy=std::async(std::launch::deferred,
//                           []{ return std::chrono::system_clock::now(); });

// auto asyncEager=std::async(std::launch::async,
//                            []{ return std::chrono::system_clock::now(); });

// std::this_thread::sleep_for(std::chrono::seconds(1));

// auto lazyStart= asyncLazy.get() - begin;
// auto eagerStart= asyncEager.get() - begin;

// auto lazyDuration= std::chrono::duration<double>(lazyStart).count();
// auto eagerDuration= std::chrono::duration<double>(eagerStart).count();

// std::cout << "asyncLazy evaluated after : " << lazyDuration
//                  << " seconds." << std::endl;
// std::cout << "asyncEager evaluated after: " << eagerDuration
//                  << " seconds." << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

Both std::async calls (lines 13 and 16) return the current time point, but the first call is lazy while the second eager; the short sleep of one second in line 19 makes that obvious. The call `asyncLazy.get()` in line 21 will trigger the execution of the promise in line 13 - the result will be available after a short nap of one second (line 19). This is not true for `asyncEager`, as `asyncEager.get()` gets the result from the immediately executed work package.

You do not have to bind a future to a variable.

```

*/
// =====
// =====
/*

```

`async`: Fire and Forget

This lesson gives an overview of fire and forget used with `std::async` in C++ for multithreading.

Fire and forget futures are special futures; they execute just in place because their future is not bound to a variable. For a “fire and forget” future, it is necessary that the promise runs in a separate thread so it can immediately start its work. This is done by the `std::launch::async` policy.

Let’s compare an ordinary future with a fire and forget future.

```

*/
// =====
// #include<iostream>
// #include<future>

// int main(){

```

```

// auto fut= std::async([]{ return 2011; });
// std::cout << fut.get() << std::endl;

// std::async(std::launch::async,
//           []{ std::cout << "fire and forget" << std::endl; });
// */
// =====
/*
Fire and forget futures look very promising but have a big drawback. A future that is created by std::async waits on its destructor, until its promise is done. In this context, waiting is not very different from blocking. The future blocks the progress of the program in its destructor. This becomes more obvious when you use fire and forget futures. What seems to be concurrent actually runs sequentially.
*/
// =====
// fireAndForgetFutures.cpp

```

```

// #include <chrono>
// #include <future>
// #include <iostream>
// #include <thread>

// int main(){

// std::cout << std::endl;

// std::async(std::launch::async, []{
//   std::this_thread::sleep_for(std::chrono::seconds(2));
//   std::cout << "first thread" << std::endl;
// });

// std::async(std::launch::async, []{
//   std::this_thread::sleep_for(std::chrono::seconds(1));
//   std::cout << "second thread" << std::endl;
// });

// std::cout << "main thread" << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

The program executes two promises in their own threads. The resulting futures are fire and forget futures. These futures block in their destructors until the associated promise is done. The result is that the promises will be executed in the sequence which you find in the source code. That being said, the execution sequence is independent of the execution time; this is exactly what you see in the output of the program.

std::async is a convenient mechanism used to distribute a bigger compute job on more shoulders.

```

*/
// =====
// =====

```

```
/*
async: Concurrent Calculation
This lesson gives an overview of concurrent calculation used with std::async in C++ for multithreading.
```

The calculation of the scalar product can be spread across four asynchronous function calls.

```
/*
// =====
// dotProductAsync.cpp

// #include <iostream>
// #include <future>
// #include <random>
// #include <vector>
// #include <numeric>

// using namespace std;

// static const int NUM= 100000000;

// long long getDotProduct(vector<int>& v, vector<int>& w){

// auto vSize = v.size();

// auto future1 = async([&]{
//     return inner_product(&v[0], &v[vSize/4], &w[0], OLL);
// });

// auto future2 = async([&]{
//     return inner_product(&v[vSize/4], &v[vSize/2], &w[vSize/4], OLL);
// });

// auto future3 = async([&]{
//     return inner_product(&v[vSize/2], &v[vSize* 3/4], &w[vSize/2], OLL);
// });

// auto future4 = async([&]{
//     return inner_product(&v[vSize * 3/4], &v[vSize], &w[vSize * 3/4], OLL);
// });

// return future1.get() + future2.get() + future3.get() + future4.get();
// }

// int main(){

// cout << endl;

// random_device seed;

// // generator
// mt19937 engine(seed());
```

```

// // distribution
// uniform_int_distribution<int> dist(0, 100);

// // fill the vectors
// vector<int> v, w;
// v.reserve(NUM);
// w.reserve(NUM);
// for (int i=0; i< NUM; ++i){
//   v.push_back(dist(engine));
//   w.push_back(dist(engine));
// }

// cout << "getDotProduct(v, w): " << getDotProduct(v, w) << endl;

// cout << endl;

// }
// =====
/*

```

The program uses the functionality of the random and time libraries - both libraries are part of C++11. The two vectors v and w are created and filled with random numbers (lines 50 - 56). Each of the vectors (lines 53 - 56) gets one hundred million elements. dist(engine) in lines 54 and 55 generates the random numbers, which are uniformly distributed in the range 0 to 100. The calculation of the scalar product takes place in getDotProduct (lines 13 - 34). Internally, std::async uses the standard template library algorithm std::inner_product. The return statement sums up the results of the futures.

std::packaged_task is also usually used to perform a concurrent computation.

```

*/
// =====
// =====
/*

```

Introduction to std::packaged_task

This lesson gives an introduction to std::packaged_task which is used in C++ for multithreading.

We'll cover the following

Explanation:

std::packaged_task pack is a wrapper for a callable in order for it to be invoked asynchronously. By calling pack.get_future() you get the associated future. Invoking the call operator on pack (pack()) executes the std::packaged_task and, therefore, executes the callable.

Dealing with std::packaged_task usually consists of four steps:

I. Wrap your work:

```
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a + b; });
```

II. Create a future:

```
std::future<int> sumResult= sumTask.get_future();
```

III. Perform the calculation:

```
sumTask(2000, 11);
```

IV. Query the result:

```
sumResult.get();
```

```
sumResult.get();
```

```
*/
```

```
// =====
```

```
// packagedTask.cpp
```

```
// #include <utility>
```

```
// #include <future>
```

```
// #include <iostream>
```

```
// #include <thread>
```

```
// #include <deque>
```

```
// class SumUp{
```

```
// public:
```

```
// int operator()(int beg, int end){
```

```
//     long long int sum{0};
```

```
//     for (int i = beg; i < end; ++i ) sum += i;
```

```
//     return sum;
```

```
// }
```

```
//};
```

```
// int main(){
```

```
// std::cout << std::endl;
```

```
// SumUp sumUp1;
```

```
// SumUp sumUp2;
```

```
// SumUp sumUp3;
```

```
// SumUp sumUp4;
```

```
// // wrap the tasks
```

```
// std::packaged_task<int(int, int)> sumTask1(sumUp1);
```

```
// std::packaged_task<int(int, int)> sumTask2(sumUp2);
```

```
// std::packaged_task<int(int, int)> sumTask3(sumUp3);
```

```
// std::packaged_task<int(int, int)> sumTask4(sumUp4);
```

```
// // create the futures
```

```
// std::future<int> sumResult1 = sumTask1.get_future();
```

```
// //std::future<int> sumResult2 = sumTask2.get_future();
```

```
// auto sumResult2 = sumTask2.get_future();
```

```
// std::future<int> sumResult3 = sumTask3.get_future();
```

```
// //std::future<int> sumResult4 = sumTask4.get_future();
```

```
// auto sumResult4 = sumTask4.get_future();
```

```

// // push the tasks on the container
// std::deque<std::packaged_task<int(int,int)>> allTasks;
// allTasks.push_back(std::move(sumTask1));
// allTasks.push_back(std::move(sumTask2));
// allTasks.push_back(std::move(sumTask3));
// allTasks.push_back(std::move(sumTask4));

// int begin{1};
// int increment{2500};
// int end = begin + increment;

// // perform each calculation in a separate thread
// while (not allTasks.empty()){
//   std::packaged_task<int(int, int)> myTask = std::move(allTasks.front());
//   allTasks.pop_front();
//   std::thread sumThread(std::move(myTask), begin, end);
//   begin = end;
//   end += increment;
//   sumThread.detach();
// }

// // pick up the results
// auto sum = sumResult1.get() + sumResult2.get() +
//           sumResult3.get() + sumResult4.get();

// std::cout << "sum of 0 .. 10000 = " << sum << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

The purpose of the program is to calculate the sum of all numbers from 0 to 10000 with the help of four std::packaged_task, each running in a separate thread. The associated futures are used to sum up the final result; of course, you can also use the Gaußschen Summenformel.

Explanation:#

I. Wrap the tasks: I pack the work packages in std::packaged_task (lines 28 - 31) objects. Work packages are instances of the class SumUp (lines 9 - 16). The work is done in the call operator (lines 11 - 15) which sums up all numbers from beg to end -1 and returns the sum as the result. std::packaged_task (lines 28 - 31) can deal with callables that need two ints and return an int: int(int, int).

II. Create the futures: I have to create the future objects with the help of std::packaged_task objects (lines 34 to 3). The packaged_task is the promise in the communication channel. The type of the future is defined explicitly as std::future<int> sumResult1= sumTask1.get_future(), but the compiler can do that job for me with auto sumResult1= sumTask1.get_future().

III. Perform the calculations: Now the calculation takes place. First, the packaged_task is moved onto the std::deque (lines 43 - 46), then each packaged_task (lines 54 - 59) is executed in the while loop. To do that, I move the head of the std::deque in an std::packaged_task (line 54), move the packaged_task in a new thread

(line 56) and let it run in the background (line 59). I used the move semantic in lines 54 and 56 because std::packaged_task objects are not copyable. This restriction holds for all promises, but also for futures and threads. However, there is one exception to this rule: std::shared_future.

IV. Pick up the results: In the final step I ask all futures for their value and sum them up (line 63).

The class templates std::promise and std::future provide you the full control over tasks.

```
/*
// =====
// =====
/*
```

Introduction to Promises and Futures

This lesson gives an introduction to std::promise and std::future which are used in C++ for multithreading.

We'll cover the following

```
std::promise
std::future
```

Promise and future are a mighty pair. A promise can put a value, an exception, or simply a notification into the shared data channel. One promise can serve many std::shared_future futures. With C++20 we may get extended futures that are compose-able.

Here is an introductory example of the usage of std::promise and std::future. Both communication endpoints can be moved to separate threads, so the communication takes place between threads.

```
/*
// =====

// promiseFuture.cpp

// #include <future>
// #include <iostream>
// #include <thread>
// #include <utility>

// void product(std::promise<int>&& intPromise, int a, int b){
//   intPromise.set_value(a*b);
// }

// struct Div{

//   void operator() (std::promise<int>&& intPromise, int a, int b) const {
//     intPromise.set_value(a/b);
//   }

// };

// int main(){

//   int a = 20;
//   int b = 10;
```

```

// std::cout << std::endl;

// // define the promises
// std::promise<int> prodPromise;
// std::promise<int> divPromise;

// // get the futures
// std::future<int> prodResult = prodPromise.get_future();
// std::future<int> divResult = divPromise.get_future();

// // calculate the result in a separate thread
// std::thread prodThread(product, std::move(prodPromise), a, b);
// Div div;
// std::thread divThread(div, std::move(divPromise), a, b);

// // get the result
// std::cout << "20*10 = " << prodResult.get() << std::endl;
// std::cout << "20/10 = " << divResult.get() << std::endl;

// prodThread.join();

// divThread.join();

// std::cout << std::endl;

// }
// =====
/*

```

Thread prodThread (line 36) gets the function product (lines 8 -10), the prodPromise (line 32) and the numbers a and b. To understand the arguments of prodThread, you have to look at the signature of the function. prodThread needs as its first argument a callable; this is the previously mentioned function product. The function product requires a promise of the kind rvalue reference (`std::promise<int>&& intPromise`) and two numbers. These are the last three arguments of prodThread. `std::move` in line 36 creates an rvalue reference - and the rest is a piece of cake. divThread (line 38) divides the two numbers a and b. For its job, it uses the instance div of the class Div (lines 12 - 18). div is an instance of a function object.

The future picks up the results by calling `prodResult.get()` and `divResult.get()`.

`std::promise#`

`std::promise` enables you to set a value, a notification, or an exception. In addition, the promise can provide its result in a delayed fashion.

Method Description

`prom.swap(prom2)` and Swaps the promises.

`std::swap(prom, prom2)`

`prom.get_future()` Returns the future.

`prom.set_value(val)` Sets the value.

`prom.set_exception(ex)` Sets the exception.

`prom.set_value_at_thread_exit(val)` Stores the value and makes it ready if the promise exits.

`prom.set_exception_at_thread_exit(ex)` Stores the exception and makes it ready if the promise exits.

If the value or the exception is set by the promise more than once, a std::future_error exception is thrown.

std::future#

A std::future enables you to

pick up the value from the promise.

ask the promise if the value is available.

wait for the notification of the promise. This waiting can be done with a relative time duration or an absolute time point.

create a shared future (std::shared_future).

Method Description

fut.share() Returns a std::shared_future. Afterwards, the result is not available anymore.

fut.get() Returns the result which can be a value or an exception.

fut.valid() Checks if the result is available. After calling fut.get() it returns false.

fut.wait() Waits for the result.

fut.wait_for(relTime) Waits for the result, but not longer than for a relTime.

fut.wait_until(absTime) Waits for the result, but not longer than until abstime.

If a future fut asks for the result more than once, a std::future_error exception is thrown.

There is a one-to-one relationship between the promise and the future. In contrast, std::shared_future supports one-to-many relations between a promise and many futures which we will discuss in the next lesson.

*/

// =====

// =====

/*

Promise and Future : Return an Exception

This lesson teaches how to return an exception while using std::promise and future in C++ for multithreading.

The function executeDivision displays the result of the calculation or the exception.

*/

// =====

// promiseFutureException.cpp

```
// #include <exception>
// #include <future>
// #include <iostream>
// #include <thread>
// #include <utility>

// struct Div{
// void operator()(std::promise<int>&& intPromise, int a, int b){
// try{
// if ( b==0 ){
// std::string errMess = std::string("Illegal division by zero: ") +
// std::to_string(a) + "/" + std::to_string(b);
// throw std::runtime_error(errMess);
// }
// intPromise.set_value(a/b);
// }
```

```

// catch (...){
//   intPromise.set_exception(std::current_exception());
// }
// }
//};

// void executeDivision(int nom, int denom){
// std::promise<int> divPromise;
// std::future<int> divResult= divPromise.get_future();

// Div div;
// std::thread divThread(div,std::move(divPromise), nom, denom);

// // get the result or the exception
// try{
//   std::cout << nom << "/" << denom << " = " << divResult.get() << std::endl;
// }
// catch (std::runtime_error& e){
//   std::cout << e.what() << std::endl;
// }

// divThread.join();
// }

// int main(){

// std::cout << std::endl;

// executeDivision(20, 0);
// executeDivision(20, 10);

// std::cout << std::endl;

// }
// =====
/*

```

The promise deals with the issue that the denominator is 0. If the denominator is 0, it sets the exception as return value: `intPromise.set_exception(std::current_exception())` in line 20. Following that, the future has to deal with the exception in its try-catch block (lines 33 - 38).

Here is the output of the program.

If possible, use tasks as a safe replacement of condition variables.

```

*/
// =====
// =====
/*

```

Promise and Future: Return a Notification

This lesson teaches how to return a notification while using `std::promise` and `future` in C++ for multithreading.

If you use promises and futures to synchronize threads, they have a lot in common with condition variables. Most of the time, promises and futures are the better choices. Before I present you an example, here is the big picture.

Criteria	Condition Variables	Tasks
Multiple synchronizations	Yes	No
Critical section	Yes	No
Error handling in receiver	No	Yes
Spurious wakeup	Yes	No
Lost wakeup	Yes	No

The advantage of a condition variable to a promise and future is that you can use condition variables to synchronize threads multiple times. In contrast to that, a promise can send its notification only once, so you have to use more promise and future pairs to get the functionality of a condition variable. If you use the condition variable for only one synchronization, the condition variable is a lot more difficult to use in the right way. A promise and future pair needs no shared variable and, therefore, it doesn't have a lock, and isn't prone to spurious or lost wakeups. In addition to that, tasks can handle exceptions. There are lots of reasons to prefer tasks to condition variables.

Do you remember how difficult it was to use condition variables? If not, here are the key parts required to synchronize two threads.

```
*/
// =====
// void waitingForWork(){
//   std::cout << "Worker: Waiting for work." << std::endl;

//   std::unique_lock<std::mutex> lck(mutex_);
//   condVar.wait(lck, []{ return dataReady; });
//   doTheWork();
//   std::cout << "Work done." << std::endl;
// }

// void setDataReady(){
//   std::lock_guard<std::mutex> lck(mutex_);
//   dataReady=true;
//   std::cout << "Sender: Data is ready." << std::endl;
//   condVar.notify_one();
// }
// =====
/*
```

The function `setDataReady` performs the notification part of the synchronization - with the function `waitingForWork` as the waiting part of the synchronization.

Here is the same workflow with tasks.

```
/*
// =====
// promiseFutureSynchronise.cpp

// #include <future>
// #include <iostream>
```

```

// #include <utility>

// void doTheWork(){
//   std::cout << "Processing shared data." << std::endl;
// }

// void waitingForWork(std::future<void>&& fut){

//   std::cout << "Worker: Waiting for work." << std::endl;
//   fut.wait();
//   doTheWork();
//   std::cout << "Work done." << std::endl;

// }

// void setDataReady(std::promise<void>&& prom){

//   std::cout << "Sender: Data is ready." << std::endl;
//   prom.set_value();

// }

// int main(){

//   std::cout << std::endl;

//   std::promise<void> sendReady;
//   auto fut = sendReady.get_future();

//   std::thread t1(waitingForWork, std::move(fut));
//   std::thread t2(setDataReady, std::move(sendReady));

//   t1.join();
//   t2.join();

//   std::cout << std::endl;

// }
// =====
/*
That was quite easy.

```

Thanks to `sendReady` (line 32) you get a future `fut` (line 34). The promise communicates using its return value `void (std::promise<void> sendReady)` that it is only capable of sending notifications. Both communication endpoints are moved into threads `t1` and `t2` (lines 35 and 36). The future waits using the call `fut.wait()` (line 15) for the notification of the promise: `prom.set_value()` (line 24).

The structure and the output of the program matches the corresponding program in the section condition variable.

```

*/
// =====

```

```
// =====
/*
```

Introduction to std::shared_future

This lesson gives an introduction to std::shared_future which is used in C++ for multithreading.

We'll cover the following

std::shared_future

std::shared_future#

The future creates a shared future by using fut.share(). Shared future is associated with its promise and can independently ask for the result. A std::shared future has the same interface as a std::future.

In addition to the std::future, a std::shared_future enables you to query the promise independently of the other associated futures.

There are two ways to create a std::shared_future:

Invoke fut.share() on a std::future fut. Afterwards, the result is no longer available. That means valid == false
Initialize a std::shared_future from a std::promise: std::shared_future<int> divResult = divPromise.get_future()
The handling of a std::shared_future is special.

```
*/
```

```
// =====
```

```
// sharedFuture.cpp
```

```
// #include <future>
// #include <iostream>
// #include <thread>
// #include <utility>
```

```
// std::mutex coutMutex;
```

```
// struct Div{
```

```
// void operator()(std::promise<int>&& intPromise, int a, int b){
//   intPromise.set_value(a/b);
// }
```

```
// };
```

```
// struct Requestor{
```

```
// void operator ()(std::shared_future<int> shaFut){
//
//   // lock std::cout
//   std::lock_guard<std::mutex> coutGuard(coutMutex);

//   // get the thread id
//   std::cout << "threadId(" << std::this_thread::get_id() << "):" ;

//   std::cout << "20/10= " << shaFut.get() << std::endl;
```

```

// }

//};

// int main(){

// std::cout << std::endl;

// // define the promises
// std::promise<int> divPromise;

// // get the futures
// std::shared_future<int> divResult = divPromise.get_future();

// // calculate the result in a separate thread
// Div div;
// std::thread divThread(div, std::move(divPromise), 20, 10);

// Requestor req;
// std::thread sharedThread1(req, divResult);
// std::thread sharedThread2(req, divResult);
// std::thread sharedThread3(req, divResult);
// std::thread sharedThread4(req, divResult);
// std::thread sharedThread5(req, divResult);

// divThread.join();

// sharedThread1.join();
// sharedThread2.join();
// sharedThread3.join();
// sharedThread4.join();
// sharedThread5.join();

// std::cout << std::endl;

// }
// =====
/*

```

Both work packages, that of the promise and that of the future, are function objects in this current example. In line 46 `divPromise` will be moved and executed in thread `divThread`. Accordingly, `std::shared_future`'s are copied in all five threads (lines 57 - 61). It's important to emphasize it once more: In contrast to an `std::future` object that can only be moved, you can copy an `std::shared_future` object.

The main thread waits in lines 57 to 61 for its child threads to finish their jobs and to display their results.

Dividing a number by 0 is undefined behavior.

```

*/
// =====
// =====
/*

```

Introduction

An introduction to a problem of calculating the sum of a vector, and how to solve it through various methods in C++.

We'll cover the following

Calculating the Sum of a Vector

After providing the theory on the memory model and the multithreading interface, I will now apply the theory in practice and provide you a few performance numbers.

Calculating the Sum of a Vector#

What is the fastest way to add the elements of a std::vector? To get the answer, I will fill a std::vector with one hundred million arbitrary but uniformly distributed numbers between 1 and 10. The task is to calculate the sum of the numbers in various ways; I use the performance of a single threaded addition as the reference execution time. In this chapter, I will also discuss atomics, locks, thread local data, and tasks.

Let's start with the single-threaded scenario in the next lesson.

```
*/  
// ======  
// ======  
/*
```

Single Threaded Summation: Ranged Based for Loops

Explaining the solution for calculating the sum of a vector problem using ranged-based for loops in C++.

The obvious strategy is it to add the numbers in a range-based for loop like we did in the code below.

The summation takes place in line 27:

```
*/  
// ======  
// calculateWithLoop.cpp  
  
// #include <chrono>  
// #include <iostream>  
// #include <random>  
// #include <vector>  
  
// constexpr long long size = 100000000;  
  
// int main()  
// {  
  
//     std::cout << std::endl;  
  
//     std::vector<int> randValues;  
//     randValues.reserve(size);  
  
//     // random values  
//     std::random_device seed;  
//     std::mt19937 engine(seed());  
//     std::uniform_int_distribution<int> uniformDist(1, 10);
```

```

// for (long long i = 0; i < size; ++i)
//   randValues.push_back(uniformDist(engine));

// const auto sta = std::chrono::steady_clock::now();

// unsigned long long sum = {};
// for (auto n : randValues)
//   sum += n;

// const std::chrono::duration<double> dur =
//   std::chrono::steady_clock::now() - sta;

// std::cout << "Time for mySumition " << dur.count()
//       << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;
//}
//=====================================================================
/*

```

You should not use loops explicitly. Most of the time you can use an algorithm from the Standard Template Library.

```

*/
//=====================================================================
// =====
/*
```

Single Threaded Summation: Addition with std::accumulate

This lesson explains the solution for calculating the sum of a vector problem using std::accumulate in C++.

std::accumulate is the right way to calculate the sum of a vector. For the sake of simplicity, I will only show the application of std::accumulate.

```

*/
//=====================================================================
// calculateWithStd.cpp

// ...

// const auto sum = std::accumulate(randValues.begin(),
//                                 randValues.end(), 0);

// ...

//=====================================================================
/*

```

```

*/
//=====================================================================
// calculateWithStd.cpp

// #include <chrono>
// #include <iostream>
```

```
// #include <random>
// #include <vector>

// constexpr long long size = 100000000;

// int main(){

// std::cout << std::endl;

// std::vector<int> randValues;
// randValues.reserve(size);

// // random values
// std::random_device seed;
// std::mt19937 engine(seed());
// std::uniform_int_distribution<> uniformDist(1, 10);
// for (long long i = 0 ; i < size ; ++i)
//     randValues.push_back(uniformDist(engine));

// const auto sta = std::chrono::steady_clock::now();

// const auto sum = std::accumulate(randValues.begin(),
//                                 randValues.end(), 0);
// const std::chrono::duration<double> dur =
//     std::chrono::steady_clock::now() - sta;

// std::cout << "Time for mySumition " << dur.count()
// << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }

// =====
/*
//calculateWithStd.cpp
```

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

constexpr long long size = 100000000;

int main(){

    std::cout << std::endl;

    std::vector<int> randValues;
    randValues.reserve(size);
```

```

// random values
std::random_device seed;
std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(1, 10);
for (long long i = 0 ; i < size ; ++i)
    randValues.push_back(uniformDist(engine));

const auto sta = std::chrono::steady_clock::now();

const auto sum = std::accumulate(randValues.begin(),
                                randValues.end(), 0);
const std::chrono::duration<double> dur =
    std::chrono::steady_clock::now() - sta;

std::cout << "Time for mySumition " << dur.count()
    << " seconds" << std::endl;
std::cout << "Result: " << sum << std::endl;

std::cout << std::endl;

}
*/
// =====
// =====
/*
Single Threaded Summation: Protection with Locks
This lesson explains the solution for calculating the sum of a vector problem using locks in C++.

```

If I protect access to the summation variable with a lock, I will get the answers to two questions.

How expensive is the synchronization of a lock without contention?

How fast can a lock be in the optimal case?

I can draw an interesting conclusion from question 2. If there is contention on a lock, the access time will decrease. That being said, I will only show the application of `std::lock_guard`.

```

*/
// =====
// calculateWithLock.cpp

// ...

// std::mutex myMutex;

// for (auto i: randValues){
//     std::lock_guard<std::mutex> myLockGuard(myMutex);
//     sum += i;
// }

// ...
// =====
/*
Let's see the above fragment of code in action:

```

```

*/
// =====
// =====
/*
// calculateWithLoop.cpp

#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include <mutex>

constexpr long long size = 100000000;

int main(){

    std::cout << std::endl;

    std::vector<int> randValues;
    randValues.reserve(size);

    // random values
    std::random_device seed;
    std::mt19937 engine(seed());
    std::uniform_int_distribution<> uniformDist(1, 10);
    for (long long i = 0 ; i < size ; ++i)
        randValues.push_back(uniformDist(engine));

    const auto sta = std::chrono::steady_clock::now();

    std::mutex myMutex;
    unsigned long long sum = {};
    for (auto i: randValues){
        std::lock_guard<std::mutex> myLockGuard(myMutex);
        sum += i;
    }

    const std::chrono::duration<double> dur =
        std::chrono::steady_clock::now() - sta;

    std::cout << "Time for mySummation " << dur.count()
        << " seconds" << std::endl;
    std::cout << "Result: " << sum << std::endl;

    std::cout << std::endl;

}
*/
// =====
// =====
/*

```

The execution time is as expected; the access to the protected variable add is slower. Using a std::lock_guard without contention is about 50 - 150 times slower than using std::accumulate.

Let's finally get to atomics! See you in the next lesson.

```
*/  
// ======  
// ======  
/*
```

Single Threaded Summation: Protection with Atomics

This lesson explains the solution for calculating the sum of a vector problem using atomics in C++.

Accordingly, I have the same questions for atomics that I had for locks.

How expensive is the synchronization of an atomic?

How fast can an atomic be if there is no contention?

I have an additional question: what is the performance difference of an atomic compared to a lock?

```
*/  
// ======  
// calculateWithAtomic.cpp
```

```
// #include <atomic>  
// #include <chrono>  
// #include <iostream>  
// #include <numeric>  
// #include <random>  
// #include <vector>  
  
// constexpr long long size = 100000000;  
  
// int main(){  
  
//     std::cout << std::endl;  
  
//     std::vector<int> randValues;  
//     randValues.reserve(size);  
  
//     // random values  
//     std::random_device seed;  
//     std::mt19937 engine(seed());  
//     std::uniform_int_distribution<> uniformDist(1, 10);  
//     for (long long i = 0 ; i < size ; ++i)  
//         randValues.push_back(uniformDist(engine));  
  
//     std::atomic<unsigned long long> sum = {};  
//     std::cout << std::boolalpha << "sum.is_lock_free(): "  
//             << sum.is_lock_free() << std::endl;  
//     std::cout << std::endl;  
  
//     auto sta = std::chrono::steady_clock::now();  
  
//     for (auto i: randValues) sum += i;
```

```

// std::chrono::duration<double> dur = std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//     << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// sum = 0;
// sta = std::chrono::steady_clock::now();

// for (auto i: randValues) sum.fetch_add(i);

// dur = std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//     << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

First, I check line 28 to see if the atomic sum has a lock. That is crucial because, otherwise, there would be no difference between using locks and atomics. On all mainstream platforms I know, atomics are lock-free. Second, I calculate the sum in two ways. I use the `+=` operator in line 33, and the `fetch_add` method in line 45. In the single-threaded case, both variants have comparable performance. However, for `fetch_add` I can explicitly specify the memory model. More about that point in the next subsection.

I want to stress three points:

Atomics are 12 - 50 times slower on Linux and Windows than `std::accumulate` without synchronization.

Atomics are 2 - 3 times faster on Linux and Windows than locks.

`std::accumulate` seems to be highly optimized on Windows.

*/

```

// =====
// =====
/*

```

Multithreaded Summation: Using `std::lock_guard`

This lesson explains the solution for calculating the sum of a vector problem using `std::lock_guard` in C++.

We'll cover the following

Using a `std::lock_guard`

You may have already guessed that using a shared variable for the summation with four threads is not optimal; the synchronization overhead will outweigh the performance benefit. Let me show you the numbers. The questions I want to answer are still the same.

What is the difference in performance between the summation using a lock and an atomic?

What is the difference in performance between single threaded and multithreaded execution of std::accumulate?

The simplest way to make the thread-safe summation is to use a std::lock_guard.

Using a std::lock_guard#

```
/*
// =====
// synchronisationWithLock.cpp

// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// std::mutex myMutex;

// void sumUp(unsigned long long& sum, const std::vector<int>& val,
//            unsigned long long beg, unsigned long long end){
//     for (auto it = beg; it < end; ++it){
//         std::lock_guard<std::mutex> myLock(myMutex);
//         sum += val[it];
//     }
// }

// int main(){
//     std::cout << std::endl;
//
//     std::vector<int> randValues;
//     randValues.reserve(size);
//
//     std::mt19937 engine;
//     std::uniform_int_distribution<> uniformDist(1,10);
//     for (long long i = 0 ; i < size ; ++i)
//         randValues.push_back(uniformDist(engine));
//
//     unsigned long long sum = 0;
//     const auto sta = std::chrono::steady_clock::now();
//
//     std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
```

```

// std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//     << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

The program is easy to explain. The function sumUp (lines 20 - 26) is the work package that each thread executes. sumUp gets the summation variable sum and the std::vector val by reference. Also, beg and end specify the range of the summation, and the std::lock_guard (line 23) is used to protect the shared sum. That being said, each thread (lines 43 - 46) performs a quarter of the summation.

The bottleneck of the program is the shared variable sum because it is heavily synchronized by an std::lock_guard. With that, one obvious solution comes immediately to mind: replace the heavyweight lock with a lightweight atomic.

```

*/
// =====
// =====
/*

```

Multithreaded Summation: Using Atomic Variable

This lesson explains the solution for calculating the sum of a vector problem using an atomic variable in C++.

Now, the summation variable sum is an atomic; that means I don't need the std::lock_guard anymore. Here is the modified sumUp function.

```
// synchronisationWithAtomic.cpp
```

```
...
```

```

void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum += val[it];
    }
}

```

```
...
```

```
/*
// =====
```

```
// synchronisationWithAtomic.cpp

// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>
// #include <atomic>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// std::atomic<unsigned long long> sum = {};

// void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
//            unsigned long long beg, unsigned long long end){
//     for (auto it = beg; it < end; ++it){
//         sum += val[it];
//     }
// }

// int main(){

//     std::cout << std::endl;

//     std::vector<int> randValues;
//     randValues.reserve(size);

//     std::mt19937 engine;
//     std::uniform_int_distribution<int> uniformDist(1,10);
//     for (long long i = 0 ; i < size ; ++i)
//         randValues.push_back(uniformDist(engine));

//     const auto sta = std::chrono::steady_clock::now();

//     std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
//     std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
//     std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
//     std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

//     t1.join();
//     t2.join();
//     t3.join();
//     t4.join();
```

```
// std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//           << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;
```

```
//}
// =====
```

```
/*
Multithreaded Summation: Using fetch_add Method
```

This lesson explains the solution for calculating the sum of a vector problem using the `fetch_add` method in C++.

The modification of the source code is minimal. I have only changed the summation expression to `sum.fetch_add(val[it])`.

```
// synchronisationWithFetchAdd.cpp
```

```
...
```

```
void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it]);
    }
}
```

```
...
```

```
*/
```

```
//
// =====
```

```
// synchronisationWithFetchAdd.cpp
```

```
// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>
// #include <atomic>
```

```
// constexpr long long size = 100000000;
```

```
// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;
```

```
// std::mutex myMutex;
```

```

// std::atomic<unsigned long long> sum = {};
// void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
//            unsigned long long beg, unsigned long long end){
//    for (auto it = beg; it < end; ++it){
//        sum.fetch_add(val[it]);
//    }
// }

// int main(){
//    std::cout << std::endl;
//    std::vector<int> randValues;
//    randValues.reserve(size);
//    std::mt19937 engine;
//    std::uniform_int_distribution<> uniformDist(1,10);
//    for (long long i = 0 ; i < size ; ++i)
//        randValues.push_back(uniformDist(engine));
//    const auto sta = std::chrono::steady_clock::now();
//    std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
//    std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
//    std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
//    std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
//    t1.join();
//    t2.join();
//    t3.join();
//    t4.join();
//    std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
//    std::cout << "Time for addition " << dur.count()
//    << " seconds" << std::endl;
//    std::cout << "Result: " << sum << std::endl;
//    std::cout << std::endl;
//}

// =====
/*
Now we have similar pursuance as the previous example, and there is little difference between the operator
+= and fetch_add.
*/

```

Although there is no performance difference between the `+=` operation and the `fetch_add` method on an `atomic`, `fetch_add` has an advantage; it allows me to explicitly weaken the memory model and to apply relaxed semantic.

```
// =====
// =====
/*

```

Now we have similar pursuance as the previous example, and there is little difference between the operator += and fetch_add.

Although there is no performance difference between the += operation and the fetch_add method on an atomic, fetch_add has an advantage; it allows me to explicitly weaken the memory model and to apply relaxed semantic.

```
// synchronisationWithFetchAddRelaxed.cpp
```

```
...
```

```
void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it], std::memory_order_relaxed);
    }
}
```

```
...
```

```
*/
```

```
// =====
// synchronisationWithFetchAddRelaxed.cpp
```

```
// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>
// #include <atomic>
```

```
// constexpr long long size = 100000000;
```

```
// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;
```

```
// std::mutex myMutex;
// std::atomic<unsigned long long> sum = {};
```

```
// void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
//           unsigned long long beg, unsigned long long end){
//     for (auto it = beg; it < end; ++it){
//         sum.fetch_add(val[it], std::memory_order_relaxed);
//     }
// }
```

```
// int main(){

// std::cout << std::endl;

// std::vector<int> randValues;
// randValues.reserve(size);

// std::mt19937 engine;
// std::uniform_int_distribution<> uniformDist(1,10);
// for (long long i = 0 ; i < size ; ++i)
//     randValues.push_back(uniformDist(engine));

// const auto sta = std::chrono::steady_clock::now();

// std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
// std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//           << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }
// /**
// =====
```

The default behavior for atomics is sequential consistency. This statement is true for the addition and assignment of an atomic, and of course for the `fetch_add` method, but we can optimize even more. I adjust the memory model in the summation expression to the relaxed semantic: `sum.fetch_add(val[it], std::memory_order_relaxed)`. The relaxed semantic is the weakest memory model and, therefore, the endpoint of my optimization.

The relaxed semantic is fine in this use-case because we have two guarantees: each addition with `fetch_add` will take place in an atomic fashion, and the threads synchronize with the `join` calls. Because of the weakest memory model, we have the best performance.

* /

```
// ======  
// ======  
/*
```

Thread Local Summation: Using Local Variable

This lesson explains the solution for calculating the sum of a vector problem using a local variable in C++.

We'll cover the following

Using a Local Variable

std::lock_guard

Explanation:

Let's combine the two previous strategies for adding the numbers. I will use four threads and minimize the synchronization between the threads.

There are different ways to minimize the synchronization: local variables, thread-local data, and tasks

Using a Local Variable#

Since each thread can use a local summation variable, it can do its job without synchronization; synchronization is only necessary to sum up the local variables. The summation of the local variables is a critical section that must be protected. This can be done in various ways. A quick remark: since only four additions take place, it doesn't matter from a performance perspective which synchronization I use. Anyway, I will use an std::lock_guard - an atomic with sequential consistency and relaxed semantic - for the summation.

```
std::lock_guard#
*/
// =====
// localVariable.cpp

// #include <mutex>
// #include <chrono>
// #include <iostream>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// std::mutex myMutex;

// void sumUp(unsigned long long& sum, const std::vector<int>& val,
//            unsigned long long beg, unsigned long long end){
//     unsigned long long tmpSum{};
//     for (auto i = beg; i < end; ++i){
//         tmpSum += val[i];
//     }
//     std::lock_guard<std::mutex> lockGuard(myMutex);
//     sum += tmpSum;
// }

// int main(){
```

```

// std::cout << std::endl;

// std::vector<int> randValues;
// randValues.reserve(size);

// std::mt19937 engine;
// std::uniform_int_distribution<> uniformDist(1, 10);
// for (long long i = 0; i < size; ++i)
//     randValues.push_back(uniformDist(engine));

// unsigned long long sum{};
// const auto sta = std::chrono::system_clock::now();

// std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
// std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// const std::chrono::duration<double> dur=
//     std::chrono::system_clock::now() - sta;

// std::cout << "Time for addition " << dur.count()
//     << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }
// =====
/*
Explanation:#
```

Lines 26 and 27 are the interesting lines; these are the lines where the local summation result tmpSum is added to the global summation variable sum.

In the next two variations using a local variable, only the function sumUp will change; therefore, I will only display the function. For the entire program, please refer to the source files.

In the next lesson, we'll throw some light on thread local summation using an atomic variable with sequential consistency.

```
/*
// =====
// =====
/*
```

Thread Local Summation: Using an Atomic Variable with Sequential Consistency

This lesson explains the solution for calculating the sum of a vector problem using an atomic variable with sequential consistency in C++.

We'll cover the following

Using an Atomic Variable with Sequential Consistency

Using an Atomic Variable with Sequential Consistency#

Let's replace the non-atomic global summation variable sum with an atomic.

```
// localVariableAtomic.cpp
```

```
...
```

```
void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    unsigned int long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    sum+= tmpSum;
}
```

```
...
```

```
*/
```

```
// =====
// localVariableAtomic.cpp
```

```
// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>
// #include <atomic>
```

```
// constexpr long long size = 100000000;
```

```
// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;
```

```
// std::mutex myMutex;
// std::atomic<unsigned long long> sum = {};
```

```
// void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
//             unsigned long long beg, unsigned long long end){
//     unsigned int long long tmpSum{};
```

```

//   for (auto i = beg; i < end; ++i){
//     tmpSum += val[i];
//   }
//   sum+= tmpSum;
// }

// int main(){

// std::cout << std::endl;

// std::vector<int> randValues;
// randValues.reserve(size);

// std::mt19937 engine;
// std::uniform_int_distribution<> uniformDist(1,10);
// for (long long i = 0 ; i < size ; ++i)
//   randValues.push_back(uniformDist(engine));

// const auto sta = std::chrono::steady_clock::now();

// std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
// std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
// << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }

// =====
/*

```

Thread Local Summation: Using an Atomic Variable with Relaxed Semantic

This lesson explains the solution for calculating the sum of a vector problem using an atomic variable with relaxed semantic in C++.

We'll cover the following

Using an Atomic Variable with Relaxed Semantic

Using an Atomic Variable with Relaxed Semantic#

We can do better. I will use relaxed semantic now instead of the default memory model. That's well defined because the only guarantee we need is that all summations take place and are atomic. With relaxed semantic the summations can even take place out of order.

```
// localVariableAtomicRelaxed.cpp

void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    unsigned int long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    sum.fetch_add(tmpSum, std::memory_order_relaxed);
}

...

*/
// =====
// localVariableAtomicRelaxed.cpp

// #include <chrono>
// #include <iostream>
// #include <mutex>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>
// #include <atomic>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// std::mutex myMutex;
// std::atomic<unsigned long long> sum = {};

// void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
//            unsigned long long beg, unsigned long long end){
//    unsigned int long long tmpSum{};

//    for (auto i = beg; i < end; ++i){
//        tmpSum += val[i];
//    }
//    sum.fetch_add(tmpSum, std::memory_order_relaxed);
// }

// int main(){
```

```

// std::cout << std::endl;

// std::vector<int> randValues;
// randValues.reserve(size);

// std::mt19937 engine;
// std::uniform_int_distribution<> uniformDist(1,10);
// for (long long i = 0 ; i < size ; ++i)
//   randValues.push_back(uniformDist(engine));

// const auto sta = std::chrono::steady_clock::now();

// std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
// std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//       << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

As expected, it doesn't make any difference whether I use a `std::lock_guard` or an atomic with sequential consistency or relaxed semantic.

Thread-local data is a special kind of local data. Its lifetime is bound to the scope of the thread, not to the scope of the function, such as for the variable `tmpSum` in this example.

We'll examine thread local summation using thread local data afterwards.

```

*/
// =====
// =====
/*

```

Thread Local Summation: Using Thread Local Data

This lesson explains the solution for calculating the sum of a vector problem using thread local data in C++.

Thread-local data belongs to the thread in which it was created; it will only be created when needed. Thread-local data is an ideal fit for the local summation variable `tmpSum`.

```

*/
// =====

```

```
// threadLocalSummation.cpp

// #include <atomic>
// #include <chrono>
// #include <iostream>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// thread_local unsigned long long tmpSum = 0;

// void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
//            unsigned long long beg, unsigned long long end){
//     for (auto i = beg; i < end; ++i){
//         tmpSum += val[i];
//     }
//     sum.fetch_add(tmpSum, std::memory_order_relaxed);
// }

// int main(){

//     std::cout << std::endl;

//     std::vector<int> randValues;
//     randValues.reserve(size);

//     std::mt19937 engine;
//     std::uniform_int_distribution<int> uniformDist(1, 10);
//     for (long long i = 0; i < size; ++i)
//         randValues.push_back(uniformDist(engine));

//     std::atomic<unsigned long long> sum{};
//     const auto sta = std::chrono::system_clock::now();

//     std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
//     std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
//     std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
//     std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);

//     t1.join();
//     t2.join();
//     t3.join();
//     t4.join();
}
```

```

// const std::chrono::duration<double> dur=
//     std::chrono::system_clock::now() - sta;

// std::cout << "Time for addition " << dur.count()
//     << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// std::cout << std::endl;

//}
//=====
/*
I declare the thread local variable tmpSum in line 18 and use it for the addition in lines 23 and 25.

```

In the next lesson and the last scenario, I will use tasks.

```

*/
//=====
//=====
/*
Thread Local Summation: Using Tasks

```

This lesson explains the solution for calculating the sum of a vector problem using tasks in C++.

Using tasks, we can do the whole job without synchronization. Each partial summation is performed in a separate thread and the final summation takes place in the main thread.

Here is the program:

```

/*
// =====
// tasksSummation.cpp

// #include <chrono>
// #include <future>
// #include <iostream>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// void sumUp(std::promise<unsigned long long>&& prom, const std::vector<int>& val,
//     unsigned long long beg, unsigned long long end){
//     unsigned long long sum={};
//     for (auto i = beg; i < end; ++i){
//         sum += val[i];
//     }
//     prom.set_value(sum);
// }

// int main(){
//     std::vector<int> val(size);
//     std::iota(val.begin(), val.end(), 1);
//     std::promise<unsigned long long> prom;
//     sumUp(prom, val, 0, size);
//     std::cout << prom.get_future().get();
// }

```

```

//  }
//  prom.set_value(sum);
//}

// int main(){

// std::cout << std::endl;

// std::vector<int> randValues;
// randValues.reserve(size);

// std::mt19937 engine;
// std::uniform_int_distribution<> uniformDist(1,10);
// for (long long i = 0; i < size; ++i)
//   randValues.push_back(uniformDist(engine));

// std::promise<unsigned long long> prom1;
// std::promise<unsigned long long> prom2;
// std::promise<unsigned long long> prom3;
// std::promise<unsigned long long> prom4;

// auto fut1= prom1.get_future();
// auto fut2= prom2.get_future();
// auto fut3= prom3.get_future();
// auto fut4= prom4.get_future();

// const auto sta = std::chrono::system_clock::now();

// std::thread t1(sumUp, std::move(prom1), std::ref(randValues), 0, fir);
// std::thread t2(sumUp, std::move(prom2), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::move(prom3), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::move(prom4), std::ref(randValues), thi, fou);

// auto sum= fut1.get() + fut2.get() + fut3.get() + fut4.get();

// std::chrono::duration<double> dur= std::chrono::system_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
//       << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// std::cout << std::endl;

// }

// =====
/*
Thread Local Summation: Using Tasks

```

This lesson explains the solution for calculating the sum of a vector problem using tasks in C++.

Using tasks, we can do the whole job without synchronization. Each partial summation is performed in a separate thread and the final summation takes place in the main thread.

Here is the program:

```
/*
// =====
// tasksSummation.cpp

// #include <chrono>
// #include <future>
// #include <iostream>
// #include <random>
// #include <thread>
// #include <utility>
// #include <vector>

// constexpr long long size = 100000000;

// constexpr long long fir = 25000000;
// constexpr long long sec = 50000000;
// constexpr long long thi = 75000000;
// constexpr long long fou = 100000000;

// void sumUp(std::promise<unsigned long long> &&prom, const std::vector<int> &val,
//            unsigned long long beg, unsigned long long end)
//{
//    unsigned long long sum = {};
//    for (auto i = beg; i < end; ++i)
//    {
//        sum += val[i];
//    }
//    prom.set_value(sum);
//}

// int main()
//{
//    std::cout << std::endl;

//    std::vector<int> randValues;
//    randValues.reserve(size);

//    std::mt19937 engine;
//    std::uniform_int_distribution<int> uniformDist(1, 10);
//    for (long long i = 0; i < size; ++i)
//        randValues.push_back(uniformDist(engine));

//    std::promise<unsigned long long> prom1;
//    std::promise<unsigned long long> prom2;
```

```

// std::promise<unsigned long long> prom3;
// std::promise<unsigned long long> prom4;

// auto fut1 = prom1.get_future();
// auto fut2 = prom2.get_future();
// auto fut3 = prom3.get_future();
// auto fut4 = prom4.get_future();

// const auto sta = std::chrono::system_clock::now();

// std::thread t1(sumUp, std::move(prom1), std::ref(randValues), 0, fir);
// std::thread t2(sumUp, std::move(prom2), std::ref(randValues), fir, sec);
// std::thread t3(sumUp, std::move(prom3), std::ref(randValues), sec, thi);
// std::thread t4(sumUp, std::move(prom4), std::ref(randValues), thi, fou);

// auto sum = fut1.get() + fut2.get() + fut3.get() + fut4.get();

// std::chrono::duration<double> dur = std::chrono::system_clock::now() - sta;
// std::cout << "Time for addition " << dur.count()
// << " seconds" << std::endl;
// std::cout << "Result: " << sum << std::endl;

// t1.join();
// t2.join();
// t3.join();
// t4.join();

// std::cout << std::endl;
// }
// =====
/*

```

In lines 39 - 47 I define the four promises and the associated futures. In lines 51 - 54 each promise is moved to its own thread. A promise can only be moved but not copied. The threads execute the function sumUp (lines 18 - 25); sumUp takes a promise by rvalue reference as its first argument. In line 56, the futures ask for the result of the summation by using the blocking get call.

```

*/
// =====
// =====
/*

```

Calculate Sum of a Vector: Conclusion

This lesson concludes all the methods used to solve the problem of calculating the sum of a vector in C++.

We'll cover the following

Single Threaded

Multithreading with a Shared Variable

Thread-local Summation

Let's conclude what we learned from this chapter:

Single Threaded#

The range-based for loop and the STL algorithm std::accumulate are in the same performance range. This observation holds for the most optimized version. In the optimized version, the compiler uses the optimized version vectorized SIMD instruction (SSE or AVX) for the summation case; therefore, the loop counter will be increased by 2 (SSE) or 4 (AVX).

Multithreading with a Shared Variable#

The usage of a shared variable for the summation variable makes one point clear: synchronization is very expensive and should be avoided as much as possible. Although I used an atomic variable and even broke the sequential consistency, the four threads are 100 times slower than one thread. From a performance perspective, minimizing expensive synchronization has to be our first goal.

Thread-local Summation#

The thread-local summation is only two times faster than the single-threaded range-based for loop or std::accumulate; that holds, even though each of the four threads can work independently. That surprised me because I was expecting a nearly fourfold improvement, and it surprised me even more that my four cores were not fully utilized.

The reason is simple: the cores can't get the data fast enough from the memory. To that point, the execution is memory bound - i.e. it slows down the cores.

Introduction

This lesson gives an introduction to the problem of the thread-safe initialization of singletons, and how to solve it through various methods in C++.

Before we start with this case study, let me emphasize that I am not advocating the use of the singleton pattern; I only use it here because it is a classic example for a variable that must be initialized in a thread-safe way. For a more elaborate discussion about the pros and cons of the singleton pattern, please refer to the referenced articles in the Wikipedia page for the singleton pattern.

I want to start my discussion of the thread-safe initialization of the singleton pattern with a detour.

```
*/  
// ======  
// ======  
/*
```

Double-Checked Locking Pattern

This lesson gives an overview of the double-checked locking pattern for the problem of the thread-safe initialization of a singleton in C++.

The double-checked locking pattern is the classic way to initialize a singleton in a thread-safe way. What sounds like established best practice - or a pattern - is more a kind of an anti-pattern. It assumes guarantees in the classical implementation, which aren't given by the Java, C#, or C++ memory model. The wrong assumption is that the creation of a singleton is an atomic operation; therefore, a solution that seems to be thread-safe is not thread-safe.

What is the double-checked locking pattern? The first idea for implementing a thread-safe singleton is to protect the initialization of the singleton with a lock.

```
*/
```

```

// =====
// #include <iostream>
// #include <mutex>
// #include <thread>

// std::mutex myMutex;

// class MySingleton{
// public:
// static MySingleton* getInstance(){
// std::lock_guard<std::mutex> myLock(myMutex);
// if(!instance) instance = new MySingleton();
// return instance;
// }
// private:
// MySingleton() = default;
// ~MySingleton() = default;
// MySingleton(const MySingleton&) = delete;
// MySingleton& operator= (const MySingleton&) = delete;
// static MySingleton* instance;
// };

// MySingleton* MySingleton::instance = nullptr;

// int main(){

// std::cout << std::endl;

// std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;
// std::cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << std::endl;

// std::cout << std::endl;

// }
// =====
/*
Any issues? Yes and no. Yes, because there is a large performance penalty; No, because the implementation is thread-safe. Each access to the singleton in line 7 is protected by a heavyweight lock. This also applies to the read access, which is not necessary after the initial construction of MySingleton. With that, here comes the double-checked locking pattern to our rescue. Let's have a look at the getInstance function.

static MySingleton& getInstance(){
if (!instance){          // check
    lock_guard<mutex> myLock(myMutex);      // lock
    if(!instance) instance = new MySingleton(); // check
}
return *instance;
}
*/
// =====
// =====

```

```

/*
#include <iostream>
#include <mutex>
#include <thread>

std::mutex myMutex;

class MySingleton{
public:
    static MySingleton* getInstance(){
        if (!instance){                                // check
            std::lock_guard<std::mutex> myLock(myMutex);      // lock
            if (!instance) instance = new MySingleton();      // check
        }
        return instance;
    }
private:
    MySingleton() = default;
    ~MySingleton() = default;
    MySingleton(const MySingleton&) = delete;
    MySingleton& operator= (const MySingleton&) = delete;
    static MySingleton* instance;
};

MySingleton* MySingleton::instance = nullptr;

int main(){

    std::cout << std::endl;

    std::cout << "MySingleton::getInstance(): " << MySingleton::getInstance() << std::endl;
    std::cout << "MySingleton::getInstance(): " << MySingleton::getInstance() << std::endl;

    std::cout << std::endl;

}

*/
// =====
// =====
/*
Instead of the heavyweight lock, I use a lightweight pointer comparison in line 10. If I get a null pointer, I apply
the heavyweight lock on the singleton (line 11). Because there is the possibility that another thread will
initialize the singleton between the pointer comparison in line 10 and the lock call in line 11, I have to perform
an additional pointer comparison in line 12. So the name is obvious: two times a check and one time a lock.

```

Smart? Yes. Thread-safe? No.

What is the issue? The call `instance= new MySingleton()` in line 12 consists of at least three steps.

Allocate memory for `MySingleton`

Initialise the MySingleton object

Let instance refer to the fully initialized MySingleton object

The issue is that the C++ runtime provides no guarantee that the steps will be performed in that sequence. For example, it is possible that the processor may reorder the steps to the sequence 1,3, and 2. So in the first step, the memory will be allocated, and in the second step instance refers to a non-initialised singleton. If just at that moment another thread t2 tries to access the singleton and makes the pointer comparison, the comparison will succeed. The consequence is that thread t2 refers to a non-initialised singleton and the program behavior is undefined.

```
/*
// =====
// =====
/*
```

Performance Measurement

This lesson gives an overview of the performance measurement for the problem of thread-safe initialization of a singleton in C++.

I want to measure how expensive it is to access a singleton object. For reference timing, I will use a singleton which I will access 40 million times sequentially. Of course, the first access will initialize the singleton object. In contrast, the accesses from four threads will be done concurrently. I'm only interested in the performance numbers; therefore, I will sum up the execution time of the four threads. I will measure the performance using a static variable with block scope (Meyers Singleton), a lock std::lock_guard, the function std::call_once in combination with the std::once_flag, and atomics with sequential consistency and acquire release semantic.

The program will run on two PCs. My Linux PC with the GCC compiler has four cores while my Windows PC with the cl.exe compiler has two. That being said, I compile the program with maximum optimization. Now, I want to answer two questions:

What are the performance numbers of the various singleton implementations?

Is there a significant difference between Linux (GCC) and Windows (cl.exe)?

Finally, I will collect all the numbers in a table. You will be seeing it in the conclusion.

```
/*
// =====
// =====
/*
```

Classical Meyers Singleton

This lesson gives an overview of a classical Meyers Singleton in C++.

Here is the sequential program. The getInstance method is not thread-safe with the C++03 standard.

```
/*
// =====
// singletonSingleThreaded.cpp
```

```
// #include <chrono>
```

```
// #include <iostream>
```

```
// constexpr auto tenMill = 10000000;
```

```
// class MySingleton
```

```
// {
```

```

// public:
// static MySingleton &getInstance()
// {
//   static MySingleton instance;
//   volatile int dummy{};
//   return instance;
// }

// private:
// MySingleton() = default;
// ~MySingleton() = default;
// MySingleton(const MySingleton &) = delete;
// MySingleton &operator=(const MySingleton &) = delete;
// };

// int main()
//{
// constexpr auto fourtyMill = 4 * tenMill;

// const auto begin = std::chrono::system_clock::now();

// for (size_t i = 0; i <= fourtyMill; ++i)
// {
//   MySingleton::getInstance();
// }

// const auto end = std::chrono::system_clock::now() - begin;

// std::cout << std::chrono::duration<double>(end).count() << std::endl;
//}

// =====
/*
Classical Meyers Singleton
This lesson gives an overview of a classical Meyers Singleton in C++.

```

Here is the sequential program. The getInstance method is not thread-safe with the C++03 standard.

```

*/
// =====
// singletonSingleThreaded.cpp

// #include <chrono>
// #include <iostream>

// constexpr auto tenMill = 10000000;

// class MySingleton{
// public:
// static MySingleton& getInstance(){
//   static MySingleton instance;
//   volatile int dummy{};
// }
```

```

// return instance;
// }
// private:
// MySingleton() = default;
// ~MySingleton() = default;
// MySingleton(const MySingleton&) = delete;
// MySingleton& operator=(const MySingleton&) = delete;

// };

// int main(){

// constexpr auto fourtyMill = 4 * tenMill;

// const auto begin= std::chrono::system_clock::now();

// for ( size_t i = 0; i <= fourtyMill; ++i){
//   MySingleton::getInstance();
// }

// const auto end = std::chrono::system_clock::now() - begin;

// std::cout << std::chrono::duration<double>(end).count() << std::endl;

// }
// =====
/*

```

As the reference implementation, I use the so-called Meyers Singleton, named after Scott Meyers. The elegance of this implementation is that the singleton object in line 11 is a static variable with a block scope; therefore, instance will be initialized only once. This initialization happens when the static method getInstance (lines 10 - 14) will be executed the first time.

The volatile Variable dummy

When I compiled the program with maximum optimisation, the compiler removed the call MySingleton::getInstance() in line 30 because the call has no effect; therefore, I got very fast execution, but wrong performance numbers. By using the volatile variable dummy (line 12), the compiler is not allowed to optimise away the MySingleton::getInstance() call in line 30.

The beauty of the Meyers Singleton is that it becomes thread-safe with C++11, so let's see how in the next lesson.

```

*/
// =====
// =====
/*

```

Introduction to Thread-Safe Meyers Singleton

This lesson gives an introduction to a thread-safe version of Meyers singleton.

The C++11 standard guarantees that static variables with block scope will be initialized in a thread-safe way. The Meyers Singleton uses a static variable with block scope, so we are done. The only work that is left to do is to rewrite the previously used classical Meyers Singleton for the multithreading use-case.

```

*/
// =====
// singletonMeyers.cpp

// #include <chrono>
// #include <iostream>
// #include <future>

// constexpr auto tenMill = 10000000;

// class MySingleton{
// public:
//   static MySingleton& getInstance(){
//     static MySingleton instance;
//     volatile int dummy{};
//     return instance;
//   }
// private:
//   MySingleton() = default;
//   ~MySingleton() = default;
//   MySingleton(const MySingleton&) = delete;
//   MySingleton& operator=(const MySingleton&) = delete;

// };

// std::chrono::duration<double> getTime(){

//   auto begin = std::chrono::system_clock::now();
//   for (size_t i = 0; i <= tenMill; ++i){
//     MySingleton::getInstance();
//   }
//   return std::chrono::system_clock::now() - begin;

// };

// int main(){

//   auto fut1= std::async(std::launch::async, getTime);
//   auto fut2= std::async(std::launch::async, getTime);
//   auto fut3= std::async(std::launch::async, getTime);
//   auto fut4= std::async(std::launch::async, getTime);

//   const auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();

//   std::cout << total.count() << std::endl;

// }
// =====
/*

```

I use the singleton object in the function `getTime` (lines 24 - 32). The function is executed by the four promises in lines 36 - 39. The results of the associated futures are summed up in line 41. That's all.

I reduce the examples to the singleton implementation

The function `get_Time` for calculating the execution time and the main function will be identical; therefore, I will skip them in the remaining examples of this subsection.

Let's go for the most obvious one, and use a lock in the next lesson.

```
*/  
// ======  
// ======  
/*
```

Thread-Safe Singleton: `std::lock_guard`

This lesson explains the solution for thread-safe initialization of a singleton problem using `std::lock_guard` in C++.

The mutex wrapped in an `std::lock_guard` guarantees that the singleton will be initialized in a thread-safe way.

```
*/  
// ======  
// singletonLock.cpp  
  
// #include <chrono>  
// #include <iostream>  
// #include <future>  
// #include <mutex>  
  
// constexpr auto tenMill = 10000000;  
  
// std::mutex myMutex;  
  
// class MySingleton{  
// public:  
//     static MySingleton& getInstance(){  
//         std::lock_guard<std::mutex> myLock(myMutex);  
//         if (!instance){  
//             instance= new MySingleton();  
//         }  
//         volatile int dummy{};  
//         return *instance;  
//     }  
// private:  
//     MySingleton() = default;  
//     ~MySingleton() = default;  
//     MySingleton(const MySingleton&) = delete;  
//     MySingleton& operator=(const MySingleton&) = delete;  
  
//     static MySingleton* instance;  
// };  
  
// MySingleton* MySingleton::instance = nullptr;
```

```

// int main(){

// constexpr auto fourtyMill = 4 * tenMill;

// const auto begin= std::chrono::system_clock::now();

// for ( size_t i = 0; i <= fourtyMill; ++i){
//   MySingleton::getInstance();
// }

// const auto end = std::chrono::system_clock::now() - begin;

// std::cout << std::chrono::duration<double>(end).count() << std::endl;

// }
// =====
/*
You may have already guessed that this approach is the slowest one.

```

The next version of the thread-safe singleton pattern is also based on the multithreading library. It uses `std::call_once` in combination with the `std::once_flag`.

```

*/
/*
Thread-Safe Singleton: std::call_once with std::once_flag
This lesson explains the solution for the thread-safe initialization of a singleton problem using std::call_once
with std::once_flag in C++.
```

You can use the function `std::call_once` together with the `std::once_flag` to register callables that will be executed exactly once in a thread-safe way.

```

*/
/*
*/
// =====
// singletonCallOnce.cpp

// #include <chrono>
// #include <iostream>
// #include <future>
// #include <mutex>
// #include <thread>

// constexpr auto tenMill = 10000000;

// class MySingleton{
// public:
// static MySingleton& getInstance(){
// std::call_once(initInstanceFlag, &MySingleton::initSingleton);
// volatile int dummy{};
// return *instance;
// }

```

```

// private:
// MySingleton() = default;
// ~MySingleton() = default;
// MySingleton(const MySingleton&) = delete;
// MySingleton& operator=(const MySingleton&) = delete;

// static MySingleton* instance;
// static std::once_flag initInstanceFlag;

// static void initSingleton(){
//   instance= new MySingleton;
// }
//};

// MySingleton* MySingleton::instance = nullptr;
// std::once_flag MySingleton::initInstanceFlag;

// int main(){

// constexpr auto fourtyMill = 4 * tenMill;

// const auto begin= std::chrono::system_clock::now();

// for ( size_t i = 0; i <= fourtyMill; ++i){
//   MySingleton::getInstance();
// }

// const auto end = std::chrono::system_clock::now() - begin;

// std::cout << std::chrono::duration<double>(end).count() << std::endl;

// }
//=====
/*

```

Thread-Safe Singleton: Atomicics

This lesson explains the solution for the thread-safe initialization of singleton problem using atomics in C++.

We'll cover the following

Sequential Consistency

Acquire-Release Semantic

With atomic variables, my implementation becomes a lot more challenging; I can even specify the memory model for my atomic operations. The following two implementations of the thread-safe singletons are based on the previously mentioned double-checked locking pattern.

Sequential Consistency#

In my first implementation, I use atomic operations without specifying the memory model; therefore, sequential consistency applies.

*/

```
//=====
```

```
// singletonSequentialConsistency.cpp

// #include <atomic>
// #include <iostream>
// #include <future>
// #include <mutex>
// #include <thread>

// constexpr auto tenMill = 10000000;

// class MySingleton{
// public:
//     static MySingleton* getInstance(){
//         MySingleton* sin = instance.load();
//         if (!sin){
//             std::lock_guard<std::mutex> myLock(myMutex);
//             sin = instance.load(std::memory_order_relaxed);
//             if(!sin){
//                 sin= new MySingleton();
//                 instance.store(sin);
//             }
//         }
//         volatile int dummy{};
//         return sin;
//     }
// private:
//     MySingleton() = default;
//     ~MySingleton() = default;
//     MySingleton(const MySingleton&) = delete;
//     MySingleton& operator=(const MySingleton&) = delete;

//     static std::atomic<MySingleton*> instance;
//     static std::mutex myMutex;
// };

// std::atomic<MySingleton*> MySingleton::instance;
// std::mutex MySingleton::myMutex;

// int main(){

//     constexpr auto fourtyMill = 4 * tenMill;

//     const auto begin= std::chrono::system_clock::now();

//     for ( size_t i = 0; i <= fourtyMill; ++i){
//         MySingleton::getInstance();
//     }

//     const auto end = std::chrono::system_clock::now() - begin;

//     std::cout << std::chrono::duration<double>(end).count() << std::endl;
```

```
//}  
// ======  
/*
```

In contrast to the double-checked locking pattern, I now have the guarantee that the expression `sin = new MySingleton()` in line 19 will happen before the store expression `instance.store(sin)` in line 20. This is due to using sequential consistency as the default memory model for atomic operations. Have a look at line 17:

```
sin = instance.load(std::memory_order_relaxed);
```

This additional load is necessary because, between the first load in line 14 and the usage of the lock in line 16, another thread may kick in and change the value of `instance`. That being said, we can optimize the program even more.

Acquire-Release Semantic#

Let's have a closer look at the previous thread-safe implementation of the singleton pattern using atomics. The loading (or reading) of the singleton in line 14 is an acquire operation, the storing (or writing) in line 20 is a release operation. Both operations take place on the same atomic, therefore sequential consistency is overkill. The C++11 standard guarantees that an acquire operation synchronizes with a release operation on the same atomic and establishes an ordering constraint. This means that all subsequent read and write operations cannot be moved before an acquire operation, and all read and write operations cannot be moved after a release operation.

These are the minimal guarantees required to implement a thread-safe singleton.

```
*/  
// ======  
// singletonAcquireRelease.cpp  
  
// #include <atomic>  
// #include <iostream>  
// #include <future>  
// #include <mutex>  
// #include <thread>  
  
// constexpr auto tenMill = 10000000;  
  
// class MySingleton{  
// public:  
//     static MySingleton* getInstance(){  
//         MySingleton* sin = instance.load(std::memory_order_acquire);  
//         if (!sin){  
//             std::lock_guard<std::mutex> myLock(myMutex);  
//             sin = instance.load(std::memory_order_relaxed);  
//             if (!sin){  
//                 sin = new MySingleton();  
//                 instance.store(sin, std::memory_order_release);  
//             }  
//         }  
//         volatile int dummy{};  
//         return sin;
```

```

// }
// private:
// MySingleton() = default;
// ~MySingleton() = default;
// MySingleton(const MySingleton&) = delete;
// MySingleton& operator=(const MySingleton&) = delete;

// static std::atomic<MySingleton*> instance;
// static std::mutex myMutex;
// };

// std::atomic<MySingleton*> MySingleton::instance;
// std::mutex MySingleton::myMutex;

// int main(){

// constexpr auto fourtyMill = 4 * tenMill;

// const auto begin= std::chrono::system_clock::now();

// for ( size_t i = 0; i <= fourtyMill; ++i){
//   MySingleton::getInstance();
// }

// const auto end = std::chrono::system_clock::now() - begin;

// std::cout << std::chrono::duration<double>(end).count() << std::endl;

// }
// =====
/*

```

The acquire-release semantic has a similar performance to the sequential consistency. This is not surprising because both memory models are very similar in the x86 architecture. We would probably get a greater difference in the performance numbers on the ARMv7 or PowerPC architecture. You can read the details on Jeff Preshings blog [Preshing on Programming](#).

```

*/
// =====
// =====
/*

```

Conclusion

This lesson concludes the performance measures of methods used in solving the thread-safe initialization of singleton problems in C++.

The numbers give a clear indication; the Meyers Singleton is the easiest to understand and the fastest one. It's about two times faster than the atomic versions. As expected, the synchronization with the lock is the most heavyweight and, therefore, the slowest. In particular, `std::call_once` on Windows is a lot slower than on Linux.

Operating System (Compiler)	Single Threaded	Meyers Singleton	<code>std::lock_guard</code>	<code>std::call_once</code>
Sequential Consistency		Acquire-Release Semantic		
Linux (GCC)	0.03 0.04	12.47 0.22	0.09	0.07

```
Windows (cl.exe) 0.02 0.03 15.48 1.74 0.07 0.07
```

I want to stress one point about the numbers explicitly: These are the summed up numbers for all four threads. That means that we get optimal concurrency with the Meyers Singleton because the Meyers Singleton is nearly as fast as the single threaded implementation.

```
/*
// =====
// =====
/*
Introduction to CppMem
```

This lesson gives an introduction to the case study of ongoing optimization with CppMem.

I will start with a small program and successively improve it, then verify each step of my process with CppMem. CppMem is an interactive tool for exploring the behavior of small code snippets using the C++ memory model.

First, here is the small program.

```
/*
// =====
// ongoingOptimisation.cpp

// #include <iostream>
// #include <thread>

// int x = 0;
// int y = 0;

// void writing(){
//   x = 2000;
//   y = 11;
// }

// void reading(){
//   std::cout << "y: " << y << " ";
//   std::cout << "x: " << x << std::endl;
// }

// int main(){
//   std::thread thread1(writing);
//   std::thread thread2(reading);
//   thread1.join();
//   thread2.join();
// }
// =====
/*
```

The program is quite simple. It consists of the two threads `thread1` and `thread2`. `thread1` writes the values `x` and `y`. `thread2` reads the values `x` and `y` in the opposite sequence. This sounds straightforward, but even in this simple program we can get different results if we run it several times.

I have two questions in mind for my process of ongoing optimization.

Is the program well-defined? In particular: is there a data race?

Which values for x and y are possible?

The first question is often very challenging to answer. In the first step, I will think about the answer to the first question and in the second step, I will verify my reasoning with CppMem. Once I have answered the first question, the second answer can easily be determined from the first. I will also present the possible values for x and y in a table.

But still, I haven't explained what I mean by ongoing optimization. It's pretty simple; I will successively optimize the program by weakening the C++ memory model. These are my optimization steps:

Non-atomic variables

Locks

Atomics with sequential consistency

Atomics with acquire-release semantic

Atomics with relaxed semantic

Volatile variables

Before I start my process of ongoing optimization, there is a short detour I have to make. I have to introduce CppMem in the next lesson.

```
*/  
// ======  
// ======  
/*  
CppMem: An Overview  
This lesson gives a brief overview of CppMem and how it helps in optimization.
```

We'll cover the following

The Overview

1. Model
2. Program
3. Display Relations
4. Display Layout
5. Model Predicates

The Test Run

Explanation:

CppMem is an interactive tool for exploring the behavior of small code snippets using the C++ memory model. It has to be in the toolbox of each programmer who seriously deals with the memory model.

The online version of CppMem - you can also install it on your PC - provides very valuable services in a twofold way:

CppMem verifies the behavior of small code snippets. Based on the C++ memory model, the tool considers all possible interleavings of threads, visualizes each of them in a graph, and annotates these graphs with additional details.

The very accurate analysis of CppMem gives you deep insight into the C++ memory model. In short, CppMem is a tool that helps you to get a better understanding of the memory model.

Of course, it's often the nature of powerful tools that you first have to overcome a few hurdles. The nature of things is that CppMem is highly configurable, and it gives you a very detailed analysis related to this extremely challenging topic. Therefore, my plan is to present the components of the tool.

The Overview#

My simplified overview of CppMem is based on the default configuration. This overview will only provide you with the base for further experiments and should help you to understand my process of ongoing optimization.

For the sake of simplicity, I will refer to the red numbers in the screenshot.

1. Model#

Specifies the C++ memory model. preferred is a simplified variant of the C++11 memory model.

2. Program#

Contains the executable program (e.g. syntax) in a simplified *C or C++. To be precise, you cannot directly copy C or C++ code programs into CppMem.

You can choose between a lot of programs that implement typical multithreading scenarios. To get the details of these programs, read the very well written article [Mathematizing C++ Concurrency](#). Of course, you can also run your own code.

CppMem is about multithreading; therefore, there are shortcuts.

You can easily define two threads using the expression {{{{ ... } ||| ... }}}. The three dots (...) represent the work package of each thread.

If you use the expression `x.readvalue(1)`, CppMem will evaluate the interleavings of the threads for which the thread execution gives the value 1 for `x`.

3. Display Relations#

Describes the relations between the read, write, and read-write modifications on atomic operations, fences, and locks.

You can explicitly enable the relations in the annotated graph with the switches.

There are three classes of relations. The coarser distinction between original and derived relations is the most interesting one. Here are the default values.

Original relations:

sb: sequenced-before

rf: read from

mo: modification order

sc: sequentially consistent

lo: lock order

Derived relations:

sw: synchronises-with

dob: dependency-ordered-before

unsequenced_races: races in a single thread

data_races: inter-thread data races

4. Display Layout#

With this switch, you can choose which Doxygen graph is used.

5. Model Predicates#

With this button, you can set the predicates for the chosen model - which can cause a non-consistent (not data-race-free) execution; therefore, if you get a non-consistent execution, you see exactly the reason for the non-consistent execution. I will not use this button in this course.

See the documentation for more details.

This is enough as a starting point for my ongoing optimization. Now, it is time to give CppMem a try.

The Test Run#

You have to choose the program `data_race.c` from the CppMem samples. The run button shows immediately that there is a data race.

Explanation:#

The data race is quite easy to see. A thread writes `x` (`x = 3`) and another thread reads `x` (`x==3`) without synchronization.

Two interleavings of threads are possible due to the C++ memory model; only one of them is consistent to the chosen model. This is the case if, in the expression `x==3`, the value of `x` is written by the expression `int x = 2` in the main function. The graph displays this relation in the edge annotated with `rf` and `sw`.

It is extremely interesting to switch between the different interleaving of threads.

The graph shows all relations in the format display layout, which you enabled in the Display Relations.

`a:Wna x=2` is in the graphic as the `a`-th statement, which is a non-atomic write. `Wna` stands for “Write non-atomic”.

The key edge in the graph is the edge between the writing of `x` (`b:Wna`) and the reading of `x` (`C:Rna`). That’s the data race on `x`.

Let’s move on to the non-atomic variables in the next lesson.

```
*/  
// ======  
// ======  
/*
```

CppMem: Non-Atomic Variables

This lesson gives an overview of non-atomic variables used in the context of CppMem.

We'll cover the following

The Analysis

First Execution

Second Execution

Third Execution

Fourth Execution

A Short Conclusion

Using the run button immediately shows that there is a data race. To be more precise, it has two data races. Neither the access to the variable `x` nor to the variable `y` are protected. As a result, the program has undefined behavior. In C++ jargon, this means that the program has the so-called catch fire semantic; therefore, all results are possible. Your PC can even catch fire.

So, we are not able to draw conclusions about the values of `x` and `y`.

Guarantees for int variables

Most of the mainstream architectures guarantee that access to an `int` variable is atomic as long as the `int` variable is aligned naturally. Naturally aligned means that the `int` variable on a 32-bit architecture must have an address divisible by 4; On a 64-bit architecture, it's divisible by 8. I mention this so explicitly because you can adjust the alignment of your data types with C++.

I have to emphasize that I'm not advising you to use an int as an atomic int. I only want to point out that the compiler guarantees more in this case than the C++11 standard. If you rely on the compiler guarantee, your program will not be compliant with the C++ standard and, therefore, may break in the future.

This was my reasoning. Now we should have a look at what CppMem will report about the undefined behavior of the program. As it stands, CppMem allows me to reduce the program to its bare minimum.

```
int main() {
    int x = 0;
    int y = 0;
    {{{
        x = 2000;
        y = 11;
    }
    ||| {
        y;
        x;
    }
    }})
}
*/
// =====
// =====
/*
```

You can just define a thread in CppMem with the curly braces (lines 4 and 12) and the pipe symbol (line 8). The additional curly braces in lines 4 and 7 or lines 8 and 11 define the work package of the thread. Because I'm not interested in the output of the variables x and y, I only read them in lines 9 and 10. That was the theory for CppMem, now to the practice.

The Analysis#

When I execute the program, CppMem complains (1) (in red) that one of the four possible interleavings of threads is not race free. Only the first execution is consistent. Now I can use CppMem to switch between the four executions (2) and analyze the annotated graph (3).

You get the most out of CppMem by analyzing the various graphs.

First Execution#

What conclusions can we derive from the following graph?

The nodes of the graph represent the expressions of the program, and the edges represent the relationship between the expressions. In my explanation, I will refer to the names (a) to (f). What can I derive from the annotations in this graph?

a:Wna x = 0: Is the first expression (a), which is a non-atomic write of x.

sb (sequenced-before): The writing of the first expression (a) is sequenced before the writing of the second expression (b). These relations also holds between the expressions (c) and (d), and (e) and (f).

rf (read from): The expression (e) reads the value of y from the expression (b). Accordingly, (f) reads from (a).

sw (synchronizes-with): The expression (a) synchronises with (f). This relation holds true because the expression (f) takes place in a separate thread. Creation of a thread is a synchronization point; everything that happens before the thread creation is visible in the thread. For symmetry reasons, the same argument holds true between (b) and (e).

dr (data race): Here is the data race between the reading and writing of the variables x and y. The program has undefined behavior.

Why is the execution consistent?

The execution is consistent because the values x and y are read from the values in the main thread (a) and (b). If the values were read from x and y using a separate thread (not main-thread) in the expressions (c) and (d), it can happen that the values of x and y in (e) and (f) are only partial reads; this is not consistent. Or to say it differently, in the concrete execution x and y get the value 0. You can see the values of x and y that were read in the expressions (e) and (f).

The next three executions are not consistent.

Second Execution#

The expression (e) reads the value of y from the expression (d) in this non-consistent execution. Also, the writing of (d) happens concurrently with the reading of (e).

Third Execution#

This execution is symmetric to the previous execution. That being said, the expression (f) reads concurrently from expression (c).

Fourth Execution#

Now everything goes wrong. The expressions (e) and (f) read from the expressions (d) and (c) concurrently.

A Short Conclusion#

Although I just used the default configuration of CppMem, I got a lot of valuable information and insight. In particular, the graphs from CppMem showed:

All four combinations of x and y are possible: (0,0), (11,0), (0,2000), (11,2000).

The program has at least one data race and, therefore, has undefined behavior.

Only one of the four possible executions is consistent.

Using volatile

Using the qualifier volatile for x and y makes no difference from the memory model perspective, compared with using non-synchronised access to x and y.

```
int main() {
    volatile int x = 0;
    volatile int y = 0;
    {{{
        x = 2000;
        y = 11;
    }
    ||
}
```

```
    y;  
    x;  
}  
}  
}
```

CppMem will generate identical graphs to those seen in the previous example. The reason is quite simple: In C++, volatile has no multithreading semantic.

The access to x and y in this example was not synchronized and we got a data race; therefore, undefined behavior. The most obvious way for synchronization is to use locks.

```
*/  
// ======  
// ======  
/*  
CppMem: Locks
```

This lesson gives an overview of locks used in the context of CppMem.

Both threads - thread1 and thread2 - use the same mutex, and they're wrapped in a std::lock_guard.

```
*/  
// ======  
// ongoingOptimisationLock.cpp
```

```
// #include <iostream>  
// #include <mutex>  
// #include <thread>  
  
// int x = 0;  
// int y = 0;  
  
// std::mutex mut;  
  
// void writing(){  
//   std::lock_guard<std::mutex> guard(mut);  
//   x = 2000;  
//   y = 11;  
// }  
  
// void reading(){  
//   std::lock_guard<std::mutex> guard(mut);  
//   std::cout << "y: " << y << " ";  
//   std::cout << "x: " << x << std::endl;  
// }  
  
// int main(){  
//   std::thread thread1(writing);  
//   std::thread thread2(reading);  
//   thread1.join();  
//   thread2.join();  
// };  
// ======
```

```
/*
```

The program is well-defined. Depending on the execution order (thread1 vs thread2), either both values are first read and then overwritten, or they're first overwritten and then read. The following values for x and y are possible.

y	x	Values possible?
---	---	------------------

0	0	Yes
---	---	-----

11	0	
----	---	--

0	2000	
---	------	--

11	2000	Yes
----	------	-----

Using std::lock_guard in CppMem

I could not find a way to use std::lock_guard in CppMem. If you know how to achieve this, please let me know.

Locks are easy to use but the synchronization is often too heavyweight. I will now switch to a more lightweight strategy and will use atomics.

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

CppMem: Atomics with Sequential Consistency

This lesson gives an overview of atomics with sequential consistency used in the context of CppMem.

We'll cover the following

CppMem

Execution for (y = 0, x = 0)

Executions for (y = 0, x = 2000)

Execution for (y = 11, x = 2000)

Sequence of Instructions

Cases:

If you don't specify the memory model, sequential consistency will be applied. Sequential consistency guarantees two properties: each thread executes its instructions in source code order, and all threads follow the same global order. Here is the optimized version of the program using atomics.

```
*/
```

```
// =====
```

```
// ongoingOptimisationSequentialConsistency.cpp
```

```
// #include <atomic>
```

```
// #include <iostream>
```

```
// #include <thread>
```

```
// std::atomic<int> x{0};
```

```
// std::atomic<int> y{0};
```

```
// void writing()
```

```
// {
```

```
// x.store(2000);
```

```
// y.store(11);
```

```

// }

// void reading()
//{
// std::cout << y.load() << " ";
// std::cout << x.load() << std::endl;
//}

// int main()
//{
// std::thread thread1(writing);
// std::thread thread2(reading);
// thread1.join();
// thread2.join();
//}
// =====
/*

```

Let's analyze the program. The program is data race free because x and y are atomics. Therefore, only one question is left to answer: What values are possible for x and y? The question is easy to answer. Thanks to the sequential consistency, all threads have to follow the same global order.

It holds true:

```

x.store(2000); happens-before y.store(11);
std::cout << y.load() << " "; happens-before std::cout << x.load() << std::endl;
Hence the value of x.load() cannot be 0 if y.load() has the value 11, because x.store(2000) happens before
y.store(11).

```

All other values for x and y are possible. Here are three possible interleavings resulting in the three different values for x and y.

thread1 will be completely executed before thread2.
 thread2 will be completely executed before thread1.
 thread1 will execute its first instruction x.store(2000) before thread2 will be completely executed.

Here are all the values for x and y.

y	x	Values possible?
0	0	Yes
11	0	
0	2000	Yes
11	2000	Yes

Let me verify my reasoning with CppMem.

CppMem#

Here is the corresponding program in CppMem.

```

int main(){
atomic_int x = 0;
atomic_int y = 0;

```

```

{{{
    x.store(2000);
    y.store(11);
}
|||
    y.load();
    x.load();
}
}}
}

```

First, a little bit of syntax. CppMem uses the typedef atomic_int for std::atomic<int> in lines 2 and 3. When I execute the program, I'm overwhelmed by the number of execution candidates.

There are 384 (1) possible execution candidates, but only 6 of them are consistent; no candidate has a data race. I'm only interested in the 6 consistent executions and ignore the other 378 non-consistent executions. Non-consistent means, for example, that they will not respect the modification order. I use the interface (2) to get the six annotated graphs.

We already know that all values for x and y are possible except for y = 11 and x = 0. This is because of the sequential consistency. Now I'm curious, which interleaving of threads will produce which values for x and y?

Execution for (y = 0, x = 0) #

Executions for (y = 0, x = 2000) #

Execution for (y = 11, x = 2000) #

I'm not done with my analysis. I'm interested in which sequence of instructions corresponds to which of the six graphs.

Sequence of Instructions #

I have assigned each graph to its corresponding sequence of instructions.

Cases: #

Let me start with the simpler cases.

(1): It's quite simple to assign the graph (1) to the sequence (1). In the sequence (1) x and y have the values 0 because y.load() and x.load() are executed before the operations x.store(2000) and y.store(11).

(6): The reasoning for the execution (6) is similar. y has the value 11 and x the value 2000 because all load operations happen after all store operations.

(2), (3), (4), (5): Now to the more interesting cases in which y has the value 0 and x has the value 2000. The yellow arrows (sc) in the graph are the key to my reasoning because they stand for the sequence of instructions. For example, let's look at execution (2).

(2): Here is the sequence of the yellow arrows (sc) in the graph (2): write x = 2000 => read y = 0 => write y = 11 => read x = 2000. This sequence corresponds to the sequence of instructions of the second interleaving of threads (2).

Let's break the sequential consistency with the acquire-release semantic in the next lesson.

*/

```
// =====
```

```
// =====
```

/*

CppMem: Atomics with an Acquire-Release Semantic

This lesson gives an overview of atomics with acquire-release semantic used in the context of CppMem.

We'll cover the following

Explanation

CppMem

Possible Executions

Execution for (y = 0, x = 0)

Execution for (y = 0, x = 2000)

Execution for (y = 11, x = 2000)

The synchronization in the acquire-release semantic takes place between atomic operations on the same atomic. This is in contrast to the sequential consistency where we have synchronization between threads. Due to this fact, the acquire-release semantic is more lightweight and, therefore, faster.

Here is the program with acquire-release semantic:

*/

```
// =====
```

```
// ongoingOptimisationAcquireRelease.cpp
```

```
// #include <atomic>
```

```
// #include <iostream>
```

```
// #include <thread>
```

```
// std::atomic<int> x{0};
```

```
// std::atomic<int> y{0};
```

```
// void writing(){
```

```
// x.store(2000, std::memory_order_relaxed);
```

```
// y.store(11, std::memory_order_release);
```

```
// }
```

```
// void reading(){
```

```
// std::cout << y.load(std::memory_order_acquire) << " ";
```

```
// std::cout << x.load(std::memory_order_relaxed) << std::endl;
```

```
// }
```

```
// int main(){
```

```
// std::thread thread1(writing);
```

```
// std::thread thread2(reading);
```

```

// thread1.join();
// thread2.join();
// };
// =====
/*

```

On first glance you will notice that all operations are atomic, so the program is well-defined. But the second glance shows more; the atomic operations on y are attached with the flag std::memory_order_release (line 12) and std::memory_order_acquire (line 16). In contrast to that, the atomic operations on x are annotated with std::memory_order_relaxed (lines 11 and 17), so there are no synchronizations and ordering constraints for x. The answer to the possible values for x and y can only be given by y.

It holds:

y.store(11, std::memory_order_release) synchronizes-with y.load(std::memory_order_acquire)
x.store(2000, std::memory_order_relaxed) is visible before y.store(11, std::memory_order_release)
y.load(std::memory_order_acquire) is visible before x.load(std::memory_order_relaxed)

Explanation#

I will elaborate a little bit more on these three statements. The key idea is that the store of y in line 12 synchronizes with a load of y in line 16. This is due to the fact that the operations take place on the same atomic and they use the acquire-release semantic. y uses std::memory_order_release in line 12 and std::memory_order_acquire in line 16. The pairwise operation on y has another very interesting property. They establish a kind of barrier relative to y; so, x.store(2000, std::memory_order_relaxed) cannot be executed after y.store(std::memory_order_release) and x.load() cannot be executed before y.load().

The reasoning in the case of the acquire-release semantic is more sophisticated than in the case of the previous sequential consistency, but the possible values for x and y are the same; Only the combination y == 11 and x == 0 is not possible.

There are three different interleavings of the threads possible, which produce the three different combinations of the values x and y.

thread1 will be executed before thread2.

thread2 will be executed before thread1.

thread1 executes x.store(2000) before thread2 will be executed.

To make a long story short, here are all possible values for x and y.

y	x	Values possible?
0	0	Yes
11	0	
0	2000	Yes
11	2000	Yes

Once more, let's verify our thinking with CppMem.

CppMem#

Here is the corresponding program:

```

*/
// =====
// =====

```

```
/*
```

Possible Executions#

I will only refer to the three graphs with consistent execution. The graphs show that there is an acquire-release semantic between the store-release of y and the load-acquire operation of y. It will not make any difference if the reading of y (rf) takes places in the main thread or in a separate thread. The graphs show the synchronizes-with relation using a sw annotated arrow.

Execution for (y = 0, x = 0)#

Execution for (y = 0, x = 2000)#

Execution for (y = 11, x = 2000)#

x does not have to be atomic. This was my first and wrong assumption; see why in the next lesson. :)

```
*/
```

```
// =====
```

```
// =====
```

```
/*
```

CppMem: Atomics with Non-Atomics

This lesson highlights atomics with non-atomics used in the context of CppMem.

We'll cover the following

CppMem

A typical misunderstanding in the application of the acquire-release semantic is to assume that the acquire operation is waiting for the release operation. Based on this wrong assumption, you m

```
*/
```

```
// =====
```

```
// ongoingOptimisationAcquireReleaseBroken.cpp
```

```
// #include <atomic>
// #include <iostream>
// #include <thread>
```

```
// int x = 0;
// std::atomic<int> y{0};
```

```
// void writing(){
//   x = 2000;
//   y.store(11, std::memory_order_release);
// }
```

```
// void reading(){
//   std::cout << y.load(std::memory_order_acquire) << " ";
//   std::cout << x << std::endl;
// }
```

```
// int main(){
//   std::thread thread1(writing);
//   std::thread thread2(reading);
```

```

// thread1.join();
// thread2.join();
// };
// =====
/*
The program has a data race on x and, therefore, undefined behavior. The acquire-release semantic
guarantees that if y.store(11, std::memory_order_release) (line 12) is executed before
y.load(std::memory_order_acquire) (line 16), then x = 2000 (line 11) will be executed before the reading of x
in line 17. If not, the reading of x will be executed at the same time as the writing of x. So, we have concurrent
access to a shared variable and one of them is a write operation. That is by definition a data race.

```

To make my point more clear, let me use CppMem.

```

CppMem#
*/
// =====
// int main(){
//   int x = 0;
//   atomic_int y = 0;
//   {{{
//     x = 2000;
//     y.store(11, memory_order_release);
//   }
//   |||
//     y.load(memory_order_acquire);
//     x;
//   }}}
// }
// =====
/*

```

The data race occurs when one thread is writing $x = 2000$ and the other thread is reading x . Therefore, we get a dr symbol (data race) on the corresponding yellow arrow.

The final step in the process of ongoing optimization is still missing: relaxed semantic.

```

*/
// =====
// =====
/*

```

CppMem: Atomics with a Relaxed Semantic

This lesson gives an overview of atomics with a relaxed semantic used in the context of CppMem.

We'll cover the following

CppMem

With the relaxed semantic, we don't have synchronization or ordering constraints on atomic operations; only the atomicity of the operations is guaranteed.

```

*/
// =====

```

```

// ongoingOptimisationRelaxedSemantic.cpp

// #include <atomic>
// #include <iostream>
// #include <thread>

// std::atomic<int> x{0};
// std::atomic<int> y{0};

// void writing(){
//   x.store(2000, std::memory_order_relaxed);
//   y.store(11, std::memory_order_relaxed);
// }

// void reading(){
//   std::cout << y.load(std::memory_order_relaxed) << " ";
//   std::cout << x.load(std::memory_order_relaxed) << std::endl;
// }

// int main(){
//   std::thread thread1(writing);
//   std::thread thread2(reading);
//   thread1.join();
//   thread2.join();
// }
// =====
/*

```

For the relaxed semantic, my key questions are very easy to answer. These are my questions:

Does the program have well-defined behavior?

Which values for x and y are possible?

On one hand, all operations on x and y are atomic, so the program is well-defined. On the other hand, there are no restrictions on the possible interleavings of threads. The result may be that thread2 sees the operations on thread1 in a different order. This is the first time in our process of ongoing optimizations that thread2 can display x == 0 and y == 11 and all combinations of x and y are therefore possible.

```

*/
// =====
// =====
/*
```

y	x	Values possible?
0	0	Yes
11	0	Yes
0	2000	Yes
11	2000	Yes

Now I'm curious how the graph of CppMem will look like for x == 0 and y == 11?

```

CppMem#
12345678910111213
int main(){
```

```

atomic_int x = 0;
atomic_int y = 0;
{{{
    x.store(2000, memory_order_relaxed);
    y.store(11, memory_order_relaxed);
}
|||
    y.load(memory_order_relaxed);
    x.load(memory_order_relaxed);
}
}

```

That was the CppMem program. Now, let's go to the graph that produces counter-intuitive behavior.

x reads the value 0 (line 10), but y reads the value 11 (line 9); this happens, although the writing of x (line 5) is sequenced before the writing of y (line 6).

Conclusion

This lesson gives a conclusion of CppMem and how it helps in optimization of programs.

Using a small program and successively improving it was quite enlightening. First, more interleavings of threads were possible with each step; therefore, more different values for x and y were also possible. Second, the challenge of the program increases with each improvement. Even for this small program, CppMem provides invaluable services.

```

*/
// =====
// =====
/*

```

Execution Policies

This lesson explains execution policies (introduced in C++ 17) in detail.

We'll cover the following

Execution Policies

The Standard Template Library has more than 100 algorithms for searching, counting, and manipulating ranges and their elements. With C++17, 69 of them are overloaded and 8 new ones are added. The overloaded and new algorithms can be invoked with a so-called execution policy.

By using an execution policy, you can specify whether the algorithm should run sequentially, in parallel, or in parallel with vectorization.

Execution Policies#

The policy tag specifies whether a program should run sequentially, in parallel, or in parallel with vectorization.

`std::parallel::seq`: runs the program sequentially

`std::parallel::par`: runs the program in parallel on multiple threads

`std::parallel::par_unseq`: runs the program in parallel on multiple threads and allows the interleaving of individual loops; this permits a vectorized version with SIMD (Single Instruction Multiple Data) extensions.

The following code snippet shows all execution policies:

```

12345678910111213
vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};

// standard sequential sort
std::sort(v.begin(), v.end());

// sequential execution
std::sort(std::parallel::seq, v.begin(), v.end());

// permitting parallel execution
std::sort(std::parallel::par, v.begin(), v.end());

```

The example shows that you can still use the classic variant of `std::sort` (line 4). Also, in C++17 you can specify explicitly whether the sequential (line 7), parallel (line 10), or the parallel and vectorized (line 13) version should be used. In the next lesson, I will discuss parallel and vectorized execution in more detail.

```

*/
// =====
// =====
/*

```

Parallel & Vectorized Execution

This lesson explains parallel and vectorized execution policies, which were introduced in C++ 17, in detail.

We'll cover the following

Parallel & Vectorized Execution

Without Optimisation?

With maximum Optimization?

Hazards of Data Races and Deadlocks

Parallel & Vectorized Execution#

Whether an algorithm runs in a parallel and vectorized way depends on many factors. For example, it depends on whether the CPU and the operating system support SIMD instructions. Additionally, it also depends on the compiler and the optimization level that you used to translate your code.

The following example shows a simple loop for creating a new vector.

```

const int SIZE= 8;

int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
int res[] = {0, 0, 0, 0, 0, 0, 0, 0};
int main(){
    for (int i= 0; i < SIZE; ++i) {
        res[i]= vec[i]+5;
    }
}
/*
// =====
// =====
/*
const int SIZE= 8;

```

```

int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
int res[] = {0, 0, 0, 0, 0, 0, 0, 0};
int main(){
    for (int i= 0; i < SIZE; ++i) {
        res[i]= vec[i]+5;
    }
}
*/
// =====
// =====
/*

```

Line 8 is the key line in this small example. Thanks to the compiler explorer, we can have a closer look at the assembler instructions generated by clang 3.6.

Without Optimisation?

Here are the assembler instructions. Each addition is done sequentially.

```

*/
// =====
// =====
/*

```

With maximum Optimization?

By using the highest optimization level, -O3, special registers such as xmm0 are used that can hold 128 bits or 4 ints. This means that the addition takes place in parallel on four elements of the vector.

Hazards of Data Races and Deadlocks

The parallel algorithm does not automatically protect you from data races and deadlocks.

```
int numComp= 0;
```

```
std::vector<int> vec={1,3,8,9,10};
```

```
std::sort(std::parallel::vec, vec.begin(), vec.end(),
          [&numComp](int fir, int sec){ numComp++; return fir < sec; });
```

The small code snippet has a data race. numComp counts the number of operations, which means that numComp in particular is concurrently modified in the lambda-function. In order to be well-defined, numComp has to be protected.

```

*/
// =====
// =====
/*

```

Algorithms

This lesson gives an overview of all 69 algorithms introduced in C++17.

Here are the 69 algorithms with parallelised versions.

```
std::adjacent_difference     std::adjacent_find     std::all_of     std::any_of
```

```

std::copy      std::copy_if    std::copy_n    std::count
std::count_if  std::equal      std::fill     std::fill_n
std::find      std::find_end   std::find_first_of   std::find_if
std::find_if_not  std::generate  std::generate_n    std::includes
std::inner_product  std::inplace_merge  std::is_heap    std::is_heap_until
std::is_partitioned  std::is_sorted  std::is_sorted_until  std::lexicographical_compare
std::max_element  std::merge      std::min_element  std::minmax_element
std::mismatch  std::move      std::none_of     std::nth_element
std::partial_sort  std::partial_sort_copy std::partition  std::partition_copy
std::remove    std::remove_copy  std::remove_copy_if  std::remove_if
std::replace    std::replace_copy  std::replace_copy_if  std::replace_if
std::reverse    std::reverse_copy  std::rotate      std::rotate_copy
std::search     std::search_n    std::set_difference  std::set_intersection
std::set_symmetric_difference  std::set_union  std::sort      std::stable_partition
std::stable_sort  std::swap_ranges  std::transform  std::uninitialized_copy
std::uninitialized_copy_n  std::uninitialized_fill  std::uninitialized_fill_n  std::unique
std::unique_copy

```

Also, we get 8 new algorithms that I have discussed in the next lesson.

```

*/
// =====
// =====
/*

```

The New Algorithms

This lesson gives an overview of the new algorithms that are a part of C++17.

The new algorithms are in the `std` namespace. `std::for_each` and `std::for_each_n` require the header `<algorithm>`, but the remaining 6 algorithms require the header `<numeric>`.

Here is an overview of the new algorithms:

Algorithm	Description
-----------	-------------

`std::for_each` Applies a unary callable to the range.

`std::for_each_n` Applies a unary callable to the first n elements of the range.

`std::exclusive_scan` Applies from the left a binary callable up to the ith (exclusive) element of the range. The left argument of the callable is the previous result. Stores intermediate results.

`std::inclusive_scan` Applies from the left a binary callable up to the ith (inclusive) element of the range. The left argument of the callable is the previous result. Stores intermediate results.

`std::transform_exclusive_scan` First applies a unary callable to the range and then applies `std::exclusive_scan`.

`std::transform_inclusive_scan` First applies a unary callable to the range and then applies `std::inclusive_scan`.

`std::reduce` Applies from the left a binary callable to the range.

`std::transform_reduce` Applies first a unary callable to the range and then `std::reduce`.

Admittedly, this description is not easy to digest; therefore, I will first present an exhaustive example and then write about the functional heritage of these functions. I will ignore the new `std::for_each` algorithm. In contrast to the C++98 variant that returns a unary function, the additional C++17 variant returns nothing.

As far as I know, at the time this course is being written (June 2017) there is no standard-conforming implementation of the parallel STL available. Therefore, I used the HPX implementation to produce the output. The HPX (High-Performance ParallelX) is a framework that is a general-purpose C++ runtime system for parallel and distributed applications of any scale.

```
12345678910111213141516171819202122232425262728293031
```

```
// newAlgorithm.cpp
```

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

int main(){
```

I apply the new algorithms to a `std::vector<int>` (line 16) and a `std::vector<std::string>` (line 57).

The `std::for_each_n` algorithm in line 17 maps the first n ints of the vector to their powers of 2.

`std::exclusive_scan` (line 26) and `std::inclusive_scan` (line 36) are quite similar; both apply a binary operation to their elements. The difference is that `std::exclusive_scan` excludes the last element in each iteration.

Let me explain the `std::transform_exclusive_scan` in line 47. In the first step, I apply the lambda function `[](int arg){ return arg *= arg; }` to each element of the range `resVec3.begin()` to `resVec3.end()`. In the second step, I apply the binary operation `[](int fir, int sec){ return fir + sec; }` to the intermediate vector. This means to sum up all elements using 0 as the initial value. The result is placed in `resVec4.begin()`.

The `std::transform_inclusive_scan` function in line 60 is similar. This function maps each element to its length.

The `std::reduce` function puts ":" characters between every two elements of the input vector. The resulting string should not start with a ":" character; therefore, the range starts at the second element (`strVec2.begin() + 1`) and uses the first element of the vector `strVec2[0]` as the initial element.

I will now discuss the `std::transform_reduce` function in line 79. First of all, the C++ algorithm `transform` is often called `map` in other languages; therefore, we can also call `std::transform_reduce` `std::map_reduce`. `std::transform_reduce` is the well-known parallel MapReduce algorithm implemented in C++. Accordingly, `std::transform_reduce` maps a unary callable (`[](std::string s){ return s.length(); }`) onto a range and reduces the pair to a output value: `[](std::size_t a, std::size_t b){ return a + b; }`.

Studying the output of the program will help you.

More overloads

All C++ variants of `reduce` and `scan` have more overloads. In the simplest form, you can invoke them without an initial element and without a binary callable. If you do not use an initial element, the first element will be used. If you do not use a binary callable, the addition will be used as the binary operation.

In the next lesson, I will discuss new algorithms from a functional perspective.

```
main = do let ints = [1..9]
        let strings =["Only","for","testing","purpose"]
        print (map (\a -> a * a) ints)
```

```

print (scanl (*) 1 ints)
print (scanl (+) 0 ints)
print (scanl (+) 0 . map(\a -> a * a) $ints)
print (scanl1 (+) . map(\a -> length a) $strings)
print (foldl1 (\l r -> l++ ":" ++r) strings)
print (foldl (+) 0 . map (\a -> length a) $strings)

```

(1) and (2) define a list of integers and a list of strings. In (3), I apply the lambda function ($\lambda a \rightarrow a * a$) to the list of integers. That being said, (4) and (5) are more sophisticated. The expression (4) multiplies (*) all pairs of integers starting with the 1 as neutral element of multiplication. Expression (5) does the corresponding for addition. Expressions (6), (7), and (9) are, for the imperative eye, quite challenging. You have to read them from right to left. $scanl1 (+) . map(\lambda a \rightarrow length a)$ (7) is a function composition. The dot (.) symbol composes the two functions. The first function maps each element to its length, the second function adds the list of lengths together. (9) is similar to (7), the difference being that foldl produces one value and requires an initial element that is in case 0. Now expression (8) should be readable; it successively joins two strings with the ":" character.

```

*/
// =====
// =====
/*

```

New Algorithms - A Functional Perspective

This lesson gives an overview of the new algorithms that are a part of C++17.

All new functions have a pendant in the purely functional language Haskell.

Functions Haskell

std::for_each_n	map
std::exclusive_scan	scanl
std::inclusive_scan	scanl1
std::transform_exclusive_scan and std::transform_inclusive_scan	composition of map and scanl or scanl1
std::reduce	foldl or foldl1
transform_reduce	composition of map and foldl or foldl1

Before I show you Haskell in action, let me briefly discuss the different functions.

Functions Description

map	applies a function to a list
foldl and foldl1	apply a binary operation to a list and reduce the list to a value. foldl needs, in contrast to foldl1, an initial value.
scanl and scanl1	apply the same strategy as foldl and foldl1 but produce all intermediate results so that you will get back a list

Note: foldl, foldl1, scanl, and scanl1 start their job from the left.

Let's have a look at the running example of Haskell functions:

```

main = do let ints = [1..9]
          let strings = ["Only", "for", "testing", "purpose"]
          print (map (\a -> a * a) ints)
          print (scanl (*) 1 ints)
          print (scanl (+) 0 ints)

```

```
print (scanl (+) 0 . map(\a -> a * a) $ints)
print (scanl1 (+) . map(\a -> length a) $strings)
print (foldl1 (\l r -> l++ ":" ++r) strings)
print (foldl (+) 0 . map (\a -> length a) $strings)
```

(1) and (2) define a list of integers and a list of strings. In (3), I apply the lambda function ($\lambda a \rightarrow a * a$) to the list of integers. That being said, (4) and (5) are more sophisticated. The expression (4) multiplies (*) all pairs of integers starting with 1 as a neutral element of multiplication. Expression (5), on the other hand, does the corresponding for addition. Expressions (6), (7), and (9) are, for the imperative eye, quite challenging; you have to read them from right to left. `scanl1 (+) . map(\a -> length) (7)` is a function composition. The dot (.) symbol composes the two functions. The first function maps each element to its length; the second function adds the list of lengths together. (9) is similar to (7), the difference being that `foldl` produces one value and requires an initial element that is 0 in this case. Now expression (8) should be readable; it successively joins two strings with the ":" character.

```
/*
// =====
// =====
/*
```

Atomic Smart Pointers

This lesson gives an overview of the atomic smart pointers, predicted to be introduced in C++20.

We'll cover the following

Atomic Smart Pointers

This chapter is about the future of C++. In this chapter, my intent is not to be as precise as I was in the other chapters in this course. That's for two reasons: First, not all of the presented features will make it into the C++20 standard; second, if a feature makes it into the C++20 standard, the interface of that feature will very likely change. My aim in this chapter is just to give you an idea about the upcoming concurrency features in C++.

```
/*
// =====
// =====
/*
```

Atomic Smart Pointers

This lesson gives an overview of the atomic smart pointers, predicted to be introduced in C++20.

We'll cover the following

Atomic Smart Pointers

This chapter is about the future of C++. In this chapter, my intent is not to be as precise as I was in the other chapters in this course. That's for two reasons: First, not all of the presented features will make it into the C++20 standard; second, if a feature makes it into the C++20 standard, the interface of that feature will very likely change. My aim in this chapter is just to give you an idea about the upcoming concurrency features in C++.

```
/*
// =====
// =====
```

```
/*
```

Thread-Safe Linked List Using Atomic Pointers

This lesson describes the use of thread-safe singly linked list using atomic pointers.

Let's see the C++11 version of the thread-safe singly linked list first; then we'll enhance it using atomic smart pointers from C++20.

```
/*
// =====
// #include <iostream>
// #include <atomic>
// #include <memory> //for shared_ptr
// using namespace std;

// template<typename T> class concurrent_stack {
//   struct Node {
//     T t;
//     shared_ptr<Node> next;
//   };
//   shared_ptr<Node> head;
//   concurrent_stack(concurrent_stack &) = delete;
//   void operator=(concurrent_stack &) = delete;

// public:
//   concurrent_stack() = default;
//   ~concurrent_stack() = default;

//   class reference{
//     shared_ptr<Node> p;
//   public:
//     reference(shared_ptr<Node> p_) : p{p_}{} }
//     T& operator* () {return p->t;}
//     T* operator->() {return &p->t;}
//   };

//   auto find( T t) const {
//     auto p = atomic_load(&head);
//     while(p && p->t != t)
//       p = p->next;
//     return reference(move(p));
//   }

//   auto front() const{
//     return reference(atomic_load(&head));
//   }

//   void push_front(T t){
//     auto p = make_shared<Node>();
//     p->t = t;
//     p->next = atomic_load(&head);
//     while(p && !atomic_compare_exchange_weak(&head,&p->next,p)){ }
//   }

//   void pop_front(){
//     auto p = atomic_load(&head);
```

```
//    while(p && !atomic_compare_exchange_weak(&head,&p,p->next)){ }
// }

//};

// int main(){
//    concurrent_stack<int> cS;
//    cS.push_front(3);
//    cS.push_front(6);
//    cS.find(6);
//    cS.pop_front();
//    cS.front();
//}

//=====
/*
*/
// =====
// template <typename T>
// class concurrent_stack
//{
//    struct Node
//    {
//        T t;
//        shared_ptr<Node> next;
//    };
//    atomic_shared_ptr<Node> head;
//    // in C++11: remove "atomic_" and remember to use the special.
//    // functions every time you touch the variable
//    concurrent_stack(concurrent_stack &) = delete;
//    void operator=(concurrent_stack &) = delete;

// public:
//    concurrent_stack() = default;
//    ~concurrent_stack() = default;

//    class reference
//    {
//        shared_ptr<Node> p;

//    public:
//        reference(shared_ptr<Node> p_) : p{p_} {}
//        T &operator*() { return p->t; }
//        T *operator->() { return &p->t; }
//    };

//    auto find(T t) const
//    {
//        auto p = head.load(); // in C++11: atomic_load(&head)
//        while (p && p->t != t)
//            p = p->next;
//        return reference(move(p));
//    }
}
```

```

// }

// auto front() const
// {
//   return reference(head); // in C++11: atomic_load(&head)
// }

// void push_front(T t)
// {
//   auto p = make_shared<Node>();
//   p->t = t;
//   p->next = head; // in C++11: atomic_load(&head)
//   while (!head.compare_exchange_weak(p->next, p))
//   {
//   }
//   // in C++11: atomic_compare_exchange_weak(&head,&p->next,p)
// }

// void pop_front()
// {
//   auto p = head.load(); // in C++11: atomic_load(&head)
//   while(p && !head.compare_exchange_weak(p,p->next){ }
//   // in C++11: atomic_compare_exchange_weak(&head,&p,p->next)
// }
// =====
/*

```

Introduction to Extended Futures

This lesson gives an overview of extended futures, predicted to be introduced in C++20.

We'll cover the following

std::future

valid vs ready

Tasks in the form of promises and futures have an ambivalent reputation in C++11. On the one hand, they are a lot easier to use than threads or condition variables; on the other hand, they have a great deficiency. They cannot be composed. C++20 will overcome this deficiency.

I have written about tasks in the form of std::async, std::packaged_task, or std::promise and std::future. The details are here: [tasks](#). With C++20 we may get extended futures.

std::future#

The name extended futures is quite easy to explain. First, the interface of the C++11 std::future was extended; second, there are new functions for creating special futures that are compostable. I will start with my first point.

The extended future has three new methods:

The unwrapping constructor that unwraps the outer future of a wrapped future (`future<future<T>>`).
The predicate `is_ready` that returns if a shared state is available.

The method then that attaches a continuation to a future.

At first, the state of a future can be valid or ready.

valid vs ready#

valid: a future is valid if it has a shared state (with a promise). This does not have to be the case because you can default-construct an std::future without a promise

ready: a future is ready if the shared state is available, i.e. the promise has already produced its value

Therefore, (valid == true) is a requirement for (ready == true). My mental model of promise and future is that they are the endpoints of a data channel.

widget

Now the difference between valid and ready becomes quite natural. The future is valid if there is a data channel to a promise. The future is ready if the promise has already put its value into the data channel. It is possible to attach one future to another; I will discuss this in the next lesson.

```
/*
// =====
// =====
```

```
/*
```

Attaching Extended Futures

This lesson discusses how one extended future can be attached to another in C++20.

We'll cover the following

std::async, std::packaged_task, and std::promise

Creating new Futures

std::make_ready_future and std::make_exceptional_future

std::when_any and std::when_all

std::when_all

std::when_any

The method then empowers you to attach a future to another future. It often happens that a future will be packed into another future. The job of the unwrapping constructor is to unwrap the outer future.

The proposal N3721

Before I show the first code snippet, I have to say a few words about proposal N3721. Most of this section is from the proposal on “Improvements for std::future and Related APIs”, including my examples. Strangely, the original authors frequently did not use the final get call to get the result from the future. Therefore, I added the res.get call to the examples and saved the result in a variable myResult. Additionally, I fixed a few typos

```
/*
// =====
// #include <future>
// using namespace std;
// int main()
//{
//
//    future<int> f1 = async([](){}
//                           { return 123; });
//    future<string> f2 = f1.then([](future<int> f)
//                           {
```

```
// return to_string(f.get()); // here .get() won't block );
```

```
// auto myResult = f2.get();
```

```
// =====
```

```
/*
```

There is a subtle difference between the `to_string(f.get())` call (line 7) and the `f2.get()` call in line 10. As I already mentioned in the code snippet, the first call is non-blocking/asynchronous and the second call is blocking/synchronous. The `f2.get()` call waits until the result of the future-chain is available. This statement will also hold for chains such as `f1.then(...).then(...).then(...).then(...)` as it will hold for the composition of extended futures. The final `f2.get()` call is blocking.

`std::async`, `std::packaged_task`, and `std::promise`#

There is not much to say about the extensions of `std::async`, `std::package_task`, and `std::promise`. I only have to add that in C++20, all three return extended futures.

The composition of futures is more exciting. In the next lesson, I will discuss how we can compose asynchronous tasks.

`Creating new Futures#`

C++20 gets four new functions for creating special futures. These functions are `std::make_ready_future`, `std::make_exceptional_future`, `std::when_all`, and `std::when_any`.

First, let's look at the functions `std::make_ready_future` and `std::make_exceptional_future`.

`std::make_ready_future` and `std::make_exceptional_future`#

Both functions create a future that is immediately ready. In the first case, the future has a value; in the second case, an exception. Therefore, what seems to be strange at first actually makes a lot of sense. In C++11 the creation of a ready future requires a promise. This is necessary even if the shared state is immediately available.

```
future<int> compute(int x) {
if (x < 0) return make_ready_future<int>(-1);
if (x == 0) return make_ready_future<int>(0);
future<int> f1 = async([]() { return do_work(x); });
return f1;
}
/*
```

```
// =====
```

```
/*
```

Hence, the result must only be calculated by a promise if $(x > 0)$ holds. Now let's begin with future composition. A short remark: both functions are the pendant to the `return` function in a monad.

`std::when_any` and `std::when_all`#

Both functions have a lot in common. First, let's look at the input.

```
*/
```

```
// =====
```

```
// template < class InputIt >
```

```
// auto when_any(InputIt first, InputIt last)
```

```

// -> future<when_any_result<std::vector<typename std::iterator_traits<InputIt>::value_type>>>;
// template < class... Futures >
// auto when_any(Futures&&... futures)
// -> future<when_any_result<std::tuple<std::decay_t<Futures>...>>>;
// template < class InputIt >
// auto when_all(InputIt first, InputIt last)
// -> future<std::vector<typename std::iterator_traits<InputIt>::value_type>>;
// template < class... Futures >
// auto when_all(Futures&&... futures)
// -> future<std::tuple<std::decay_t<Futures>...>>;
// =====
/*
Both functions accept a pair of iterators for a future range or an arbitrary number of futures. The big difference is that in the case of the pair of iterators, the futures have to be of the same type; while in the case of the arbitrary number of futures, the futures can have different types and even std::future and std::shared_future can be used.

```

The output of the function depends on whether a pair of iterators or an arbitrary number of futures (variadic template) was used; either way, both functions return a future. If a pair of iterators were used, you would get a future of futures in an std::vector: future<vector<future<R>>. If you use a variadic template, you will get a future of futures in a std::tuple: future<tuple<future<R0>, future<R1>, ... >.

This covers their commonalities. The future that both functions return will be ready if all (when_all) or any (when_any) of the input futures are ready. The next two examples show the usage of std::when_all and std::when_any.

```

std::when_all#
*/
// =====
// #include <future>

// using namespace std;

// int main() {

// shared_future<int> shared_future1 = async([] { return intResult(125); });
// future<string> future2 = async([]() { return stringResult("hi"); });

// future<tuple<shared_future<int>, future<string>>> all_f =
//     when_all(shared_future1, future2);

// future<int> result = all_f.then(
//     [](future<tuple<shared_future<int>, future<string>>> f){
//         return doWork(f.get());
//     });
// auto myResult = result.get();

```

```
//}  
// =====  
/*  
The future all_f (line 10) composes both the future shared_future1 (line 7) and future2 (line 8). The future  
result in line 13 will be executed if all underlying futures are ready. In this case, the future all_f in line 15 will  
be executed. The result is in the future result and can be used in line 18.
```

std::when_any#

The future in when_any can be taken by result in line 11 below. result provides the information indicating
which input future is ready. If you don't use when_any_result, you have to ask each future if it is ready - which
is tedious.

```
*/  
// =====  
// #include <future>  
// #include <vector>  
  
// using namespace std;  
  
// int main(){  
  
//   vector<future<int>> v{ .... };  
//   auto future_any = when_any(v.begin(), v.end());  
  
//   when_any_result<vector<future<int>>> result = future_any.get();  
  
//   future<int>& ready_future = result.futures[result.index];  
  
//   auto myResult = ready_future.get();  
  
// }
```

```
/*  
future_any is the future that will be ready if one of its input futures is ready. future_any.get() in line 11  
returns the future result. By using result.futures[result.index] (line 13) you have the ready_future, and thanks  
to ready_future.get(), you can ask for the result of the job.
```

```
*/  
// =====  
// =====  
/*
```

Creating New Futures

This lesson discusses how one extended future can be attached to another in C++20.

We'll cover the following

```
std::make_ready_future and std::make_exceptional_future  
std::when_any and std::when_all  
std::when_all  
std::when_any
```

C++20 gets four new functions for creating special futures: `std::make_ready_future`, `std::make_exceptional_future`, `std::when_all`, and `std::when_any`. First, let's look at the functions `std::make_ready_future`, and `std::make_exceptional_future`.

`std::make_ready_future` and `std::make_exceptional_future`#

Both functions create a future that is immediately ready. In the first case, the future has a value; in the second case, an exception. Therefore, what seems to be strange at first actually makes a lot of sense. In C++11, the creation of a ready future requires a promise. This is necessary even if the shared state is immediately available.

```
future<int> compute(int x) {
    if (x < 0) return make_ready_future<int>(-1);
    if (x == 0) return make_ready_future<int>(0);
    future<int> f1 = async([]() { return do_work(x); });
    return f1;
}
```

```
/*
// =====
// =====
/*
```

Hence, the result must only be calculated by a promise if $(x > 0)$ holds. Now let's begin with future composition. A short remark: both functions are the pendant to the return function in a monad.

`std::when_any` and `std::when_all`#

Both functions have a lot in common. First, let's look at the input.

```
template < class InputIt >
auto when_any(InputIt first, InputIt last)
    -> future<when_any_result<std::vector<typename std::iterator_traits<InputIt>::value_type>>;
```

```
template < class... Futures >
auto when_any(Futures&&... futures)
    -> future<when_any_result<std::tuple<std::decay_t<Futures>...>>;
```

```
template < class InputIt >
auto when_all(InputIt first, InputIt last)
    -> future<std::vector<typename std::iterator_traits<InputIt>::value_type>;
```

```
template < class... Futures >
auto when_all(Futures&&... futures)
    -> future<std::tuple<std::decay_t<Futures>...>;
```

```
/*
// =====
// =====
/*
```

Both functions accept a pair of iterators for a future range or an arbitrary number of futures. The big difference is that in the case of the pair of iterators, the futures have to be of the same type; while in the case

of the arbitrary number of futures, they can have different types and even std::future and std::shared_future can be used.

The output of the function depends on whether a pair of iterators or an arbitrary number of futures (variadic template) was used; either way, both functions return a future. If a pair of iterators were used, you would get a future of futures in an std::vector: future<vector<future<R>>. If you use a variadic template, you will get a future of futures in a std::tuple: future<tuple<future<R0>, future<R1>, ... >.

This covers their commonalities. The future that both functions return will be ready if all (when_all) or any (when_any) of the input futures are ready. The next two examples show the usage of std::when_all and std::when_any.

```
std::when_all#
*/
// =====
// #include <future>

// using namespace std;

// int main() {

//   shared_future<int> shared_future1 = async([] { return intResult(125); });
//   future<string> future2 = async([]() { return stringResult("hi"); });

//   future<tuple<shared_future<int>, future<string>>> all_f =
//     when_all(shared_future1, future2);

//   future<int> result = all_f.then(
//     [](<tuple<shared_future<int>, future<string>>> f){
//       return doWork(f.get());
//     });
//   auto myResult = result.get();

// }

// =====
/*
```

The future all_f (line 10) composes both the future shared_future1 (line 7) and future2 (line 8). The future result in line 13 will be executed if all underlying futures are ready. In this case, the future all_f in line 15 will be executed. The result is in the future result and can be used in line 18.

```
std::when_any#
The future in when_any can be taken by result in line 11 below. result provides the information indicating which input future is ready. If you don't use when_any_result, you have to ask each future if it is ready - which is tedious.
*/
// =====
// #include <future>
// #include <vector>
```

```

// using namespace std;

// int main()
//{
// vector<future<int>> v{...};
// auto future_any = when_any(v.begin(), v.end());

// when_any_result<vector<future<int>>> result = future_any.get();

// future<int> &ready_future = result.futures[result.index];

// auto myResult = ready_future.get();
//}

//=====
/*
future_any is the future that will be ready if one of its input futures is ready; future_any.get() in line 11
returns the future result. By using result.futures[result.index] (line 13) you have the ready_future, and thanks
to ready_future.get(), you can ask for the result of the job.
*/
//=====
//=====
```

/*
Latches and Barriers
This lesson gives an overview of latches and barriers, predicted to be introduced in C++20.

We'll cover the following

std::latch
std::barrier
std::flex_barrier

Latches and barriers are simple thread synchronization mechanisms which enable some threads to wait until a counter becomes zero. At first, don't confuse the new barriers with memory barriers (also known as fences). In C++20, we will presumably get latches and barriers in three variations: std::latch, std::barrier, and std::flex_barrier.

First, there are two questions:

What are the differences between these three mechanisms to synchronize threads? You can use an std::latch only once, but you can use an std::barrier and an std::flex_barrier more than once. Additionally, an std::flex_barrier enables you to execute a function when the counter becomes zero.

What use cases do latches and barriers support that cannot be done in C++11 and C++14 with futures, threads, or condition variables in combination with locks? Latches and barriers address no new use cases, but they are a lot easier to use; they are also more performant because they often use a lock-free mechanism internally.

Now, I will have a closer look at these three coordination mechanisms.

std::latch#

`std::latch` is a countdown counter; its value is set in the constructor. A thread can decrement the counter by using the method `thread.count_down_and_wait` and wait until the counter becomes zero. In addition, the method `thread.count_down` only decrements the counter by 1 without waiting. `std::latch` also has the method `thread.is_ready` that can be used to test if the counter is zero, and the method `thread.wait` to wait until the counter becomes zero. You cannot increment or reset the counter of a `std::latch`, hence you cannot reuse it.

Here is a short code snippet from the N4204 proposal:

```
/*
// =====
// void DoWork(threadpool* pool) {
//   latch completion_latch(NTASKS);
//   for (int i = 0; i < NTASKS; ++i) {
//     pool->add_task([&] {
//       // perform work
//       ...
//       completion_latch.count_down();
//     }));
//   }
//   // Block until work is done
//   completion_latch.wait();
// }
// =====
/*
```

I set the `std::latch completion_latch` in its constructor to `NTASKS` (line 2). The thread pool executes `NTASKS` (lines 4 - 8). At the end of each task (line 7), the counter will be decremented. Line 11 is the barrier for the thread running the function `DoWork` and, hence, for the small workflow. This thread has to wait until all tasks have been finished. In this case, an `std::barrier` is quite similar to an `std::latch`.

`std::barrier`#

The subtle difference between `std::latch` and `std::barrier` is that you can use `std::barrier` more than once because the counter will be reset to its previous value. Immediately after the counter becomes zero, the so-called completion phase starts. `std::barrier` has an empty completion phase; this changes with `std::flex_barrier`. `std::barrier` has two interesting methods: `std::arrive_and_wait` and `std::arrive_and_drop`. While `std::arrive_and_wait` waits at the synchronization point, `std::arrive_and_drop` removes itself from the synchronization mechanism.

The proposal N4204

The proposal uses a `vector<thread*>` and pushes the dynamically allocated threads onto the vector `workers.push_back(new thread([&]{ ... });`; this is a memory leak. Instead, you should put the threads into a `std::unique_ptr` or create them directly in the vector: `workers.emplace_back[&]{ ... }`. This observation holds for the example with `std::barrier` and `std::flex_barrier`. The names in the example with `std::flex_barrier` are a little bit confusing. For example, `std::flex_barrier` is called `notifying_barrier`, so I changed the name to `flex_barrier`. Additionally, the variable `n_threads` (representing the number of threads) was not initialized or was missing; I initialized it to `NTASKS`.

Before I take a closer look at `std::flex_barrier` and the completion phase in particular, I will give a short example demonstrating the usage of `std::barrier`.

```
/*
// =====
// void DoWork() {
```

```

// Tasks& tasks;
// int n_threads{NTASKS};
// vector<thread*> workers;

// barrier task_barrier(n_threads);

// for (int i = 0; i < n_threads; ++i) {
//   workers.push_back(new thread([&] {
//     bool active = true;
//     while(active) {
//       Task task = tasks.get();
//       // perform task
//       ...
//       task_barrier.arrive_and_wait();
//     }
//   }));
// }

// // Read each stage of the task until all stages are complete.
// while (!finished()) {
//   GetNextStage(tasks);
// }
// =====
/*

```

The barrier in line 6 is used to coordinate a number of threads that perform their task multiple times; in this case, there are n_threads (line 3). Each thread takes its task at line 12 via tasks.get(), performs it and waits - once it is done with its task (line 15) - until all threads are done with their tasks. After that, it takes a new task in line 12 while active returns true in line 11. In contrast to std::barrier, std::flex_barrier has an additional constructor.

std::flex_barrier#

This additional constructor can be parameterized by a callable unit that will be invoked in the completion phase. The callable has to return a number; this number sets the value of the counter in the completion phase. A return of -1 means that the counter keeps the same counter value in the next iteration. Numbers smaller than -1 are not allowed.

The completion phase performs the following steps:

All threads are blocked.

An arbitrary thread is unblocked and executes the callable unit.

If the completion phase is done, all threads will be unblocked.

The code snippet shows the usage of a std::flex_barrier.

```

*/
// =====
// void DoWork() {
//   Tasks& tasks;
//   int initial_threads;
//   int n_threads{NTASKS};
//   atomic<int> current_threads(initial_threads);
//   vector<thread*> workers;

```

```

// // Create a flex_barrier, and set a lambda that will be
// // invoked every time the barrier counts down. If one or more
// // active threads have completed, reduce the number of threads.
// std::function rf = [&] { return current_threads;};
// flex_barrier task_barrier(n_threads, rf);

// for (int i = 0; i < n_threads; ++i) {
//   workers.push_back(new thread([&] {
//     bool active = true;
//     while(active) {
//       Task task = tasks.get();
//       // perform task
//       ...
//       if (finished(task)) {
//         current_threads--;
//         active = false;
//       }
//       task_barrier.arrive_and_wait();
//     }
//   }));
// }

// // Read each stage of the task until all stages are complete.
// while (!finished()) {
//   GetNextStage(tasks);
// }
// =====
/*

```

The example follows a similar strategy as the previous example with `std::barrier`. The difference is that this time the `std::flex_barrier` counter is adjusted at runtime; hence, the `std::flex_barrier task_barrier` in line 11 gets a lambda function. This lambda function captures its variable `current_threads` by reference: `[&] { return current_threads;}`. The variable will be decremented in line 21, and `active` will be set to false if the thread has completed its task. Therefore, the counter is decremented in the completion phase.

`std::flex_barrier` can increase the counter in contrast with `std::barrier` or `std::latch`. You can read further details of `std::latch`, `std::barrier`, and `std::flex_barrier` at cppreference.com.

```

*/
// =====
// =====
/*

```

Transactional Memory - An Overview

This lesson gives an outline of transactional memory, predicted to be introduced in C++20.

We'll cover the following

ACI(D)

Transactional memory is based on the idea of a transaction from database theory. Transactional memory makes working with threads a lot easier for two reasons: first, data races and deadlocks disappear; second, transactions are composable.

A transaction is an action that has the following properties: Atomicity, Consistency, Isolation, and Durability (ACID). Except for the durability or storing the result of an action, all properties hold for transactional memory in C++. Now three short questions are left.

ACI(D)#[/h4>

What do atomicity, consistency, and isolation mean for an atomic block consisting of some statements?

```
atomic{  
    statement1;  
    statement2;  
    statement3;  
}
```

Atomicity: Either all or none of the statements in the block are performed.

Consistency: The system is always in a consistent state. All transactions establish a total order.

Isolation: Each transaction runs in total isolation from other transactions.

How do these properties apply? A transaction remembers its initial state and will be performed without synchronization. If a conflict happens during its execution, the transaction will be interrupted and restored to its initial state. This rollback causes the transaction to be executed again. If the initial state of the transaction is maintained at the end of the transaction, the transaction will be committed.

A transaction is a kind of speculative activity that is only committed if the initial state holds. In contrast to a mutex, it is an optimistic approach. A transaction is performed without synchronization; it will only be published if no conflict occurs. That being said, a mutex is a pessimistic approach. First, the mutex ensures that no other thread can enter the critical region. Next, the thread will enter the critical region if it is the exclusive owner of the mutex, hence all other threads are blocked.

```
*/  
// =====
```

```
// =====  
/*
```

The Two Flavors of Transactional Memory

This lesson explains synchronized and atomic blocks in transactional memory.

We'll cover the following

Synchronized & Atomic Blocks

Synchronized Blocks

Atomic Blocks

transaction_safe versus transaction_unsafe Code

C++ supports transactional memory in two flavors: synchronized blocks and atomic blocks.

Synchronized & Atomic Blocks#[/h4>

Up to now, I only wrote about transactions. Now, I will write about synchronized blocks and atomic blocks. Both can be encapsulated in each other; specifically, synchronized blocks are not atomic blocks because they

can execute transaction-unsafe. An example would be code like the output to the console which can not be undone. For this reason, synchronized blocks are often called relaxed blocks.

Synchronized Blocks#

Synchronized blocks behave like they are protected by a global lock, i.e. This means that all synchronized blocks follow a total order. In particular: all changes to a synchronized block are available in the next synchronized block. There is a synchronizes-with relation between the synchronized blocks because of the commit of the transaction synchronizes-with the next start of a transaction. Synchronized blocks cannot cause a deadlock because they create a total order. While a classical lock protects a memory area, a global lock of a synchronized block protects the total program. This is the reason the following program is well-defined:

```
/*
// =====
// synchronized.cpp

// #include <iostream>
// #include <vector>
// #include <thread>

// int i= 0;

// void increment(){
//   synchronized{
//     std::cout << ++i << ",";
//   }
// }

// int main(){

// std::cout << std::endl;

// std::vector<std::thread> vecSyn(10);
// for(auto& thr: vecSyn)
//   thr = std::thread([]{ for(int n = 0; n < 10; ++n) increment(); });
// for(auto& thr: vecSyn) thr.join();

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

// }
// =====
/*
```

Although the variable `i` in line 7 is a global variable and the operations in the synchronized block are transaction-unsafe, the program is well-defined. 10 threads concurrently invoke the function `increment` (line 21) ten times, incrementing the variable `i` in line 11. The access to `i` and `std::cout` happens in total order. This is the characteristic of the synchronized block. Afterwards, the program returns the expected result; the values for `i` are written in an increasing sequence, separated by a comma.

What about data races? You can have them with synchronized blocks. A small modification of the source code is sufficient to introduce a data race

```
/*
// =====
```

```

// nonsynchronized.cpp

// #include <chrono>
// #include <iostream>
// #include <vector>
// #include <thread>

// using namespace std::chrono_literals;
// using namespace std;

// int i= 0;

// void increment(){
//   synchronized{
//     cout << ++i << ",";
//     this_thread::sleep_for(1ns);
//   }
// }

// int main(){

// cout << endl;

// vector<thread> vecSyn(10);
// vector<thread> vecUnsyn(10);

// for(auto& thr: vecSyn)
//   thr = thread([]{ for(int n = 0; n < 10; ++n) increment(); });
// for(auto& thr: vecUnsyn)
//   thr = thread([]{ for(int n = 0; n < 10; ++n) cout << ++i << ","; });

// for(auto& thr: vecSyn) thr.join();
// for(auto& thr: vecUnsyn) thr.join();

// cout << "\n\n";

// }
// =====
/*

```

To observe the data race, I let the synchronized block sleep for a nanosecond (line 16). At the same time I access the output stream std::cout without a synchronized block (line 30). In total, 20 threads increment the global variable i - half of them without synchronization. The C++11 standard guarantees that the characters will be written atomically; that is not an issue. What is worse is that the variable i is written by at least 2 threads. This is a data race, hence, the program has undefined behavior. The total order of synchronized blocks also holds for atomic blocks.

Atomic Blocks#

You can execute transaction-unsafe code in a synchronized block, but not in an atomic block. Atomic blocks are available in three forms: `atomic_noexcept`, `atomic_commit`, and `atomic_cancel`. The three suffixes `_noexcept`, `_commit`, and `_cancel` define how an atomic block should manage an exception.

`atomic_noexcept`: If an exception is thrown, `std::abort` will be called and the program aborts.

`atomic_cancel`: In the default case, `std::abort` is called. This will not hold if a transaction-safe exception is thrown that is responsible for ending the transaction. In this case the transaction will be canceled, put to its initial state, and the exception will be thrown.

`atomic_commit`: If an exception is thrown, the transaction will be committed.

Transaction-safe exceptions are: `std::bad_alloc`, `std::bad_array_length`, `std::bad_array_new_length`, `std::bad_cast`, `std::bad_typeid`, `std::bad_exception`, `std::exception`, and all exceptions are derived from one of these.

`transaction_safe` versus `transaction_unsafe` Code#

You can declare a function as `transaction_safe` or attach the `transaction_unsafe` attribute to it.

```
int transactionSafeFunction() transaction_safe;  
[[transaction_unsafe]] int transactionUnsafeFunction();  
  
*/  
// ======  
// ======  
/*
```

`transaction_safe` belongs to the type of the function. What does `transaction_safe` mean? A `transaction_safe` function is, according to the proposal N4265, a function that has a `transaction_safe` definition. This holds true if the following properties do not apply to its definition:

It has a volatile parameter or a volatile variable.

It has `transaction-unsafe` statements.

If the function uses a constructor or destructor of a class in its body that has a volatile non-static member. Of course this definition of `transaction_safe` is not sufficient because it uses the term `transaction_unsafe`. You can read the proposal N4265 for the details.

```
*/  
// ======  
// ======  
/*
```

Introduction to Coroutines

This lesson gives an overview of coroutines, predicted to be introduced in C++20.

We'll cover the following

A Generator Function

Coroutines are functions that can suspend and resume their execution while keeping their state. The evolution of functions goes one step further in C++20.

What I present in this section as a new idea in C++20 is actually quite old. The term coroutine was coined by Melvin Conway; He used it in his publication on compiler construction in 1963. Likewise, Donald Knuth called procedures a special case of coroutines. Sometimes, it just takes a while to get your ideas accepted.

With the new keywords `co_await` and `co_yield`, C++20 will extend the execution of a C++ function with two new concepts.

Thanks to `co_await` expression it will be possible to suspend and resume the execution of the expression. If you use `co_await` expression in a function `func`, the call `auto getResult = func()` will not block if the result of the function is not available. Instead of resource-consuming blocking, you have resource-friendly waiting.

`co_yield` expression allows it to write a generator function that returns a new value each time. A generator function is a kind of data stream from which you can pick values. The data stream can be infinite, therefore, we are in the center of lazy evaluation with C++.

A Generator Function#

The following program is as simple as possible. The function `getNumbers` returns all integers from `begin`

```
/*
// =====
// greedyGenerator.cpp

// #include <iostream>
// #include <vector>

// std::vector<int> getNumbers(int begin, int end, int inc = 1){

//     std::vector<int> numbers;
//     for (int i = begin; i < end; i += inc){
//         numbers.push_back(i);
//     }

//     return numbers;
// }

// int main(){

//     std::cout << std::endl;

//     const auto numbers = getNumbers(-10, 11);

//     for (auto n: numbers) std::cout << n << " ";

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

//     for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";

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

// }
// =====
/*
```

Of course, I am reinventing the wheel with `getNumbers` because that job could be done with `std::iota` since C++11.

Two observations about the program are important. On one hand, the `numbers` vector in line 8 always gets all values. This holds even if I'm only interested in the first 5 elements of a vector with 1000 elements. On the other hand, it's quite easy to transform the function `getNumbers` into a lazy generator.

```
/*
// =====
// lazyGenerator.cpp

// #include <iostream>
// #include <vector>

// generator<int> generatorForNumbers(int begin, int inc = 1){

//   for (int i = begin;; i += inc){
//     co_yield i;
//   }

// }

// int main(){

//   std::cout << std::endl;

//   const auto numbers= generatorForNumbers(-10);

//   for (int i= 1; i <= 20; ++i) std::cout << numbers << " ";

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

//   for (auto n: generatorForNumbers(0, 5)) std::cout << n << " ";

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

// }
// =====
/*
```

While the function `getNumbers` in the file `greedyGenerator.cpp` returns a `std::vector<int>`, the coroutine `generatorForNumbers` in `lazyGenerator.cpp` returns a generator. The generator `numbers` in line 18 or `generatorForNumbers(0, 5)` in line 24 returns a new number on request. The query is triggered by the range-based for-loop; to be more precise, the query of the coroutine returns the value `i` via `co_yield i` and immediately suspends its execution. If a new value is requested, the coroutine resumes its execution exactly at that place.

The expression `generatorForNumbers(0, 5)` in line 24 is a just-in-place usage of a generator.

I want to explicitly stress one point: the coroutine `generatorForNumbers` creates an infinite data stream because the for-loop in line 8 has no end condition. This is fine if I only ask for a finite number of values such as in line 20, but this will not hold for line 24 since there is no end condition. Therefore, the expression runs forever. Because coroutines are a totally new concept to C++, I want to provide a few details about them.

```
*/  
// ======  
// ======  
/*
```

oroutines: More Details

This lesson clarifies more details regarding coroutines, including use-cases, design goals, and underlying concepts.

We'll cover the following

Typical Use-Cases

Underlying Concepts

Design Goals

Becoming a Coroutine

co_return, co_yield, and co_await

Typical Use-Cases#

Coroutines are the natural way to write event-driven applications; e.g. simulations, games, servers, user interfaces, or even algorithms. Coroutines are typically used for cooperative multitasking. The key to cooperative multitasking is that each task takes as much time as it needs. This is in contrast to pre-emptive multitasking, for which we have a scheduler that decides how long each task gets the CPU.

That being said, there are different kinds of coroutines.

Underlying Concepts#

Coroutines in C++20 are asymmetric, first-class, and stackless.

The workflow of an asymmetric coroutine goes back to the caller. This will not hold for a symmetric coroutine. A symmetric coroutine can delegate its workflow to another coroutine.

First-class coroutines are similar to First-Class Functions since coroutines behave like data. This means that you can use them as an argument to return value from a function, or store them in a variable.

A stackless coroutine enables it to suspend and resume the top-level coroutine, but this coroutine can not invoke another coroutine.

Proposal N4402 describes the design goals of coroutines.

Design Goals#

Coroutines should

be highly scalable (to billions of concurrent coroutines).

have highly efficient resume and suspend operations comparable in cost to the overhead of a function.
seamlessly interact with existing facilities with no overhead.

have open ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics such as generators, goroutines, tasks and more.

usable in environments where exceptions are forbidden or not available.

There are four reasons for a function to become a coroutine.

Becoming a Coroutine#

A function will become a coroutine if it uses,

co_return, or
co_await, or
co_yield, or a
co_await expression in a range-based for-loop.

This explanation was from proposal N4628.

Finally, I will discuss the new keywords co_return, co_yield, and co_await.

co_return, co_yield, and co_await#

co_return: a coroutine uses co_return as its return statement.

co_yield: thanks to co_yield you can implement a generator. This means you can create a generator that will generate an infinite data stream from which you can successively query values. The return type of the generator generator<int> generatorForNumbers(int begin, int inc= 1) is generator<int>. generator<int> internally holds a special promise p such that a call co_yield i is equivalent to a call co_await p.yield_value(i). co_yield i can be called an arbitrary number of times. Immediately after the call, the execution of the coroutine will be suspended.

co_await: co_await eventually causes the execution of the coroutine to be suspended and resumed. The expression exp in co_await exp has to be a so-called awaitable expression, and exp has to implement a specific interface. This interface consists of the three functions: e.await_ready, e.await_suspend, and e.await_resume.

The typical use case for co_await is a server that waits for events.

```
Acceptor acceptor{443};  
while (true){  
    Socket socket= acceptor.accept();          // blocking  
    auto request= socket.read();                // blocking  
    auto response= handleRequest(request);  
    socket.write(response);                   // blocking  
}  
*/  
// ======  
// ======  
/*
```

The server is quite simple because it sequentially answers each request in the same thread. The server listens on port 443 (line 1), accepts its connections (line 3), reads the incoming data from the client (line 4), and writes its answer to the client (line 6). All calls in lines 3, 4, and 6 are blocking.

Thanks to co_await, the blocking calls can now be suspended and resumed.

```
*/  
// ======  
// Acceptor acceptor{443};  
// while (true){  
//     Socket socket= co_await acceptor.accept();  
//     auto request= co_await socket.read();  
//     auto response= handleRequest(request);
```

```
// co_await socket.write(response);
// }
// =====
/*
```

} Task Blocks

This lesson gives an overview of task-blocks, predicted to be introduced in C++20.

We'll cover the following

Fork & Join

define_task_block versus define_task_block_restore_thread

The Interface

The Scheduler

Task blocks use the well-known fork-join paradigm for the parallel execution of tasks.

Who invented it in C++? Both Microsoft with its Parallel Patterns Library (PPL) and Intel with its Threading Building Blocks (TBB) were involved in the proposal N4441. Additionally, Intel used its experience with its Cilk Plus library.

The name fork-join is quite easy to explain.

Fork & Join#

The simplest approach to explain the fork-join paradigm is through a graph.

widget

How does it work?

The creator invokes define_task_block or define_task_block_restore_thread. This call creates a task block that can create tasks or it can wait for their completion. The synchronization is at the end of the task block. The creation of a new task is the fork phase; the synchronization of the task block is the join phase of the workflow. Admittedly, that was the simplistic description. Let's have a look at a piece of code.

```
/*
// =====
// template <typename Func>
// int traverse(node& n, Func && f){
//   int left = 0, right = 0;
//   define_task_block(
//     [&](task_block& tb){
//       if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
//       if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
//     }
//   );
//   return f(n) + left + right;
// }
// =====
/*
```

traverse is a function template that invokes function f on each node of its tree. The keyword define_task_block defines the task block. The task block tb can start a new task in this block; that's exactly what happens at the left and right branches of the tree in lines 6 and 7. Line 9 is the end of the task block and, hence, the synchronization point.

HPX (High Performance ParalleX)

The above example is from the documentation for the HPX (High Performance ParalleX) framework, which is a general purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented many features of the upcoming C++20 standard.

You can define a task block by using either the function `define_task_block` or the function `define_task_block_restore_thread`.

`define_task_block` versus `define_task_block_restore_thread`#

The subtle difference is that `define_task_block_restore_thread` guarantees that the creator thread of the task block is the same thread that will run after the task block.

```
...
define_task_block([&](auto& tb){
    tb.run([&]{[]() { func(); });
    define_task_block_restore_thread([&](auto& tb){
        tb.run([&]{[]() { func2(); });
        define_task_block([&](auto& tb){
            tb.run([&]{ func3(); }
        });
        ...
    });
    ...
});
...
...
});
...
...
*/
// =====
// =====
/*
```

Task blocks ensure that the creator thread of the outermost task block (lines 2 - 14) is exactly the same thread that will run the statements after finishing the task block. This means that the thread that executes line 2 is the same thread that executes lines 15 and 16. However, this guarantee will not hold for nested task blocks; therefore, the creator thread of the task block in lines 6 - 8 will not automatically execute lines 9 and 10. If you need that guarantee, you should use the function `define_task_block_restore_thread` (line 4). Now it holds that the creator thread executing line 4 is the same thread executing lines 12 and 13.

The Interface#

A `task_block` has a very limited interface. You can not explicitly define it; you have to use either function `define_task_block` or `define_task_block_restore_thread`. The `task_block` `tb` is in the scope of the defined task block active and can, therefore, start new tasks (`tb.run`) or wait (`tb.wait`) until the task is done.

```
define_task_block([&](auto& tb){
    tb.run([&]{ process(x1, x2) });
    if (x2 == x3) tb.wait();
    process(x3, x4);
```

```
});  
*/  
// ======  
  
// ======  
/*  
What is the code snippet doing? In line 2 a new task is started. This task needs the data x1 and x2. Line 4 uses  
the data x3 and x4. If x2 == x3 is true, the variables have to be protected from shared access. This is the reason  
why the task block tb waits until the task in line 2 is done.
```

The Scheduler#

The scheduler manages which thread is running. This means that it is no longer the responsibility of the programmer to decide who executes the task. In this case, threads are just an implementation detail.

There are two strategies for executing the newly created task. The parent represents the creator thread and the child the new task.

Child stealing: The scheduler steals the task and executes it.

Parent stealing: The task block tb itself executes the task. Now the scheduler steals the parent. Proposal N4441 supports both strategies.

```
*/  
// ======  
// ======  
/*  
ABA
```

This lesson explains the ABA problem: An analogy for a value being changed in between two successive reads for that value giving the same result.

We'll cover the following

ABA

An Analogy

Non-critical ABA

A lock-free data structure

ABA in Action

Remedies

Tagged state reference

Garbage Collection

Hazard Pointers

RCU

Programming concurrent applications are inherently complicated. This still holds true if you use C++11 and C++14 features, and that is before I mention the memory model.

ABA#

ABA means you read a value twice and it returns the same value A each time. Therefore, you conclude that nothing changed in between, but you missed the fact that the value was updated to B somewhere in between.

Let me first use a simple scenario to introduce the problem.

An Analogy#

The scenario consists of you sitting in a car and waiting for the traffic light to become green. Green stands for B in our case, and red stands for A. What's happening?

You look at the traffic light and it is red (A).

Because you are bored, you begin to check the news on your smartphone and forget the time.

You look once more at the traffic light. Damn, it is still red (A).

Of course, the traffic light became green (B) between your two checks. Therefore, what seems to be one red phase was actually a full cycle.

What does this mean for threads (processes)? Here is a more formal explanation:

Thread 1 reads the variable var with value A.

Thread 1 is preempted and thread 2 runs.

Thread 2 changes the variable var from A to B to A.

Thread 1 continues to run and checks the value of variable var and gets A. Because of the value A, thread 1 proceeds.

Often that is not a problem and you can simply ignore it.

Non-critical ABA#

The functions `compare_exchange_strong` and `compare_exchange_weak` suffer the ABA problem that can be observed in the `fetch_mult` (line 7). Here, it is non-critical; `fetch_mult` multiplies a `std::atomic<T>&` shared by `mult`.

```
/*
// =====
/// // fetch_mult.cpp

// #include <atomic>
// #include <iostream>

// template <typename T>
// T fetch_mult(std::atomic<T>& shared, T mult){
//   T oldValue = shared.load();
//   while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
//   return oldValue;
// }

// int main(){
//   std::atomic<int> myInt{5};
//   std::cout << myInt << std::endl;
//   fetch_mult(myInt,5);
//   std::cout << myInt << std::endl;
// }
// =====
/*
```

The key observation is that there is a small time window between the reading of the old value `T oldValue = shared.load` in line 8 and the comparison with the new value in line 9. Therefore, another thread can kick in

and change the oldValue from oldValue to another value, then back to oldValue. The oldValue is the A, another value is the B in ABA.

Often it makes no difference if the first value is from the second read operation as long as the value is the original one. But in a lock-free concurrent data structure, ABA may have a great impact.

A lock-free data structure#

I will not present a lock-free data structure in detail here. I will use a lock-free stack that is implemented as a singly linked list. The stack supports only two operations.

Pop the top object and return a pointer to it.

Push the specified object to stack.

Let me describe the pop operation in pseudo-code to give you an idea of the ABA problem. The pop operation executes the following steps until the operation is successful.

Get the head node: head

Get the subsequent node: headNext

Make headNext to the new head if head is still the head of the stack

Here are the first two nodes of the stack:

Stack: TOP -> head -> headNext -> ...

```
/*
// =====
// =====
/*
```

Let's construct the ABA problem.

ABA in Action#

Let's start with the following stack:

Stack: TOP -> A -> B -> C

```
/*
// =====
// =====
/*
```

Thread 1 is active and wants to pop the head of the stack.

Thread 1 stores

```
head = A
headNext = B
/*
// =====
// =====
/*
```

Before thread 1 finishes the pop step, thread 2 kicks in.

Thread 2 pops A

```
Stack: TOP -> B -> C
*/
// =====
// =====
/*
Thread 2 pops B and deletes B
```

```
Stack: TOP -> C
*/
// =====
// =====
/*
*/
/*
// =====
// =====
/*
Thread 2 pushed A back
```

```
Stack: TOP -> A -> C
*/
// =====
// =====
/*
Thread 1 is rescheduled and checks if A == head. Because of A == head, headNext (B) becomes the new head - but B was already deleted; therefore, the program has undefined behavior.
```

There are a few remedies to the ABA problem.

Remedies#

The conceptional problem of ABA is quite easy to understand. A node such as B == headNext was deleted although another node A == head was referring to it. The solution to our problem is to get rid of the premature deletion of the node. Here are a few remedies.

Tagged state reference#

You can add a tag to each node indicating how often the node has been successfully modified. The result is that the compare and swap (CAS) method will eventually fail although the check returns true.

The next three techniques are based on the idea of deferred reclamation.

Garbage Collection#

Garbage collection guarantees that the variables will only be deleted if it is not needed anymore. This sounds promising but has a big drawback. Most garbage collectors are not lock-free, therefore, even if you have a lock-free data structure, the overall system won't be lock-free.

Hazard Pointers#

From Wikipedia: Hazard Pointers:

In a hazard-pointer system, each thread keeps a list of hazard pointers indicating which nodes the thread is currently accessing. (In many systems this “list” may be provably limited to only one or two elements.) Nodes on the hazard pointer list must not be modified or deallocated by any other thread. (...) When a thread wishes

to remove a node, it places it on a list of nodes “to be freed later”, but does not actually deallocate the node’s memory until no other thread’s hazard list contains the pointer. This manual garbage collection can be done by a dedicated garbage-collection thread (if the list “to be freed later” is shared by all the threads); alternatively, cleaning up the “to be freed” list can be done by each worker thread as part of an operation such as “pop”.

RCU#

RCU stands for Read Copy Update and is a synchronization technique for almost all read-only data structures. RCU was created by Paul McKenney and has been used in the Linux Kernel since 2002.

The idea is quite simple and follows the acronym. In order to modify data, you make a copy of the data and modify that copy. In contrast, all readers work with the original data. If there is no reader, you can safely replace the data structure with its copy. For more details about RCU, read the article [What is RCU, Fundamentally?](#) by Paul McKenney.

Two new proposals: As part of a concurrency toolkit, there are two proposals for upcoming C++ standards. The proposal P0233r0 for hazard pointers and the proposal P0461R0 for RCU.

```
*/  
// ======  
// ======  
/*
```

Blocking Issues

This lesson explains the challenges of blocking issues while using a condition variable in C++.

To make my point clear, you have to use a condition variable in combination with a predicate. If you don’t, your program may become a victim of a spurious wakeup or lost wakeup.

If you use a condition variable without a predicate, it may happen that the notifying thread sends its notification before the waiting thread is in the waiting state; Therefore, the waiting thread waits forever. This phenomenon is called a lost wakeup. Here is the program.

```
*/  
// ======  
// conditionVariableBlock.cpp  
  
// #include <iostream>  
// #include <condition_variable>  
// #include <mutex>  
// #include <thread>  
  
// std::mutex mutex_;  
// std::condition_variable condVar;  
  
// bool dataReady;  
  
// void waitingForWork(){  
  
//     std::cout << "Worker: Waiting for work." << std::endl;  
  
//     std::unique_lock<std::mutex> lck(mutex_);  
//     condVar.wait(lck);  
//     // do the work
```

```

// std::cout << "Work done." << std::endl;
// }

// void setDataReady(){

// std::cout << "Sender: Data is ready." << std::endl;
// condVar.notify_one();

// }

// int main(){

// std::cout << std::endl;

// std::thread t1(setDataReady);
// std::thread t2(waitingForWork);

// t1.join();
// t2.join();

// std::cout << std::endl;

// }
// =====
/*

```

By chance, the first invocation of the program works fine. The second invocation locks because the notify call (line 28) happens before the thread t2 (line 37) is waiting (line 19).

Of course, deadlocks and livelocks are side effects of race conditions. A deadlock depends in general on the interleaving of the threads and may sometimes occur or not. A livelock is similar to a deadlock; while a deadlock blocks, a livelock seems to make progress, with the emphasis on "seems." Think about a transaction in a transactional memory use case. Each time the transaction should be committed, a conflict happens and, therefore, a rollback takes place.

```

*/
// =====
// =====
/*

```

Breaking of Program Invariants

This lesson explains challenges related to the breaking of program invariants during the implementation of concurrency in C++.

Program invariants are invariants that should hold for the entire lifetime of your program.

Malicious race condition breaks an invariant of the program. The invariant of the following program is that the sum of all balances should be the same amount. In our case, this is 200 euros because each account starts with 100 euro (line 9). I neither want to create money by transferring it nor do I want to destroy it.

```

*/
// =====
// breakingInvariant.cpp

```

```
// #include <atomic>
// #include <functional>
// #include <iostream>
// #include <thread>

// struct Account{
//   std::atomic<int> balance{100};
// };

// void transferMoney(int amount, Account& from, Account& to){
//   using namespace std::chrono_literals;
//   if (from.balance >= amount){
//     from.balance -= amount;
//     std::this_thread::sleep_for(1ns);
//     to.balance += amount;
//   }
// }

// void printSum(Account& a1, Account& a2){
//   std::cout << (a1.balance + a2.balance) << std::endl;
// }

// int main(){
//   std::cout << std::endl;
//   Account acc1;
//   Account acc2;
//   std::cout << "Initial sum: ";
//   printSum(acc1, acc2);
//
//   std::thread thr1(transferMoney, 5, std::ref(acc1), std::ref(acc2));
//   std::thread thr2(transferMoney, 13, std::ref(acc2), std::ref(acc1));
//   std::cout << "Intermediate sum: ";
//   std::thread thr3(printSum, std::ref(acc1), std::ref(acc2));
//
//   thr1.join();
//   thr2.join();
//   thr3.join();
//
//   std::cout << "    acc1.balance: " << acc1.balance << std::endl;
//   std::cout << "    acc2.balance: " << acc2.balance << std::endl;
//
//   std::cout << "Final sum: ";
//   printSum(acc1, acc2);
//
//   std::cout << std::endl;
// }
```

```
/*
In the beginning, the sum of the accounts is 200 euros. Line 33 displays the sum by using the function
printSum in lines 21 - 23. Line 38 makes the invariant visible. Because there is a short sleep of 1ns in line 16,
the intermediate sum is 182 euro. In the end, all is fine; each account has the right balance (line 44 and line
45) and the sum is 200 euro (line 48).
```

```
*/
```

```
// =====
// =====
/*
```

Data Races

This lesson gives an overview of data race problems which might occur during the implementation of concurrency in C++.

A data race is a situation in which at least two threads access a shared variable at the same time. Within that, at least one thread tries to modify the variable.

If your program has a data race, it will have undefined behavior. This means all outcomes are possible and, therefore, reasoning about the program makes no sense anymore. Let me show you a program with a data race.

```
/*
// =====
// addMoney.cpp

// #include <functional>
// #include <iostream>
// #include <thread>
// #include <vector>

// struct Account{
//   int balance{100};
// };

// void addMoney(Account& to, int amount){
//   to.balance += amount;
// }

// int main(){

//   std::cout << std::endl;

//   Account account;

//   std::vector<std::thread> vecThreads(100);

//   for (auto& thr: vecThreads) thr = std::thread(addMoney, std::ref(account), 50);

//   for (auto& thr: vecThreads) thr.join();

//   std::cout << "account.balance: " << account.balance << std::endl;

//   std::cout << std::endl;
```

```
//}  
//=====  
/*  
100 threads are adding 50 euros (line 25) to the same account (line 20). They use the function addMoney. The  
key observation is that the writing to the account is done without synchronization, therefore we have a data  
race and the result is not valid. This is undefined behavior and the final balance (line 30) differs between 5000  
and 5100 euro.
```

```
*/  
//=====  
//=====  
/*
```

False Sharing

This lesson gives an overview of a false sharing problem which might occur during the implementation of concurrency in C++.

When a processor reads a variable such as an int from main memory, it will read more than the size of an int from memory; the processor will read an entire cache line (typically 64 bytes) from memory. False sharing occurs if two threads read different int's at the same time, a and b that are located on the same cache line. Although a and b are logically separated, they are physically connected. An expensive hardware synchronization on the cache line is necessary because a and b share the same one. The result is that you will get the right results, but the performance of your concurrent application decreases.

`std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size` with C++17

Both functions let you deal in a portable way with the cache line size.

`std::hardware_destructive_interference_size` returns the minimum offset between two objects to avoid false sharing and `std::hardware_constructive_interference_size` returns the maximum size of contiguous memory to promote true sharing.

```
*/  
//=====  
//=====  
/*
```

Lifetime Issues of Variables

This lesson explains challenges related to lifetime issues of variables in C++.

Creating a C++ example with lifetime related issues is quite easy. Let the created thread t run in the background (i.e. it was detached with a call to `t.detach()`) and let it be only half completed; the creator thread will not wait until its child is done. In this case, you have to be extremely careful not to use anything in the child thread that belongs to the creator thread.

```
*/  
//=====  
// lifetimelssues.cpp
```

```
// #include <iostream>  
// #include <string>  
// #include <thread>  
  
// int main(){  
  
// std::cout << "Begin:" << std::endl;
```

```

// std::string mess{"Child thread"};

// std::thread t([&mess]{ std::cout << mess << std::endl;});
// t.detach();

// std::cout << "End:" << std::endl;

// }
// =====
/*

```

This is too simple. The thread `t` is using `std::cout` and the variable `mess` - both of which belong to the main thread. The effect is that we don't see the output of the child thread in the second run. Only "Begin:" (line 9) and "End:" (line 16) are printed.

In order to see the output of child thread, creator thread will have to wait for it.

Let's see the solution:

```

*/
// =====
// lifetimelssuesSolution.cpp

// #include <iostream>
// #include <string>
// #include <thread>

// int main(){

// std::cout << "Begin:" << std::endl;

// std::string mess{"Child thread"};

// std::thread t([&mess]{ std::cout << mess << std::endl;});
// t.join();

// std::cout << "End:" << std::endl;

// }
// =====
/*

```

Moving Threads

This lesson gives an overview of problems and challenges related to moving threads in C++.

Moving threads make the lifetime issues of threads even harder.

A thread supports the move semantic but not the copy semantic, the reason being the copy constructor of `std::thread` is set to delete: `thread(const thread&) = delete;`. Imagine what will happen if you copy a thread while the thread is holding a lock.

Let's move a thread.

```
*/
```

```

// =====
// threadMoved.cpp

// #include <iostream>
// #include <thread>
// #include <utility>

// int main(){

// std::thread t([]{std::cout << std::this_thread::get_id();});
// std::thread t2([]{std::cout << std::this_thread::get_id();});

// t = std::move(t2);
// t.join();
// t2.join();

//}
// =====
/*

```

Both threads t and t2 should do their simple job: printing their IDs. In addition to that, thread t2 will be moved to t (line 12). At the end, the main thread takes care of its children and joins them. But wait, the result is very different from my expectations.

What is going wrong? We have two issues:

By moving the thread t2 (taking ownership), t gets a new callable unit and its destructor will be called. As a result, t's destructor calls std::terminate because it is still joinable.
 Thread t2 has no associated callable unit. The invocation of join on a thread without callable unit leads to the exception std::system_error.

Knowing this, fixing the errors is straightforward.

```

*/
// =====
// threadMovedFixed.cpp

// #include <iostream>
// #include <thread>
// #include <utility>

// int main(){

// std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});
// std::thread t2([]{std::cout << std::this_thread::get_id() << std::endl;});

// t.join();
// t = std::move(t2);
// t.join();

// std::cout << "\n";
// std::cout << std::boolalpha << "t2.joinable(): " << t2.joinable() << std::endl;

```

```
//}  
// ======  
/*  
Deadlocks
```

This lesson gives an overview of deadlocks which might occur during implementation of concurrency in C++.

A deadlock is a state in which two or more threads are blocked because each thread waits for the release of a resource before it releases its own resource.

There are two main reasons for deadlocks:

A mutex has not been unlocked.

You lock your mutexes in a different order.

For overcoming the second issue, techniques such as lock hierarchies are used in classical C++.

For the details about deadlocks and how to overcome them with modern C++, read the subsection issues of mutexes and locks.

Race Conditions

This lesson gives an overview of race condition problems that might occur during the implementation of concurrency in C++.

A race condition is a situation in which the result of an operation depends on the interleaving of certain individual operations.

Race conditions are quite difficult to spot; It depends on the interleaving of the threads whether they occur. That means the number of cores, the utilization of your system, or the optimization level of your executable may all be reasons why a race condition appears or does not.

Race conditions are not bad per se. It is the nature of threads that they will interleave in different ways, but this can often cause serious problems. In this case, I will call them malign race conditions. Typical effects of malign race conditions are data races, breaking of program invariants, blocking issues of threads, or lifetime issues of variables.

```
*/  
// ======  
// ======  
/*
```

General

This lesson provides a simple set of rules for writing well-defined and fast, concurrent programs in modern C++.

We'll cover the following

Code Reviews

Minimize Data Sharing of Mutable Data

Minimize Waiting

Prefer Immutable Data

Look for the Right Abstraction

Use Static Code Analysis Tools

Use Dynamic Enforcement Tools

Multithreading, parallelism, and concurrency, in particular, are quite new topics in C++; therefore, more and more best practices will be discovered in the coming years. Consider the rules in this chapter not as a complete list, but rather as a necessary starting point that will evolve over time. This holds particularly true for the parallel STL. At the time of writing this course (08/2017), the new C++17 standard - including the parallel algorithms - hasn't been published yet; therefore, it is too early to formulate best practices for it.

Let's start with a few very general best practices that will apply to atomics and threads.

Code Reviews#

Code reviews should be part of each professional software development process; this holds especially true when you deal with concurrency. Concurrency is inherently complicated and requires a lot of thoughtful analysis and experience.

To make the review most effective, send the code you want to discuss to the reviewers before the review. Explicitly state which invariants should apply to your code. The reviewers should have enough time to analyze the code before the official review starts.

Minimize Data Sharing of Mutable Data#

You should minimize data sharing of mutable data for two reasons: performance and safety. Safety is mainly about data races. Let me focus on performance in this paragraph. I will deal with correctness in the following best practices section.

You may have heard of Amdahl's law. It predicts the theoretical maximum speedup you can get using multiple processors. The law is quite simple: If p is the proportion of your code that can run concurrently, you will get a maximum speedup of $\frac{1}{1-p}$

$$\frac{1}{1-p}$$

. So, if 90% of your code can run concurrently, you will get at most a 10 times speedup: $\frac{1}{1-0.9} = \frac{1}{0.1} = 10$

$$\frac{1}{1-0.9}$$

1

=

$$1-0.9$$

1

=

$$0.1$$

1

$$=10$$

.

From the opposite perspective: if 10% of your code has to run sequentially because you use a lock, you will get at most a 10 times speedup. Of course, I assumed that you have access to infinite processing resources.

Minimize Waiting#

Waiting has at least two drawbacks. First, when a thread waits it cannot make any progress; therefore, your performance goes down. Even worse: if the waiting is busy, the underlying CPU will be fully utilized (In the memory model chapter, I compared the busy waiting of a spinlock with the non-busy waiting of a mutex). Second, the more waiting you have in your program in order to synchronize the threads, the more likely it will become that a bad interleaving of waiting periods causes a deadlock.

Prefer Immutable Data#

A data race is a situation in which at least two threads access a shared variable at the same time. In that case, at least one thread tries to modify the variable; the definition makes it quite obvious. A requirement for getting a data race is mutable data. If you have immutable data, no data race can happen. You only have to guarantee that the immutable data will be initialized in a thread-safe way.

Functional programming languages such as Haskell, having no mutable data, are very suitable for concurrent programming.

Look for the Right Abstraction#

There are various ways to initialize a Singleton in a multithreading environment. You can rely on the standard library using a lock_guard or std::call_once, rely on the core language using a static variable, or rely on atomics using acquire-release semantic. The acquire-release semantic is by far the most challenging one. It's a big challenge in various aspects: you have to implement it, maintain it, and explain it to your coworkers. In contrast to your effort, the well-known Meyers Singleton is a lot easier to implement and runs faster.

The story with the right abstractions goes on. Instead of implementing a parallel loop for summing up a container, use std::reduce. You can parametrise std::reduce with a binary callable and the parallel execution policy.

The more you go for the right abstraction, the less likely it will become that you shoot yourself in the foot.

Use Static Code Analysis Tools#

In the chapter on case studies, I introduced CppMem as an interactive tool for exploring the behavior of small code snippets using the C++ memory model. CppMem can help you in two aspects: First, you can verify the correctness of your code; Second, you get a deeper understanding of the memory model and, therefore, of the multithreading issues in general.

Use Dynamic Enforcement Tools#

ThreadSanitizer is a data races detector for C/C++; it's also part of Clang 3.2 and GCC 4.8. To use ThreadSanitizer, you have to compile and link your program using the flag -fsanitize=thread, or more generally,

```
g++ -std=c++11 dataRace.cpp -fsanitize=thread pthread -g -o dataRace
```

The following program has a data race.

```
/*
// =====
// dataRace.cpp
```

```
// #include <thread>

// int main(){
//   int globalVar{};

//   std::thread t1([&globalVar]{ ++globalVar; });
//   std::thread t2([&globalVar]{ ++globalVar; });

//   t1.join();
//   t2.join();

//}

// =====
/*
```

Memory Model

This lesson provides a simple set of rules for efficiently using C++ memory models.

We'll cover the following

Don't use Volatile for Synchronization

Don't Program Lock-Free

If you program Lock-Free, use well-established patterns

Don't build your own abstraction, use guarantees of the language

The foundation of multithreading is a well-defined memory model. Having a basic understanding of the memory helps a lot to get a deeper insight into the multithreading challenges.

Don't use Volatile for Synchronization#

In C++, in contrast to C# or Java, volatile has no multithreading semantic . In C# or Java, volatile declares an atomic such as std::atomic declares an atomic in C++, and it is typically used for objects which can change independently of the regular program flow. Due to this characteristic, no optimized storing in caches takes place.

Don't Program Lock-Free#

This advice sounds ridiculous after writing a course about concurrency and having an entire chapter dedicated to the memory model. However, the reason for this advice is quite simple. Lock-free programming is very error-prone and requires an expert level of knowledge. In particular, if you want to implement a lock-free data structure, be aware of the ABA problem.

If you program Lock-Free, use well-established patterns#

If you have identified a bottleneck that could benefit from a lock-free solution, apply established patterns.

Sharing an atomic boolean or an atomic counter is straightforward.

Use a thread-safe or even lock-free container to support consumer/producer scenario. If your container is thread-safe, you can put and get values from the container without worrying about synchronization; you simply shift the application challenges to the infrastructure.

Don't build your own abstraction, use guarantees of the language#

Thread-safe initialization of a shared variable can be done in various ways: you can rely on guarantees of the C++ runtime such as constant expressions, static variables with block scope, or use the function `std::call_once` in combination with the flag `std::once_flag`. We program in C++; therefore, you can build your own abstraction based on atomics using even the highly sophisticated acquire-release semantic. Don't do this in the first place, unless you have to do it – i.e. if you have identified a bottleneck by measuring the performance of a critical code path. Only make the change if you know that your handcrafted version will outperform the default guarantees of the language.

Multithreading: Threads

This lesson illustrates the best practices used to implement multithreaded applications in C++.

We'll cover the following

Threads

Use tasks instead of threads

Be extremely careful if you detach a thread

Consider using an atomic joining thread

Threads#

Threads are the basic building blocks for writing concurrent programs.

Use tasks instead of threads#

```
/*
// =====
// asyncVersusThread.cpp

// #include <future>
// #include <thread>
// #include <iostream>

// int main(){

// std::cout << std::endl;

// int res;
// std::thread t([&]{ res = 2000 + 11; });
// t.join();
// std::cout << "res: " << res << std::endl;

// auto fut= std::async([]{ return 2000 + 11; });
// std::cout << "fut.get(): " << fut.get() << std::endl;

// std::cout << std::endl;

//}

// =====
/*
```

Based on the program, there are a lot of reasons for preferring tasks over threads. These are the main reasons:

you can use a safe communication channel for returning the result of the communication. If you use a shared variable, you have to synchronize the access to it.

you can quite easily return values, notifications, and exceptions to the caller.

With extended futures we get the possibility to compose futures and build highly sophisticated workflows.

These workflows will be based on the continuation then, and the combinations when_any and when_all.

Be extremely careful if you detach a thread#

The following code snippet requires our full attention.

```
/*
// =====
// #include <iostream>
// #include <thread>

// int main(){

// std::string s{"C++11"};
// std::thread t([&s]{ std::cout << s << std::endl; });
// t.detach();
//}
// =====
/*
```

Because thread t is detached from the lifetime of its creator, two race conditions can cause undefined behavior.

Thread t may outlive the lifetime of its creator. The consequence is that t refers to a non-existing std::string. The program shuts down before thread t can do its work because the lifetime of the output stream std::cout is bound to the lifetime of the main thread

Consider using an atomic joining thread#

A thread t with a callable unit is called joinable if neither a t.join() nor a t.detach() call happened. The destructor of a joinable thread throws the std::terminate exception. In order not to forget the t.join(), you can create your own wrapper around std::thread. This wrapper checks in the constructor, if the given thread is still joinable, and joins the given thread in the destructor.

You don't have to build this wrapper on your own; Use the scoped_thread from Anthony Williams or the gsl::joining_thread from the guideline support library.

Now let's practice more about data sharing and multithreading.

```
/*
// =====
// =====
/*
```

Multithreading: Shared Data

This lesson gives an overall guide for best practices used to manage shared data in multithreaded applications in C++.

We'll cover the following

Data Sharing

Pass data per default by copy

Minimize the time holding a lock

Put a mutex into a lock

Use std::lock or std::scoped_lock for locking more mutexes atomically

Never call unknown code while holding a lock

Data Sharing#

With data sharing, the challenges in multithreading programming start.

Pass data per default by copy#

```
/*
// =====
// #include <iostream>
// #include <thread>
// #include <string>

// int main(){

// std::string s{"C++11"};

// std::thread t1([s]{std::cout << s << std::endl; }); // do something with s
// t1.join();

// std::thread t2([&s]{ std::cout << s << std::endl; }); // do something with s
// t2.join();

// // do something with s

// s.replace(s.begin(), s.end(), 'C', 'Z');
// }

// =====
/*
#include <iostream>
#include <thread>
#include <string>

int main(){

std::string s{"C++11"};

std::thread t1([s]{std::cout << s << std::endl; }); // do something with s
t1.join();

std::thread t2([&s]{ std::cout << s << std::endl; }); // do something with s
t2.join();

// do something with s

s.replace(s.begin(), s.end(), 'C', 'Z');
}
*/
```

```

// #include <iostream>
// #include <condition_variable>
// #include <mutex>
// #include <thread>

// std::mutex mutex_;
// std::condition_variable condVar;

// bool dataReady{false};

// void setDataReadyBad(){
//     std::lock_guard<std::mutex> lck(mutex_);
//     //Work on Shared Variable
//     dataReady = true;
//     std::cout << "Data prepared" << std::endl;
//     condVar.notify_one();
// }           // unlock the mutex

// void setDataReadyGood(){
//     //Work on Shared Variable
// {
//     std::lock_guard<std::mutex> lck(mutex_);
//     dataReady = true;
// }           // unlock the mutex
//     std::cout << "Data prepared" << std::endl;
//     condVar.notify_one();
// }

// int main(){

// std::cout << std::endl;

// std::thread t1(setDataReadyBad);
// std::thread t2(setDataReadyGood);

// t1.join();
// t2.join();

// std::cout << std::endl;

// }
// =====
/*
Multithreading: Shared Data
This lesson gives an overall guide for best practices used to manage shared data in multithreaded applications in C++.

```

We'll cover the following

Data Sharing

Pass data per default by copy

Minimize the time holding a lock

Put a mutex into a lock

Use std::lock or std::scoped_lock for locking more mutexes atomically

Never call unknown code while holding a lock

Data Sharing#

With data sharing, the challenges in multithreading programming start.

Pass data per default by copy#

```
/*
// =====
// #include <iostream>
// #include <thread>
// #include <string>

// int main()
//{
//    std::string s("C++11");

//    std::thread t1([s]
//    {
//        std::cout << s << std::endl; // do something with s
//    } t1.join();

//    std::thread t2(&s)
//    {
//        std::cout << s << std::endl; // do something with s
//    } t2.join();

//    do something with s

//    s.replace(s.begin(), s.end(), 'C', 'Z');
//}

// =====
/*
```

If you pass data such as the std::string s to a thread t1 by copy, the creator thread and the created thread t1 will use independent data; this is in contrast to the thread t2. It gets its std::string s by reference. This means you have to synchronize the access to s in the creator thread and the created thread t2 preventively. This is error-prone and expensive.

Minimize the time holding a lock#

If you hold a lock, only one thread can enter the critical section and make progress.

```
/*
// =====
// #include <iostream>
// #include <condition_variable>
// #include <mutex>
// #include <thread>

// std::mutex mutex_;
// std::condition_variable condVar;
```

```

// bool dataReady{false};

// void setDataReadyBad(){
//   std::lock_guard<std::mutex> lck(mutex_);
//   //Work on Shared Variable
//   dataReady = true;
//   std::cout << "Data prepared" << std::endl;
//   condVar.notify_one();
// }           // unlock the mutex

// void setDataReadyGood(){
//   //Work on Shared Variable
//   {
//     std::lock_guard<std::mutex> lck(mutex_);
//     dataReady = true;
//   }           // unlock the mutex
//   std::cout << "Data prepared" << std::endl;
//   condVar.notify_one();
// }

// int main(){

// std::cout << std::endl;

// std::thread t1(setDataReadyBad);
// std::thread t2(setDataReadyGood);

// t1.join();
// t2.join();

// std::cout << std::endl;

// }
// =====
/*

```

The functions `setDataReadyBad` and `setDataReadyGood` are the notification components of a condition variable. The variable `dataReady` is necessary to protect against spurious wakeups and lost wakeups. Because `dataReady` is a non-atomic variable, it has to be synchronized using the lock `lck`. To make the lifetime of the lock as short as possible, use an artificial scope (`{ ... }`) such as in the function `setDataReadyGood`.

```

*/
// =====
// =====
/*

```

Put a mutex into a lock#

You should not use a mutex without a lock.

```

std::mutex m;
m.lock();
// critical section

```

```
m.unlock();
```

Something unexpected may happen in the critical section or you could simply forget to unlock the mutex; anyway, the result is the same. If you don't unlock a mutex, another thread requiring the mutex will be blocked and you will end with a deadlock.

Thanks to locks that automatically take care of the underlying mutex, your risks of getting a deadlock are considerably reduced. According to the RAIID idiom, a lock automatically binds its mutex in the constructor and releases it in the destructor.

```
{
    std::mutex m,
    std::lock_guard<std::mutex> lockGuard(m);
    // critical section
}      // unlock the mutex
```

The artificial scope (`{ ... }`) ensures that the lifetime of the lock automatically ends; therefore, the underlying mutex will be unlocked.

Use `std::lock` or `std::scoped_lock` for locking more mutexes atomically#

If a thread needs more than one mutex, you have to be extremely careful that you lock the mutex always in the same sequence. If not, you get a data race and a bad interleaving of threads may cause a deadlock.

```
void deadLock(CriticalData& a, CriticalData& b){
    std::lock_guard<std::mutex> guard1(a.mut);
    // some time passes
    std::lock_guard<std::mutex> guard2(b.mut);
    // do something with a and b
}
```

...

```
std::thread t1([&]{deadLock(c1,c2);});
std::thread t2([&]{deadLock(c2,c1);});
```

...

Thread t1 and t2 need two CriticalData resources to perform their job, as CriticalData has its own mutex mut to synchronize the access. Unfortunately, both invoke the function deadlock with the arguments c1 and c2 in a different sequence. Now we have a data race. If thread t1 can lock the first mutex (a.mut) but not the second one (b.mut) because thread t2 locks the second one in the meantime, then we will get a deadlock.

Thanks to `std::unique_lock`, you can defer the locking of its mutex; it's done by the function `std::lock`, which can lock an arbitrary number of mutexes in an atomic way.

```
void deadLock(CriticalData& a, CriticalData& b){
    unique_lock<mutex> guard1(a.mut,defer_lock);
    // some time passes
    unique_lock<mutex> guard2(b.mut,defer_lock);
    std::lock(guard1,guard2);
```

```
// do something with a and b  
}
```

...

```
std::thread t1([&]{deadLock(c1,c2);});  
std::thread t2([&]{deadLock(c2,c1);});
```

...

C++17 has a new lock std::scoped_lock, which can get an arbitrary number of mutexes and locks them atomically. Now, the workflow becomes even simpler.

```
void deadLock(CriticalData& a, CriticalData& b){  
    std::scoped_lock(a.mut, b.mut);  
    // do something with a and b  
}
```

...

```
std::thread t1([&]{deadLock(c1,c2);});  
std::thread t2([&]{deadLock(c2,c1);});
```

...

ever call unknown code while holding a lock#

Calling an unknownFunction while holding a mutex is a recipe for undefined behavior.

```
{  
    std::mutex m,  
    std::lock_guard<std::mutex> lockGuard(m);  
    sharedVariable= unknownFunction();  
}
```

I can only speculate about the unknownFunction. If unknownFunction

tries to lock the mutex m, that will be undefined behavior. Most of the times, you will get a deadlock.
starts a new thread that tries to lock the mutex m, you will get a deadlock.

will not directly or indirectly try to lock the mutex m, all seems to be fine. "Seems" because your coworker can modify the function or the function is dynamically linked and you get a different version. All bets are open what may happen.

```
*/  
// ======  
// ======  
/*
```

I can only speculate about the unknownFunction. If unknownFunction

tries to lock the mutex m, that will be undefined behavior. Most of the times, you will get a deadlock.
starts a new thread that tries to lock the mutex m, you will get a deadlock.

will not directly or indirectly try to lock the mutex m, all seems to be fine. “Seems” because your coworker can modify the function or the function is dynamically linked and you get a different version. All bets are open what may happen.

```
/*
// =====
// conditionVariableLostWakeup.cpp

// #include <condition_variable>
// #include <mutex>
// #include <thread>

// std::mutex mutex_;
// std::condition_variable condVar;

// void waitingForWork(){
//     std::unique_lock<std::mutex> lck(mutex_);
//     condVar.wait(lck);
//     // do the work
// }

// void setDataReady(){
//     condVar.notify_one();
// }

// int main(){
//     std::thread t1(setDataReady);
//     std::thread t2(waitingForWork);

//     t1.join();
//     t2.join();

// }
```

```
/*
// =====
/*
If the thread t1 runs before the thread t2, you will get a deadlock. t1 will send its notification before t2 can accept it, and the notification is lost. This will happen very often because thread t1 starts before thread t2, and thread t1 has less work to perform.
```

Adding a bool variable dataReady to the workflow will solve this issue. dataReady will also protect against a spurious wakeup, as the waiting thread first checks if the notification was from the right thread.

```
/*
// =====
// conditionVariableLostWakeupSolved.cpp

// #include <condition_variable>
// #include <mutex>
// #include <thread>

// std::mutex mutex_;
```

```

// std::condition_variable condVar;

// bool dataReady{false};

// void waitingForWork(){
//   std::unique_lock<std::mutex> lck(mutex_);
//   condVar.wait(lck, []{ return dataReady; });
//   // do the work
// }

// void setDataReady(){
// {
//   std::lock_guard<std::mutex> lck(mutex_);
//   dataReady = true;
// }
// condVar.notify_one();
// }

// int main(){

// std::thread t1(waitingForWork);
// std::thread t2(setDataReady);

// t1.join();
// t2.join();

// }
// =====
/*
Use Promises and Futures instead of Condition Variables#
For one-time notifications, promises and futures are the better choice. The workflow of the previous program
conditioVarialbleLostWakeupSolved.cpp can directly be implemented with a promise and a future.
*/
// =====
// notificationWithPromiseAndFuture.cpp

// #include <future>
// #include <utility>

// void waitingForWork(std::future<void>&& fut){
//   fut.wait();
//   // do the work
// }

// void setDataReady(std::promise<void>&& prom){
//   prom.set_value();
// }

// int main(){

// std::promise<void> sendReady;

```

```

// auto fut = sendReady.get_future();

// std::thread t1(waitingForWork, std::move(fut));
// std::thread t2(setDataReady, std::move(sendReady));

// t1.join();
// t2.join();

//}

//=====
/*
// notificationWithPromiseAndFuture.cpp

#include <future>
#include <utility>

void waitingForWork(std::future<void>&& fut){
    fut.wait();
    // do the work
}

void setDataReady(std::promise<void>&& prom){
    prom.set_value();
}

int main(){

    std::promise<void> sendReady;
    auto fut = sendReady.get_future();

    std::thread t1(waitingForWork, std::move(fut));
    std::thread t2(setDataReady, std::move(sendReady));

    t1.join();
    t2.join();

}
*/
// =====
// =====
/*

```

The workflow is reduced to its bare minimum. The promise `prom.set_value()` sends the notification the future `fut.wait()` is waiting for. The program needs no mutexes and locks because there is no critical section. Because no lost wakeup or spurious wakeup can happen, a predicate is also not necessary.

If your workflow requires that you use a condition variable many times, then a promise and a future pair is no alternative.

Promises and Futures#

`std::async` can often be used as an easy-to-use replacement for threads or condition variables.

If possible, go for std::async#
If possible, you should go for std::async to execute an asynchronous task.

```
auto fut = std::async([]{ return 2000 + 11; });  
// some time passes  
std::cout << "fut.get(): " << fut.get() << std::endl;
```

Let's see this in action:

```
*/  
// ======  
// #include <future>  
// #include <thread>  
// #include <iostream>  
// #include <chrono>  
  
// int main(){  
  
//   std::cout << std::endl;  
  
//   auto fut= std::async([]{ return 2000 + 11; });  
//   std::this_thread::sleep_for (std::chrono::seconds(2)); //Work for 2 seconds  
//   std::cout << "fut.get(): " << fut.get() << std::endl;  
  
//   std::cout << std::endl;  
  
// }  
// ======  
/*
```

By invoking `auto fut = std::async([]{ return 2000 + 11; })`, you say to the C++ runtime: “Run my job”. I don’t care if it will be executed immediately, if it will run on the same thread, if it will run on a thread pool, or if it will run on a GPU; You are only interested in picking up the result in the future: `fut.get()`.

From a conceptional view, a thread is just an implementation detail for running your job. You only specify what should be done and not how it should be done.

```
*/  
// ======  
// ======  
/*
```

The Interplay of Time Point, Time Duration, and Clock

This lesson highlights the interplay of time point, time duration, and clock.

This course would not be complete without writing a chapter about the time library. The time library consists of three parts: time point, time duration, and clock; they all depend on each other.

Time point:

The time point is given by its starting point - the so-called epoch - and the time that has elapsed since the epoch (expressed as a time duration).

Time duration:

The time duration is the difference between two time points. It is measured in the number of time ticks.

Clock:

The clock consists of a starting point and a timer tick. This information enables you to calculate the current time.

You can compare time points. When you add a time duration to a time point, you get a new time point. The time tick is the accuracy of the clock in which you measure the time duration. The birth of Jesus - in my culture - is the starting time point, and a year is a typical time tick.

I will illustrate the three concepts using the lifetime of Dennis Ritchie - the creator of C who died in 2011. For the sake of simplicity, I'm only interested in the years. Here is the lifetime:

widget

The birth of Jesus is our epoch; the time points 1941 and 2011 are defined by the epoch and the time duration. (Of course, the epoch is also a time point.) When I subtract 1941 from 2011, I get the time duration. This time duration is measured to an accuracy of one year in our example. Dennis Ritchie died at 70.

Let's dive deeper into the components of the time library.

```
*/
// =====
// =====
/*
Time Point
```

This lesson gives a brief introduction to time point and its usage in C++ with the help of interactive examples.

The time point `std::chrono::time_point` is defined by the starting point (epoch) and the additional time duration. The class template consists of two components: clock and time duration. By default, the time duration is derived from the clock.

```
template<
    class Clock,
    class Duration=typename Clock::duration
>
class time_point;
*/
// =====
// =====
/*
```

The following four special time points depend on the clock:

epoch: the starting point of the clock

now: the current time

min: the minimum time point that the clock can have

max: the maximum time point that the clock can have

The accuracy of the minimum and maximum time point depends on the clock used:
std::system::system_clock, std::chrono::steady_clock or std::chrono::high_resolution_clock.

C++ gives no guarantee about the accuracy, the starting point, or the valid time range of a clock. The starting point of std::chrono::system_clock is typically 1st January 1970, the so-called

```
/*
// =====
// =====
/*
*/
/*From Time Point to Calendar Time
```

This lesson gives a brief introduction to calendar time and its usage in C++ with the help of interactive examples.

We'll cover the following

Cross the valid Time Range

Thanks to std::chrono::system_clock::to_time_t, you can convert a time point that internally uses std::chrono::system_clock to an object of type std::time_t. Further conversion of the std::time_t object with the function std::gmtime gives you the calendar time, expressed in Coordinated Universal Time (UTC). In the end, this calendar time can be used as the input for the function std::asctime to get a textual representation of the calendar time.

```
/*
// =====
// timepoint.cpp

// #include <chrono>
// #include <ctime>
// #include <iostream>
// #include <string>

// int main(){

//     std::cout << std::endl;

//     std::chrono::time_point<std::chrono::system_clock> sysTimePoint;
//     std::time_t tp= std::chrono::system_clock::to_time_t(sysTimePoint);
//     std::string sTp= std::asctime(std::gmtime(&tp));
//     std::cout << "Epoch: " << sTp << std::endl;

//     tp= std::chrono::system_clock::to_time_t(sysTimePoint.min());
//     sTp= std::asctime(std::gmtime(&tp));
//     std::cout << "Time min: " << sTp << std::endl;

//     tp= std::chrono::system_clock::to_time_t(sysTimePoint.max());
//     sTp= std::asctime(std::gmtime(&tp));
//     std::cout << "Time max: " << sTp << std::endl;

//     sysTimePoint= std::chrono::system_clock::now();
//     tp= std::chrono::system_clock::to_time_t(sysTimePoint);
//     sTp= std::asctime(std::gmtime(&tp));}
```

```
// std::cout << "Time now: " << sTp << std::endl;
```

```
// }
```

```
// =====
```

```
/*
```

The output of the program shows the valid range of `std::chrono::system_clock`. On my Linux PC, `std::chrono::system_clock` has the UNIX-epoch as the starting point, and can have time points between the years 1677 and 2262. You can add time durations to time points to get new time points; However, note that adding time durations beyond the valid time range is undefined behavior.

Cross the valid Time Range#

The following example uses the current time and adds or subtracts 1000 years. For the sake of simplicity, I ignore leap years and assume that a year has 365 days.

```
*/
```

```
// =====
```

```
// timepointAddition.cpp
```

```
// #include <chrono>
```

```
// #include <ctime>
```

```
// #include <iostream>
```

```
// #include <string>
```

```
// using namespace std::chrono;
```

```
// using namespace std;
```

```
// string timePointAsString(const time_point<system_clock>& timePoint){
```

```
// time_t tp= system_clock::to_time_t(timePoint);
```

```
// return asctime(gmtime(&tp));
```

```
// }
```

```
// int main(){
```

```
// cout << endl;
```

```
// time_point<system_clock> nowTimePoint= system_clock::now();
```

```
// cout << "Now: " << timePointAsString(nowTimePoint) << endl;
```

```
// const auto thousandYears= hours(24*365*1000);
```

```
// time_point<system_clock> historyTimePoint= nowTimePoint - thousandYears;
```

```
// cout << "Now - 1000 years: " << timePointAsString(historyTimePoint) << endl;
```

```
// time_point<system_clock> futureTimePoint= nowTimePoint + thousandYears;
```

```
// cout << "Now + 1000 years: " << timePointAsString(futureTimePoint) << endl;
```

```
// }
```

```
// =====
```

```
/*
```

For readability, I introduced the namespace `std::chrono`. The output of the program shows that an overflow of the time points in lines 25 and 28 causes incorrect results. Subtracting 1000 years from the current time point gives a time point in the future; adding 1000 years to the current time point gives a time point in the past.

The difference between two time points is a time duration. Time durations support the basic arithmetic and can be displayed in different time ticks.

```
*/
// =====
// =====
/*
```

Time Duration

This lesson gives a brief introduction to a time duration class template and explains it with the help of interactive examples.

Time duration `std::chrono::duration` is a class template that consists of the type of the tick `Rep` and the length of a tick `Period`.

```
template<
    class Rep,
    class Period = std::ratio<1>
> class duration;
```

The tick length is `std::ratio<1>` by default; `std::ratio<1>` stands for a second and can also be written as `std::ratio<1,1>`. The rest is quite easy. `std::ratio<60>` is a minute and `std::ratio<1,1000>` a millisecond. When the type of `Rep` is a floating-point number, you can use it to hold fractions of time ticks.

C++11 predefines the most important time durations:

```
typedef duration<signed int, nano> nanoseconds;
typedef duration<signed int, micro> microseconds;
typedef duration<signed int, milli> milliseconds;
typedef duration<signed int> seconds;
typedef duration<signed int, ratio< 60>> minutes;
typedef duration<signed int, ratio<3600>> hours;
```

How much time has passed since the UNIX epoch (1.1.1970)? Thanks to type aliases for the different time durations, I can answer the question quite easily. In the following example, I ignore leap years and assume that a year has 365 days.

```
/*
// =====

// timeSinceEpoch.cpp

// #include <chrono>
// #include <iostream>

// using namespace std;

// int main(){

//     cout << fixed << endl;

//     cout << "Time since 1.1.1970:\n" << endl;
```

```

// const auto timeNow= chrono::system_clock::now();
// const auto duration= timeNow.time_since_epoch();
// cout << duration.count() << " nanoseconds " << endl;

// typedef chrono::duration<long double, ratio<1, 1000000>> MyMicroSecondTick;
// MyMicroSecondTick micro(duration);
// cout << micro.count() << " microseconds" << endl;

// typedef chrono::duration<long double, ratio<1, 1000>> MyMilliSecondTick;
// MyMilliSecondTick milli(duration);
// cout << milli.count() << " milliseconds" << endl;

// typedef chrono::duration<long double> MySecondTick;
// MySecondTick sec(duration);
// cout << sec.count() << " seconds " << endl;

// typedef chrono::duration<double, ratio<60>> MyMinuteTick;
// MyMinuteTick myMinute(duration);
// cout << myMinute.count() << " minutes" << endl;

// typedef chrono::duration<double, ratio<60*60>> MyHourTick;
// MyHourTick myHour(duration);
// cout << myHour.count() << " hours" << endl;

// typedef chrono::duration<double, ratio<60*60*24*365>> MyYearTick;
// MyYearTick myYear(duration);
// cout << myYear.count() << " years" << endl;

// cout << endl;

// */
// =====
/*

```

The typical time durations are microsecond (line 18), millisecond (line 22), second (line 26), minute (line 30), hour (line 34), and year (line 38). Also, I define the German school hour (45 min) in line 42.

As the next lesson illustrates, it's quite convenient to calculate with time durations.

```

*/
// =====
// =====
/*

```

Time Duration Calculations

This lesson will explain which time duration operations are supported in C++.

The time durations support basic arithmetic operations, meaning you can multiply or divide a time duration by a number. Of course, you can compare time durations. I explicitly want to emphasize that all these calculations and comparisons respect the units.

With the C++14 standard, it gets even better; the C++14 standard supports the typical time literals.

Type Suffix Example

std::chrono::hours	h	5h
std::chrono::minutes	min	5min
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns

How much time does my son Marius (17 years old) spend in a typical school day? I will answer the question in the following example and show the result in various time durations formats.

```
/*
// =====
// schoolDay.cpp

// #include <iostream>
// #include <chrono>

// using namespace std::literals::chrono_literals;
// using namespace std::chrono;
// using namespace std;

// int main(){

// cout << endl;

// constexpr auto schoolHour= 45min;

// constexpr auto shortBreak= 300s;
// constexpr auto longBreak= 0.25h;

// constexpr auto schoolWay= 15min;
// constexpr auto homework= 2h;

// constexpr auto schoolDaySec= 2*schoolWay + 6 * schoolHour + 4 * shortBreak +
// longBreak + homework;

// cout << "School day in seconds: " << schoolDaySec.count() << endl;

// constexpr duration<double, ratio<3600>> schoolDayHour = schoolDaySec;
// constexpr duration<double, ratio<60>> schoolDayMin = schoolDaySec;
// constexpr duration<double, ratio<1,1000>> schoolDayMilli= schoolDaySec;

// cout << "School day in hours: " << schoolDayHour.count() << endl;
// cout << "School day in minutes: " << schoolDayMin.count() << endl;
// cout << "School day in milliseconds: " << schoolDayMilli.count() << endl;
```

```
// cout << endl;  
  
// }  
// ======  
/*  
I have time durations for a German school hour (line 14), for a short break (line 16), for a long break (line 17),  
for Marius's way to school (line 19), and his homework (line 20). The result of the calculation  
schoolDaysInSeconds (line 22) is available at compile time.
```

Evaluation at Compile Time

The time literals (lines 14 - 20), the schoolDaySec in line 22, and the various durations (lines 27 - 29) are all constant expressions (`constexpr`). Therefore, all values will be evaluated at compile time; just the output is performed at runtime.

The accuracy of the time tick is dependent on the clock used. In C++ we have the clocks `std::chrono::system_clock`, `std::chrono::steady_clock`, and `std::chrono::high_resolution_clock`.

```
*/  
// ======  
// ======  
/*  
Clocks
```

This lesson gives a brief introduction to clocks and their usage in C++ with the help of interactive examples.

We'll cover the following

Accuracy and Steadiness

Epoch

The fact that there are three different types of clocks begs the question: What are the differences?

`std::chrono::system_clock`: is the system-wide real time clock (wall-clock). The clock has the auxiliary functions `to_time_t` and `from_time_t` to convert time points into calendar time

`std::chrono::steady_clock`: is the only clock to provide the guarantee that you can not adjust it. Therefore, `std::chrono::steady_clock` is the preferred clock to wait for a time duration or until a time point

`std::chrono::high_resolution_clock`: is the clock with the highest accuracy, but it can simply be an alias for the clocks `std::chrono::system_clock` or `std::chrono::steady_clock`

No guarantees about accuracy, starting point, and valid time range

The C++ standard provides no guarantee about the accuracy, the starting point, or the valid time range of the clocks. Typically, the starting point of `std::chrono::system_clock` is the 1.1.1970, the so called UNIX-epoch, while for `std::chrono::steady_clock` it is typically the boot time of your PC.

Accuracy and Steadiness#

It is quite interesting to know which clocks are steady and what accuracy they provide. Steady means that the time points can not decrease. You can get the answers directly from the clocks.

```
*/
```

```

// =====
// clockProperties.cpp

// #include <chrono>
// #include <iomanip>
// #include <iostream>

// using namespace std::chrono;
// using namespace std;

// template <typename T>
// void printRatio(){
//   cout << " precision: " << T::num << "/" << T::den << " second " << endl;
//   typedef typename ratio_multiply<T,kilo>::type MillSec;
//   typedef typename ratio_multiply<T,mega>::type MicroSec;
//   cout << fixed;
//   cout << "      " << static_cast<double>(MillSec::num)/MillSec::den
//   << " milliseconds " << endl;
//   cout << "      " << static_cast<double>(MicroSec::num)/MicroSec::den
//   << " microseconds " << endl;
// }

// int main(){

//   cout << boolalpha << endl;

//   cout << "std::chrono::system_clock: " << endl;
//   cout << " is steady: " << system_clock::is_steady << endl;
//   printRatio<chrono::system_clock::period>();

//   cout << endl;

//   cout << "std::chrono::steady_clock: " << endl;
//   cout << " is steady: " << chrono::steady_clock::is_steady << endl;
//   printRatio<chrono::steady_clock::period>();

//   cout << endl;

//   cout << "std::chrono::high_resolution_clock: " << endl;
//   cout << " is steady: " << chrono::high_resolution_clock::is_steady
//   << endl;
//   printRatio<chrono::high_resolution_clock::period>();

//   cout << endl;

// }
// =====
/*

```

I show in lines 27, 33, and 39 whether or not each clock is steady. The function `printRatio` (lines 10 - 20) is more challenging to read. First, I display the accuracy of the clocks as a fraction with the unit in seconds. Additionally, I use the function template `std::ratio_multiply` and the constants `std::kilo` and `std::mega` to

adjust the units to milliseconds and microseconds displayed as floating-point numbers. You can get the details of the calculation at [compile time at cppreference.com](http://cppreference.com).

The output on Linux differs from that on Windows. `std::chrono::system_clock` is far more accurate on Linux; `std::chrono::high_resolution_clock` is steady on Windows. Although the C++ standard doesn't specify the epoch of the clock, you can calculate it.

Epoch#

Thanks to the auxiliary function `time_since_epoch`, each clock returns how much time has passed since the epoch.

```
/*
// =====
// now.cpp

// #include <chrono>
// #include <iomanip>
// #include <iostream>

// using namespace std::chrono;

// template <typename T>
// void durationSinceEpoch(const T dur){
//   std::cout << "  Counts since epoch: " << dur.count() << std::endl;
//   typedef duration<double, std::ratio<60>> MyMinuteTick;
//   const MyMinuteTick myMinute(dur);
//   std::cout << std::fixed;
//   std::cout << "  Minutes since epoch: " << myMinute.count() << std::endl;
//   typedef duration<double, std::ratio<60*60*24*365>> MyYearTick;
//   const MyYearTick myYear(dur);
//   std::cout << "  Years since epoch: " << myYear.count() << std::endl;
// }

// int main(){
//   std::cout << std::endl;
//   system_clock::time_point timeNowSysClock = system_clock::now();
//   system_clock::duration timeDurSysClock= timeNowSysClock.time_since_epoch();
//   std::cout << "system_clock: " << std::endl;
//   durationSinceEpoch(timeDurSysClock);

//   std::cout << std::endl;
//   const auto timeNowStClock = steady_clock::now();
//   const auto timeDurStClock= timeNowStClock.time_since_epoch();
//   std::cout << "steady_clock: " << std::endl;
//   durationSinceEpoch(timeDurStClock);

//   std::cout << std::endl;
//   const auto timeNowHiRes = high_resolution_clock::now();
```

```

// const auto timeDurHiResClock= timeNowHiRes.time_since_epoch();
// std::cout << "high_resolution_clock: " << std::endl;
// durationSinceEpoch(timeDurHiResClock);

// std::cout << std::endl;

// }
// =====
/*

```

The variables `timeDurSysClock` (line 26), `timeNowStClock` (line 32), and `timeNowHiResClock` (line 39) contain the amount of time that has passed since the starting point of the corresponding clock. Without automatic type deduction with `auto`, the explicit types of the time point and time duration are extremely verbose to write. In the function `durationSinceEpoch` (lines 9 - 19), I display the time duration in different resolutions. First, I display the number of time ticks (line 11), then the number of minutes (line 15), with the years (lines 18) since the epoch at the end; all values depend on the clock used. For the sake of simplicity, I ignore leap years and assume that a year has 365 days.

`std::chrono::system_clock` and `std::chrono::high_resolution_clock` have the UNIX-epoch as starting point on my linux PC. The starting point of `std::chrono::steady_clock` is the boot time of my PC. While it seems that `std::high_resolution_clock` is an alias for `std::system_clock` on Linux, `std::high_resolution_clock` seems to be an alias for `std::steady_clock` on Windows. This conclusion is in accordance with the result from the previous subsection: Accuracy and Steadiness.

Thanks to the time library, you can put a thread to sleep. The arguments of the sleep and wait functions are time points or time durations.

```

*/
// =====
// =====
/*

```

Sleep and Wait

This lesson gives a brief introduction to sleep/wait and its usage in C++ with the help of interactive examples.

We'll cover the following

Conventions

Various waiting strategies

One important feature that multithreading components such as threads, locks, condition variables, and futures have in common is the notion of time.

Conventions#

The methods for handling time in multithreading programs follow a simple convention: Methods ending with `_for` have to be parametrized by a time duration; methods ending with `_until` by a time point. Here is a concise overview of the methods that deal with sleeping, blocking, and waiting:

Multithreading Component	<code>_until</code>	<code>_for</code>
<code>std::thread</code>	<code>th.sleep_until(in2min)</code>	<code>th.sleep_for(2s)</code>
<code>std::unique_lock</code>	<code>lk.try_lock_until(in2min)</code>	<code>lk.try_lock(2s)</code>
<code>std::condition_variable</code>	<code>cv.wait_until(in2min)</code>	<code>cv.wait_for(2s)</code>
<code>std::future</code>	<code>fu.wait_until(in2min)</code>	<code>fu.wait_for(2s)</code>
<code>std::shared_future</code>	<code>shFu.wait(in2min)</code>	<code>shFu.wait_for(2s)</code>

in2min stands for a time 2 minutes in the future; 2s is a time duration of 2 seconds. Although I use auto in the initialization of the time point in2min, the following is still verbose:

```
1
auto in2min= std::chrono::steady_clock::now() + std::chrono::minutes(2);
Time literals from C++14 come to our rescue when using typical time durations, e.g. 2s stands for 2 seconds.
Let's look at different waiting strategies.
```

Various waiting strategies#

The main idea of the following program is that the promise provides its result for four shared futures. That is possible because more than one shared_future can wait for the notification of the same promise. Each future has a different waiting strategy. Both the promise and every future will be executed in different threads. For simplicity reasons, I will only speak about a waiting thread in the rest of this subsection, although it will be the corresponding future that is waiting. Below are the details of the promises and the futures.

Here are the strategies for the four waiting threads:

consumeThread2: waits up to 20 seconds for the result of the promise.

consumeThread3: asks the promise for the result and goes back to sleep for 700 milliseconds.

consumeThread4: asks the promise for the result and goes back to sleep. Its sleep duration starts with 1 millisecond and doubles each time.

Here is the program:

```
*/
// =====
// sleepAndWait.cpp

// #include <utility>
// #include <iostream>
// #include <future>
// #include <thread>
// #include <utility>

// using namespace std;
// using namespace std::chrono;

// mutex coutMutex;

// long double getDifference(const steady_clock::time_point& tp1,
//                           const steady_clock::time_point& tp2){
//   const auto diff= tp2 - tp1;
//   const auto res= duration<long double, milli>(diff).count();
//   return res;
// }

// void producer(promise<int>&& prom){
//   cout << "PRODUCING THE VALUE 2011\n\n";
//   this_thread::sleep_for(seconds(5));
```

```

//  prom.set_value(2011);
// }

// void consumer(shared_future<int> fut,
//               steady_clock::duration dur){
//   const auto start = steady_clock::now();
//   future_status status= fut.wait_until(steady_clock::now() + dur);
//   if (status == future_status::ready ){
//     lock_guard<mutex> lockCout(coutMutex);
//     cout << this_thread::get_id() << " ready => Result: " << fut.get()
//     << endl;
//   }
//   else{
//     lock_guard<mutex> lockCout(coutMutex);
//     cout << this_thread::get_id() << " stopped waiting." << endl;
//   }
//   const auto end= steady_clock::now();
//   lock_guard<mutex> lockCout(coutMutex);
//   cout << this_thread::get_id() << " waiting time: "
//   << getDifference(start,end) << " ms" << endl;
// }

// void consumePeriodically(shared_future<int> fut){
//   const auto start = steady_clock::now();
//   future_status status;
//   do {
//     this_thread::sleep_for(milliseconds(700));
//     status = fut.wait_for(seconds(0));
//     if (status == future_status::timeout) {
//       lock_guard<mutex> lockCout(coutMutex);
//       cout << "    " << this_thread::get_id()
//       << " still waiting." << endl;
//     }
//     if (status == future_status::ready) {
//       lock_guard<mutex> lockCout(coutMutex);
//       cout << "    " << this_thread::get_id()
//       << " waiting done => Result: " << fut.get() << endl;
//     }
//   } while (status != future_status::ready);
//   const auto end= steady_clock::now();
//   lock_guard<mutex> lockCout(coutMutex);
//   cout << "    " << this_thread::get_id() << " waiting time: "
//   << getDifference(start,end) << " ms" << endl;
// }

// void consumeWithBackoff(shared_future<int> fut){
//   const auto start = steady_clock::now();
//   future_status status;
//   auto dur= milliseconds(1);
//   do {
//     this_thread::sleep_for(dur);

```

```

//    status = fut.wait_for(seconds(0));
//    dur *= 2;
//    if (status == future_status::timeout) {
//        lock_guard<mutex> lockCout(coutMutex);
//        cout << "      " << this_thread::get_id()
//            << " still waiting." << endl;
//    }
//    if (status == future_status::ready) {
//        lock_guard<mutex> lockCout(coutMutex);
//        cout << "      " << this_thread::get_id()
//            << " waiting done => Result: " << fut.get() << endl;
//    }
// } while (status != future_status::ready);
// const auto end= steady_clock::now();
// lock_guard<mutex> lockCout(coutMutex);
// cout << "      " << this_thread::get_id()
//     << " waiting time: " << getDifference(start,end) << " ms" << endl;
//}

// int main(){

//    cout << endl;

//    promise<int> prom;
//    shared_future<int> future= prom.get_future();
//    thread producerThread(producer, move(prom));

//    thread consumerThread1(consumer, future, seconds(4));
//    thread consumerThread2(consumer, future, seconds(20));
//    thread consumerThread3(consumePeriodically, future);
//    thread consumerThread4(consumeWithBackoff, future);

//    consumerThread1.join();
//    consumerThread2.join();
//    consumerThread3.join();
//    consumerThread4.join();
//    producerThread.join();

//    cout << endl;

//}

//=====
/*

```

I create the promise in the main function (line 98), use the promise to create the associated future (line 99), and move the promise into a separate thread (line 100). I have to move the promise into the thread because it does not support the copy semantic. That is not necessary for the shared futures (lines 102 - 105); they support the copy semantic and can, therefore, be copied.

Before I talk about the work package of the thread, let me say a few words about the auxiliary function `getDifference` (lines 14 - 19). The function takes two-time points and returns the time duration in milliseconds. I will use the function a few times.

What about the five created threads?

producerThread: executes the function producer (lines 21 - 25) and publishes its result 2011 after 5 seconds of sleep. This is the result the futures are waiting for.

consumerThread1: executes the function consumer (lines 27 - 44). The thread is waiting for 4 seconds at most (line 30) before it continues with its work; This waiting period is not long enough to get the result from the promise.

consumerThread2: executes the function consumer (lines 27 - 44). The thread is waiting 20 seconds at most before it continues with its work.

consumerThread3: executes the function consumePeriodically (lines 46 - 67). It sleeps for 700 milliseconds (line 50) and asks for the result of the promise (line 60). Because of the std::chrono::seconds(0) in line 51, there is no waiting. If the result of the calculation is available, it will be displayed in line 60.

consumerThread4: executes the function consumeWithBackoff (lines 69 - 92). It sleeps in the first iteration 1 second and doubles its sleeping period every iteration. Otherwise, its strategy is similar to the strategy of consumerThread3.

Now to the synchronization of the program. Both the clock determining the current time and std::cout are shared variables, but no synchronization is necessary. Firstly, the method call std::chrono::steady_clock::now() is thread-safe (for example in lines 30 and 40); Secondly, the C++ runtime guarantees that the characters will be written thread-safe to std::cout. I only used a std::lock_guard to wrap std::cout (for example in lines 32, 37, and 41).

Although the threads write one after the other to std::cout, the output is not easy to understand.

The first output is from the promise, with the remaining outputs from the futures. At first consumerThread4 asks for the result. The output is indented by 8 characters. consumerThread4 also displays its ID. consumerThread3 immediately follows. Its output is indented by 4 characters. The output of consumerThread1 and consumerThread2 is not indented.

consumeThread1: waits unsuccessfully 4000.18 ms seconds without getting the result.

consumeThread2: gets the result after 5000.3 ms although its waiting duration is up to 20 seconds.

consumeThread3: gets the result after 5601.76 ms. That's about 5600 milliseconds= $8 * 700$ milliseconds.

consumeThread4: gets the result after 8193.81 ms. To say it differently. It waits 3 seconds too long.

```
*/  
// ======  
// ======  
/*
```

Glossary

This lesson serves as a reference for the most important terms used in this course of concurrency with modern C++.

We'll cover the following

ACID
Callable Unit
Concurrency
CriticalSection
Function Objects
Lambda Functions
Lock-free
Lost Wakeup
Modification Order
Monad
Non-blocking
Parallelism
Predicate
RAII
Sequential Consistency
Sequence Point
Spurious Wakeup
Thread
Total Order
volatile
Wait-free

The idea of this glossary is by no means to be exhaustive.

ACID#

A transaction is an action that has the properties Atomicity, Consistency, Isolation, and Durability (ACID). Except for durability, all properties hold for transactional memory in C++.

Atomicity: either all or no statement of the block is performed.

Consistency: the system is always in a consistent state. All transactions build a total order.

Isolation: each transaction runs in total isolation from the other transactions.

Callable Unit#

A callable unit (short callable) is something that behaves like a function. Not only are these named functions, but also function objects and lambda functions. If a callable accepts one argument, it's called unary callable; accepting two arguments is called binary callable.

Predicates are special functions that return a boolean as a result.

Concurrency#

Concurrency means that the execution of several tasks overlaps. Concurrency is a superset of parallelism.

Critical Section#

A critical section is a section of code that should be executed in an atomic fashion following the ACID properties.

The ACID properties are typically guaranteed with mutual exclusion primitives such as mutexes, or with transactions such as transactional memory.

Function Objects#

First of all, don't call them functors. That's a well-defined term from a branch of mathematics called category theory.

Function objects are objects that behave like functions. They achieve this by implementing the function call operator. As function objects are objects, they can have attributes and therefore state.

```
/*
// =====
// #include <iostream>
// #include <vector>
// #include <algorithm>

// struct Square{
//   void operator()(int& i){i= i*i;}
// };

// int main(){
//   std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

//   std::for_each(myVec.begin(), myVec.end(), Square());

//   for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
// }
// =====
/*
Instantiate function objects to use them
```

It's a common error that the name of the function object (Square) is used in an algorithm instead of the instance of function object (Square()) itself: std::for_each(myVec.begin(), myVec.end(), Square). Of course, that's a typical error. You have to use the instance: std::for_each(myVec.begin(), myVec.end(), Square())

Lambda Functions#

Lambda functions provide their functionality in-place. The compiler gets its information right on the spot and, therefore, has great optimization potential. Lambda functions can receive their arguments by value or by reference. They can capture the variables of their defining environment by value or by reference as well.

```
/*
// =====
// #include <iostream>
// #include <vector>
// #include <algorithm>

// int main(){
//   std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

//   std::for_each(myVec.begin(), myVec.end(), [](int& i){ i= i*i; });

//   for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
// }
// =====
/*
Lambda functions should be your first choice
```

If the functionality of your callable is short and self-explanatory, use a lambda function. A lambda function is generally faster than a function, or a function object and easier to understand.

Lock-free#

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

Lost Wakeup#

A lost wakeup is a situation in which a thread misses its wakeup notification due to a race condition.

That may happen if you use a condition variable without a predicate.

Modification Order#

The modification order is the order in which each memory location is modified.

The memory model guarantees that each memory location has a total modification order; i.e. memory operations performed by the same thread on the same memory location cannot be reordered.

Monad#

Haskell as a pure functional language has only pure functions. A key feature of these pure functions is that they will always return the same result when given the same arguments. Thanks to this property - called referential transparency - a Haskell function cannot have side effects; therefore, Haskell has a conceptional issue. The world is full of calculations that have side effects. These are calculations that can fail, that can return an unknown number of results, or that are dependent on the environment. To solve this conceptional issue, Haskell uses monads and embeds them in the purely functional language.

The classical monads encapsulate one side effect:

I/O monad: Calculations that deal with input and output.

Maybe monad: Calculations that maybe return a result.

Error monad: Calculations that can fail.

List monad: Calculations that can have an arbitrary number of results.

State monad: Calculations that build a state.

Reader monad: Calculations that read from the environment.

The concept of the monad is from category theory, which is a part of the mathematics that deals with objects and mapping between these objects. Monads are abstract data types (type classes), which transform simple types into enriched types. Values of these enriched type are called monadic values. Once in a monad, a value can only be transformed by a function composition into another monadic value.

This composition respects the special structure of a monad; therefore, the error monad will interrupt its calculation if an error occurs or the state monad builds its state.

To make this happen, a monad consists of three components:

Type constructor: The type constructor defines how the simple data type becomes a monadic data type.

Functions:

Identity function: Introduces a simple value into the monad.

Bind operator: Defines how a function is applied to a monadic value to get a new monadic value.

Rules for the functions:

The identity function has to be the left and the right identity element.

The composition of functions has to be associative.

In order for the error monad to become an instance of the type class monad, the error monad has to support the identity function and the bind operator. Both functions define how the error monad deals with an error in the calculation. If you use an error monad, the error handling is done in the background.

A monad consists of two control flows: the explicit control for calculating the result, and the implicit control flow for dealing with the specific side effect.

Of course, you can define a monad in fewer words: "A monad is just a monoid in the category of endofunctors."

Monads are becoming more and more import in C++. With C++17, we get std::optional which is a kind of a Maybe monad. With C++20, we will probably get extended futures and the ranges library from Eric Niebler; both are monads.

Non-blocking#

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. This definition is from the excellent book Java concurrency in practice.

Parallelism#

Parallelism means that several tasks will be performed at the same time. Parallelism is a subset of Concurrency.

Predicate#

Predicates are special callable units that return a boolean as the result. If a predicate has one argument, it's called a unary predicate. If a predicate has two arguments, it's called a binary predicate.

RAII#

Resource Acquisition Is Initialization (RAII) stands for a popular technique in C++, in which the resource acquisition and release are bound to the lifetime of an object. For a lock, this means that the mutex will be locked in the constructor and unlocked in the destructor.

Typical use cases in C++ are locks that handle the lifetime of its underlying mutex, or a smart pointer that handles the lifetime of its resource (memory).

Sequential Consistency#

Sequential consistency has two key characteristics:

The instructions of a program are executed in source code order

There is a global order of all operations on all threads

Sequence Point#

A sequence point defines any point in the execution of a program at which it is guaranteed that all effects of previous evaluations will have been performed, and no effects from subsequent evaluations have yet been performed.

Spurious Wakeup#

A spurious wakeup is a phenomenon of condition variables. It may happen that the waiting component of the condition variable erroneously gets a notification, although the notification component didn't notify it.

Thread#

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation

of threads and processes differs between operating systems, but a thread is a component of a process in most cases. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time. For the details, read the Wikipedia article about threads.

Total Order#

A total order is a binary relation (\leq) on some set X which is antisymmetric, transitive, and total.

Antisymmetric: if $a \leq b$ and $b \leq a$ then $a = b$

Transitivity: if $a \leq b$ and $b \leq c$ then $a \leq c$

Totality: $a \leq b$ or $b \leq a$

volatile#

volatile is typically used to denote objects which can change independently of the regular program flow. These are, for example, objects in embedded programming which represent an external device (memory-mapped I/O). Because these objects can change independently of the regular program flow and their value will directly be written into main memory, no optimized storing in caches takes place.

Wait-free#

A non-blocking algorithm is wait-free if there is guaranteed per-thread progress.

```
*/  
// ======  
// ======  
/*
```

Running Source Code on your own machine

Teach yourself how to interact with and run source code examples of this course.

We'll cover the following

Run the Programs

All source code examples are complete; that means, assuming you have a conforming compiler, you can compile and run them. The name of the source file is in the title of the listing. Only, if necessary, will I use the using namespace std directive in the source files.

Run the Programs#

Compiling and running the examples is quite easy for the C++11 and C++14 examples in this course. Every modern C++ compiler should support them. For both the GCC and the clang compiler, the C++ standard must be specified as well as the threading library to link against. For example, the g++ compiler from GCC creates an executable program called thread with the following command-line: g++ -std=c++14 -pthread thread.cpp -o thread.

-std=c++14: use the language standard C++14

-pthread: add support for multithreading with the pthread library

thread.cpp: source file

-o thread: executable program

The same command-line works for the clang++ compiler. The Microsoft Visual Studio 17 C++ compiler supports C++14 as well. If you have no modern C++ compiler at your disposal, there are a lot of online compilers available. Arne Mertz' blog post C++ Online Compiler gives a great overview.

With C++17 and C++20, the story becomes quite complicated. I installed the HPX (High-Performance ParalleX) framework, which is a general purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented the Parallel STL of C++17 and many of the concurrency features of C++20. Please refer to the corresponding sections in the chapter The Future: C++20, and read about how you can see the code examples in action.

```
/*
// =====
// =====
/*
```

Summary#

Starting with a quick overview, we learned about the details of the memory model like the contract, atomics, synchronization, and fences. Then, we discussed core concepts of multithreading. We also went through a lot of case studies to apply the theory. In the end, we went through the parallel algorithm of the Standard Template Library, a few coding examples, and the time library.

What's next?#

Now that we have familiarized ourselves with the concepts of concurrency in C++, we'll learn about the graph algorithms in the next "Learn Graph Algorithms in C++" module.

```
*/
```

```
/*
```

algorithms:

Introduction to Graph Algorithms

Learn the concepts of graph theory and graph algorithms by studying the Königsberg bridge problem.

We'll cover the following

A bit of history

An algorithm for Eulerian trails

Fleury's algorithm

Implementing Fleury's algorithm

Next steps

This course is about algorithms on graphs, which are data structures that represent a set of nodes

Although graphs are an abstract concept of mathematics and theoretical computer science, they can be used to model various problems from real-life applications. Some examples are:

routing of IP packages on the internet.

finding the fastest connection in public transportation systems.

detecting deadlocks in operating system processes.

As such, graph algorithms can be a powerful and versatile tool in every software developer's toolbox.

A bit of history#

The study of graph theory perhaps originated with the so-called Königsberg bridge problem. It is a neat example of modeling a real-life puzzle using graphs.

The historical Prussian city of Königsberg (today Kalinigrad) consisted of four landmasses separated by rivers and connected by a total of seven bridges. In the early 18th century, mathematician Leonard Euler studied

whether there was any way to devise a walk through the city that crosses each of the bridges exactly once. Today, hiss analysis of this problem is today seen as one of the founding moments of graph theory.

Euler modeled the city as a graph with each section of the city being represented by a node and each bridge between two sections represented by an edge. This graph is illustrated below.

Although graphs are an abstract concept of mathematics and theoretical computer science, they can be used to model various problems from real-life applications. Some examples are:

routing of IP packages on the internet.

finding the fastest connection in public transportation systems.

detecting deadlocks in operating system processes.

As such, graph algorithms can be a powerful and versatile tool in every software developer's toolbox.

A bit of history#

The study of graph theory perhaps originated with the so-called Königsberg bridge problem. It is a neat example of modeling a real-life puzzle using graphs.

The historical Prussian city of Königsberg (today Kalinigrad) consisted of four landmasses separated by rivers and connected by a total of seven bridges. In the early 18th century, mathematician Leonard Euler studied whether there was any way to devise a walk through the city that crosses each of the bridges exactly once. Today, hiss analysis of this problem is today seen as one of the founding moments of graph theory.

Euler modeled the city as a graph with each section of the city being represented by a node and each bridge between two sections represented by an edge. This graph is illustrated below.

The resulting graph now has only two nodes with an odd number of edges, which makes it possible to construct an Eulerian trail from south Königsberg to the eastern island. Fleury's algorithm describes how to achieve this:

Start from a node with an odd number of edges.

Select the next edge from the current node and choose an edge that will disconnect its endpoints once removed.

Travel along the selected edge and remove it from the graph.

Let's run the algorithm on the Königsberg bridges graph:

Implementing Fleury's algorithm#

Conceptually, Fleury's algorithm seems relatively simple. It can be described in a short paragraph of text, and it was not very challenging to apply it on our small example graph. Implementing it in a programming language like C++ comes with its own set of challenges though.

So far, our perspective on graphs has been that they are drawings made up of circles and lines. Our first task will be to transfer this graphical representation into something a machine can work with, using efficient data structures like vector from the C++ standard library.

When looking at the image of the four-node Königsberg graph, it was easy for us to ascertain at a glance whether removing an edge would disconnect the graph. But neither a computer program nor a human can see the whole graph at once the graph has hundreds of millions of nodes. We'll need strategies to traverse graphs and check whether removing certain edges would split them into separate components, which will also require efficient graph algorithms.

Next steps#

During the rest of the course, you'll learn:

the fundamental concepts of graph theory.

how graphs can be represented in code using adequate data structures.

algorithms for traversing graphs efficiently.

how to implement optimization algorithms that solve for the shortest path, network flow, and other problems.

We use the C++ programming language in this course, but the code examples should also be readily transferable to other programming languages like Java or Python. After finishing the course, you will be well equipped to implement graph algorithms like Fleury's algorithm for finding Eulerian trails.

Directed graphs

Undirected graphs

Before we start diving into graph algorithms, let's introduce some of the basic concepts of graph theory that we'll use throughout the course.

Directed graphs#

We'll start with the most basic question: what even is a graph? Let's look at an example.

a

b

c

A graph with three nodes and four edges.

The above graph consists of three nodes (named a

a

, b

b

, c

c

) and four edges (drawn as arrows between nodes). The edges have a direction, indicated by the arrowhead. Hence, we call such a graph a directed graph:

a \to b

a → b

a \to c

a → c

b \to c

b → c

c \to b

c → b

The actual positioning of the nodes in the graph does not matter. In other words, it has no significance that in the above drawing a

a

is "above" b

b

. In fact, here is the same graph drawn in a different layout:

a

b

c

The same graph as above, just drawn differently.

In other words, a graph is completely described by its set of nodes and edges.

By convention, the capital letter V

V

(for vertices) is used to denote the set of nodes. In our example graph, we have

$$V = \{a, b, c\}.$$

$$V = \{a, b, c\}.$$

To denote the set of edges, we use the capital letter E

E

. Each edge is a pair of two vertices, its source and target. In our example graph,

$$E = \{(a, b), (a, c), (b, c), (c, b)\}.$$

$$E = \{(a, b), (a, c), (b, c), (c, b)\}.$$

In total, a graph G

G

can then be written as:

$$G = (V, E) \quad \text{with } E \subseteq V \times V.$$

$$G = (V, E) \quad \text{with } E \subseteq V \times V.$$

For our example graph,

$$G = (\{a, b, c\}, \{(a, b), (a, c), (b, c), (c, b)\}).$$

$$G = (\{a, b, c\}, \{(a, b), (a, c), (b, c), (c, b)\}).$$

In this course, we'll only consider graphs whose edges fulfill the following properties:

There are never two edges with the same source and the same target.

There are no "self-loop" edges of the form (v, v)

(v, v)

Such graphs are called simple graphs. The following figure shows the two forbidden situations that were described above.

a

b

c

Edges that cannot exist in simple graphs.

Undirected graphs#

So far, the graphs we've considered have had directed edges with a source and target. However, many times we'll find that all of the connections between vertices are bidirectional. One example is a traffic network, where the intersections are the nodes and the roads are the edges. Usually, the roads can all be traversed in both directions.

We can describe such a graph as a directed graph, for example.

a
b
c
d

A directed graph where all connections are bidirectional.

The above graph has the set of edges:

$$E = \{(a, b), (b, a), (a, c), (c, a), (b, d), (d, b)\}.$$
$$E=\{(a,b),(b,a),(a,c),(c,a),(b,d),(d,b)\}.$$

But, if we know that all connections are bidirectional anyway, there is no need to draw two arrows between each pair of connected nodes. To simplify the drawing, we can just draw a straight line between connected nodes, implying they are connected bidirectionally.

a
b
c
d

The same graph, drawn with undirected edges.

We call such a graph an undirected graph.

How should we define the set of edges E

E

of an undirected graph? We could use a new definition where every edge is not an ordered pair of source and origin, but instead is an unordered set of two vertices. However, it would be inconvenient to have two different definitions of graphs.

The more pragmatic solution is to use the same definition as before: edges are a pair of source and origin. But in an undirected graph, we always require that if: (u, v)

(u, v)
is an edge in the graph, then also its reverse (v, u)

(v, u)

is an edge. Formally,

$$\text{for all } u, v \in V: (u, v) \in E \rightarrow (v, u) \in E.$$
$$\text{for all } u, v \in V: (u, v) \in E \rightarrow (v, u) \in E.$$

So, the set of edges of our undirected example graph is still

$$E = \{(a, b), (b, a), (a, c), (c, a), (b, d), (d, b)\}.$$
$$E=\{(a,b),(b,a),(a,c),(c,a),(b,d),(d,b)\}.$$

The graph itself has not changed, but the way we have drawn it and interpreted it is different! We can think of the undirected connection between a

a
and b
b
as being represented by the two “half-edges” (a, b)
 (a, b)
and (b, a)
 (b, a)

Graph Terminology I

Learn about the fundamental concepts of graph theory, such as adjacency and paths.

We'll cover the following

Adjacency

Sparse and dense

Walks, trails, and paths

Adjacency#

For an edge $e = (u, v)$

$e = (u, v)$

that connects node u

u

to node v

v

, we say that v

v

is adjacent to u

u

, or that v

v

is a neighbor of u

u

. The edge e

e

is called incident to both u

u

and v

v

.

The number of neighbors of a node v

v

is called the degree of v

v

, written $\deg(v$

v

$).$

In the following example graph, c

c

is a neighbor of b

b

, as there is an edge (b, c)

(b, c)

. But b

b

is not a neighbor of c

c

, as there is no edge (c, b)

(c,b)

. The degree of node a

a

is $\deg(a) = 2$

(a)=2

, as it has two neighbors.

a

c

b

An example graph with three nodes.

Sparse and dense#

Graphs can be classified as sparse or dense, depending on how many edges they have.

Let's consider a graph with $n = |V|$

$n=|V|$

nodes. The maximum number of edges in a directed graph is $n^2 - n$

n

2

-n

, and the maximum number of edges in an undirected graph is

$\{n \choose 2} = \frac{n^2 - n}{2}$.

(

2

n

)=

2

n

2

-n

.

In both cases, the maximum number of edges is quadratic with respect to the number of nodes. If the actual number of edges $m = |E|$

$m=|E|$

is close to that maximum number, we call the graph dense. On the other hand, if the number of edges is only linear in the number of nodes (for example, $m \sim 2 \cdot n$

$m \sim 2 \cdot n$

), the graph is called sparse.

Walks, trails, and paths#

A walk is a sequence of nodes v_1, \dots, v_k

v

1

,...,v

k

such that each successive node is adjacent to the previous one. As the name suggests, we can think of a walk as starting at some node $v_1 \in V$

v_1

$\in V$

, and then walking along edges of the graph to v_2, v_3

v_2

$, v_3$

and so on.

In a walk we are allowed to repeat edges and nodes. There are also two more limited versions of walks that can be useful concepts.

A trail is a walk in which no edge may be used more than once.

A path is a walk in which no node may be used more than once.

Let's illustrate these concepts using the following example graph.

a
b
c
d
e
f
g
j
h
i

An example of a walk (blue), trail (green) and path (red)

First, let's look at the edges marked in blue. They form a walk $a \rightarrow b \rightarrow c \rightarrow b \rightarrow c \rightarrow d$

. This walk is not a trail, as the edge (b, c)

(b, c)
is traversed twice. It is also not a path as nodes b
 b
and c
 c
are repeated.

The green edges form the trail $e \rightarrow g \rightarrow f \rightarrow e \rightarrow j$
 $e \rightarrow g \rightarrow f \rightarrow e \rightarrow j$

. This is a trail because no edge is repeated. However, it is not a path, because the node e
 e
is visited twice.

Finally, the node sequence e \to h \to i
e→h→i
, marked in red, is a path. It does not repeat any edges or nodes.

Graph Terminology II
Study further graph theory concepts, such as cycles and connectivity.

We'll cover the following

Cycles
Reachability
Lengths and distances
Cycles#

A cycle is similar to a path, except that its first and last nodes are identical. In other words, a cycle is like a path that loops back into itself.

The following example graph shows the cycle a \to c \to d \to a
a→c→d→a
marked in blue.

a
c
b
d

A cycle with three edges.

In an undirected graph, a single edge between nodes u

u
and v

v
consists of the two “half-edges” (u, v)

(u,v)
and (v, u)

(v,u)

. However, the walk u \to v \to u

u→v→u

, forward and backward along the edge is not considered a cycle.

In general, in an undirected cycle, each undirected edge may only be traversed once. This also means that an undirected cycle needs to be at least of length three, while a directed cycle can be of length two.

a
b
u
v

Left: A two node cycle in a directed graph. Right: a single undirected edge is not a cycle.

A graph that contains at least one cycle is called cyclic, and a graph that does not contain any cycle is called acyclic.

Reachability#
If there is a path from node u

u
to node v
v
, then v
v
is called reachable from node u
u
.

An undirected graph is called connected if every node is reachable from every other node. If it is not reachable, it is called disconnected.

For example, the graph below is disconnected, because the node c

c
is not reachable from node d
d
. If we add an edge (c, d)
(c,d)
, the graph becomes connected.

a
d
b
c
e

A disconnected undirected graph.

In a directed graph, there are two notions of connectivity: weak connectivity and strong connectivity.

A directed graph is strongly connected if every node is reachable from every other node. This means that for each pair of distinct nodes (u, v)

(u,v)
, there must be both a path from u
u
to v
v
and a path from v
v
to u
u
.

The directed graph is called weakly connected if, for every pair of distinct nodes (u, v)

(u,v)
, there is at least one path from u
u
to v
v
, or from v
v
to u
u

For example, the following directed graph is weakly connected. It is not strongly connected because there are several pairs of nodes that can not reach each other in both directions. For instance, there is no path that leads from c

c
to a
a
.

a
b
c
d

A weakly connected graph.

There is another way to view weak connectivity. We can turn a directed graph into an undirected graph by adding a reverse edge (u, v)

(u, v)
for each edge (v, u)
 (v, u)
in E

E

. Let's call this the corresponding undirected graph of a directed graph. Then a directed graph is weakly connected if and only if its corresponding undirected graph is connected.

Lengths and distances#

The length of a walk (or trail, or path) is the number of edges that it follows. In a walk v_1, \dots, v_k

v
1

, ..., v
k

, its length is $k-1$

$k-1$

.

If there is a walk that leads from u

u
to v
v

, then there must also be the shortest walk from u

u
to v
v

. This shortest walk is always a path because it would not make sense to repeat nodes or edges on the shortest walk. There can be several shortest paths of the same length.

The length of a shortest path from u

u

to v
v
is called the distance of node v
v
from node u
u
.

Let's check out these concepts in the following example graph.

a
b
c
d
A shortest path from a to c.
A shortest path from d
d
to c
c
is marked in blue. Its length is 2
2
.

There are other paths from d
d
to c
c
, but none of them are shorter. There is, however, one other shortest path of length 2
2
: d \to b \to c
d→b→c
.

Weighted Graphs
Discover graphs with edge weights.

We'll cover the following

Introducing weighted graphs
Paths in weighted graphs
A new perspective on unweighted graphs
Introducing weighted graphs#

Weighted graphs are a slight generalization of graphs that are useful for modeling more complex problems using graph theory. In an (edge)-weighted graph, each edge is assigned a weight, which is usually a number that represents a property of the edge.

For example, consider this undirected graph of subway stations, where each edge represents a train connection. The weight of an edge is a real number representing the distance between the two nodes it connects.

Piccadilly Circus

Holborn

1

Embankment

0.7

Baker Street

1.6

Liverpool Street

King's Cross

2.4

1.3

1.8

2

1.7

A weighted graph where the edge weights are distances between subway stations.

Formally, a weighted graph $G = (V, E, w)$

$G = (V, E, w)$

is a graph with nodes V

V

, edges E

E

and an additional edge weight function w

w

. In the most cases, the weights are numeric:

if the weights are integers, then w

w

is a function $w : E \rightarrow \mathbb{N}$

$w : E \rightarrow \mathbb{N}$

if the weights are real numbers, then w

w

is a function $w : E \rightarrow \mathbb{R}$

$w : E \rightarrow \mathbb{R}$

Other weight functions are also possible (for example, each edge could be labeled with a text) but these are less common. In this course, most weighted graphs will have integer weights.

If the graph is undirected, we need to deal with a slight technicality. For each “half-edge” (u, v)

(u, v)

, also its opposite (v, u)

(v, u)

is in E

E

. But since these two “half-edges” together form a single undirected edge, they must both have the same weight. In other words, the following symmetry condition holds in weighted, undirected graphs:

$$w(u, v) = w(v, u) \quad \forall (u, v) \in E.$$

$$w(u, v) = w(v, u) \quad \forall (u, v) \in E.$$

Paths in weighted graphs#

In the previous lesson, we saw that for unweighted graphs, the length of a path is simply the number of edges on the path. For weighted graphs, we'll adapt our definition of the length of a path to make use of the weights.

For a path in a weighted graph, its length is defined as the sum of the weights of the edges along the path. We can think of the weight of an edge as representing the distance between the nodes it connects.

For example, consider the path from Piccadilly Circus to Holborn marked in blue below.

Piccadilly Circus

Holborn

1

Embankment

0.7

Baker Street

1.6

Liverpool Street

King's Cross

2.4

1.3

1.8

2

1.7

A path in the weighted graph.

The length of the blue path is $0.7 + 2 + 1.8 = 4.5$

$0.7+2+1.8=4.5$

.

A new perspective on unweighted graphs#

We can also think of unweighted graphs as special kinds of weighted graphs, where each edge has a weight of 1

1

. Our definition of weighted path lengths then coincides with the original definition of path lengths in unweighted graphs.

Excursion: Algorithm Analysis

Use big-O-notation for analyzing algorithms.

We'll cover the following

Counting operations

Towards big-O-notation

Only the largest growing term counts

Ignoring constant factors

Big-O-notation for graph algorithms

This lesson introduces the concept of big-\mathcal{O}

O

-notation for describing the worst-case runtime of an algorithm. If you are already familiar with big-\mathcal{O}

O

-notation, feel free to skip this lesson.

Counting operations#

Computer science problems can be solved with various different algorithms. Naturally, this leads to the task of comparing different algorithms for the same problem to find out which one is the best. More often than not, this boils down to the question “which algorithm is the fastest?”

To measure and compare the runtime behavior of algorithms, we can count the number of operations that the algorithm must perform compared to the size of its input. For illustration, consider the following C++-function that checks whether there are two elements in a vector that sum to zero.c

```
/*
//=====
=====

// #include <iostream>
// #include <vector>
// using namespace std;

// bool twoSum(vector<int> numbers) {
//   for (int a : numbers) {
//     for (int b : numbers) {
//       int sum = a + b;
//       if (sum == 0) {
//         return true;
//       }
//     }
//   }
//   return false;
// }

// int main() {
//   vector<int> example {-2, 3, 2, 6};
//   cout << (twoSum(example) ? "true" : "false");
// }
////
=====

=====
/*
Let's say that we pass a vector with n
n
elements to twoSum. How many operations will it perform?
```

Actually, this will depend on the contents of the vector. If the first two elements are already summed to zero, the function will finish much quicker than if the last two elements sum to zero. Therefore, we usually focus on the worst-case runtime of an algorithm. In the case of twoSum, the worst case happens when there are no two numbers of the vector that sum to zero. In that case, both loops run over all elements and finally return false.

So, let's count the operations that twoSum performs in the worst case on an input vector of size n
n

. There are two nested for-loops looping over n
n
elements, so the code inside these loops is executed n^2

n
2

times. There are two statements happening there: the addition and the if-condition. This makes $2n^2$

2n
2

operations. Finally, there is the return-statement after the loops, accounting for one more operation. Our twoSum function performs $2n^2 + 1$

2n
2
+1
operations in the worst case.

Towards big-\mathcal{O}

O
-notation#

While operation counting is a valid and precise way to measure an algorithm's runtime, it's also quite tedious. Computer scientists usually use the so-called big-\mathcal{O}

O
-notation instead. In this notation, our twoSum function's runtime would be written simply as $\mathcal{O}(n^2)$

$\mathcal{O}(n^2)$
)

.

There is a rigorous mathematical definition of the \mathcal{O}

O
-symbol, but we will not cover that in this course. An intuitive understanding of its meaning will be sufficient.

Only the largest growing term counts#

The first rule of big-\mathcal{O}

O
-notation is that we'll only keep the fastest growing out of a sum of several terms. In our example of $2n^2 + 1$

2n
2
+1
operations of twoSum, we'll only keep the quadratic term $2n^2$

2n
2

and ignore the constant 1

1
.

To give another example, if an algorithm takes $0.5n^3 + 50\log n + 15n$

0.5n
3
+50logn+15n
operations, we'll only keep the $0.5n^3$

0.5n
3

term. This is because the cubic function grows much faster than the linear and the logarithmic component, so the cubic term will dominate the runtime as the input n
n
grows large.

Ignoring constant factors#

Another simplification applied in big-\mathcal{O}

O

-notation is that constant factors before terms are simply ignored. This makes our $2n^2+1$

$2n$

2

$+1$

operations from twoSum simply become \mathcal{O}(n²)

O(n

2

)

. In our second example from above, $0.5n^3 + 50\log n + 15n$

0.5n

3

$+50\log n + 15n$

becomes \mathcal{O}(n³)

O(n

3

)

.

The reasoning behind this simplification is that the influence of constant factors is ultimately much less relevant than the difference between terms of different growth. For example, in the plot above, we can see that the $50\log n$

$50\log n$

function is larger than the $0.5n^3$

0.5n

3

function for small values of n

n

. However, once n

n

becomes larger, the constant factors become irrelevant and the cubic term dominates.

There is also another, more practical argument for ignoring constant factors. - They are ultimately just difficult to count with exact precision. In our twoSum example above, we have counted C++ statements, but that is not the same as counting actual machine operations. Since we are not dealing with assembly code, but with high-level programming languages, it is not clear how many actual machine instructions a statement like int sum = a + b or if (sum == 0) will compile to, and this is also very impractical to count accurately.

All in all, these arguments suggest ignoring constant factors. Therefore we do so in big-\mathcal{O}

O

-notation. Still, if we compare a $2n^2$

$2n$

2

against a $4n^2$

$4n$

2

algorithm, the former will be twice as fast, but both will be described as $\mathcal{O}(n^2)$

$O(n^2)$

2

)

. Hence, constant factors can make a difference in practice. But for large instances, an $\mathcal{O}(n^3)$

$O(n^3)$

3

)

algorithm will never be faster than an $\mathcal{O}(n^2)$

$O(n^2)$

2

)

algorithm, no matter what the constant factors are.

Big- \mathcal{O}

O

-notation for graph algorithms#

When analyzing algorithms on graphs, the runtime is expressed relative to the size of the input graph $G = (V, E)$

$G=(V,E)$

. There are two components contributing to that size: the set of nodes V

V

and the set of edges E

E

. The influence of both sets is treated separately, to describe the algorithm runtime more precisely. For example,

an algorithm that performs an operation for each edge of the graph has a

$\mathcal{O}(|E|)$

$O(|E|)$

runtime.

an algorithm that performs some checks for each pair of nodes has a

$\mathcal{O}(|V|^2)$

$O(|V|$

2

)

runtime.

*/

```
/*
// =====
Adjacency Matrix
Learn how graphs can be represented in code using adjacency matrices.
```

We'll cover the following

Representing the vertices

Adjacency matrix

Implementing the adjacency matrix

Let's check out how to implement data structures to represent graphs. We'll work with the following example graph:

```
c  
a  
d  
b
```

An example graph with 4 nodes and 6 edges.

Representing the vertices#

When discussing graphs, we usually think of the vertices as a set V

```
V  
. In the example graph, the set of vertices is  $V = \{a, b, c, d\}$   
V={a,b,c,d}  
.
```

But when working with graphs in code, it is more convenient to assign increasing integer IDs to the nodes so that information about them can be stored efficiently in an array-like data structure, such as a C++ vector.

In our example graph, we could map the node names to integer IDs like this:

```
a \to 0  
a→0  
b \to 1  
b→1  
c \to 2  
c→2  
d \to 3  
d→3
```

The transformed graph is

```
2  
0  
3  
1
```

The example graph with integer node names

If a graph has $n = |V|$

```
n=|V|  
nodes, the node IDs are integers 0, 1, \dots, n-1  
0,1,...,n-1  
. That means that we actually only need the single number n
```

to represent the set of nodes.

All of our graph algorithms will work on these integer node IDs. If the original names of the nodes (a, b, c, d
a,b,c,d
) are still needed in the application, they can be stored for future reference.

```
vector<string> nodeNames = {"a", "b", "c", "d"};
map<string, int> nameToInt = {"a", 0}, {"b", 1}, {"c", 2}, {"d", 3};
```

Adjacency matrix#

Now that we've got the nodes covered, let's look into data structures for the edges. There are two common ways to represent them: adjacency matrices and adjacency lists. Recall that two vertices are called adjacent if they are connected by an edge. In this lesson, we'll check out the adjacency matrix data structure.

The adjacency matrix is a square matrix A

A
with n
n
rows and n
n
columns. The entries of the matrix are binary, either 0
0
or 1
1
. The entry A[i, j]
A[i,j]
in the i
i
-th row and the j
j
-th column is 1
1
when the nodes i
i
and j
j
are connected by an edge, otherwise, it is 0
0
. .

Since our nodes are represented by integers starting with 0

0
, our matrix is also 0
0
-indexed: A[0, 0]
A[0,0]
is the top left entry, and A[n-1, n-1]
A[n-1,n-1]
is the bottom right entry.

Let's look again at our example graph from above.

2
0
3
1

The example graph with integer node names.

Its adjacency matrix is

$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$

0
1
1
0

1
0
0
0

0
1
0
0

0
1
1
0

}

For example, the entry $A[0, 1]$

$A[0,1]$
is 1
1
, because there is an edge from node 0
0

to node 1

1
in our graph. On the other hand, the entry $A[3, 2]$
 $A[3,2]$

```

is 0
0
, because there is no edge from node 3
3
to node 2
2
in our graph.

```

Since we only consider simple graphs without self-loops, the matrix's main diagonal is always filled with zeros.

In an undirected graph, the adjacency matrix will always be symmetrical:

$$A[i,j] = A[j,i] \quad \text{for all } 0 \leq i \leq n-1, 0 \leq j \leq n-1$$

$$A[i,j]=A[j,i]\text{for all}0\leq i\leq n-1,0\leq j\leq n-1$$

This is due to the symmetry condition on the edge set.

Implementing the adjacency matrix#

In code, the adjacency matrix can be implemented naively using vectors of integers.

```

/*
// =====

// #include <vector>
// #include <iostream>
// using namespace std;

// int main() {
//   int n = 4;
//   vector<vector<int>> adjacencyMatrix(n, vector<int>(n, 0)); // initialize with zeros

//   // add edges using a loop over a vector of (source, target) pairs
//   vector<pair<int, int>> edges {{0, 1}, {1, 0}, {1, 2}, {1, 3}, {2, 0}, {2, 3}};
//   for (auto [u, v] : edges) {
//     adjacencyMatrix[u][v] = 1;
//   }
//   cout << "Edge from 0 to 1: " << adjacencyMatrix[0][1] << endl;
//   cout << "No edge from 0 to 3: " << adjacencyMatrix[0][3] << endl;
// }

/*

```

Some optimizations are possible in this implementation:

Instead of a nested vector of vectors, we can use a single vector with n^2

n^2

entries, where $A[i, j]$

$A[i,j]$

is stored at index n^2i+j

n^2i+j

Instead of a vector<int> , we can use a vector<bool> , which stores only one bit per entry.

When going with these optimizations, it is best to write a wrapper class that handles the indexing and also hides the `vector<bool>` object from the outside because it does not conform to the STL container interface.

```
/*
// =====
// #include <vector>
// #include <iostream>
// using namespace std;

// class AdjacencyMatrix
//{
// private:
//   vector<bool> adj;
//   int n;

// public:
//   AdjacencyMatrix(int _n)
//     : n{_n}, adj{vector<bool>(_n * _n, 0)}
//   {
//   }
//   void addEdge(int source, int target)
//   {
//     this->adj[this->n * source + target] = true;
//   }
//   void removeEdge(int source, int target)
//   {
//     this->adj[this->n * source + target] = false;
//   }
//   bool getEdge(int source, int target)
//   {
//     return this->adj[this->n * source + target];
//   }
// };

// int main()
//{
//   int n = 4;
//   AdjacencyMatrix adjacencyMatrix(n); // create empty matrix

//   // add edges using a loop over a vector of (source, target) pairs
//   vector<pair<int, int>> edges{{0, 1}, {1, 0}, {1, 2}, {1, 3}, {2, 0}, {2, 3}};
//   for (auto [u, v] : edges)
//   {
//     adjacencyMatrix.addEdge(u, v);
//   }
//   cout << boolalpha; // print booleans as true / false
//   cout << "Edge from 0 to 1: " << adjacencyMatrix.getEdge(0, 1) << endl;
//   cout << "No edge from 0 to 3: " << adjacencyMatrix.getEdge(0, 3) << endl;
// }
// =====
/*
```

In the above code snippet, we implemented an `AdjacencyMatrix` class, which uses these optimizations. The interface of the class consists of the functions `getEdge`, `addEdge`, and `removeEdge` that can be used to check and modify the edges. The actual representation of the adjacency matrix as a `vector<bool>` is an implementation detail hidden from the user of the class.

```
/*
// =====
// =====
/*
```

Adjacency List

Discover how graphs can be represented in code using adjacency lists.

We'll cover the following

Adjacency list

Implementing the adjacency list

The second option to represent graph edges in code is the adjacency list.

Adjacency list#

In this data structure, we store a list of each node's neighbors.

Let's look at our example graph again. We use the same integer encoding of nodes that we did for the adjacency matrix.

```
2
0
3
1
```

The example graph with integer node names

The node with index 1

```
1
has three neighbors: 0
```

```
0
, 2
2
```

and 3

```
3
. Its adjacency list looks like this:
```

```
0
2
3
null
```

Adjacency list for node 1

The order of the neighbors in the adjacency list does not matter, so this would be another representation of the adjacency list of node 1

```
1
:
```

```
2
0
3
```

null
Another representation of the adjacency list for node 1
The node 3
3
has no neighbors, so its adjacency list is the empty list.

null
The adjacency list of node 3 is the empty list.
Implementing the adjacency list#
For the implementation, we can store the adjacency list for each node as a vector<int> that stores the indices of the neighbors. In total, we have n
n
adjacency lists, that we can store in another, outer vector.
*/
=====// #include <vector>
// #include <iostream>
// using namespace std;

// int main()
// {
// int n = 4;
// vector<vector<int>> adjacencyList(n); // create empty adjacency list

// // add edges using a loop over a vector of (source, target) pairs
// vector<pair<int, int>> edges{{0, 1}, {1, 0}, {1, 2}, {1, 3}, {2, 0}, {2, 3}};
// for (auto [u, v] : edges)
// {
// adjacencyList[u].push_back(v);
// }

// cout << "Neighbors of node 1:";
// for (int u : adjacencyList[1])
// {
// cout << " " << u;
// }
// cout << endl;
// }
=====/*

Implementation of the adjacency list data structure

Note that the type vector<vector<int>>, which we use to store our adjacency lists, is the same as with our naive implementation of the adjacency matrix. The interpretation, however, is different:

For the adjacency matrix, each inner vector has n
n
elements, one for each node in the graph. Each element is either 0
0
or 1
1
, depending on whether there is an edge to that node.

For the adjacency list, each inner vector has only as many elements as the degree of its corresponding node, one for each neighbor. Each element is the node ID of that neighbor.

```
*/  
// ======  
/*
```

Representing Weighted Graphs

Explore how weighted graphs can be implemented.

We'll cover the following

Weighted adjacency matrix

What if weights can be zero?

Weighted adjacency list

Real numbered weights

Let's see how we can adapt the graph representations introduced in the previous lessons to weighted graphs.

Throughout this lesson, we'll use the following weighted graph as a running example:

```
0  
1  
5  
3  
2  
7  
3  
4  
11  
4
```

An example weighted graph

Weighted adjacency matrix#

To represent a weighted graph using its adjacency matrix, we can simply use a matrix of integers instead of booleans. The entry $A[i,j]$

$A[i,j]$

will be 0

0

when there is no edge from i

i

to j

j

, otherwise, it will be the weight of the edge (i, j)

(i,j)

. Here is the weighted adjacency matrix of our example graph:

For the implementation, we can simply reuse our naive implementation of the adjacency matrix as a nested vector of integers, this time corresponding to the weighted adjacency matrix.

```
*/  
// ======  
// #include <vector>  
// #include <tuple>  
// #include <iostream>  
// using namespace std;
```

```

// int main() {
//   int n = 4;
//   vector<vector<int>> adjacencyMatrix(n, vector<int>(n, 0)); // initialize with zeros

//   // add edges using a loop over a vector of (source, target, weight) tuples
//   vector<tuple<int, int, int>> edges {
//     make_tuple(0, 1, 5),
//     make_tuple(1, 0, 3),
//     make_tuple(1, 2, 7),
//     make_tuple(1, 3, 4),
//     make_tuple(2, 0, 11),
//     make_tuple(2, 3, 4)
//   };
//   for (auto [v, u, w] : edges) {
//     adjacencyMatrix[v][u] = w;
//   }
//   cout << "Edge from 0 to 1: weight " << adjacencyMatrix[0][1] << endl;
//   cout << "No edge from 0 to 3: " << adjacencyMatrix[0][3] << endl;
// }

// =====
/*

```

What if weights can be zero?

So far, we have implicitly assumed that all weights in our graph are nonzero. This way, we could use 0 as a special value to represent that there is no edge (i, j)

(i,j)
between two nodes.

In general, weighted graphs can also have edges with zero weights, which should be treated differently than absent edges.

Depending on the application, we might get away with using a different special value to represent absent edges, like -1. But in the most general case, our graph can contain edges of arbitrary integer weight, including zero and negative numbers.

In such a case, we can represent the graph using two matrices. The first one is the ordinary boolean adjacency matrix. The second one is the weight matrix that contains the weight of each edge (and has zero entries for absent edges).

```

*/
// =====
// #include <vector>
// #include <tuple>
// #include <iostream>
// using namespace std;

// int main()
// {
//   int n = 4;
//   vector<vector<int>> adjacencyMatrix(n, vector<int>(n, 0)); // unweighted adjacency matrix
//   vector<vector<int>> weightMatrix(n, vector<int>(n, 0)); // integer weight matrix

```

```

// // add edges using a loop over a vector of (source, target, weight) tuples
// vector<tuple<int, int, int>> edges{
//   make_tuple(0, 1, -2),
//   make_tuple(1, 0, 3),
//   make_tuple(1, 2, 0),
//   make_tuple(1, 3, 0),
//   make_tuple(2, 0, -6),
//   make_tuple(2, 3, 4)};
// for (auto [v, u, w] : edges)
// {
//   adjacencyMatrix[v][u] = 1;
//   weightMatrix[v][u] = w;
// }
// cout << "Edge from 1 to 2: adjacency " << adjacencyMatrix[1][2] << ", weight " << weightMatrix[1][2] << endl;
// cout << "No edge from 0 to 3: adjacency " << adjacencyMatrix[0][3] << endl;
// }
// =====
/*

```

In the code snippet above, we use the variable `adjacencyMatrix` to store the binary edge relation and the variable `weightMatrix` to store the actual weights of the edges. To get the weight of an edge, we first check whether the edge is present in the `adjacencyMatrix`, and then look up the weight in the `weightMatrix`.

We have chosen the most straightforward implementation of the adjacency matrix using `vector<vector<int>>` here, but the more efficient `vector<bool>` implementation, discussed in the Adjacency Matrix lesson, can also be used.

Weighted adjacency list#

Now, let's check how we can represent weighted graphs using adjacency lists.

In ordinary adjacency lists, each list entry was simply an integer, or the target of the corresponding edge. For weighted graphs, each entry becomes a pair of the target node and the weight of the edge.

Let's recall our example weighted graph.

The weighted adjacency list of node 1

1
is:

(0, 3)
(2, 7)
(3, 4)
null

The weighted adjacency list of node 1

For example, the second entry (2, 7)

(2,7)
means that there is an edge from node 1

1
to node 2

```
2
of weight 7
7
.
```

In code, we'll also represent the adjacency list for each node as a vector of pairs of integers. The adjacency lists for the whole graph are then once again represented as a nested vector of the individual lists.

```
/*
// =====
// #include <vector>
// #include <tuple>
// #include <iostream>
// using namespace std;

// int main() {
//   int n = 4;
//   vector<vector<pair<int, int>>> adjacencyList(n); // create empty adjacency list

//   // add edges using a loop over a vector of (source, target, weight) tuples
//   vector<tuple<int, int, int>> edges {
//     make_tuple(0, 1, 5),
//     make_tuple(1, 0, 3),
//     make_tuple(1, 2, 7),
//     make_tuple(1, 3, 4),
//     make_tuple(2, 0, 11),
//     make_tuple(2, 3, 4)
//   };
//   for (auto [v, u, w] : edges) {
//     adjacencyList[v].push_back({u, w});
//   }
//   cout << "Neighbors of node 1:";
//   for (int i = 0; i < adjacencyList[1].size(); ++i) {
//     auto [u, w] = adjacencyList[1][i];
//     cout << " " << u << " (weight " << w << ")";
//     if (i < adjacencyList[1].size() - 1) {
//       cout << ",";
//     }
//   }
//   cout << endl;
// }
// =====
/*
```

Real numbered weights#

The same data structures can be used to represent weighted graphs where the weights are real numbers. We simply need to use a floating-point type (like double) to represent the weights. For example,

`vector<vector<double>>` for the weighted adjacency matrix.

`vector<vector<pair<int, double>>>` for the weighted adjacency list.

Since floating-point numbers have a finite precision, they are only an approximation to real numbers. Hence, there might be rounding errors in computations.

The double type guarantees 15 significant digits of precision. If higher precision is needed, the only option is to use specialized big integer or arbitrary-precision floating-point types.

```
/*
// =====
// =====
```

```
/*
Comparison of Graph Representations
```

After studying both adjacency matrices and adjacency lists, let's see how they compare for common operations that we'll perform in graph algorithms.

In most applications of graph algorithms, the set of nodes V

is constant. No nodes are added or removed over time. We can therefore focus on operations that work on the edge set E

E

.

Memory usage#

The adjacency matrix stores at least one bit for every ordered pair of nodes (u, v)
 (u, v)

. Since there are exactly $(|V|^2 - |V|)$

$(|V|$

2

$-|V|)$

such pairs, the memory footprint is $\mathcal{O}(|V|^2)$

$O(|V|$

2

)

.

The adjacency list representation needs to store one list for each node in the graph, which takes at least $\mathcal{O}(|V|)$

$O(|V|)$

space. Additionally, every edge of the graph needs to be stored in one of the lists, taking up further $\mathcal{O}(|E|)$

$O(|E|)$

space. The total memory requirements are thus $\mathcal{O}(|V| + |E|)$

$O(|V|+|E|)$

.

In most cases, the adjacency list representation is more memory efficient, as it stores only the present edges, while the adjacency matrix also contains entries for all absent edges. The two representations are comparable only in dense graphs. Note that in a dense graph, the number of edges is close to the maximum number of edges $(|V|^2 - |V|$

$|V|$

2

$-|V|$

), so we have $|E| = \mathcal{O}(|V|^2)$

$|E| = O(|V|)$

2
)
.

Operations on single edges#

Let's assume that we are given two nodes u, v

u, v

, and we want to perform an operation on the edge (u, v)

(u, v)

, for example:

Test whether the edge exists.

Add or remove the edge.

Modify the edge's weight.

u

v

?

Performing an operation on the edge (u, v)

In the adjacency matrix representation, such operations take only constant time ($\mathcal{O}(1)$)

$O(1)$

, as we have random access to the adjacency status $A[u, v]$

$A[u, v]$

of the node pair.

In the adjacency list representation, we'll have to iterate over the elements of the adjacency list of node u

and check whether any of them leads to the node v

v

. In the worst case, we need to iterate over the whole list, which has $\text{deg}(u)$
 $\text{deg}(u)$

many elements. The operation thus takes $\mathcal{O}(\text{deg}(u))$

$O(\text{deg}(u))$

time.

a_1

...

v

...

$a_{\text{deg}(u)}$

null

Illustration of the adjacency list of node u where we need to iterate over the list to find v

The adjacency matrix is the clear winner here.

Operations on all outgoing edges#

The second kind of edge operation that we consider is the following. We are given a node u

u

and want to do something for each of its outgoing edges, such as:

list all the outgoing edges of u

u

.

add/remove some of these edges.
modify the weight of some of these edges.

u
v_1
?
v_2
?
v_3
?

Performing an operation on all neighboring edges of u

In the adjacency matrix representation, we need to check the row of the matrix corresponding to node u

for the nonzero entries. As the row has $|V|$
 $|V|$ elements, this takes $\mathcal{O}(|V|)$
 $\mathcal{O}(|V|)$ time.

In the adjacency list representation, we simply need to iterate over all entries of the adjacency list of u

u. This again takes $\mathcal{O}(|\text{deg}(u)|)$
 $\mathcal{O}(|\text{deg}(u)|)$ time.

The adjacency list representation is more efficient here, especially when the graph is sparse.

Summary#

All in all, the adjacency matrix is better when we often perform operations targeting a specific edge (u, v) .
. The adjacency list wins when considering the memory footprint and operations performed on all neighbors of a node. This is especially true in sparse graphs.

The following table summarizes the results:

Aspect	Adjacency matrix	Adjacency list
Memory usage	$\mathcal{O}(V ^2)$	$\mathcal{O}(V \cdot E)$
Single edge operation	$\mathcal{O}(1)$	$\mathcal{O}(\text{deg}(u))$
Neighboring edges operation	$\mathcal{O}(V)$	$\mathcal{O}(\text{deg}(u))$

So, which representation should we use after all? In general, the correct answer can only be “it depends on the application,” but we can still make a strong case for using adjacency lists as the default graph representation.

A central component of many graph algorithms is the traversal of a graph. In other words, we start from one or more nodes and explore the graph along the edges leaving these nodes. Therefore, operations on all of the outgoing edges of a node are very natural in graph algorithms. Additionally, most graphs encountered in real-life applications are sparse rather than dense.

Considering these facts, the adjacency list representation is preferable in most applications, and we will make use of it in the majority of graph algorithms discussed in this course. A notable exception is the Floyd Warshall algorithm, which can be implemented more efficiently using adjacency matrices.

Challenge: Converting Graph Representations

Practice converting adjacency matrices to adjacency lists and vice versa.

In the previous lessons, we have studied the two graph representations: adjacency matrix and adjacency list. Both are equally expressive and can be converted into the other.

Here is a function that converts an adjacency matrix into an adjacency list in the naive implementation:

```
/*
// =====
// #include <vector>
// #include <iostream>
// using namespace std;

// vector<vector<int>> adjacencyMatrixToList(vector<vector<int>> adjacencyMatrix)
//{
// int n = adjacencyMatrix.size();
// vector<vector<int>> adjacencyList(n);
// for (int i = 0; i < n; ++i)
// {
//     for (int j = 0; j < n; ++j)
//     {
//         if (adjacencyMatrix[i][j])
//         {
//             adjacencyList[i].push_back(j);
//         }
//     }
// }
// return adjacencyList;
//}

// int main()
//{
// vector<vector<int>> adjacencyMatrix{{0, 1, 0}, {1, 0, 1}, {1, 0, 0}};
// auto adjacencyList = adjacencyMatrixToList(adjacencyMatrix);
// cout << "Neighbors of node 1:";
// for (auto u : adjacencyList[1])
// {
// }
```

```

//   cout << " " << u;
// }
// cout << endl;
//=====
/*
As an exercise, implement a function which converts an adjacency list to an adjacency matrix.
*/
// =====
// #include <vector>
// #include <iostream>
// using namespace std;

// vector<vector<int>> adjacencyListToMatrix(vector<vector<int>> adjacencyList)
//{
//   // your code here
//   vector<vector<int>> adjacencyMatrix;
//   return adjacencyMatrix;
//}
//=====
/*
Depth-first Search (DFS)
Explore how to traverse graphs using Depth-first search.

```

We'll cover the following

Explanation of DFS

DFS results are not unique

Simple applications

Connectivity check

Cycle detection

After settling the basic terminology and learning about graph representations, let's dive into our first actual graph algorithm. In this section, we'll look into different ways of traversing graphs.

Unlike totally or partially ordered data structures, such as lists or trees, there is no canonical order in which the nodes of a graph should be explored. However, there are two standard algorithms to traverse graphs that define an order in which the nodes are visited: depth-first search (DFS) and breadth-first search (BFS). Although the algorithms are elementary, they can already be applied to solve a wide variety of graph problems with just basic modifications.

In this lesson, we will start by looking into depth-first search.

Explanation of DFS#

Depth-first search is a method to traverse a graph where the deepest not yet explored node is always explored first.

To illustrate the concept let's consider the following example graph:

b
d

c
a
e

Our 5 node example graph for running DFS.

1 of 11

The node e

e was not discovered during our search, because it is unreachable from our starting node a

a

. Depending on our application, we might want to continue running another DFS starting from node e

e

to explore the remaining nodes and edges.

To recap, during DFS we can classify nodes into three states:

Undiscovered.

In progress, has unchecked outgoing edges.

Finished.

We also classify edges as either:

Discovery edges, which leads to previously undiscovered nodes.

Back edges, which leads from in-progress nodes to other in-progress nodes.

Redundant edges, which leads from in-progress nodes to finished nodes.

In literature, redundant edges are sometimes further classified into forward edges and cross edges, but we do not need this distinction for this course.

When running DFS on an undirected graph, there can be no redundant edges.

All edges are either discovery edges or back edges.

DFS results are not unique#

Note that in general, the starting node will determine which nodes will be discovered during the DFS. Also, the order in which the neighbors of a node are visited can determine whether an edge is labeled as a discovery edge, back edge, or redundant edge. For example, here is the result of running DFS from starting node e

e

instead:

b
d
c
a
e

The result of running DFS from node

As we can see, all nodes are marked as finished (blue) when starting from e

e

, and there are no redundant edges. However, we still have back edges. If one DFS execution that visits all nodes finds back edges, then every such DFS execution will find back edges.

Simple applications#

The basic DFS algorithms can already be used to solve some elementary (but important!) graph problems.

Connectivity check#

We can use DFS to check whether an undirected graph is connected. Hence, we can also check whether a directed graph is weakly connected. To do so, we run DFS from an arbitrary starting vertex. Once the vertex is marked as finished (blue), we check if there are any white vertices left. If so, then the graph is definitely not connected. Otherwise, it is at least weakly connected.

We can also apply DFS to find out whether a graph is strongly connected, but the algorithm is a bit more complex. We'll discuss this algorithm in its own lesson.

Cycle detection#

We can use DFS to detect cycles in graphs. For this, we'll start with DFS from an arbitrary vertex. After finishing that vertex, we'll continue with DFS from a remaining white vertex, until all vertices are finished (blue).

If we find a back edge during the execution of DFS that leads from the current node to another node that is still in progress (gray), we've found a cycle in the graph. This indicates that if a node is still in progress, there must be a path from it to the current node, and the back edge thus completes a cycle. If such a back edge is never discovered, the graph is acyclic.

```
/*
// =====
// =====
/*
```

Implementation of DFS

Learn how to implement depth first search.

We'll cover the following

Implementation notes

Avoiding stack overflows

Implementing cycle detection with DFS

Runtime analysis

Trying out the cycle detection

Detecting cycles in undirected graphs

In this lesson, we'll solve the problem of detecting cycles in directed graphs by implementing DFS.

Implementation notes#

The easiest way to implement DFS is as a recursive procedure:

Mark the current node as "in progress."

Recursively process all nodes behind discovery edges.

Mark the current node as done.

Due to the recursion occurring between starting and finishing a node, we automatically explore deeper nodes first before going up to shallower nodes again.

So, in principle, our depth-first search procedure should be a recursive function

```
void dfs(int u) {}  
that performs DFS from the starting node u.
```

As we mostly need to perform operations on all neighboring edges of a node, we'll make use of the adjacency list graph representation. We are considering unweighted graphs here, but the implementation for weighted graphs is very similar. We'll simply ignore the values of the weights.

But now our DFS procedure additionally needs to access some "global" state such as:

the adjacency list of the graph.
the current state or color of each node.
One option would be to pass this state to the dfs routine additionally, like in

```
void dfs(int u, vector<vector<int>> const &adjacencyList, vector<NodeState> &state) {}
```

However, this is unwieldy, especially when we want to keep track of more state during the search, such as node predecessors or distances.

Instead, we'll implement our cycle detection algorithm as a class where this state is stored in the member variables.

Avoiding stack overflows#

There is one downside of implementing DFS recursively. As each recursive function call creates a stack frame, it is possible to run into stack overflows on very large graphs.

```
dfs(u1)  
dfs(u2)  
dfs(u3)  
...  
dfs(u1000000)  
dfs(u1000001)
```

An easy workaround for this problem is to simply increase the maximum stack size, for example, using the ulimit -s command on Linux systems.

But some operating systems do not support raising the stack size past a hard limit. In that case, the only remedy is to implement DFS iteratively by keeping track of a vector of "in progress" nodes. This vector can be allocated on the heap and thus we can avoid stack overflow issues.

Implementing cycle detection with DFS#

Let's start by setting up a basic class to run the cycle detection.

```
#include <vector>  
#include <iostream>  
using namespace std;  
  
// shorthand for adjacency list type  
typedef vector<vector<int>> AdjacencyList;
```

```

// define an enum for node states during DFS execution
enum class NodeState {UNVISITED, IN_PROGRESS, FINISHED};

class CycleDetector {
private:
    const AdjacencyList adjacencyList;
    vector<NodeState> nodeStates;
    bool cycleFound = false;
public:
    CycleDetector(AdjacencyList _adjacencyList)
        : adjacencyList {_adjacencyList}
        // initialize all nodes as unvisited
        , nodeStates {vector<NodeState>(adjacencyList.size(), NodeState::UNVISITED)}
    {}
};

```

To keep track of the node states during DFS, we have defined the enumeration `NodeState`.

The states correspond to the colors white, gray, and blue from the previous lesson.

In our class, we use a vector to keep track of the current state of each node. Initially, all nodes are in the `UNVISITED` state. We also maintain a boolean variable `cycleFound`, initially false, which keeps track of whether we have discovered a cycle yet.

Next, let's write the DFS routine.

```

void dfs(int u) {
    // mark current node as in progress
    this->nodeStates[u] = NodeState::IN_PROGRESS;

    for (int v : this->adjacencyList[u]) {
        switch (this->nodeStates[v]) {
            // discovery edge: recursively call dfs
            case NodeState::UNVISITED: dfs(v); break;
            // back edge: mark cycle as found
            case NodeState::IN_PROGRESS: this->cycleFound = true; break;
            // redundant edge: skip
            case NodeState::FINISHED: break;
        }
    }

    // mark current node as done
    this->nodeStates[u] = NodeState::FINISHED;
}

```

After marking the current node `u` as `IN_PROGRESS`, we'll check all of its outgoing edges. Our action will depend on the type of each edge:

For discovery edges leading to unvisited nodes, we'll recursively call `dfs`.

For back edges leading to “in progress” nodes, we'll note that we have found a cycle, but will not follow them. We'll ignore redundant edges leading to finished nodes.

Finally, we'll mark the current node as FINISHED.

Now, we only need to write a function that uses dfs to discover whether there is a cycle anywhere in the graph.

```
bool containsCycle() {
    for (int u = 0; u < (int)this->adjacencyList.size(); ++u) {
        // skip nodes that were already discovered
        if (this->nodeStates[u] == NodeState::FINISHED) continue;
        dfs(u);
    }
    return this->cycleFound;
}
```

Our containsCycle function calls dfs from every possible starting vertex in the graph. This is necessary because the graph might be disconnected. Trying dfs from one vertex might not find an existing cycle in another connected component.

However, to save time, we won't consider vertices in the FINISHED state, as we have already explored them and all of their neighbors. In the end, we'll simply return cycleFound. It was set to true only if we have ever found a cycle in the graph.

The following code snippet contains the complete implementation of our CycleDetector class again for reference.

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

typedef vector<vector<int>> AdjacencyList;
enum class NodeState {UNVISITED, IN_PROGRESS, FINISHED};

class CycleDetector {
private:
    const AdjacencyList adjacencyList;
    vector<NodeState> nodeStates;
    bool cycleFound = false;
public:
    CycleDetector(AdjacencyList _adjacencyList)
        : adjacencyList {_adjacencyList}
        , nodeStates {vector<NodeState>(adjacencyList.size(), NodeState::UNVISITED)}
    {}

    bool containsCycle() {
        for (int u = 0; u < (int)this->adjacencyList.size(); ++u) {
            if (this->nodeStates[u] == NodeState::FINISHED) continue;
            dfs(u);
        }
        return this->cycleFound;
    }

    void dfs(int u) {
```

```

this->nodeStates[u] = NodeState::IN_PROGRESS;

for (int v : this->adjacencyList[u]) {
    switch (this->nodeStates[v]) {
        case NodeState::UNVISITED: dfs(v); break;
        case NodeState::IN_PROGRESS: this->cycleFound = true; break;
        case NodeState::FINISHED: break;
    }
}

this->nodeStates[u] = NodeState::FINISHED;
}
};


```

Runtime analysis#

Let's analyze the time complexity of our DFS implementation. In the worst case, a single call to `dfs` might inspect every edge and explore the whole graph, which takes $\mathcal{O}(|E|)$ time.

In our cycle detection, we'll run DFS from all starting nodes that have not been previously visited. This is a common technique in graph traversal code. Since we only start subsequent DFS runs on unvisited nodes, we'll still inspect each edge of the graph once. This entails that the total runtime of all these `dfs` calls is actually still $\mathcal{O}(|E|)$

$\mathcal{O}(|E|)$

However, since we're also iterating over each node to check whether it was visited yet, we have an additional $\mathcal{O}(|V|)$

$\mathcal{O}(|V|)$

work to do. The total complexity of exploring the whole graph using DFS is thus $\mathcal{O}(|V| + |E|)$

$\mathcal{O}(|V| + |E|)$

Note: In most graphs we have $|V| \leq |E|$

$|V| \leq |E|$

, and thus $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$

$\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$

Trying out the cycle detection#

Now, let's verify that our implementation works. We'll run it on our DFS example graph of the previous lesson.

We've already renamed the nodes to integer indices from zero to four.

The graph contains two cycles,

$0 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0$

$0 \rightarrow 1 \rightarrow 3 \rightarrow 0$

so we expect our algorithm to return true.

```

int main() {
    int n = 5;
    AdjacencyList adjacencyList(n);
    vector<pair<int, int>> edges {{0, 1}, {0, 2}, {1, 3}, {2, 1}, {3, 0}, {4, 1}};
    for (auto [u, v] : edges) {
        adjacencyList[u].push_back(v);
    }

    CycleDetector cycleDetector(adjacencyList);
    cout << "Contains cycle: " << cycleDetector.containsCycle() << endl;
}

```

Now, let's make a slight modification to the graph and remove the edge (1, 3)
 $(1,3)$
(marked in red above). Since both cycles in the original graph are using that edge, our algorithm will now return false.

```

int main() {
    int n = 5;
    AdjacencyList adjacencyList(n);
    vector<pair<int, int>> edges {{0, 1}, {0, 2}, {2, 1}, {3, 0}, {4, 1}};
    for (auto [u, v] : edges) {
        adjacencyList[u].push_back(v);
    }

    CycleDetector cycleDetector(adjacencyList);
    cout << "Contains cycle: " << cycleDetector.containsCycle() << endl;
}

```

Detecting cycles in undirected graphs#

The algorithm we implemented above only correctly detects cycles in directed graphs. For undirected graphs, we'll need to modify our DFS implementation slightly to account for the fact that cycles in undirected graphs need to have at least three nodes.

In an undirected graph, if a discovery (half-)edge (u, v)
 (u,v)
leads us to a node v
 v , then we should not count the symmetric half-edge (v, u)
 (v,u)
as a back edge. Otherwise, every undirected edge would be counted as a cycle.

```

a
b
u
v

```

Left: A two node cycle in a directed graph. Right: a single undirected edge is not a cycle.
Thankfully, we can fix that by making a small change to our implementation of dfs. Whenever a discovery edge (u, v)
 (u,v)
leads to the node v

```

v
, let's call the node u
u
the parent of v
v
. The starting node of the DFS has no parent.

```

In our implementation, we'll then need to keep track of the parent, and never consider edges leading back to the parent as back edges. When there is no parent (in the starting node), we'll use a dummy value of -1 for the parent.

```

// keep track of parent vertex, default -1 (no parent)
void dfs(int u, int parent = -1) {
    this->nodeStates[u] = NodeState::IN_PROGRESS;

    for (int v : this->adjacencyList[u]) {
        // skip edge back to the parent
        if (v == parent) continue;
        switch (this->nodeStates[v]) {
            // call dfs and pass the parent
            case NodeState::UNVISITED: dfs(v, u); break;
            case NodeState::IN_PROGRESS: this->cycleFound = true; break;
            case NodeState::FINISHED: break;
        }
    }

    this->nodeStates[u] = NodeState::FINISHED;
}

*/
// =====
// =====
/*
Breadth First Search (BFS)
Discover how to traverse graphs using breadth first search.

```

We'll cover the following

Explanation of BFS

Edge types in BFS

Application: Shortest paths

The second important traversal algorithm for graphs is breadth-first search (BFS). The main concept of BFS is that the closest not yet explored node is explored first. This is exactly the opposite strategy compared to DFS.

Explanation of BFS#

Let's execute the BFS algorithm on an example graph as we did for DFS.

```

a
b

```

c
d
e
f

An example graph for running BFS. We'll start from node a
1 of 8

In the end result of a BFS execution, all nodes are marked as finished and all distances to the starting node are recorded.

We'll only discover nodes reachable from the starting vertex, just like we saw for DFS. If we want to explore the whole graph, running further BFS executions from yet undiscovered vertices is needed.

Edge types in BFS#

When discussing DFS, we'll split up the edges into different types: discovery edges, back edges, and redundant edges. The back edges were very useful in discovering cycles. So why are we not doing the same in BFS?

It is not easy to identify back edges in BFS, due to the different node traversal order. When we find an edge that leads "back" to an already discovered or finished node, we cannot be sure that this guarantees a cycle.

For example, check out the following graph where BFS was started from node a, and the current node is d. We find edges leading from d to the finished node c and the discovered node e, but neither of them are part of a cycle. In fact, the graph is acyclic.

a (0)
c (1)
b (1)
d (2)
e (2)

Identifying cycles through back edges in BFS is not easily possible.

Since the detailed edge classifications for BFS are not as useful, they are usually omitted. However, keeping track of which edges are discovery edges can be important.

Application: Shortest paths#

We already saw that executing BFS yields us the distance of each node to the starting vertex. We can also use BFS to compute the shortest path between two vertices u

u
and v
v
.

To do so, we simply run BFS from the starting vertex u

u
. Each time we discover a new node b
b
from the current node a

a
, we mark a
a
as the parent of b
b
(just like we did in our implementation of DFS for undirected graphs).

Once we reach node v

v
, we can go back to its parent, grandparent, and so on, until we reach node u
u
again. The sequence of ancestors, from oldest to youngest, corresponds to the shortest path from u
u
to v
v
.

The following image shows the result of our BFS on the example graph, with all discovery edges marked in green (signifying the parent relationship between nodes).

a (0)
b (1)
c (1)
d (2)
e (2)
f (3)

BFS result with marked discovery edges

To obtain the shortest path from a to e, we need to go back along discovery edges from e to a. The parent of e is c, and the grandparent is a itself. Thus, a -> c -> e is a shortest path from a to e.

Shortest paths are not always unique, as the following graph shows:

a
b
c
d

Shortest paths are not always unique.

Let's say we run BFS from node a

a
. There are two shortest paths from a
a
to d
d
:

a -> b -> d
a -> c -> d

It is not certain which path BFS discovers, because both b and d could be the second node to explore.

Implementation of BFS

Learn how to implement Breadth-first Search.

We'll cover the following

Implementation Notes

Implementing shortest paths using BFS

Trying out the shortest path computation

In this lesson, we will implement Breadth-first Search to solve the shortest path problem for unweighted graphs.

Implementation Notes#

When implementing DFS previously, we wanted to explore the deepest unexplored node first. This corresponds to using a stack for the unexplored nodes, which is a Last-In-First-Out (LIFO) data structure. Usually the stack is implicit in the recursive implementation.

For implementing BFS, we want to do the opposite and explore the closest unexplored node first. Hence, we'll use a queue to store the unexplored nodes, which is a First-In-First-Out (FIFO) data structure.

u_1 (front)

u_2

u_3 (back)

The queue of unexplored nodes

The illustration above shows what the queue of unexplored nodes can look. When new vertices are discovered, they'll be pushed to the back of the queue. However, when we select the next node to explore, we'll bring it from the front of the queue.

Implementing shortest paths using BFS#

We'll implement a class for the shortest path computation to keep track of the state of the search as we did for BFS. For each node, we'll store"

its distance to the starting vertex.

the parent vertex from which we discovered the node.

Again, we're using the adjacency list representation, which is preferable for graph traversals.

```
/*
// =====
// #include <queue>
// #include <vector>
// #include <algorithm>
// using namespace std;

/// shorthand for adjacency list type
// typedef vector<vector<int>> AdjacencyList;

// class ShortestPaths
// {
// private:
//   AdjacencyList adjacencyList;
//   vector<int> distances;
//   vector<int> parents;
//   // undiscovered nodes will have -1 as distance / parent
//   static constexpr int UNKNOWN = -1;
```

```

// public:
// ShortestPaths(AdjacencyList &_adjacencyList)
//   : adjacencyList(_adjacencyList), distances{vector<int>(_adjacencyList.size(), UNKNOWN)},
parents{vector<int>(_adjacencyList.size(), UNKNOWN)}
// {
// }
// =====
/*

```

We are using two vector<int> variables, parents and distances, to keep track of the BFS state. Initially, they are both filled with the value -1, which means “yet unknown.”

Unlike in our DFS implementation, we do not explicitly keep track of which nodes are unvisited, in progress, or finished, because we do not want to classify edges. Still, the nodes with a parent of -1 are the yet unvisited nodes.

Next, let’s write the bfs method.

```

*/
// =====
// void
// bfs(int start)
// {
//   // // initialization
//   this->distances.assign(this->distances.size(), UNKNOWN);
//   this->parents.assign(this->parents.size(), UNKNOWN);
//   queue<int> unexplored;
//   unexplored.push(start);
//   distances[start] = 0;
//   parents[start] = start;

//   // // bfs main loop
//   while (!unexplored.empty())
//   {
//     int u = unexplored.front();
//     unexplored.pop();
//     for (int v : this->adjacencyList[u])
//     {
//       if (this->parents[v] == UNKNOWN)
//       {
//         parents[v] = u;
//         distances[v] = distances[u] + 1;
//         unexplored.push(v);
//       }
//     }
//   }
// =====
/*
```

First, we need to do some initialization (lines 3-8). We’ll fill the parents and distances variables with -1 to support running the method multiple times. Next, we’ll initialize our queue of unexplored nodes with the

starting vertex and assign its distance and parent. For the special case of the starting vertex, we'll say that it is its own parent.

In the BFS main loop (lines 11-21), we'll continue exploring nodes as long as our queue of unexplored nodes is not empty. We'll bring the closest unexplored node from the front of the queue and check all of its neighbors. For each unvisited neighbor, we'll assign the distance and parent. Then we'll push it to the back of our queue of unexplored nodes.

As we assign the parent before pushing to the queue, we can be sure that each node is pushed once at most, so our while-loop will terminate.

All that's left is to actually compute the shortest path based on the BFS result.

```
/*
// =====
// vector<int> computeShortestPath(int start, int end)
//{
//  this->bfs(start);
//  // end was not discovered -> no path
//  if (this->parents[end] == UNKNOWN)
//  {
//    throw NoPathExistsException();
//  }

//  vector<int> path;
//  path.push_back(end);
//  int current = end;
//  // go back along parents to build the path
//  while (current != start)
//  {
//    current = parents[current];
//    path.push_back(current);
//  }
//  // path was assembled in reverse order
//  std::reverse(path.begin(), path.end());
//  return path;
//}
// =====
/*
```

To compute the shortest path from a vertex start to a vertex end, we'll first run bfs from the starting vertex. If we never discover the end vertex, then there is no path to it from the start vertex because the graph is disconnected. In this case, we'll throw an exception that can be defined like so:

```
class NoPathExistsException: public runtime_error {
public:
  NoPathExistsException() : runtime_error("No path exists between the nodes.") {}
};
```

If a path exists, we can recover it from the parents relation. Starting at the end vertex, we'll go back using parents until we reach the start vertex. As the path is constructed in reverse order, we'll call the reverse function before returning it.

The following code snippet contains our completed ShortestPaths class.

```
/*
// =====
// #include <queue>
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef vector<vector<int>> AdjacencyList;

// class NoPathExistsException : public runtime_error
//{
// public:
//  NoPathExistsException() : runtime_error("No path exists between the nodes.") {}
//};

// class ShortestPaths
//{
// private:
//  AdjacencyList adjacencyList;
//  vector<int> distances;
//  vector<int> parents;
//  static constexpr int UNKNOWN = -1;

//  void bfs(int start)
//  {
//    this->distances.assign(this->distances.size(), UNKNOWN);
//    this->parents.assign(this->parents.size(), UNKNOWN);
//    queue<int> unexplored;
//    unexplored.push(start);
//    distances[start] = 0;
//    parents[start] = start;

//    while (!unexplored.empty())
//    {
//      int u = unexplored.front();
//      unexplored.pop();
//      for (int v : this->adjacencyList[u])
//      {
//        if (this->parents[v] == UNKNOWN)
//        {
//          parents[v] = u;
//          distances[v] = distances[u] + 1;
//          unexplored.push(v);
//        }
//      }
//    }
//  }

// public:
//  ShortestPaths(AdjacencyList &_adjacencyList)
```

```

//      : adjacencyList{_adjacencyList}, distances{vector<int>(_adjacencyList.size(), UNKNOWN)},
parents{vector<int>(_adjacencyList.size(), UNKNOWN)}
// {
// }

// vector<int> computeShortestPath(int start, int end)
// {
//   this->bfs(start);
//   if (this->parents[end] == UNKNOWN)
//   {
//     throw NoPathExistsException();
//   }

//   vector<int> path;
//   path.push_back(end);
//   int current = end;
//   while (current != start)
//   {
//     current = parents[current];
//     path.push_back(current);
//   }

//   std::reverse(path.begin(), path.end());
//   return path;
// }
// =====
/*

```

The complexity analysis for BFS is the same as for DFS. Performing a BFS from one node takes $\mathcal{O}(|E|)$ time, and running BFS from all nodes takes $\mathcal{O}(|V| + |E|)$. Hence, our shortest path computation from one starting node has complexity $\mathcal{O}(|E|)$.

Trying out the shortest path computation#

Let's try out our shortest path implementation on the BFS example graph from the previous lesson.

We want to compute the shortest path from starting vertex 0

0

to goal vertex 4

4

, marked in red above.

```

*/
// =====
// int main() {
//   int n = 6;
//   AdjacencyList adjacencyList(n);
//   vector<pair<int, int>> edges {{0, 1}, {0, 2}, {1, 3}, {2, 4}, {3, 0}, {3, 4}, {4, 3}, {4, 5}};

```

```

// for (auto [u, v] : edges) {
//   adjacencyList[u].push_back(v);
// }

// ShortestPaths shortestPaths(adjacencyList);
// vector<int> path04 = shortestPaths.computeShortestPath(0, 4);
// std::cout << "Shortest path from 0 to 4: ";
// for (int u : path04) {
//   std::cout << u << " ";
// }
// =====
/*

```

We correctly receive the shortest path $0 \rightarrow 2 \rightarrow 4$ as the output of our program.

We can also try to compute the shortest path between nodes that have no path between them.

```

*/
// =====
// int main() {
//   int n = 6;
//   AdjacencyList adjacencyList(n);
//   vector<pair<int, int>> edges {{0, 1}, {0, 2}, {1, 3}, {2, 4}, {3, 0}, {3, 4}, {4, 3}, {4, 5}};
//   for (auto [u, v] : edges) {
//     adjacencyList[u].push_back(v);
//   }

//   ShortestPaths shortestPaths(adjacencyList);
//   try {
//     vector<int> path50 = shortestPaths.computeShortestPath(5, 0);
//   } catch (NoPathExistsException exc) {
//     cout << "No path from 5 to 0.";
//   }
// }
// =====
/*

```

Application 1: Topological Sort

Practice applying DFS to compute topological sorts.

We'll cover the following

The topological sorting problem

Computing topological sorts with DFS

Implementation

Trying out the topological sort implementation

In this lesson, we'll apply the DFS algorithm to solve a task ordering problem.

The topological sorting problem#

Our goal is to order a set of tasks in a way that respects the dependencies between them. For example, let's assume that you're planning your activities for the day. Unfortunately, there is a collection of household chores that you'll need to complete first, such as washing the dishes, withdrawing cash, grocery shopping, and

cooking. Some of these activities may depend on each other. For instance, you need to shop for groceries before you can cook dinner. And you need to withdraw cash before you can shop for groceries.

These tasks and dependencies can be modeled in a task graph. Each task is a node of the graph, and there is an edge (u, v)

(u, v)

whenever task v

v

directly depends on task u

u

. The following figure shows an example task graph.

Whenever there is a path $u \rightarrow v$

$u \rightarrow v$

in the task graph, the task v

v

(directly or indirectly) depends on the task u

u

. This means that we have to perform task u

u

before we can perform task v

v

!

We are looking for an order in which we can perform the tasks, such that all such dependencies are respected. Speaking in graph-theoretic terms, we are looking for an ordering of the nodes with this property:

Whenever there is an edge (u, v)

(u, v)

, then u

u

occurs before v

v

in the ordering.

Such an ordering is called a topological sort.

For example, the order

withdraw cash \rightarrow shop groceries \rightarrow wash dishes \rightarrow cook lunch \rightarrow cook dinner

is a valid topological sort. There can be more than one topological sort, the following order is another one:

wash dishes \rightarrow withdraw cash \rightarrow shop groceries \rightarrow cook lunch \rightarrow cook dinner

Meanwhile, the order

cook lunch \rightarrow withdraw cash \rightarrow shop groceries \rightarrow wash dishes \rightarrow cook dinner

is not a topological sort. For example, there is an edge (wash dishes, cook lunch), but cook lunch appears before wash dishes in the order.

It is clear that a topological sort cannot exist when the graph contains cycles. With circular dependencies, there is no way we can complete all the tasks.

Thus, we need to verify that the task graph is acyclic when computing a topological sort. We can do so using our CycleDetector class from the DFS lessons or by checking for back edges while we compute the topological sort. For simplicity, we will assume that the input graph is acyclic for the rest of this lesson.

Computing topological sorts with DFS#

There is a clever way to compute a topological sort using DFS. We'll perform a DFS on the graph, possibly multiple times from different starting nodes, until all vertices are marked as finished. Whenever we finish a node, we'll add it to a list. In the end, the reversed list will be a topological sort. In other words:

Ordering the nodes in the reversed finishing order of DFS is a topological sort.

Let's verify why this is the case. We need to make sure that the defining property of a topological sort holds:

Whenever there is an edge (u, v)

(u, v)

, then u

u

occurs before v

v

in the ordering.

So, let's say that we have an edge from u

u

to v

v

.

We want to show that u

u

occurs before v

v

in the reversed finishing order. In other words, v

v

must be finished before u

u

. There are two cases:

v

v

is discovered before u

u

in the DFS. As the graph is acyclic, there is no way to reach u

u

from v

v

. So v

v

will be marked as finished before u

u

is even discovered.

u

```

u
is discovered before v
v
in the DFS. As there is an edge (u,v)
(u,v)
, DFS will completely explore and finish the deeper node v
v
before finishing u
u
.
Therefore, whenever there is an edge (u, v)
(u,v)
, v
v
will be finished before u
u
in the DFS. This means that the reverse finishing order is in fact a topological sort.

```

Implementation#

The implementation of the topological sort is a slight modification of our CycleDetector DFS implementation. We'll again start by setting up a class for it.

```

*/
// =====
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef vector<vector<int>> AdjacencyList;
// enum class NodeState {UNVISITED, IN_PROGRESS, FINISHED};

// class TopologicalSort {
// private:
//   const AdjacencyList adjacencyList;
//   vector<NodeState> nodeStates;
//   // used to store the DFS finishing order
//   vector<int> finishOrder;
// public:
//   TopologicalSort(AdjacencyList _adjacencyList)
//     : adjacencyList {_adjacencyList}
//     , nodeStates {vector<NodeState>(adjacencyList.size(), NodeState::UNVISITED)}
//   {}
// };
// =====
/*

```

The new addition compared to our CycleDetector class is the initially empty vector<int> finishOrder (line 13) that will be used to store the DFS finishing order.

Next up is the dfs routine:

```

*/
// =====
// void dfs(int u)

```

```

//{
//  this->nodeStates[u] = NodeState::IN_PROGRESS;

//  for (int v : this->adjacencyList[u])
//  {
//    switch (this->nodeStates[v])
//    {
//      // discovery edge: recursively call dfs
//      case NodeState::UNVISITED:
//        dfs(v);
//        break;
//      // back edge: cycle found! There is no topological sort.
//      case NodeState::IN_PROGRESS:
//        throw GraphIsCyclicException();
//        break;
//      // redundant edge: skip
//      case NodeState::FINISHED:
//        break;
//    }
//  }

//  this->nodeStates[u] = NodeState::FINISHED;
//  // add to finishing order
//  this->finishOrder.push_back(u);
//}

//=====
/*

```

There are two notable changes to our previous DFS implementation. First, we'll add each node to the finishOrder vector once it is finished (line 17). Second, we'll throw an `Exception` when we encounter a back edge, as cyclic graphs have no topological sort (line 9).

All that's left now is to run dfs from all starting nodes to compute the topological sort.

```

*/
//=====
// vector<int> computeTopologicalSort {
//   for (int u = 0; u < (int)this->adjacencyList.size(); ++u) {
//     if (this->nodeStates[u] == NodeState::FINISHED) continue;
//     dfs(u);
//   }
//   // reverse finishing order
//   reverse(this->finishOrder.begin(), this->finishOrder.end());
//   return this->finishOrder;
//}
//=====
/*

```

After running the DFS from all starting nodes, we'll still need to reverse the finishing order (line 7) to end up with the final topological sort.

The following snippet contains the complete implementation for reference:

```
#include <vector>
```

```

#include <algorithm>
using namespace std;

typedef vector<vector<int>> AdjacencyList;
enum class NodeState {UNVISITED, IN_PROGRESS, FINISHED};

class GraphIsCyclicException: public runtime_error{
public:
    GraphIsCyclicException() : runtime_error("The graph contains a cycle.") {}
};

class TopologicalSort {
private:
    const AdjacencyList adjacencyList;
    vector<NodeState> nodeStates;
    vector<int> finishOrder;
public:
    TopologicalSort(AdjacencyList _adjacencyList)
        : adjacencyList {_adjacencyList}
        , nodeStates {vector<NodeState>(adjacencyList.size(), NodeState::UNVISITED)}
    {}

    vector<int> computeTopologicalSort() {
        for (int u = 0; u < (int)this->adjacencyList.size(); ++u) {
            if (this->nodeStates[u] == NodeState::FINISHED) continue;
            dfs(u);
        }
        reverse(this->finishOrder.begin(), this->finishOrder.end());
        return this->finishOrder;
    }

    void dfs(int u) {
        this->nodeStates[u] = NodeState::IN_PROGRESS;

        for (int v : this->adjacencyList[u]) {
            switch (this->nodeStates[v]) {
                case NodeState::UNVISITED: dfs(v); break;
                case NodeState::IN_PROGRESS: throw GraphIsCyclicException(); break;
                case NodeState::FINISHED: break;
            }
        }

        this->nodeStates[u] = NodeState::FINISHED;
        this->finishOrder.push_back(u);
    }
};

/*
// =====
// =====
*/

```

Let's analyze the running time of our algorithm. First, we'll perform a DFS from all nodes, which takes $\mathcal{O}(|V| + |E|)$ time. Finally, we'll reverse the list of nodes, which takes $\mathcal{O}(|V|)$ time. The final runtime is thus $\mathcal{O}(|V| + |E|)$

Trying out the topological sort implementation#

Let's run the implementation on our example task graph from above. After transforming the node names to integers, it will look like this:

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main() {
//   int n = 5;
//   AdjacencyList adjacencyList(n);
//   vector<pair<int, int>> edges {{0, 1}, {0, 4}, {2, 4}, {2, 1}, {3, 0}, {4, 1}};
//   for (auto [u, v] : edges) {
//     adjacencyList[u].push_back(v);
//   }

//   TopologicalSort topologicalSort(adjacencyList);
//   vector<int> topSort = topologicalSort.computeTopologicalSort();
//   cout << "Topological sort:";
//   for (int u : topSort) {
//     cout << " " << u;
//   }
// }
// =====
/*
```

The result 3, 2, 0, 4, 1 is indeed a valid topological sort.

```
/*
// =====
// =====
/*
```

Application 2: Strongly Connected Components

Learn how to split a graph into its strongly connected components.

We'll cover the following

Strongly connected components

Kosaraju's algorithm

In this lesson, we use graph traversal to separate a directed graph into its strongly connected components. Such a decomposition can reveal useful information on the graph structure and can be the first step for more involved graph algorithms.

Strongly connected components#

Recall that a directed graph is called strongly connected when any pair of nodes can reach each other. Even if the whole graph is not strongly connected, it can be split up into parts called its strongly connected components (SCCs).

The following illustration shows a graph G

G

with three strongly connected components, marked in green, red, and blue, respectively.

2

3

5

0

6

1

4

A graph with three strongly connected components

For example, the nodes 0

0

and 4

4

are in the same SCC, as we can reach node 0

0

from node 4

4

and also reach node 4

4

from node 0

0

(via node 6

6

). On the other hand, nodes 4

4

and 5

5

are in distinct SCCs: although we can go from node 4

4

to node 5

5

, there is no way to go from node 5

5

to node 4

4

.

After identifying strongly connected components in a graph $G = (V, E)$

$G = (V, E)$

, we can contract them to form a new graph G'

G

'

, called the contracted component graph of G

G

. The nodes of the contracted component graph are the SCCs of G

G

. There is an edge between SCCs X

X

and Y

Y

whenever there are $u \in X, v \in Y$

$u \in X, v \in Y$

such that $(u, v) \in E$

$(u, v) \in E$

. The following figure shows the contracted component graph G'

G

,

of our example graph G

G

from above.

{0,1,4,6}

{5}

{2,3}

The graph after contracting strongly connected components

An important property of G'

G

,

is that it is always acyclic. Imagine there were two SCCs X

X

, Y

Y

such that X

X

could reach Y

Y

and Y

Y

could reach X

X

- then X

X

and Y

Y

would actually be part of the same SCC. Therefore, there cannot be a cycle in G'

G

,

.

The contracted component graph can be seen as a high-level overview of the structure of a graph. It has many applications in answering structural questions about G

G

.

For example, consider the question to identify a minimal set M

M

of nodes such that all vertices of G

G

are reachable from M

M

. At first glance, it might seem like a difficult optimization problem, but after computing the contracted component graph of G

G

, it becomes easy to solve. Let's call a node of G'

G

'

that has no incoming edges a source component. We can define a set M

M

by simply taking one arbitrary node out of each source component.

Clearly, we need to choose at least one node out of each source component, as the nodes in a source component cannot be reached from any other component. At the same time, we can reach all other SCCs (and thus the nodes inside them) from the source components. So, we can reach all nodes from M

M

, and it has minimal size, as required.

In our example graph, we could, for example, select M = \{1,3\}

M=\{1,3\}

so all nodes in the graph can be reached from just two starting nodes. In total, there are eight different possible choices for M

M

.

2

3

5

0

6

1

4

All nodes of G can be reached from M = {1,3}

The good news is that computing SCCs can be done efficiently using the graph traversal algorithms we have learned in this chapter. There are several well-known algorithms for it, such as Kosaraju's algorithm and Tarjan's algorithm. We'll focus on the former, as it is a bit more intuitive and builds upon our topological sorting application from the previous lesson.

Kosaraju's algorithm#

Kosaraju's algorithm applies two passes of DFS on the whole graph to compute its SCCs. The first pass is an ordinary forward DFS, while the second pass is a backward DFS, which follows the edges in the reverse direction.

During the first DFS, we'll keep track of the finishing order of nodes, just like in topological sort. Unlike in a topological sort, we'll accept that the graph can contain cycles, so we won't stop when encountering back edges.

The key idea is that the second, backward DFS chooses its starting nodes in the reverse finishing order of the forward DFS. When following this order, each starting node of the second DFS path will discover exactly one strongly connected component of the graph.

Before covering in detail why this works, let's start with executing the algorithm on our example graph, starting with the forward DFS. We begin with running DFS from node 0

0

.

2

3

5

0

6

1

4

The example graph after forward DFS from node 0

The finishing order of this first DFS is 5, 4, 1, 6, 0

5,4,1,6,0

. Because there are still unvisited nodes, we'll run another forward DFS from starting vertex 2

2

.

2

3

5

0

6

1

4

The example graph after another forward DFS from node 2

The finishing order of the second forward DFS is 3, 2

3,2

. In total, this gives us the complete reverse finishing order

2, 3, 0, 6, 1, 4, 5.

2,3,0,6,1,4,5.

Now, we prepare for the backward DFS by reversing all edges. The resulting graph is called the transpose graph of G

G

. We also reset all nodes to the unvisited state.

2

3

5

0

6
1
4

The transposed example graph

We can start discovering SCCs through the backward DFS, starting with the first node in reverse finishing order, which is node 2

2

.

2
3
5
0
6
1
4

Discovering the first SCC in the transpose graph

We've discovered the first SCC $\{2, 3\}$

$\{2, 3\}$

. The next still unvisited node in the reverse finishing order is node 0

0

, so we'll run another DFS from it.

2
3
5
0
6
1
4

Discovering the second SCC in the transpose graph

Finally, the node 5

5

is still unvisited and thus becomes the last starting node, which discovers the third and final SCC. The end result is shown below with its edges in normal order again.

2
3
5
0
6
1
4

The original graph after discovering all SCCs

Now, let's try to understand why each run of the backward DFS discovers exactly one SCC. It all hinges on the following observation:

(Forward-Backward-Property): If u

u

occurs before v

v

in the reverse finishing order of the forward DFS, and there is a path from v

v

to u

u

, then u

u

and v

v

are in the same SCC.

Assuming that this property holds, it's easier to see that the algorithm is correct. When we run a backward DFS from node u

u

, we'll only discover nodes v

v

that have a forward path to u

u

, so they must be in the same SCC by the property. One can also verify inductively that each backward DFS pass from a starting node u

u

will in fact discover all nodes in the SCC of u

u

.

Let's see why the Forward-Backward-Property holds. Assume that u

u

occurs before v

v

in the reverse finishing order of the forward DFS. In other words, u

u

finishes after v

v

, and there is a path from v

v

to u

u

. We need to verify that there is also a path from u

u

to v

v

, to show that they are in the same SCC.

There are two cases to check, depending on which node is discovered first during the forward DFS.

The first case is that u

u

is discovered before v

v

. Since u

u

finishes after v

v
, there must be a path from u

u
to v

v
, as otherwise, u

u
would finish before v

v
is even reached.

The second case is that v

v
is discovered before u

u
. But this case actually cannot happen. Since there is a path from v

v
to u

u
, u

u
would have to finish before v

v
, but this contradicts our assumption that u

u
finishes after v

v
. .

This shows that the Forward-Backward-Property holds and explains why Kosaraju's algorithm works as intended.

*/
// ======
// ======
/*

Implementation of Kosaraju's Algorithm

Explore how to implement splitting a graph into its strongly connected components.

We'll cover the following

Implementing SCC identification

Trying out Kosaraju's algorithm

Implementing SCC identification#

Our implementation of Kosaraju's algorithm uses similar building blocks as our topological sort implementation. The setup of the class is similar, again using the adjacency list graph representation.

```
/*
// =====
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef vector<vector<int>> AdjacencyList;
// enum class NodeState
//{
//    UNVISITED,
//    IN_PROGRESS,
//    FINISHED
//};

// class StronglyConnectedComponents
//{
// private:
//    AdjacencyList adjacencyList;
//    vector<NodeState> nodeStates;
//    vector<int> finishOrder;
//    // vector to store the resulting SCCs
//    vector<vector<int>> scc;

// public:
//    StronglyConnectedComponents(AdjacencyList _adjacencyList)
//        : adjacencyList{_adjacencyList}, nodeStates{vector<NodeState>(adjacencyList.size(), NodeState::UNVISITED)}
//    {
//    }

//    vector<vector<int>> computeScc()
//    {
//        // TODO
//    }

//    // similar DFS routine as for topological sort
//    void dfs(int u)
//    {
//        this->nodeStates[u] = NodeState::IN_PROGRESS;
//        for (int v : this->adjacencyList[u])
//    }
}
```

```

// {
//   // we ignore back edges and redundant edges here
//   if (this->nodeStates[v] == NodeState::UNVISITED)
//   {
//     this->dfs(v);
//   }
// }
// this->nodeStates[u] = NodeState::FINISHED;
// this->finishOrder.push_back(u);
// }
// =====
/*

```

We've also borrowed the dfs routine from the topological sort class, which keeps track of the node finishing order during DFS. The main work will be done in the yet unfinished computeScc function (line 21). The variable scc (line 14) will be used to store the strongly connected components while they compute. Each SCC is represented as a vector<int> of the nodes it contains, and in total, all SCCs form a vector<vector<int>>.

Let's start implementing computeScc. First, we need to run the forward pass DFS.

```

*/
// =====
// for (int u = 0; u < this->adjacencyList.size(); ++u)
//{
//   if (this->nodeStates[u] == NodeState::UNVISITED)
//   {
//     this->dfs(u);
//   }
// }

// get the reverse finishing order
// vector<int> traverseOrder(finishOrder.rbegin(), finishOrder.rend());
// =====
/*

```

This is a straightforward DFS from all starting nodes. In the end, we'll obtain the reverse finishing order for the backward DFS. We now have to do some bookkeeping before we run the backward DFS.

```

fill(this->nodeStates.begin(), this->nodeStates.end(), NodeState::UNVISITED);
this->finishOrder.clear();
this->transpose();

```

Here, we'll reset all nodes to be unvisited and clear the finishOrder vector populated by the dfs. we'll use it during the backward pass to store the newly discovered SCC. We'll also call the transpose function to reverse all edges. Its implementation looks like this:

```

*/
// =====
// void transpose()
//{
//   // create new transposed adjacency list
//   AdjacencyList tmp = AdjacencyList(this->adjacencyList.size());
//   for (int u = 0; u < this->adjacencyList.size(); ++u)
//   {
//     for (int v : this->adjacencyList[u])
//   }
// }
```

```

// {
//   // add reverse edge (v, u)
//   tmp[v].push_back(u);
// }
// }
// // replace the adjacency list
// this->adjacencyList = tmp;
// }
// =====
/*

```

In transpose, we'll create a new adjacency list vector which contains an edge (v, u) (v,u) for every edge (u, v) (u,v) in the original graph.

All that's left now is to run the backward DFS and assemble the scc result.

```

*/
// =====
// for (int u : traverseOrder)
//{
//   if (this->nodeStates[u] == NodeState::UNVISITED)
//   {
//     this->dfs(u);
//     // add the newly found SCC
//     this->scc.push_back(this->finishOrder);
//     // clear the finish order to make room for the next SCC
//     this->finishOrder.clear();
//   }
// }

/// cleanup
// this->transpose();
// fill(this->nodeStates.begin(), this->nodeStates.end(), NodeState::UNVISITED);
// return this->scc;
// =====

// =====
// =====
/*

```

Here, we'll go over the nodes in the reverse finishing order of the forward DFS, as it was stored in traverseOrder. After each DFS run, we'll store the visited nodes (available in the variable finishOrder) as a new SCC and clear the finishOrder.

In the end, we reverse the edges again to return the graph to its original form and return the result.

The following code snippet contains the complete implementation of Kosaraju's algorithm.

```

*/
// =====
// =====
/*

```

```

/*
#include <vector>
#include <algorithm>
using namespace std;

typedef vector<vector<int>> AdjacencyList;
enum class NodeState {UNVISITED, IN_PROGRESS, FINISHED};

class StronglyConnectedComponents {
private:
    AdjacencyList adjacencyList;
    vector<NodeState> nodeStates;
    vector<int> finishOrder;
    vector<vector<int>> scc;

    void transpose() {
        AdjacencyList tmp = AdjacencyList(this->adjacencyList.size());
        for (int u = 0; u < this->adjacencyList.size(); ++u) {
            for (int v : this->adjacencyList[u]) {
                tmp[v].push_back(u);
            }
        }
        this->adjacencyList = tmp;
    }

public:
    StronglyConnectedComponents(AdjacencyList _adjacencyList)
        : adjacencyList {_adjacencyList}
        , nodeStates {vector<NodeState>(adjacencyList.size(), NodeState::UNVISITED)}
    {}

    vector<vector<int>> computeScc() {
        for (int u = 0; u < this->adjacencyList.size(); ++u) {
            if (this->nodeStates[u] == NodeState::UNVISITED) {
                this->dfs(u);
            }
        }
    }

    vector<int> traverseOrder(finishOrder.rbegin(), finishOrder.rend());
    this->transpose();
    fill(this->nodeStates.begin(), this->nodeStates.end(), NodeState::UNVISITED);
    this->finishOrder.clear();

    for (int u : traverseOrder) {
        if (this->nodeStates[u] == NodeState::UNVISITED) {
            this->dfs(u);
            this->scc.push_back(this->finishOrder);
            this->finishOrder.clear();
        }
    }

    this->transpose();
}

```

```

        fill(this->nodeStates.begin(), this->nodeStates.end(), NodeState::UNVISITED);
        return this->scc;
    }

void dfs(int u) {
    this->nodeStates[u] = NodeState::IN_PROGRESS;
    for (int v : this->adjacencyList[u]) {
        if (this->nodeStates[v] == NodeState::UNVISITED) {
            this->dfs(v);
        }
    }
    this->nodeStates[u] = NodeState::FINISHED;
    this->finishOrder.push_back(u);
}
};

*/
// =====
// =====
/*

```

Let's also analyze the runtime complexity. We're running two passes of DFS on the whole graph, which takes $\mathcal{O}(|V| + |E|)$
 $\mathcal{O}(|V|+|E|)$ time. Additionally, we'll transpose the graph twice, which also runs in $\mathcal{O}(|V| + |E|)$
 $\mathcal{O}(|V|+|E|)$. The total complexity is thus $\mathcal{O}(|V| + |E|)$
 $\mathcal{O}(|V|+|E|)$.

Trying out Kosaraju's algorithm#

Recall our example graph from the previous lesson:

Let's run our implementation of Kosaraju's algorithm on it!c

```

*/
// =====
// #include <iostream>
// using namespace std;

// int main()
// {
//     int n = 7;
//     AdjacencyList adjacencyList(n);
//     vector<pair<int, int>> edges{{0, 1}, {0, 6}, {1, 4}, {2, 3}, {2, 5}, {3, 2}, {3, 5}, {4, 0}, {4, 5}, {6, 4}};
//     for (auto [u, v] : edges)
//     {
//         adjacencyList[u].push_back(v);
//     }

//     StronglyConnectedComponents components(adjacencyList);
//     auto scc = components.computeScc();
//     cout << "Strongly connected components:" << endl;
//     for (auto component : scc)

```

```
// {
//   for (int u : component)
//   {
//     cout << u << " ";
//   }
//   cout << endl;
// }
// =====
/*
```

Challenge: Bipartite Graph Check

Apply graph traversal algorithms to check whether a graph is bipartite.

We'll cover the following

Bipartite graphs

2-colorings

Exercise

Bipartite graphs#

In this exercise, we'll take a look at a special kind of graph called bipartite graphs. An undirected graph G

G

is called bipartite if we can split its set of vertices up into two partitions (let's call them the "left" and the "right" vertices), such that all edges connect a left vertex to a right vertex. In other words, there are no edges between two left vertices or between two right vertices.

The following illustration shows an example of a bipartite graph. The left vertices are marked in red and the right vertices are marked in blue.

a
e
b
d
c
f

An example bipartite graph

Bipartite graphs often occur in matching problems or assignment problems.

A nice property of bipartite graphs is that many problems that are difficult or even intractable on general graphs become easier on bipartite graphs, allowing for simpler or more efficient algorithms. Therefore, it can be useful to know whether a graph is bipartite. We can apply our newfound knowledge about graph traversal algorithms to write a program that checks this property.

2-colorings#

There is an alternative characterization of bipartite graphs, which is a bit easier to use algorithmically. A graph is bipartite if and only if we can color its nodes using two colors (say, red and blue), such that there are no edges between nodes of the same color. This is called a 2-coloring of the graph.

Clearly, if a graph is bipartite, we can color it that way. Simply color the left partition red and the right partition blue. On the other hand, if we have a 2-coloring of a bipartite graph, we'll call the red nodes the left partition and the blue nodes the right partition.

In the bipartite graph example above, we already included the red and blue node coloring. Let's also look at an example graph that is not 2-colorable:

a
b
c

The smallest non-bipartite graph

There is no way to color the nodes of the above triangle graph in red and blue without assigning two neighboring nodes the same color. Thus, the graph is not bipartite, and neither is any graph that contains such a triangle. In fact, a graph is bipartite if and only if it contains no cycles of odd length.

We want to check whether a graph is bipartite by trying to construct a 2-coloring for it. We can attempt to create the 2-coloring using any of the graph traversal algorithms that we've learned: DFS or BFS.

Please take a moment to think for yourself how that might work, before looking at the hint or the solution.

Exercise#

Your task will be to implement a program that takes the adjacency list of a graph as input and checks whether the graph is bipartite, using the above strategy. You can use either BFS or DFS to solve the problem.

A class stub BipartiteCheck with a constructor and a function bool isBipartite() is already given to you to get you started. Please refrain from modifying the name of the class or of the isBipartite function. Apart from that, you can add methods and member variables to the class as you like.c

```
/*
// =====
// #include <vector>
// using namespace std;

// typedef vector<vector<int>> AdjacencyList;

// class BipartiteCheck
// {
// private:
//   AdjacencyList adjacencyList;

// public:
//   BipartiteCheck(AdjacencyList _adjacencyList)
//     : adjacencyList(_adjacencyList)
//   {
//   }
//   bool isBipartite()
//   {
//     // TODO modify the code below
//     return false;
//   }
// };
// =====
/*
```

Shortest Path Problems
Discover shortest path problems.

We'll cover the following

Navigating a city

Kinds of shortest path problems

In this chapter, we'll focus on algorithms that can compute shortest paths in weighted graphs. We already saw in the previous chapter that shortest paths in unweighted graphs can be computed using a simple Breadth-

First Search algorithm. But the most interesting real-life shortest path problems will require us to use weighted graphs to model the problem domain.

Navigating a city#

We'll start with the classical example of a shortest path problem: navigation. Let's say that we are located in London at Piccadilly Circus and want to get to Liverpool Street as quickly as possible, using the London Tube (the city's subway network).

We can model this as a graph problem where the nodes are different subway stations and the edges correspond to the subway lines running between them. We can take the physical distance between two stations in miles as the weight of each edge. All transportation lines run both ways, so the graph is undirected.

Piccadilly Circus

Holborn

1

Embankment

0.7

Baker Street

1.6

Liverpool Street

King's Cross

2.4

1.3

1.8

2

1.7

Distances in the London Tube network

Because the London subway network is notoriously complex, we'll have reduced the number of available stations and connections for the sake of the example. In the graph above, the shortest path from the node labeled Piccadilly Circus to the node labeled Liverpool Street is

Piccadilly Circus -> Embankment -> Liverpool Street

and has a length of $0.7 + 2 = 2.7$

$0.7+2=2.7$

. The path corresponds to the real-life shortest path from Piccadilly Circus to Liverpool Street along the given transport lines, which has a length of 2.7 miles.

In reality, we might not be interested in minimizing the actual distance of the connection, but rather the time spent commuting. To solve this related problem, we can use the graph with the same nodes and edges. The only required change is to take the travel times in minutes along each route as the weight of each edge. A shortest path in this graph will then correspond to the fastest connection. For simplicity, we're ignoring the time for changing lines at a stop. This is usually the kind of shortest path problem solved by car navigation systems.

Piccadilly Circus

Holborn

4

Embankment

2

Baker Street

6

Liverpool Street

King's Cross

11

6

10

18

9

Travel times in the London Tube network

In this case, the shortest path from the Piccadilly Circus node to the Liverpool Street node is

Piccadilly Circus -> Holborn -> Liverpool Street

with a length of $4 + 10 = 14$

$4+10=14$

. This path corresponds to the fastest travel route from Piccadilly Circus to Liverpool Street, which takes 14 minutes.

Another example would be to minimize the money we spend on transportation tickets. In that case, we could take the ticket price of each connection as the edge weights.

These examples show that through clever graph modeling, we can map a variety of real-life minimization problems to shortest path problems in weighted graphs, which makes the shortest path algorithms such powerful and versatile tools.

One limitation to keep in mind is that only minimization problems can be modeled as shortest path problems in general. The seemingly closely related problem of computing longest paths in graphs is NP-hard, so no efficient algorithm is known for it.

Kinds of shortest path problems#

The shortest path queries that we have asked so far have been “what is a shortest path from u

u

to v

v

?“ This is the most common, as well as the simplest type of shortest path problem, asking only for the shortest path between one pair of nodes. Let’s call this kind of question the Basic Shortest Path (BSP) problem.

BSP: Find a shortest path from vertex u

u

to vertex v

v

.

u

v

?

a

b

Illustration of the BSP problem

When searching for a shortest path from u

u

to v

v

, most shortest path algorithms will also discover paths from u
u
to other nodes in the graph, or at least those that are on the way to discovering v
v
. These paths are actually also shortest paths! This is called the optimal substructure property of shortest paths.

We can therefore see the problem of finding a shortest path from u
u
to v
v
as an instance of the slightly more general Single Source Shortest Path (SSSP) problem.

SSSP: Starting from a vertex u
u
, find shortest paths to all other vertices of the graph.

Illustration of the SSSP problem
We'll focus on algorithms for the SSSP problem, even when we're only solving BSP questions. This might seem unintuitive, because finding shortest paths to all nodes seems much more difficult than just finding a shortest path to one node. In the worst case, however, we'll actually need to discover paths to all other nodes before finally finding v

A line graph where solving the BSP can be as hard as solving the SSSP
The example graph above is actually a line with u
u
and v
v
being the two endpoints. To find the shortest path from u
u
to v
v
, we must discover all nodes of the graph along the way.

This is why the SSSP is easier than the BSP from a worst-case complexity standpoint. Still, when we're solving a BSP in practice, we can choose to stop our SSSP algorithm early once it discovers a shortest path to the target node v
v
, as we are not interested in a shortest path to any other node.

We already saw that BFS can solve the SSSP in unweighted graphs. We'll soon get to know Dijkstra's algorithm, which solves the SSSP efficiently for weighted graphs.

There is one type of shortest path problem, which is even more general called the All Source Shortest Path (ASSP) problem.

ASSP: Find shortest paths between all pairs of vertices of the graphs.

u
a
?

v
?
?
?
?

Illustration of the ASSP problem

Of course, once we know an SSSP algorithm, we can use it repeatedly for each starting vertex to solve the ASSP. We'll also learn an optimized algorithm for the ASSP that can solve it faster, which is the Floyd-Warshall algorithm.

[Back](#)

[Challenge: Bipartite Graph Check](#)

[Next](#)

```
*/  
// ======  
// ======  
/*
```

Dijkstra's Algorithm for the SSSP

Learn how to solve the SSSP using Dijkstra's algorithm.

We'll cover the following

Explanation of Dijkstra's algorithm

Constructing the shortest paths

Negative edge weights

In this lesson, we'll study Dijkstra's algorithm, which is the most common algorithm that efficiently solves the SSSP problem.

Explanation of Dijkstra's algorithm#

Let's use the following example graph to execute Dijkstra's algorithm:

```
a (0)  
b (?)  
6  
d (?)  
4  
c (?)  
10  
2  
e (?)  
5  
3  
1  
2  
1
```

An example graph for running Dijkstra's algorithm

We want to solve the SSSP problem starting from node a

a

, which is marked in gray, meaning that it is discovered. Like in BFS, we'll also keep track of the distance of each node to the starting vertex. In the beginning, we'll only know the starting vertex, and its distance is 0

.

Dijkstra's algorithm works similarly to BFS in the sense that the closest discovered, yet unexplored node is always explored first. In the beginning, we'll need to start with node a

a

, the only discovered node. We'll mark it as black, which means "finished," and note the distances of its neighbors.

a (0)

b (6)

6

c (10)

10

d (4)

4

2

3

e (?)

5

1

2

1

The graph after processing node a

Here, we use the notation $d(u)$

$d(u)$

to refer to the distance of node u

u

. For example, we currently have $d(b) = 6$

$d(b)=6$

.

The main difference compared to regular BFS is that the distances for the discovered nodes are not yet final. They can still be improved by finding a better path. A node's distance is only fixed once it is finished and corresponds to the length of the shortest path to the node. We now must select the closest unexplored node for exploration. Since we are now working with weighted graphs, the edge weights need to be taken into consideration when choosing the closest node. Currently, the closest node is d

d

, so it will be processed next. This means that we can finalize the distance of d

d

to be 4

4

.

From d

d

, we can discover the new node e

e

. We'll also find another edge to the already discovered node b

b
. Let's check whether we need to update the distance of b
b
. The currently best-known path to b
b
has length 6
6
. Our new path to node b
b
has a total length of 5
5
:

we know that our current node d

d
has a distance of 4
4
the edge (d, b)
(d,b)
has weight 1
1

So, we'll update the distance of b

b
to be 5
5
.

a (0)
b (5)
6
c (10)
10
d (4)
4
2
3
e (6)
5
1
2
1

The graph after processing node d

The process of inspecting the edge (d, b)

(d,b)
and checking whether the distance of b
b

needs updating is called relaxing the edge (d, b)
(d,b)
.

In general, to relax an edge (u, v)

(u,v)

we'll need to check whether $d(u) + w(u, v) < d(v)$

$d(u)+w(u,v)<d(v)$

. If so, we have discovered a new, shorter path to v

v

and set $d(v) = d(u) + w(u, v)$

$d(v)=d(u)+w(u,v)$

.

The next closest node to pick is node b

b

, which can thus be finished with a distance of 5

5

. From b

b

we relax all edges:

since $d(b) + w(b, c) = 5 + 3 = 8 < 10 = d(c)$

$d(b)+w(b,c)=5+3=8<10=d(c)$

, we update $d(c) = 8$

$d(c)=8$

.

since $d(b) + w(b, e) = 5 + 5 = 10 > 6$

$d(b)+w(b,e)=5+5=10>6$

, we do not update $d(e)$

$d(e)$

.

since $d(b) + w(b, a) = 5 + 2 = 7 > 0$

$d(b)+w(b,a)=5+2=7>0$

, we do not update $d(a)$

$d(a)$

.

We could have also simply ignored the edge (b, a)

(b,a)

, as node a

a

is already finished.

The graph after processing node b

The next node to process is node e

e

, as it has the smallest distance among the unexplored nodes. We can finalize its distance to 6

6

and successfully relax the edge (e, c)

(e,c)

, setting $d(c) = 7$

$d(c)=7$

. Finally, this leaves us with node c

c

, which has no more outgoing edges to relax, and we can also mark it as finished.

The final result of running Dijkstra's algorithm

The final result of running Dijkstra's algorithm now contains the final distance of each node, which corresponds to the length of the shortest path to reach that node.

Constructing the shortest paths#

So far, our algorithm only outputs the length of each shortest path from the starting vertex. To also recover the paths themselves, we can use the same technique as for the unweighted case: keep track of the parent used to discover each node.

In the case of Dijkstra, we've seen that the distance of an unexplored node can change over time and is only fixed once the node is explored. The same also accounts for the parent of a node. When we first discover a node v

v
from a node u

u , then u
 u
will become the parent of v

v . However, whenever we successfully relax an edge (w, v)
 (w, v)
and need to update $d(v)$
 $d(v)$, we also need to update the parent of v

v
to be w
 w
.

The following illustration shows the edges in the graph that correspond to the parent relation after running Dijkstra, marked in green.

a (0)
b (5)
6
c (7)
10
d (4)
4
2
3
e (6)
5
1
2
1

The final result of running Dijkstra's algorithm

Note that although node c

c
was first discovered from node a
 a

, its parent is actually node e
e
, as the last distance improvement for node c
c
was triggered from node e
e
.

To construct the shortest path from node a
a
to node c
c
, we go back from c
c
to its parent, grandparent, and so on, until we reach node a
a
. The shortest path is thus a -> d -> e -> c.

Negative edge weights#

One caveat of Dijkstra's algorithm is that it is only guaranteed to work when all edge weights are non-negative. This is because once a discovered node becomes the closest available, we'll consider it finished. Negative edge weights allow for the discovery of a shorter path from a node that is further away.

The following example shows such a situation:

a
b
4
c
2
-3
d
1

A graph with negative edge weights where Dijkstra's algorithm fails
Here, Dijkstra's algorithm would process nodes in this order:

a
a
with distance 0
c
c
with distance 2
d
d
with distance 3
b
b
with distance 4
When processing node b
b
, we encounter the negative edge (b, c)

(b,c)
, finding a shorter path to the already finished node c
c
. Even if we relax it and update the distance of c
c
to 1
1
, we've still have an incorrect distance for the node d
d
(3
3
instead of 2
2

When edge weights can be negative, Dijkstra's algorithm therefore cannot be used. In this situation, a suitable alternative is the Bellmann-Ford algorithm, which is less efficient but works in the presence of negative edges as long as there are no negative cycles.

A negative cycle is a cycle with net negative weight, as in the following graph:

a
b
4
c
-3

A graph with a negative cycle

The cycle a -> b -> c -> a has a total weight of -1, which means that it would be impossible to find shortest walks in the graph. We could simply traverse the negative cycle, again and again, yielding ever shorter walks.

Meanwhile, the shortest path problem is still well defined. Recall that a path is a walk with unique vertices, and therefore it cannot traverse the negative cycle again and again. However, most shortest path algorithms implicitly assume that no negative cycles are present, and otherwise fail. When negative cycles exist, the problem of finding shortest paths becomes NP-hard.

```
*/  
// ======  
// ======  
/*  
Implementation of Dijkstra  
Implement Dijkstra's algorithm.
```

We'll cover the following

- Implementation notes
- Priority queues in C++
- Implementing Dijkstra
- Time complexity
- Trying out the Dijkstra implementation
- Implementation notes#

As we saw in the previous lesson, Dijkstra's algorithm is very similar to BFS in the sense that the closest unexplored node is processed next within the order of processing. However, determining the closest node needs to consider the distance estimates known so far, and that they can change over time. Therefore, it is not possible to simply store the unfinished nodes in a static queue data structure.

A naive implementation could simply store the currently known distance of each node in a vector and check all the unexplored nodes during each iteration to get the node with the minimum distance. However, this would lead to a time complexity of $\mathcal{O}(|V|^2)$

$\mathcal{O}(|V|$

2

)

, and we can do better.

Instead, we should use a data structure that allows us to efficiently get the node with the lowest distance estimate. Additionally, the data structure should support updating the distance estimate of a vertex. Both properties can be fulfilled by using a priority queue data structure to store the nodes and their distances.

The following illustration shows how such a priority queue looks during the execution of Dijkstra's algorithm:

(u, 3)

(v, 6)

(w, 7)

A priority queue containing three nodes and their distances

Each element of the priority queue contains a vertex together with its currently known distance estimate. The elements of the queue are ordered by their distances. In priority queue terminology, the distance is called the key of the queue element, and elements are always ordered by their keys. When a new node is inserted or the distance of an existing node is updated, the elements of the queue are reordered to maintain the ordering of the element keys.

For example, if we relax an edge and update the distance of w

w

to 5

5

, the priority queue would look like this:

(u, 3)

(w, 5)

(v, 6)

The reordered priority queue after updating the distance of node w to 5

Under the hood, priority queues are often implemented using binary heaps. They only need logarithmic time to insert new elements, update existing elements, or extract the minimum element.

Priority queues in C++#

The C++ standard template library contains `std::priority_queue`, which is an implementation of priority queues. However, there is a major downside to this class: it does not support updating the key of an element, an operation that we need for relaxing edges in Dijkstra's algorithm.

We could implement our own priority queue data structure that supports decreasing keys, but this would be both challenging and time-consuming. Thankfully, there are viable workarounds.

One option is to use a `std::set` container as the queue, which supports extraction, insertion, and deletion operations in logarithmic time. Together with some bookkeeping, we can update elements by first deleting them and then inserting their updated version.

We could also instead use a `std::priority_queue` but still allow the insertion of the same vertex multiple times (with different distance values as keys). Since we always extract the value with the minimum key from the priority queue, we can be sure that the “closest copy” of each node is extracted first. We can then choose to ignore any other copies of the node, should we encounter it again.

```
(u, 3)  
(w, 5)  
(u, 6)  
(v, 7)
```

A priority queue containing multiple copies of a vertex with different keys

This workaround has a somewhat worse memory footprint. In the worst case, we store $\mathcal{O}(|E|)$
 $\mathcal{O}(|E|)$

elements in the priority queue, instead of the optimal $\mathcal{O}(|V|)$
 $\mathcal{O}(|V|)$

. On the plus side, it is far more convenient to implement than the `std::set` solution. Therefore, the second workaround should be preferred in all situations where memory efficiency is not a big concern. We have also chosen the second workaround for our implementation.

Implementing Dijkstra#

Our setup for the Dijkstra class is very similar to our BFS class that computes shortest paths in unweighted graphs:

```
*/  
// ======  
// #include <queue>  
// #include <vector>  
// #include <algorithm>  
// using namespace std;  
  
// // we use the weighted adjacency list here that stores pairs of target nodes and distances  
// typedef pair<int, int> NodeAndDistance;  
// typedef vector<vector<NodeAndDistance>> AdjacencyList;  
  
// class NoPathExistsException: public runtime_error {  
// public:  
//   NoPathExistsException() : runtime_error("No path exists between the nodes.") {}  
// };  
  
// class Dijkstra {  
// private:  
//   AdjacencyList adjacencyList;  
//   vector<int> distances;  
//   vector<int> parents;  
//   static constexpr int UNKNOWN = -1;  
  
//   void runDijkstra(int start) {  
//     // TODO  
//   }
```

```

// public:
// Dijkstra(AdjacencyList &_adjacencyList)
//   : adjacencyList {_adjacencyList}
//   , distances {vector<int>(_adjacencyList.size(), UNKNOWN)}
//   , parents {vector<int>(_adjacencyList.size(), UNKNOWN)}
// {}

// // return both the length of the shortest path and the path itself
// pair<int, vector<int>> computeShortestPath(int start, int end) {
//   this->runDijkstra(start);
//   if (this->parents[end] == UNKNOWN) {
//     throw NoPathExistsException();
//   }

//   int distance = this->distances[end];

//   // reconstruct the path from parents
//   vector<int> path;
//   path.push_back(end);
//   int current = end;
//   while (current != start) {
//     current = parents[current];
//     path.push_back(current);
//   }

//   std::reverse(path.begin(), path.end());
//   return make_pair(distance, path);
// }

// =====
/*

```

There are two main differences to the BFS version. First, we are now using weighted adjacency lists (line 8) to represent our input graph. Second, our computeShortestPath function now returns both the length of the shortest path as well as the path itself (line 33). We need to do this, as the length of a weighted path can no longer be inferred from the number of nodes on the path.

Now, let's look into writing the runDijkstra function. We'll start by setting up the priority_queue. Like in our illustrations above, it will hold pairs of integers $(u, d(u))$
 $(u, d(u))$ that represent a node and its currently estimated distance.

We also need to instruct the priority_queue on how to order its elements. In our case, we want to compare elements by the second integer of the tuple, or the distance. If the distances should be equal, we do not care about the order and can define it arbitrarily. Let's say that we prefer nodes with smaller IDs.

There is one more technicality of std::priority_queue: it will always store the largest element, according to its order, at the head of the queue. Therefore, we actually need to invert our element order such that elements with smaller distances look like larger elements to the priority_queue.

The correct ordering for our priority queue can be defined like so:

```

auto comparator = [](NodeAndDistance p1, NodeAndDistance p2) {
    return p1.second > p2.second || (p1.second == p2.second && p1.first > p2.first);
};

```

It returns true when p1 has a greater distance than p2, which means that p1 will occur as "smaller than p2" to the priority queue. Therefore, the node with the smallest distance will be the "largest" element according to our order and will appear at the top of the queue.

Next, we can initialize the priority_queue and an additional vector which keeps track of nodes that we already visited, since nodes can occur in the priority queue several times.

```

//initialize priority queue and visited vector
priority_queue<NodeAndDistance, vector<NodeAndDistance>, decltype(comparator)>
distanceQueue(comparator);
vector<bool> visited(this->adjacencyList.size(), false);

```

```

// handle the starting node
distanceQueue.push(make_pair(start, 0));
parents[start] = start;
distances[start] = 0;

```

To get started, we've pushed the starting node with distance 0

```

0
into the priority_queue and assigned its parent and distance entries.
```

Next up is the main loop of Dijkstra's algorithm.

```

*/
// =====
// while (!distanceQueue.empty())
//{
//     // get the closest node from the queue
//     auto [u, distance] = distanceQueue.top();
//     distanceQueue.pop();
//     // do not visit a node more than once
//     if (visited[u])
//         continue;
//     visited[u] = true;

//     for (auto [v, weight] : this->adjacencyList[u])
//     {
//         // relax the edge (u, v)
//         if (distances[v] == UNKNOWN || distances[u] + weight < distances[v])
//         {
//             distances[v] = distances[u] + weight;
//             parents[v] = u;
//             distanceQueue.push(make_pair(v, distances[v]));
//         }
//     }
// }
// =====
/*
```

At each iteration of the loop, we'll begin by extracting the node with the closest distance from the priority_queue (line 3). If the node was already visited, we'll skip it. Otherwise, we mark it as visited.

Next, we'll iterate over all outgoing edges of the current node and try to relax them (line 11). We'll update the parents and distances of the neighbors if they are previously undiscovered or if the edge could be relaxed to reveal a shorter path.

Since every loop iteration removes an element from the priority_queue, and at most $|E|$ elements can be added to it, the algorithm will terminate. The final values of distances are the lengths of the shortest paths from the starting vertex to all other nodes, and parents can be used to construct the actual paths.

Here is our complete implementation of Dijkstra's algorithm once more for reference.

```
/*
// =====
// #include <queue>
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef pair<int, int> NodeAndDistance;
// typedef vector<vector<NodeAndDistance>> AdjacencyList;

// class NoPathExistsException : public runtime_error
// {
// public:
//   NoPathExistsException() : runtime_error("No path exists between the nodes.") {}
// };

// class Dijkstra
// {
// private:
//   AdjacencyList adjacencyList;
//   vector<int> distances;
//   vector<int> parents;
//   static constexpr int UNKNOWN = -1;

//   void runDijkstra(int start)
//   {
//     auto comparator = [] (NodeAndDistance p1, NodeAndDistance p2)
//     {
//       return p1.second > p2.second || (p1.second == p2.second && p1.first > p2.first);
//     };
//     priority_queue<NodeAndDistance, vector<NodeAndDistance>, decltype(comparator)>
//     distanceQueue(comparator);
//     vector<bool> visited(this->adjacencyList.size(), false);

//     distanceQueue.push(make_pair(start, 0));
//     parents[start] = start;
//     distances[start] = 0;
```

```

// while (!distanceQueue.empty())
// {
//     auto [u, distance] = distanceQueue.top();
//     distanceQueue.pop();
//     if (visited[u])
//         continue;
//     visited[u] = true;

//     for (auto [v, weight] : this->adjacencyList[u])
//     {
//         if (distances[v] == UNKNOWN || distances[u] + weight < distances[v])
//         {
//             distances[v] = distances[u] + weight;
//             parents[v] = u;
//             distanceQueue.push(make_pair(v, distances[v]));
//         }
//     }
// }

// public:
// Dijkstra(AdjacencyList &_adjacencyList)
//     : adjacencyList{_adjacencyList}, distances{vector<int>(_adjacencyList.size(), UNKNOWN)}, parents{vector<int>(_adjacencyList.size(), UNKNOWN)}
// {
// }

// pair<int, vector<int>> computeShortestPath(int start, int end)
// {
//     this->runDijkstra(start);
//     if (this->parents[end] == UNKNOWN)
//     {
//         throw NoPathExistsException();
//     }

//     int distance = this->distances[end];

//     vector<int> path;
//     path.push_back(end);
//     int current = end;
//     while (current != start)
//     {
//         current = parents[current];
//         path.push_back(current);
//     }

//     std::reverse(path.begin(), path.end());
//     return make_pair(distance, path);
// }
// };

```

```
// =====
/*
Time complexity#
Let's analyze the time complexity of running computeShortestPath. In the worst case, every edge gets relaxed
and leads to one entry being added to the priority queue. Hence, the priority queue will contain
 $O(|E|)$ 
 $O(|E|)$ 
elements.
```

Every element of the priority queue needs to be inserted as well as deleted, which takes $O(\log|E|)$ time each. The total runtime of Dijkstra is thus $O(|E|\log|E|)$. Constructing the shortest path after running Dijkstra takes only linear time and can be neglected.

We can use a small mathematical trick here to make the complexity look even a bit better. Since $|E| = |E|^2$, we actually have

$$\begin{aligned} O(|E|\log|E|) &= O(|E|\log|V|^2) = O(|E| \cdot 2\log|V|) = \\ O(|E|\log|E|) &= O(|E|\log|V|) \\ 2 & \\) \end{aligned}$$

Thus, the time complexity of Dijkstra can be given as $O(|E|\log|V|)$.

Trying out the Dijkstra implementation#

Let's try out the implementation on our example graph from the previous lesson:

The example graph for running Dijkstra

We'll compute a shortest path and its length from node 0

```
0
to node 2
2
using Dijkstra's algorithm.
```

```
/*
// =====
// #include <iostream>
// using namespace std;
```

```
// int main()
//{
// int n = 5;
// AdjacencyList adjacencyList(n);
// vector<tuple<int, int, int>> edges{
```

```

// make_tuple(0, 1, 6),
// make_tuple(0, 2, 10),
// make_tuple(0, 3, 4),
// make_tuple(1, 0, 2),
// make_tuple(1, 2, 3),
// make_tuple(1, 4, 5),
// make_tuple(3, 1, 1),
// make_tuple(3, 4, 2),
// make_tuple(4, 2, 1)};
// for (auto [u, v, w] : edges)
// {
//   adjacencyList[u].push_back(make_pair(v, w));
// }

// Dijkstra dijkstra(adjacencyList);
// auto [distance, path02] = dijkstra.computeShortestPath(0, 2);
// cout << "Shortest path from 0 to 2: ";
// for (int u : path02)
// {
//   cout << u << " ";
// }
// cout << endl
//     << "Length of the path: " << distance;
// */
// =====
/*

```

Floyd-Warshall Algorithm for the ASSP

Use Floyd-Warshall to compute shortest paths between all pairs of nodes.

We'll cover the following

The Floyd-Warshall algorithm

Constructing the paths

Negative edges

In the next lessons, we'll focus on the All Sources Shortest Path (ASSP) problem. Of course, we could solve the ASSP problem by just running Dijkstra's algorithm once from each starting vertex. But the Floyd-Warshall algorithm, which was designed to specifically solve the ASSP problem, is a viable alternative. Its benefits are that it is very simple to implement and it runs faster than repeated Dijkstra executions on dense graphs.

The Floyd-Warshall algorithm#

To explain how the Floyd-Warshall algorithm works, let's look at a small example graph.

An example graph with 4 nodes

The Floyd-Warshall algorithm is remarkable in the sense that it is one of the few algorithms that uses the weighted adjacency matrix graph representation. Let's assume that all edge weights are positive, so we can use a single matrix to represent the graph. The weighted adjacency matrix of our example graph is

$$A = \begin{pmatrix} 0 & 6 & 3 & 12 \\ 0 & 0 & 9 & 5 \\ 0 & 0 & 0 & 2 \\ 0 & 4 & 1 & 0 \end{pmatrix}$$

The basic idea of Floyd-Warshall is to successively transform the adjacency matrix to end up with a resulting matrix that contains the lengths of the shortest paths between each pair of nodes.

In the following, let's write n

n

for the number of nodes $|V|$

$|V|$

. In particular, the adjacency matrix is an $n \times n$

$n \times n$

matrix and the nodes of the graph are numbered 0, 1, ..., $n-1$

0, 1, ..., $n-1$

.

The Floyd-Warshall algorithm successively computes a sequence of $n \times n$

$n \times n$

distance matrices

A_0, A_1, \dots, A_n

A

0

, A

1

, ..., A

n

where A_0

A

0

can be obtained directly from the adjacency matrix and A_n

A

n

contains the end result.

Let's look at the definition of these matrices A_k

A

k

.

Distance matrices: $A_k[i, j]$

A

k

[i, j]

is the length of the shortest path from node i

i
to node j

j
that visits only intermediate nodes with an index less than k

k

.

So, the matrix A_k

A
k

contains the lengths of the shortest paths between each pair of nodes, assuming that the nodes with index $\geq k$

$\geq k$
are forbidden to visit in between. We'll call such a path a \mathbf{k} -restricted shortest path.

Let's take a deeper look at the first and last path matrix, A_0

A
0

and A_n

A
n

.

The matrix A_n

A
n

contains the lengths of shortest paths between any pair of nodes that visit only nodes with index $< n$

$< n$ in between. Since all nodes have an index $< n$

$< n$, the matrix A_n

A

n

is our desired end result containing the lengths of the shortest paths between all pairs of nodes.

On the other hand, the matrix A_0

A
0

contains lengths of shortest paths between all pairs of nodes that may only visit nodes with index < 0

in between. Since there is no node with a negative index, the shortest paths can only be direct edges between nodes. Thus, we have

$$A_0[i, j] = \begin{cases} 0 & i = j \\ w(i, j) & (i, j) \in E \\ \infty & \text{otherwise} \end{cases}$$

A

0

$$[i,j] = \begin{cases} 0 & i=j \\ w(i,j) & (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

$$\begin{cases} 0 & i=j \\ w(i,j) & (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Here, we follow the convention that if no shortest path between two nodes exists, then their distance is infinite. Further, the shortest path from a node to itself is the empty path of length 0

0

.

In particular, A_0

A

0

is nearly identical to the weighted adjacency matrix. The only difference is that zeros that are not on the main diagonal are replaced with ∞

∞

.

and have found an algorithm for the ASSP problem.

Let's say that we already computed A_k

A

k

and want to compute A_{k+1}

A
k+1

. How can we compute the entry $A_{\{k+1\}}[i, j]$

A
k+1

[i,j]
, the length of the (k+1)
(k+1)
-restricted shortest path from i
i
to j
j

? There are two cases:

This (k+1)
(k+1)
-restricted shortest path does not visit the node k
k
. This (k+1)
(k+1)
-restricted shortest path does visit the node k
k
. Let's visualize the two cases:

i
j
k

The (k+1)-restricted shortest path can either visit node k or not

In the first case depicted in the dotted edge above, the (k+1)

(k+1)
-restricted shortest path from i

i
to j
j
does not visit node k
k

. Therefore, it is the same as the k

k
-restricted shortest path, and we can set $A_{\{k+1\}}[i, j] = A_k[i, j]$
A
k+1

[i,j]=A
k

[i,j]

In the second case depicted in the dotted edges below, the path from i

to j

j

consists of two sub-paths: one from i

i

to k

k

, and another from k

k

to j

j

. Since the path from i

i

to j

j

is a $(k+1)$

$(k+1)$

-restricted shortest path, the two sub-paths must also be $(k+1)$

$(k+1)$

-restricted shortest paths.

The key insight here is that both of these sub-paths do not visit the node k

k

. Clearly, it can't appear twice on a path from i

i

to j

j

. Therefore, the two sub-paths are actually also k

k

-restricted shortest paths. This means that we can combine the k

k

-restricted shortest path from i

i

to k

k

with the k

k

-restricted shortest path from k

k

to j

j

to form the $(k+1)$

$(k+1)$

-restricted shortest path from i

i

to j

j

:

$$A_{k+1}[i, j] = A_k[i, k] + A_k[k, j]$$

A

k+1

[i,j]=A

k

[i,k]+A

k

[k,j]

By combining the two cases, we now know how to compute $A_{k+1}[i, j]$

A

k+1

[i,j]

: always takes the smaller of the two available options.

$$A_{k+1}[i, j] = \min \{A_k[i, j], A_k[i, k] + A_k[k, j]\}$$

A

k+1

[i,j]=min{A

k

[i,j],A

k

[i,k]+A

k

[k,j]}

Using this formula, we can compute all of A_{k+1}

Let's go back to our example graph from above.

The example graph from above

Here are the matrices A_1

for it:

$$A_1 = \begin{pmatrix} 0 & 6 & 3 & 12 & \infty & 0 & 9 & 5 & \infty & \infty & 0 & 2 & \infty & 4 & 1 & 0 \end{pmatrix} \\ A_2 = \begin{pmatrix} 0 & 6 & 3 & 11 & \infty & 0 & 9 & 5 & \infty & \infty & 0 & 2 & \infty & 4 & 1 & 0 \end{pmatrix}$$

. This is because node 0
0
has no incoming edges and therefore cannot be used as an intermediate node on a shortest path.

In matrix A_1
A
1

, we see a difference in row 0

0

, column 3

3

. We may now use nodes 0

0

and 1

1

as intermediate vertices on shortest paths. This allows us to discover the path $0 \rightarrow 1 \rightarrow 3$

$0 \rightarrow 1 \rightarrow 3$

of length 11

11

, which is shorter than the direct edge $(0, 3)$

$(0, 3)$

of weight 12

12

.

On the other hand, the $(3, 2)$

$(3, 2)$

entry did not change from A_1

A

1

to A_2

A

2

: although we can now try the path $3 \rightarrow 1 \rightarrow 2$

$3 \rightarrow 1 \rightarrow 2$

, it is longer than the direct edge $(3, 2)$

$(3, 2)$

.

Running the Floyd-Warshall algorithm further yields the matrices

A_3 = $\begin{pmatrix} 0 & 6 & 3 & 5 & \infty & 0 & 9 & 5 & \infty & \infty & 0 & 2 & \infty & 4 & 1 & 0 \end{pmatrix}$
A_4 = $\begin{pmatrix} 0 & 6 & 3 & 5 & \infty & 0 & 6 & 5 & \infty & \infty & 6 & 0 & 2 & \infty & 4 & 1 & 0 \end{pmatrix}$
A

3

contains the lengths of shortest paths between all pairs of nodes. For example, the shortest path from node

2

2

to node 1

1

has length $A_4[2, 1] = 6$

A

4

$[2,1]=6$

.

Constructing the paths#

So far, we have only focused on computing the lengths of the shortest paths. Of course, we are also interested in constructing the shortest paths themselves. To do so, we'll again use the concept of parent vertices, although it might make more sense to call them predecessor vertices in the ASSP scenario.

In the end, we will need a $n \times n$

$n \times n$

matrix of predecessor vertices, which contains the predecessor along the shortest path for each pair of nodes. Using this matrix, we can go backward from a target node along predecessors until we reach the source node, just like in our Dijkstra- or BFS-based shortest path algorithms.

Akin to the sequence of distance matrices A_k

A

k

, we can maintain a sequence of predecessor matrices

P_0, P_1, \dots, P_n

P

0

, P

1

, \dots, P

n

with the following definition:

Predecessor matrices: $P_k[i, j]$

P

k

$[i, j]$

is the predecessor of node j

j
on the k

k
-restricted shortest path from i

i
to j

j
.

If there is no k

k
-

-restricted shortest path, we set the predecessor to be -1

-1
.

Just like for the distance matrices, P_n

P
n

is our desired end result and contains the predecessors on the actual, unrestricted shortest paths. On the other hand, P_0

P
0

can again be computed easily from the adjacency matrix:

P_0[i, j] = \begin{cases} i & \& i = j \\ \text{or } (i, j) \in E \\ -1 & \text{otherwise} \end{cases}

P
0

[i,j]={
i
-1

i=j or (i,j)∈E
otherwise

For our example graph, we have

P_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 2 \\ 0 & 1 & 2 & -1 & 3 \\ 0 & 1 & 2 & 3 & -1 \end{pmatrix}

P
0

=

(

0
-1
-1
-1

0
1
-1
3

0
1
2
3

0
1
2
3

)

We can update the predecessor matrices in conjunction with the path matrices. We saw above that there were two options for setting $A_{k+1}[i, j]$

A
 $k+1$

$[i,j]$
, depending on whether node k
 k
gets visited along the way or not. This resulted in the formula

$A_{k+1}[i, j] = \min \{A_k[i, j], A_k[i, k] + A_k[k, j]\}.$
 A
 $k+1$

$[i,j]=\min\{A$
 k

$[i,j],A$
 k

[i,k]+A

k

[k,j]}.

Updating the predecessor $P_{\{k+1\}}[i, j]$

P

k+1

[i,j]

depends on which of the two options is the better one.

If $A_k[i, j]$

A

k

[i,j]

is the shorter path length, then we can keep $P_{\{k+1\}}[i, j] = P_k[i, j]$

P

k+1

[i,j]=P

k

[i,j]

.

If $A_k[i, k] + A_k[k, j]$

A

k

[i,k]+A

k

[k,j]

is the shorter path length, then our (k+1)

(k+1)

-restricted shortest path consists of two sub-paths, the latter being the k

k

-restricted shortest path from k

k

to j

j

. Thus, we'll set $P_{\{k+1\}}[i, j] = P_k[k, j]$

P

k+1

[i,j]=P

k

[k,j]

Let's compute the predecessor matrices for our example graph.

```
0  
1  
6  
2  
3  
3  
12  
9  
5  
2  
4  
1
```

The example graph from above

If we follow the procedure to update the predecessor matrices, we'll ultimately arrive at

```
P_4 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -1 & 1 & 3 & 1 \\ -1 & 3 & 2 & 2 \\ -1 & 3 & 3 & 3 \end{pmatrix}
```

```
P  
4
```

```
\begin{pmatrix} = \\ X \end{pmatrix}
```

```
0  
-1  
-1  
-1
```

```
0  
1  
3  
3
```

```
0  
3  
2  
3
```

```
0  
1  
2  
3
```

X

Let's use P_4

P

4

to construct the shortest path from node 1

1

to node 2

2

. First, we'll lookup the predecessor of node 2

2

when starting from node 1

1

, which is $P_4[1, 2] = 3$

P

4

$[1,2]=3$

. Next, we'll need the predecessor of node 3

3

along the shortest path from starting node 1

1

to node 3

3

. This is $P_4[1, 3] = 1$

P

4

$[1,3]=1$

. Since we've reached the starting node 1

1

, the shortest path is now completely assembled as $1 \rightarrow 3 \rightarrow 2$

$1 \rightarrow 3 \rightarrow 2$

.

Negative edges#

As long as there are no negative cycles, the Floyd-Warshall algorithm yields correct results even when the graph contains negative edges.

We can also use Floyd-Warshall to detect whether a graph contains negative cycles. For that purpose, we'll run an additional iteration of the algorithm and compute the matrix $A_{\{n+1\}}$

A

$n+1$

. If the graph contains a negative cycle, then $A_{\{n+1\}}$

A
n+1

will be different from A_n

A
n

in at least one entry, because we can go around the negative cycle multiple times, finding consecutively shorter walks. If the graph contains no negative cycle, then $A_{n+1} = A_n$

A
n+1

=A
n

.Implementation of Floyd-Warshall

Learn about the details of implementing Floyd-Warshall.

We'll cover the following

Implementation notes

Implementing Floyd-Warshall

Runtime analysis

Trying out the implementation

Implementation notes#

The implementation of Floyd-Warshall is relatively straightforward compared to Dijkstra's algorithm. We'll construct the matrices A_0, P_0

A
0

,P
0

from the weighted adjacency list and then iteratively compute the distance and predecessor matrices A_k ,

P_k
A
k

,P
k

, for $k = 1, \dots, |V|$
 $k=1, \dots, |V|$

.

One important observation to reduce the memory footprint of the algorithm is that the computation of

$A_{\{k+1\}}$

A

$k+1$

and $P_{\{k+1\}}$

P

$k+1$

depends only on the immediate previous matrices A_k

A

k

and P_k

P

k

. Therefore, the updates to the distance and predecessor matrices can be done in place. In other words, we only need to store one pair of $n \times n$

$n \times n$ matrices instead of all of them. This reduces the memory requirements from $\mathcal{O}(|V|^3)$

$\mathcal{O}(|V|$

3

)

to $\mathcal{O}(|V|^2)$

$\mathcal{O}(|V|$

2

)

.

Another implementation detail is how to represent the infinite values in the distance matrices. A pragmatic and often feasible solution is to use a large constant value such as one billion to represent infinity. The main concern here is that this value should be larger than the maximum length of a shortest path in the graph, to avoid overflows or incorrect results.

Implementing Floyd-Warshall#

As usual, let's begin by setting up a class for the Floyd-Warshall algorithm.

```
/*
// =====
// #include <vector>
// #include <algorithm>
// using namespace std;

// // use weighted adjacency matrix representation
// typedef vector<vector<int>> AdjacencyMatrix;

// class NoPathExistsException : public runtime_error
```

```
// {
// public:
//  NoPathExistsException() : runtime_error("No path exists between the nodes.") {}
// };

// class FloydWarshall
//{
// private:
//  AdjacencyMatrix distanceMatrix;
//  AdjacencyMatrix predecessorMatrix;
//  // constants for unknown predecessors and infinite distances
//  static constexpr int UNKNOWN = -1;
//  static constexpr int INFINITY = 1'000'000'000;

//  void initialize(AdjacencyMatrix const &adjacencyMatrix)
//  {
//    // TODO
//  }

//  void compute()
//  {
//    // TODO
//  }

// public:
//  FloydWarshall(AdjacencyMatrix const &adjacencyMatrix)
//  {
//    this->initialize(adjacencyMatrix);
//    this->compute();
//  }

//  // return length of shortest path as well as the path itself
//  pair<int, vector<int>> getShortestPath(int start, int end)
//  {
//    int distance = this->distanceMatrix[start][end];
//    if (distance == INFINITY)
//    {
//      throw NoPathExistsException();
//    }

//    vector<int> path{end};
//    int current = end;
//    while (current != start)
//    {
//      current = this->predecessorMatrix[start][current];
//      path.push_back(current);
//    }
//    std::reverse(path.begin(), path.end());
//    return make_pair(distance, path);
//  }
//};
```

```
// =====  
/*
```

In principle, the setup is very similar to the Dijkstra implementation. A notable change is the use of the adjacency matrix representation (line 6). We also define two constants (lines 18-19):

-1 for unknown/missing predecessors

One billion for “infinite distances” that will be used in the algorithm

The distanceMatrix and predecessorMatrix member variables will be used to hold the intermediate and final results of the Floyd-Warshall algorithm. They’ll be filled in the initialize and compute functions called in the constructor.

The function to return a shortest path and its length (line 35) is very similar to the one from our Dijkstra implementation. The only difference is that the results are accessed from matrices instead of vectors.

Next, let’s look into the initialize function. It constructs the matrices A_0

A

0

and P_0

P

0

from the given adjacency matrix.

```
*/  
// =====  
// void initialize(AdjacencyMatrix const &adjacencyMatrix)  
// {  
//   int n = adjacencyMatrix.size();  
//   // default values: infinite distance, no predecessor  
//   this->distanceMatrix = AdjacencyMatrix(n, vector<int>(n, INFINITY));  
//   this->predecessorMatrix = AdjacencyMatrix(n, vector<int>(n, UNKNOWN));  
//   for (int i = 0; i < n; ++i)  
//   {  
//     for (int j = 0; j < n; ++j)  
//     {  
//       // fill main diagonal  
//       if (i == j)  
//       {  
//         this->distanceMatrix[i][j] = 0;  
//         this->predecessorMatrix[i][j] = i;  
//       }  
//       // fill distances and predecessors for edges  
//       else if (adjacencyMatrix[i][j] > 0)  
//       {  
//         this->distanceMatrix[i][j] = adjacencyMatrix[i][j];  
//         this->predecessorMatrix[i][j] = i;  
//       }  
//     }  
//   }  
// }
```

```
//}  
// =====  
/*
```

Initially, all distances are infinite and all predecessors are unknown. For each edge, we'll note the weight in the distanceMatrix and the source in the predecessorMatrix. We'll also fill the main diagonal with the respective special values.

Finally, we're only missing the compute function, which executes the Floyd-Warshall matrix updates. Its implementation is straightforward.

```
*/  
// =====  
// void compute()  
// {  
//   int n = this->distanceMatrix.size();  
//   // run n = |V| operations  
//   for (int k = 0; k < n; ++k)  
//   {  
//     // update each entry  
//     for (int i = 0; i < n; ++i)  
//     {  
//       for (int j = 0; j < n; ++j)  
//       {  
//         if (this->distanceMatrix[i][k] + this->distanceMatrix[k][j] < this->distanceMatrix[i][j])  
//         {  
//           this->distanceMatrix[i][j] = this->distanceMatrix[i][k] + this->distanceMatrix[k][j];  
//           this->predecessorMatrix[i][j] = this->predecessorMatrix[k][j];  
//         }  
//       }  
//     }  
//   }  
// }
```

```
/*
```

There are three nested for-loops in compute. Each iteration of the outermost loop corresponds to one update of A_k , P_k

A
k

,P
k

to A_{k+1} , P_{k+1}
A
 $k+1$

,P
 $k+1$

. The two innermost loops iterate over the rows and columns of the distance and predecessor matrices to update them.

Here is our completed Floyd-Warshall implementation once more for reference.

```
/*
// =====
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef vector<vector<int>> AdjacencyMatrix;

// class NoPathExistsException : public runtime_error
//{
// public:
// NoPathExistsException() : runtime_error("No path exists between the nodes.") {}
// };

// class FloydWarshall
//{
// private:
// AdjacencyMatrix distanceMatrix;
// AdjacencyMatrix predecessorMatrix;
// static constexpr int UNKNOWN = -1;
// static constexpr int INFINITY = 1'000'000'000;

// void initialize(AdjacencyMatrix const &adjacencyMatrix)
// {
// int n = adjacencyMatrix.size();
// this->distanceMatrix = AdjacencyMatrix(n, vector<int>(n, INFINITY));
// this->predecessorMatrix = AdjacencyMatrix(n, vector<int>(n, UNKNOWN));
// for (int i = 0; i < n; ++i)
// {
// for (int j = 0; j < n; ++j)
// {
// if (i == j)
// {
// this->distanceMatrix[i][j] = 0;
// this->predecessorMatrix[i][j] = i;
// }
// else if (adjacencyMatrix[i][j] > 0)
// {
// this->distanceMatrix[i][j] = adjacencyMatrix[i][j];
// this->predecessorMatrix[i][j] = i;
// }
// }
// }
// }

// void compute()
// {
```

```

// int n = this->distanceMatrix.size();
// for (int k = 0; k < n; ++k)
// {
//   for (int i = 0; i < n; ++i)
//   {
//     for (int j = 0; j < n; ++j)
//     {
//       if (this->distanceMatrix[i][k] + this->distanceMatrix[k][j] < this->distanceMatrix[i][j])
//       {
//         this->distanceMatrix[i][j] = this->distanceMatrix[i][k] + this->distanceMatrix[k][j];
//         this->predecessorMatrix[i][j] = this->predecessorMatrix[k][j];
//       }
//     }
//   }
// }

// public:
// FloydWarshall(AdjacencyMatrix const &adjacencyMatrix)
// {
//   this->initialize(adjacencyMatrix);
//   this->compute();
// }

// pair<int, vector<int>> getShortestPath(int start, int end)
// {
//   int distance = this->distanceMatrix[start][end];
//   if (distance == INFINITY)
//   {
//     throw NoPathExistsException();
//   }

//   vector<int> path{end};
//   int current = end;
//   while (current != start)
//   {
//     current = this->predecessorMatrix[start][current];
//     path.push_back(current);
//   }
//   std::reverse(path.begin(), path.end());
//   return make_pair(distance, path);
// }

// =====
/*
Runtime analysis#
The runtime of Floyd-Warshall is dominated by the compute function, which takes  $\mathcal{O}(|V|^3)$ 
 $\mathcal{O}(|V|$ 
 $^3$ 
 $)$ 
time due to the three nested for-loops.

```

Let's compare this to the runtime of repeated executions of Dijkstra's algorithm, which would be another way to solve the ASSP. Since one Dijkstra execution takes $O(|E| \log |V|)$ time, running Dijkstra from each starting vertex would take $O(|V| |E| \log |V|)$ time.

In dense graphs where $|E| = O(|V|^2)$
 $|E|=O(|V|$
 2
 $)$, the Dijkstra runtime ends up to be $O(|V|^3 \log |V|)$
 $O(|V|$
 3
 $\log|V|)$, using Floyd-Warshall is preferable.

However, in sparse graphs where $|E| = O(|V|)$
 $|E|=O(|V|)$, repeated Dijkstra runs end up with a $O(|V|^2 \log |V|)$
 $O(|V|$
 2
 $\log|V|)$ complexity, Dijkstra generally outperforms Floyd-Warshall.

Trying out the implementation#

Recall our Floyd-Warshall example graph from the previous lesson:

```
/*
// =====
// #include <iostream>
// using namespace std;

// int main()
// {
//   int n = 4;
//   AdjacencyMatrix adjacencyMatrix(n, vector<int>(n, 0));
//   vector<tuple<int, int, int>> edges{
//     make_tuple(0, 1, 6),
//     make_tuple(0, 2, 3),
//     make_tuple(0, 3, 12),
//     make_tuple(1, 2, 9),
//     make_tuple(1, 3, 5),
//     make_tuple(2, 3, 2),
//     make_tuple(3, 1, 4),
//     make_tuple(3, 2, 1)};
//   for (auto [v, u, w] : edges)
//   {
//     adjacencyMatrix[v][u] = w;
//   }
// }
```

```

// FloydWarshall floydWarshall(adjacencyMatrix);
// auto [distance, path] = floydWarshall.getShortestPath(0, 3);
// cout << "Shortest path from 0 to 3:";
// for (int u : path)
// {
//   cout << " " << u;
// }
// cout << endl
//    << "Length of the path: " << distance << endl;

// try
// {
//   floydWarshall.getShortestPath(2, 0);
// }
// catch (NoPathExistsException exc)
// {
//   cout << "No path from 2 to 0.";
// }
// */
// =====
/*

```

Challenge: Bellmann-Ford

Implement the Bellmann-Ford algorithm to solve SSSP problems in graphs with negative edge weights.

We'll cover the following

The Bellmann-Ford algorithm

Example execution

Runtime analysis

Detecting negative cycles

Challenge: Implement Bellmann-Ford

In this Challenge Lesson, we'll implement the Bellmann-Ford algorithm, an alternative algorithm for the SSSP. While it runs slower than Dijkstra, it is able to correctly handle graphs with negative edge weights as long as there are no negative cycles.

The Bellmann-Ford algorithm#

The Bellmann-Ford algorithm computes distances by repeatedly relaxing all edges of the graphs. Recall that relaxing an edge (u,v)

(u,v)

means checking whether

$$d(u) + w(u, v) < d(v),$$

$$d(u) + w(u, v) < d(v),$$

in which case we can update the tentative distance $d(v)$

$d(v)$

.

To run Bellmann-Ford from a starting node u

u

, we

```

Initialize d(u) = 0
d(u)=0
and d(v) = \infty
d(v)=\infty
for v \neq u
v
.
=u
.

Try to relax each edge of the graph.
Repeat step (2) until the distance estimates do not change anymore.
Finally, the value of d(v)
d(v)
corresponds to the distance of node v
v
from node u
u
. As usual, we can also keep track of predecessor nodes when relaxing edges to construct the shortest paths themselves.

```

Example execution#

Let's run Bellmann-Ford on this small example graph, starting from node 3

```

3
:
```

```

3
1
4
2
3
-2
0
-6
5
```

Example graph for running Bellmann-Ford

When running step (2), we have arbitrarily decided to try relaxing the edges in the following order:

```
(0, 3), (1, 0), (1, 2), (3, 1), (3, 2)
(0,3),(1,0),(1,2),(3,1),(3,2)
```

The following table shows the distance estimates after each execution of step (2).

step	d(0)
d(0)	0
d(1)	0
d(1)	0
d(2)	0
d(2)	0
d(3)	0
d(3)	0

0 \infty

\infty

\infty

\infty

0

1 \infty

\infty

4

4

3

3

0

0

2 -2

-2

4

4

2

2

0

0

3 -2

-2

4

4

2

2

0

0

3 -2

-2

4

4

2

2

0

0

In the first execution, we've only successfully relaxed the outgoing edges of the starting node 3

3

. After the second execution, we could also relax the negative edges from the now reachable node 1

1

. In the third step, the distance estimates do not change anymore, and we've found the lengths of the shortest paths.

Runtime analysis#

As long as the graph has no negative cycles, we can be sure that step (2) is executed only a finite number of times. In fact, the following property holds:

Bellmann-Ford invariant: after k

k

executions of step (2), $d(v)$

$d(v)$

is the length of the shortest walk from u

u

to v

v

that has length $\leq k$

$\leq k$

.

When there are no negative cycles, shortest walks are identical to shortest paths, and have a length of at most

$(|V|-1)$

$(|V|-1)$

. Therefore, we'll run step (2) at most $|V|$

$|V|$

times until it stops changing distances. Actually, we only need to run step (2) at most $(|V|-1)$

$(|V|-1)$

times because we know that the $|V|$

$|V|$

-th iteration will never change any distances.

If we use the weighted adjacency list representation, then one execution of step (2) takes $\mathcal{O}(|E|)$
 $\mathcal{O}(|E|)$

time. The total runtime of Bellmann-Ford is thus $\mathcal{O}(|V||E|)$

$\mathcal{O}(|V||E|)$

.

This is quite a bit slower than Dijkstra's $\mathcal{O}(|E|\log|V|)$

$\mathcal{O}(|E|\log|V|)$

, but that's the price we pay for being able to handle negative edge weights.

Detecting negative cycles#

We can even use Bellmann-Ford to detect whether a graph has negative cycles reachable from the starting node. To check this, we'll run step (2) of Bellmann-Ford exactly $|V|$

$|V|$

times. If there are still changes to the distances at the $|V|$

$|V|$

-th execution, then there must be a shortest walk of length $\geq |V|$

$\geq |V|$

. This can only happen when negative cycles are present.

Challenge: Implement Bellmann-Ford#

Now your task is to implement the Bellmann-Ford algorithm. A class stub BellmannFord that accepts a graph in weighted adjacency list representation has already been given to you. You can assume that the input graph has no negative cycles.

Your task is to use Bellmann-Ford to finish the function `vector<int> computeDistances(int start)` , which returns distances from node u to all other nodes of the graphs. Please refrain from changing the name of the class or of the `computeDistances` function. However, you are free to add further functions or member variables.

*/

// =====

// #include <vector>

// using namespace std;

// typedef vector<vector<pair<int, int>>> AdjacencyList;

// class BellmannFord {

```

// private:
//   AdjacencyList adjacencyList;
//   static constexpr int INFINITY = 1'000'000'000;
// public:
//   BellmannFord(AdjacencyList _adjacencyList)
//     : adjacencyList {_adjacencyList}
//   {}
//   vector<int> computeDistances(int start) {
//     int n = this->adjacencyList.size();
//     vector<int> distances(n, INFINITY);
//     // TODO modify the code below
//     return distances;
//   }
// };
// =====
/*

```

The Minimum Spanning Tree Problem

Get introduced to the minimum spanning tree problem.

We'll cover the following

Defining the problem

Introducing trees

Properties of trees

What about undirected graphs?

Defining the problem#

The minimum spanning tree (MST) problem deals with connected, weighted, undirected graphs $G = (V, E, w)$

$G = (V, E, w)$

. Here, the weights can be interpreted as costs, and the goal is to select some subset of edges $F \subseteq E$

$F \subseteq E$

that connect the nodes of V

V

in a cost-optimal way. In particular, we want

F

F

contains enough edges such that they connect all nodes of V

V

the total cost of the edges in F

F

is as small as possible.

As an example, consider the following weighted graph:

```

a
b
7
c
2
8
d
8

```

3
e
4
6

An example graph for the MST problem

We can think of the nodes as representing cities, and the edges representing distances between them. Now, assume that we want to build a railroad network to connect these cities. Laying down train tracks is expensive, so we'll want to find a way to connect all the nodes using the existing edges while keeping the total edge cost to a minimum.

In fact, this translates to finding a subset $F \subseteq E$

$F \subseteq E$

of minimal cost which connects V

V

. The solution is given below, with the edges of F

F

highlighted in blue:

a
b
7
c
2
8
d
8
3
e
4
6

The minimum spanning tree of the example graph

The total cost of F

F
is

$$w(F) = \sum_{e \in F} w(e) = 4 + 2 + 3 + 7 = 16.$$

$$w(F) = \sum_{e \in F}$$

Σ

$$w(e) = 4 + 2 + 3 + 7 = 16.$$

There is no other subset of E

E
that connects all nodes and has a lower cost.

Introducing trees#

In the language of graph theory, we can formulate the task like this:

(MST problem) Given a connected, weighted, undirected graph $G = (V, E, w)$
 $G = (V, E, w)$

, find a subgraph $T = (V, F, w)$

$T = (V, F, w)$

such that T

T is connected and $w(F)$

$w(F)$

is minimal.

Saying that T

T

is a subgraph of G

G

simply means that T

T

contains only nodes and edges that are also in G

G

. In this case, the set of nodes are identical, although the set of edges might only be a subset of E

E

.

As we have seen above, the weight $w(F)$

$w(F)$

of the set of edges F

F

is obtained by summing the weight of the individual edges:

$$w(F) = \sum_{e \in F} w(e).$$

$w(F) =$

$e \in F$

\sum

$w(e).$

Why do we call this task the Minimum Spanning Tree problem? Let's think a bit about what the graph T

T

will look. By definition, we already know T

T

is connected. We can also claim that T

T

must be acyclic.

This can be inferred from the fact that T

T

has a minimum cost. If there was any cycle in T

T

, we could remove one of its edges and obtain a graph that is still connected but has a lower cost. The following picture illustrates this.

a

b

7

c
2
8
d
8
3
e
4
6

A selection of edges that is not acyclic

Above, we've selected a set F

F

(marked in blue) that contains a cycle. Removing any of the dashed edges on the cycle results in a still connected subgraph of lower cost.

Hence, we know that T

T

is both connected and acyclic. Such a graph is called a tree because it looks like a real-life tree that is branching out, although the branches do not meet.

Let's draw the solution graph T

T

for our example problem and omit the edges not chosen. We also rearrange the nodes a bit.

a
b
7
c
2
d
3
e
4

The tree T that solves the MST problem on our example graph

With some imagination, it does look like a tree growing upwards from the node b

b

.

This explains why the task is called the Minimum Spanning Tree problem. We're looking for the cheapest tree that spans or connects the original graph.

Properties of trees#

A tree on a graph with n

n

vertices always has $n-1$

$n-1$

edges, which is the minimum number of edges required to connect all nodes.

Therefore, we can think of trees as minimally connected graphs. They are connected, but removing just one edge would disconnect them.

At the same time, trees are also maximally acyclic. They are acyclic, but adding just a single edge to them would introduce a cycle. For example, let's assume we add the edge (u, v)

(u, v) to a tree T

. Since T

T is connected, there is already a path from u

u

to v

v

in it, so the new edge completes a cycle.

We've actually implicitly "discovered" trees before, namely in the form of discovery edges in breadth-first-search or depth-first-search. As a reminder, these are the edges in the search that discover a node for the first time. The discovery edges of a graph search in a connected graph always form a spanning tree of the graph.

This also means that the MST problem can be solved in $\mathcal{O}(|E|)$

$\mathcal{O}(|E|)$

on unweighted graphs by running breadth-first-search. In unweighted graphs, every spanning tree is a minimum spanning tree.

On weighted graphs, BFS will also give us a spanning tree, but not necessarily a minimal one. If we run a BFS from node a

a

in our example graph, we'll discover the following spanning tree:

The result of running BFS from node a in our example graph, with discovery edges marked in blue

The result of running BFS from node a is not optimal because its total weight is 21

21

, while the MST has weight 16

16

. So, we'll need a more complex algorithm to actually compute minimum spanning trees in weighted graphs.

There are two classical algorithms for the MST problem: Kruskal's algorithm and Prim's algorithm. Both have the same runtime complexity, so we'll just focus on one. For now, we'll use Kruskal's algorithm, which is simple but elegant.

What about undirected graphs?#

So far, we've only talked about the MST problem within connected graphs. After all, we cannot find a spanning tree of a disconnected graph. We can, however, approximate this by finding the minimum spanning tree of each connected component of the graph. Such a collection of spanning trees is fittingly called a minimum spanning forest.

```
/*
// =====
// =====
*/

```

Kruskal's Algorithm

Learn to compute minimum spanning trees using Kruskal's algorithm.

We'll cover the following

Computing MSTs with Kruskal's algorithm

Why the greedy strategy works

The disconnected case

In this lesson, we'll study Kruskal's algorithm, which makes use of a surprisingly straightforward strategy to assemble a minimum spanning tree.

Computing MSTs with Kruskal's algorithm#

The input to Kruskal's algorithm is a connected, weighted, undirected graph $G = (V, E, w)$

$G = (V, E, w)$

. The algorithm iteratively builds up an acyclic graph $T = (V, F, w)$

$T = (V, F, w)$

with $F \subseteq E$

$F \subseteq E$

. In the beginning, F

F

is empty. The edges will be added until T

T

is a spanning tree of G

G

.

First, we'll sort all edges of E

E

by their weight $w(e)$

$w(e)$

in increasing order. Next, we'll inspect each edge $e = (u, v)$

$e = (u, v)$

in this order, one by one. If u

u

and v

v

are in different connected components of T

T

, we'll add e

e

to F

F

. Otherwise, we'll skip e

e

, as it would introduce a cycle in T

T

.

We'll continue this strategy until we have processed all edges, at which point T

T

is a spanning tree of G

G

.

Let's illustrate the algorithm on the example graph from the previous lesson:

a
b
7
c
2
8
d
8
3
e
4
6

The example graph for the MST problem

The edge with the lowest weight is (a, c)

(a,c)

with weight 2

2

. We'll add it as the first edge to our subgraph T

T

(marked in blue below).

a
b
7
c
2
8
d
8
3
e
4
6

Picking the first edge for T

The next edges are (b, d)

(b,d)

of weight 3

3

and (a, e)

(a,e)

of weight 4

4

. Both edges connect nodes that are still held in different connected components in T

T

. Therefore, we can add them as well.

a
b
7
c

2
8
d
8
3
e
4
6

Adding two more edges

The next edge to consider is (c, e)

(c,e)
of weight 6

6

. However, c

c

and e

e

are already connected in T

T

, via the blue edges to node a

a

. So, we skip this edge.

Instead, we'll choose the next edge (a, b)

(a,b)
of weight 7

7

. Our graph T

T

now has 4

4

edges and is thus already a spanning tree. In fact, it's a minimum spanning tree. Its weight is $2 + 3 + 4 + 7 =$

16

$2+3+4+7=16$

.

a
b
7
c
2
8
d
8
3
e
4
6

The complete MST (blue). The red edge (c, e) is not chosen

Why the greedy strategy works#

Intuitively, it is quite clear that Kruskal's algorithm will produce a spanning tree as long as the underlying graph is connected. But how can we be sure that it is indeed minimum?

After all, Kruskal just uses a greedy strategy of selecting edges. In other words, it makes local decisions by inspecting a single edge without taking into account the edges that follow.

We can show that the result of Kruskal's algorithm is indeed an MST by using a "cut-and-paste" argument. For this, we'll let T

T be the spanning tree resulting from applying Kruskal's algorithm. We'll show that every other spanning tree can be improved by making it more similar to T

T

.

Let U

U

be any other spanning tree. Since U

U

is different from T

T

, there must be at least one edge e

e

that is in T

T

, but not in U

U

. Out of these edges, pick the one with minimal weight.

Let's say that this edge is $e = (u, v)$

$e = (u, v)$

. Since U

U

is connected, there must be a path from u

u

to v

v

in it. Furthermore, there must be at least one edge $f = (a, b)$

$f = (a, b)$

on this path, which is not in T

T

. Otherwise, together with the edge e

e

, it would form a cycle in T

T

.

a

b

f

v

u

e

The edge (u, v) in T (red) and the path from u to v in U (blue)

Now, let's show that

$$w(f) \geq w(e).$$

$$w(f) \geq w(e).$$

Towards a contradiction, assume that $w(f) < w(e)$

$$w(f) < w(e)$$

. Since $f \notin T$

$f \in$

/

T

, Kruskal's algorithm decided not to choose f

f

. That means that at the time when f

f

was considered, a

a

and b

b

were already in the same connected component of T

T

, so they must have been linked by a path of previously selected edges g_1, \dots, g_k

g

1

, ..., g

k

with

$$w(g_i) \leq w(f) < w(e) \quad \text{quad } 1 \leq i \leq k$$

$$w(g$$

i

$$\leq w(f) < w(e) \quad 1 \leq i \leq k$$

There must be at least one edge g_i

g

i

that is not in U

U

. Otherwise f

f

would complete a cycle in U

U

. However, this contradicts our earlier choice of e

e
because the edge with minimum weight is in T

T
but not in U

U
. Hence, the assumption $w(f) < w(e)$
 $w(f) < w(e)$
cannot hold.

So, we know that

$w(f) \geq w(e)$.
 $w(f) \geq w(e)$.

We can now modify U

U
into a new spanning tree U_1
U
1

by removing the edge f

f
and adding the edge e

e

.

a
b
f
v
u
e

The spanning tree U_1 (blue) obtained from U by removing f and adding e

Since $w(f) \geq w(e)$

$w(f) \geq w(e)$

, this will decrease the total weight of or keep it constant:

$w(U) \geq w(U_1)$.
 $w(U) \geq w(U_1)$
1
).

We also know that U_1

U
1

has one more edge in common with T
T

than U

U

had.

If U_1

U

1

is still different from T

T

, we can now iterate this procedure and obtain spanning trees U_2, U_3

U

2

, U

3

that become more and more similar to T

T

, until finally, $U_l = T$

U

l

$=T$

for some l

l

. Since each step can only decrease the weight of the spanning tree, we get

$w(U) \leq w(U_1) \leq w(U_2) \leq \dots \leq W(U_l) = w(T).$

$w(U) \geq w(U$

1

$) \geq w(U$

2

$) \geq \dots \geq W(U$

l

$) = w(T).$

In total, we've shown that $w(U) \leq W(T)$

$w(U) \geq W(T)$

. Since U

U

was an arbitrary spanning tree different from T

T

, T

T

is indeed a minimum spanning tree.

The disconnected case#

Kruskal's algorithm can also be run on input graphs G

G

that are disconnected. In this case, the end result will be a minimum spanning forest of G

G

.

Implementation of Kruskal's Algorithm

Dive into the implementation of MST algorithms.

We'll cover the following

Implementation Notes

Implementing Kruskal

Trying out the implementation

Runtime Analysis

Implementation Notes#

During Kruskal's algorithm, we'll iteratively build up a minimum spanning tree T

T

by processing the edges one by one. For each edge $e = (u, v)$

$e=(u,v)$

we need to check whether u

u

and v

v

are already connected in T

T

. The main challenge of implementing Kruskal is performing it efficiently.

The naive approaches, such as running a BFS from u

u

to v

v

for each edge, would be quite costly. The key component in speeding up the algorithm is using the correct data structure, which is called Disjoint-Set-Union (DSU) or Union-Find.

As the name suggests, a DSU data structure stores disjoint sets. In our case, it will store the connected components of our current graph T

T

. Each set has a representative. We can efficiently check whether two elements are in the same set by comparing their representatives.

Here is an illustration of a DSU data structure containing the three disjoint sets $\{a, b, c\}$, $\{d, e\}$

$\{a,b,c\}, \{d,e\}$

and $\{f\}$

$\{f\}$

. The representative of each set is marked in gray.

b

c

a

d
e
f

Illustration of a DSU data structure

Typically, a DSU data structure supports two operations:

The find(u) operation, which returns the representative of the set that contains u

u

The unite(u, v) operation, which merges the set containing u

u

and the set containing v

v

into one bigger set

In Kruskal's algorithm, we use find to check whether two nodes are in the same component, and unite to join two components into one when adding an edge to our tree T

T

.

When implemented carefully, both unite and find essentially run in amortized $\mathcal{O}(1)$

$O(1)$

time. This means that not every operation might take constant time; but when we perform k

unite or find operations in total, the total runtime will surely be $\mathcal{O}(k)$

$O(k)$

.

The C++ standard template library does not contain an implementation of a DSU data structure. Fortunately, it is not too daunting to implement one by ourselves. An implementation in the form of the class DisjointSetUnion is given in the appendix.

Implementing Kruskal#

Since Kruskal's algorithm is relatively straightforward and does not require a very mutable state, we'll implement it as a function rather than by writing a class for it. It will take a weighted graph in adjacency list representation and return both the weight of the minimum spanning tree, as well as the tree itself, which is also a weighted adjacency list.

The first part consists of building a list of all edges and sorting it by weight.

```
/*
// =====
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef vector<vector<pair<int, int>>> AdjacencyList;
// typedef tuple<int, int, int> Edge;

// pair<int, AdjacencyList> computeMinimumSpanningTree(AdjacencyList adjacencyList) {
//     int n = adjacencyList.size();
//     vector<Edge> edges;
//     // build edge list
//     for (int u = 0; u < n; ++u) {
```

```

//      for (auto [v, w] : adjacencyList[u]) {
//          edges.push_back(make_tuple(u, v, w));
//      }
//  }
//  // sort by weight
//  sort(
//      edges.begin(),
//      edges.end(),
//      [] (auto e, auto f){ return get<2>(e) <= get<2>(f); }
//  );
//  // TODO: to be continued
//};

// =====
/*

```

After sorting the edges, we process them one by one to see if they should be added to the tree.

```

*/
// =====
// initialize |V| singleton sets
// auto dsu = DisjointSetUnion(n);
// AdjacencyList tree(n);
// int treeWeight = 0;
// for (auto [u, v, w] : edges)
//{
//  // if u and v are in different connected components
//  if (dsu.find(u) != dsu.find(v))
//  {
//    // join the component
//    dsu.unite(u, v);
//    // add edge to the tree
//    treeWeight += w;
//    tree[u].push_back(make_pair(v, w));
//    tree[v].push_back(make_pair(u, w));
//  }
//}

// return make_pair(treeWeight, tree);
// =====
/*

```

In line 2

2

, we'll initialize the DisjointSetUnion data structure with $n = |V|$

$n = |V|$

singleton sets. We'll then loop over all edges $e = (u, v)$

$e = (u, v)$

and check whether u

u

and v

v

are already in the same component, by comparing representatives with `find` in line 7

7

.

If u
 u
 and v
 v
 are not yet connected, we'll call unite to combine their components (line 9
 9
). After that, we'll add the edge (u, v)
 (u,v)
 to our tree. Finally, we'll return the MST weight and the tree itself.

The following code snippet contains the complete implementation of Kruskal's algorithm.

```
/*
// =====
// typedef vector<vector<pair<int, int>>> AdjacencyList;
// typedef tuple<int, int, int> Edge;

// pair<int, AdjacencyList> computeMinimumSpanningTree(AdjacencyList adjacencyList) {
//     int n = adjacencyList.size();
//     vector<Edge> edges;
//     for (int u = 0; u < n; ++u) {
//         for (auto [v, w] : adjacencyList[u]) {
//             edges.push_back(make_tuple(u, v, w));
//         }
//     }
//     sort(
//         edges.begin(),
//         edges.end(),
//         [](auto e, auto f){ return get<2>(e) <= get<2>(f); }
//     );
//
//     auto dsu = DisjointSetUnion(n);
//     AdjacencyList tree(n);
//     int treeWeight = 0;
//     for (auto [u, v, w] : edges) {
//         if (dsu.find(u) != dsu.find(v)) {
//             dsu.unite(u, v);
//             treeWeight += w;
//             tree[u].push_back(make_pair(v, w));
//             tree[v].push_back(make_pair(u, w));
//         }
//     }
//
//     return make_pair(treeWeight, tree);
// =====
/*
// =====
// #include <iostream>
```

```

// #include <vector>
// using namespace std;

// typedef vector<vector<pair<int, int>>> AdjacencyList;
// typedef tuple<int, int, int> Edge;

// int main()
//{
// // set up the adjacency list
// int n = 5;
// vector<Edge> edges{
//     make_tuple(0, 1, 7), make_tuple(0, 2, 2), make_tuple(0, 3, 8),
//     make_tuple(0, 4, 4), make_tuple(1, 2, 8), make_tuple(1, 3, 3),
//     make_tuple(2, 4, 6)};
// AdjacencyList adjacencyList(n);
// for (auto [u, v, w] : edges)
// {
//     adjacencyList[u].push_back(make_pair(v, w));
//     adjacencyList[v].push_back(make_pair(u, w));
// }

// // run mst computation
// auto [weight, tree] = computeMinimumSpanningTree(adjacencyList);

// // print output
// cout << "Weight of the MST: " << weight << endl;
// cout << "Edges:";
// for (int u = 0; u < n; ++u)
// {
//     for (auto [v, w] : tree[u])
//     {
//         if (u < v)
//         {
//             cout << "(" << u << ", " << v << ")";
//         }
//     }
// }
// =====
// =====
/*
Runtime Analysis#
In the beginning of Kruskal's algorithm, we'll sort the edges, which takes  $O(|E| \log |E|)$ 
 $O(|E| \log |E|)$ 
time. Later, we'll perform  $O(|V|)$ 
 $O(|V|)$ 
unite-operations and  $O(|E|)$ 
 $O(|E|)$ 
find-operations, which takes  $O(|V| + |E|)$ 

```

$O(|V| + |E|)$

time. This amount of time is negligible compared to the time spent sorting the edges. Therefore, the total runtime is $\mathcal{O}(|E| \log |E|)$

$O(|E| \log |E|)$

Just like in the discussion of Dijkstra's algorithm, we can rewrite that as $\mathcal{O}(|E| \log |V|)$
 $O(|E| \log |V|)$

```
/*
// =====
// =====
/*
```

Challenge: Maximum Spanning Trees

Modify Kruskal's algorithm to find maximum spanning trees.

The goal of this challenge is to implement an algorithm that computes the weight of a maximum spanning tree of a connected, weighted, and undirected graph. A maximum spanning tree is a spanning tree with the highest possible sum of its edge weights.

There are at least two different approaches to solving this problem:

Implementing a modified version of Kruskal's algorithm.

Applying our implementation of Kruskal's algorithm while also changing the input graph and processing the output differently.

Each approach will require different insights, so thinking about both will be worthwhile.

To recap, here is our implementation of Kruskal's algorithm once again, with a slight simplification. It will return only the weight of the minimum spanning tree, not the tree itself.

```
/*
// =====
// typedef vector<vector<pair<int, int>>> AdjacencyList;
// typedef tuple<int, int, int> Edge;

// int computeMinimumSpanningTree(AdjacencyList adjacencyList) {
//     int n = adjacencyList.size();
//     vector<Edge> edges;
//     for (int u = 0; u < n; ++u) {
//         for (auto [v, w] : adjacencyList[u]) {
//             edges.push_back(make_tuple(u, v, w));
//         }
//     }
//     sort(
//         edges.begin(),
//         edges.end(),
//         [] (auto e, auto f){ return get<2>(e) <= get<2>(f); }
//     );

//     auto dsu = DisjointSetUnion(n);
//     int treeWeight = 0;
//     for (auto [u, v, w] : edges) {
```

```

//     if (dsu.find(u) != dsu.find(v)) {
//         dsu.unite(u, v);
//         treeWeight += w;
//     }
// }

// return treeWeight;
// =====
/*
In the code challenge widget below, the function int computeMinimumSpanningTree(AdjacencyList adjacencyList) is available for yourself to use. You can also make use of the class DisjointSetUnion in your implementation.
*/
// =====
// #include <vector>
// #include <algorithm>
// using namespace std;

// typedef vector<vector<pair<int, int>>> AdjacencyList;
// typedef tuple<int, int, int> Edge;

// int computeMaximumSpanningTree(AdjacencyList adjacencyList) {
//     int n = adjacencyList.size();
//     // modify the code below
//     return -1;
// }
// =====
/*

```

The Max Flow Problem

Learn about flow problems in graph networks.

We'll cover the following

Example of a flow network

The maximum flow problem

Cyclic flows

The next lessons discuss algorithms that solve flow problems in weighted graphs. Apart from direct applications in flow networks, they can also be applied to solve various kinds of matching or assignment problems.

Example of a flow network#

As an example, let's consider the problem of transferring a large file on the internet from London to Frankfurt. We want to achieve the maximum possible throughput in doing so. On the way, we may need to pass several intermediate nodes, and each connection we use on the way has a limited data rate.

This situation can be modeled as a weighted graph where the edge weights correspond to the maximum throughput on each connection, for example, in Megabit per second:

Naturally, we'd like to utilize each connection at its full capacity. However, this is not always possible. For example, the outgoing connections from Amsterdam have a total capacity of 7 Mb/s, but the only incoming connection has a capacity of only 5 Mb/s, so the outgoing connections cannot be fully utilized.

The maximum network flow which can be achieved in this network is 10 Mb/s. There are actually several ways to obtain this maximum flow. The following image shows one of them:

The maximum flow problem#

Let's come up with a formalization of the flow maximization problem. The setting is a directed graph $G = (V, E, c)$

$G = (V, E, c)$

, called a flow network, where $c : E \rightarrow \mathbb{N}$

$c : E \rightarrow \mathbb{N}$

is a weight function. The edge weights are the capacities of the edges, written as $c(u, v)$

$c(u, v)$

.

The capacity of an edge is the maximum flow that can be pushed through that single edge. We assume that the capacities are integers, and only an integer amount of flow can be pushed through an edge.

There are also two special nodes in V

V

: the source s

s

and the sink t

t

. The source has no incoming edges, and the sink has no outgoing edges. In the example from above, the source is London, and the sink is Frankfurt. Our goal is to maximize the total network flow from s

s

to t

t

.

A flow in the graph is a specification of how many units of flow should be pushed through each edge. It is specified as a flow function $f : E \rightarrow \mathbb{N}$

$f : E \rightarrow \mathbb{N}$

. However, for the flow to be valid, it needs to follow a few intuitive rules:

Respect the capacity: The flow through an edge (u, v)

(u, v)

can be the edge capacity at most.

$f(u, v) \leq c(u, v) \quad \forall (u, v) \in E$

$f(u, v) \leq c(u, v) \quad \forall (u, v) \in E$

In-flow = out-flow: Except for the source and the sink, the flow entering a node v

v

must equal the flow exiting v

v

.

$\sum_{(u, v) \in E} f(u, v) = \sum_{(v, w) \in E} f(v, w) \quad \forall v \in V \setminus \{s, t\}$

$(u, v) \in E$

Σ

$f(u, v) =$

$(v,w) \in E$

\sum

$f(v,w) \forall v \in V \setminus \{s,t\}$

There are no constraints on the outgoing flow of the source, so we can assume that we have an infinite amount of flow available to distribute from the source.

The value $v(f)$

$v(f)$

of a flow f

f

is the amount of flow that reaches the sink t

t

:

$v(f) = \sum_{(u,t) \in E} f(u,t)$

$v(f) =$

$(u,t) \in E$

\sum

$f(u,t)$

The value of a flow is also equal to the amount of flow that leaves the source s

s

.

We are interested in finding a maximum flow, or, a flow of the highest possible value. In our example graph from above, the maximum flow has a value of 10

10

.

Cyclic flows#

The definition of a flow does not prohibit pushing flow around in cycles on the way from s

s

to t

t

. For example, here is a flow of value 5

5

, which also pushes some additional flow through the cycle $a \rightarrow b \rightarrow c$:

However, we do not usually desire the behavior of pushing flow through cycles, meaning that it would not make sense to send files in cycles on the internet.

The good news is that even if we end up computing a flow with such cyclic flow components, we can remove them by running a DFS-based cycle detection algorithm on the edges that have nonzero flow. If we find a cycle, we can identify the minimum flow on it and reduce the flow on each edge of the cycle by this amount. This will remove the cyclic flow component, but keep the value of the overall flow intact. We can iterate that approach until there are no more cycles.

For our example from above, we end up with the following adapted flow.

```
*/  
// ======  
// ======  
/*
```

The Ford-Fulkerson Method

Learn about the Ford-Fulkerson method of computing maximum flows.

We'll cover the following

Description of the method

A first attempt of finding augmenting paths

The residual network

Runtime Analysis

Correctness

This lesson will cover a general, high-level strategy to compute maximum flows, called the Ford-Fulkerson method.

Description of the method#

Given a flow network $G = (V, E, c)$

$G=(V,E,c)$

, the Ford-Fulkerson method iteratively refines a flow f

in the graph, until it becomes a maximum flow.

In the beginning, the flow f

f

has value 0

0

, or, $f(u, v) = 0$

$f(u,v)=0$

for all edges $(u, v) \in E$

$(u,v)\in E$

.

We then search for a path from s

s

to t

t

through which we can still push at least a single unit of flow. Such a path is called an augmenting path. We can use it to update our flow f

f

to a flow of higher value. We repeat this procedure until no augmenting path exists, at which point our flow f

is maximum.

A first attempt of finding augmenting paths#

So, let's look into how we can identify augmenting paths. The first approach might be to look for an s

s

$-t$

t

-path upon which each edge still has capacity remaining, or,

$f(u, v) < c(u, v)$ \quad for all edges (u, v) on the path.
 $f(u, v) < c(u, v)$ for all edges (u, v) on the path.

Let's try this out in an example flow network.

There are several s

s

-t

t

-augmenting paths, but let's assume we decide on the path $s \rightarrow a \rightarrow b \rightarrow t$. The flow that we can push through this path is 2

2

, or the minimum of the edge capacities on it. After augmenting our flow in that way, we'll arrive at the following network:

At this point, no further paths from s

s

to t

t

exists upon which every edge still has free capacity. However, the flow we've found so far is not a maximum flow. This means that our simple strategy of finding augmenting paths was not good enough.

The problem that we ran into here is that we made an incorrect decision by pushing two units of flow from a
a

to b

b

. Instead, we should "take back" one unit of flow from (a, b)

(a, b)

and move it to (a, t)

(a, t)

. This allows us to also pump one more unit of flow from s

s

to t

t

via b

b

.

The key insight here is that we've augmented the flow on the path $s \rightarrow b \rightarrow a \rightarrow t$, by

adding one unit of flow for (s, b)

(s, b)

"adding" one unit of flow for (b, a)

(b, a)

by taking back one unit of flow along (a, b)

(a, b)

adding one unit of flow for (a, t)

(a, t)

In other words, when looking for augmenting s

s

-t
t

-paths, we're not only allowed to add flow to edges by following them forward, but we're also allowed to remove flow from edges by traversing them backward. This idea motivates the definition of the residual network.

The residual network#

Given a flow network $G = (V, E, c)$

$G = (V, E, c)$

and a flow f

f in it, we'll define a residual network $G_f = (V, E_f, c_f)$

G

f

$= (V, E$

f

$, c$

f

)

. The idea of the residual network is that the residual capacities $c_f(u, v)$

c

f

(u, v)

correspond to the amount of flow that can still be pushed along the edge (u, v)

(u, v)

. Here, this amount can come from

Excess capacity along the edge: $c(u, v) - f(u, v)$

$c(u, v) - f(u, v)$

Taking back some flow along the reverse edge: $f(v, u)$

$f(v, u)$

Therefore, the definition of c_f

c

f

is

$c_f(u, v) = c(u, v) - f(u, v) + f(v, u),$

c

f

$(u, v) = c(u, v) - f(u, v) + f(v, u),$

and the edge set is

$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}.$

E
f

= $\{(u,v) \in V \times V | c_f(u,v) > 0\}$.

There is no flow in the residual network.

To illustrate the concept, let's consider our flow network from above after augmenting our flow for the first time:

For example, $c_f(a, b) = 3$

c
f

$(a,b)=3$

, as there are 3

3

free units of flow up to the maximum capacity $c(a, b) = 5$

$c(a,b)=5$

. But also $c_f(b, a) = 2$

c
f

$(b,a)=2$

, as we can push back the two units of flow from $f(a, b) = 2$

$f(a,b)=2$

.

We can use the residual network to find augmenting paths in the original network. In fact, any s

s
-t
t

-path in the residual network corresponds to an augmenting path in the original network, and vice versa.

In our example residual network, we can find the path $s \rightarrow b \rightarrow a \rightarrow t$. The minimum capacity along the path is

1

1

. This means that we can still increase the value of our flow by 1

1

by augmenting it along the path, which leads to both adding and removing flow in the original graph. This brings us to the maximum flow that we already saw above:

The residual network contains no s

s
-t
t

-path, so there are no more augmenting paths.

To recap, we can describe the Ford-Fulkerson method as follows:

Initialize a flow f

f

of value 0

0

. Construct the residual network G_f

G

f

. Try to find an s

s

$-t$

t

-path in G_f

G

f

. If there is such a path, augment the flow f

f

with it, then go to step 2

2

. If there is no such path, return the flow f

f

. Runtime Analysis#

Even though we have not made a concrete implementation yet, let's establish an upper bound for the runtime of the Ford Fulkerson method.

Constructing the residual graph G_f

G

f

can be done in $\mathcal{O}(|E|)$

$\mathcal{O}(|E|)$

. We can also construct an s

s

$-t$

t

-path in $\mathcal{O}(|E|)$

$\mathcal{O}(|E|)$

, using either BFS or DFS. Augmenting the flow along the path in $\mathcal{O}(|E|)$

$\mathcal{O}(|E|)$

time is also not a problem.

The main question remaining is how many iterations of steps (2)-(4) in the method will be needed. Let M

be the value of a maximum flow in G

G

. Since each discovered augmented path increases the value $v(f)$ by at least 1

1

, we'll never require more than M

M

iterations. An upper bound for the runtime is thus $\mathcal{O}(|E| \cdot M)$

$\mathcal{O}(|E| \cdot M)$

.

In a graph with large capacities, M

M

can be in the range of millions or even billions, which makes this runtime unacceptable. Unfortunately, there are pathological cases where this upper bound can actually be reached, depending on the choice of augmenting paths. The classical example is the following flow network:

We could alternate augmenting our flow

f

along the paths $s \rightarrow a \rightarrow b \rightarrow t$ and then $s \rightarrow b \rightarrow a \rightarrow t$, increasing the value of f

f

by 1

1

in each iteration. In total, this would require $M = 2,000,000$

$M=2,000,000$

iterations.

However, if we instead choose our augmenting paths as $s \rightarrow a \rightarrow t$ and $s \rightarrow b \rightarrow t$, only two iterations are required. This shows that the “correct” choice of augmenting paths can have a significant impact on the method’s runtime. Several refinements of the Ford-Fulkerson method can achieve a better worst-case complexity by choosing augmenting paths wisely. Two well-known variants are

Edmonds-Karp’s algorithm, which runs in $\mathcal{O}(|V| |E|^2)$

$\mathcal{O}(|V| |E|$

2

)

.

Dinic’s algorithm, which runs in $\mathcal{O}(|V|^2 |E|)$

$\mathcal{O}(|V|$

2

|E|)

.

Correctness#

Although the Ford Fulkerson method certainly seems reasonable, so far, we haven’t made a watertight argument about why the returned flow is actually a maximum flow. Can we really be sure that whenever f

f

is not yet maximum, a s

s

-t
t
-path in G_f
G
f

exists?

The next chapter, Max Flow = Min Cut, is optional. In it, we'll prove the correctness of the Ford-Fulkerson method and show the relation of the maximum flow problem to another graph problem, the minimum cut problem. If you're not interested in learning the proof, feel free to skip that chapter.

Max Flow = Min Cut

Explore the connection between maximum flows and minimum cuts.

We'll cover the following

The minimum cut problem

Cuts and flows

Correctness of the Ford-Fulkerson method

Implications of the proof

In this lesson, we want to prove that the Ford-Fulkerson method is actually guaranteed to find a maximum flow. To do this, we'll take a short detour into the territory of another graph problem, finding minimum cuts.

The minimum cut problem#

The minimum cut problem is another problem that can be posed in a flow network $G = (V, E, c)$

$G = (V, E, c)$

. Given a source s

s

and a sink t

t

, an \mathbf{s}

s

\mathbf{t}

-cut is a partition of the vertices V

V

into two subsets S

S

and T

T

, such that $s \in S$

$s \in S$

and $t \in T$

$t \in T$

.

The value $v(S, T)$

$v(S, T)$

of an s

s
-t
t

-cut is the capacity of all edges that cross the cut from S

S
to T
T
.

$$v(S, T) = \sum_{\{u \in S, v \in T, (u, v) \in E\}} c(u, v).$$

$$v(S, T) = \sum_{\{u \in S, v \in T, (u, v) \in E\}} c(u, v).$$

$$c(u, v).$$

As an example, let's take another look at the flow network from the previous lesson:

Here, we've marked up an example cut with $S = \{s, a\}$

$S = \{s, a\}$

(blue) and $T = \{b, t\}$

$T = \{b, t\}$

(red). The edges crossing through the border of the cut are marked in green. The value of the cut is

$$v(S, T) = 1 + 2 + 5 = 8.$$

$$v(S, T) = 1 + 2 + 5 = 8.$$

However, it is not a minimum cut because there is a cut with a smaller value:

This cut $S = \{s, a, b\}$, $T = \{t\}$

$S = \{s, a, b\}$, $T = \{t\}$

is a cut with the minimum value of 3

3

.

Cuts and flows#

There is an interesting relationship between cuts and flows in flow networks. For any cut (S, T)

(S, T)

and any flow f

f

, it holds that

$$v(S, T) \geq v(f).$$

$$v(S, T) \geq v(f).$$

In other words, flows can't be larger than cuts, and cuts can't be smaller than flows.

Let's see why this is the case. We have a flow f

f

which transports $v(f)$

$v(f)$

units from the source s

s

to the sink t

t

. Since $s \in S, t \in T$

$s \in S, t \in T$

, at some point each unit of flow must leave S

S

and enter T

T

, In other words, it must flow through an edge crossing the border between S

S

and T

T

. Hence, $v(S, T)$

$v(S, T)$

, which is the capacity of these border-crossing edges, must be at least $v(f)$

$v(f)$

$).$

This inequality between cuts and flows also allows us to link maximum flows and minimum cuts. Assume that we find a cut (S^*, T^*)

$(S$

*

, T

*

)

and a flow f^*

f

*

of equal value:

$$v(S^*, T^*) = v(f^*).$$

$v(S$

*

, T

*

) = $v(f$

*

$).$

In that case, f^*

f

*

is a maximum flow, and (S^*, T^*)

$(S$

*

, T

*

)

is a minimum cut. There is no larger flow because flows can't be larger than cuts, and there is no smaller cut because cuts can't be smaller than flows.

Correctness of the Ford-Fulkerson method#

We can apply what we've learned about cuts and flows to prove the correctness of the Ford-Fulkerson method.

Recall that Ford-Fulkerson attempts to augment a flow f

by finding an s

s

$-t$

t

-path in the residual network G_f

G

f

. The algorithm terminates when such a path no longer exists.

Let f^*

f

*

be the flow returned by the Ford-Fulkerson method. We can use f^*

f

*

to define an s

s

$-t$

t

-cut (S^*, T^*)

$(S$

*

, T

*

)

. Choose S^*

S

*

to be the vertices that are reachable from s

s

in the residual network. Because there is no s

s

$-t$

t

-path in G_f

G

f

, we have $t \notin S^*$

$t \in$

/

S

*

and can set $T^* = V \setminus S^*$

T

*

$=V \setminus S$

*

to define a cut.

To illustrate this, consider the flow computed by Ford-Fulkerson in our example network:

At this point, the residual network G_f

G

f

is:

Since s

s

, a

a

and b

b

are reachable from s

s

in the residual network, the cut defined by f^*

f

*

is

$S^* = \{s, a, b\}$, $T^* = \{t\}$.

S

*

$=\{s,a,b\}, T$

*

$=\{t\}$.

This is the cut of value 3

3

that we already saw above.

The key insight regarding the cut (S^*, T^*)

(S
*
,T
*
)

is that all edges crossing from S^*

S
*

to T^*

T
*

must be filled to their capacity by f^*

f
*

. On the other hand, assume that there is an edge (u, v)
 (u, v)

with $u \in S^*, v \in T^*$

$u \in S$
*
, $v \in T$
*

with $f^*(u, v) < c(u, v)$

f
*

$(u, v) < c(u, v)$

. This implies that there is an edge (u, v)
 (u, v)

in the residual network G_f

G
f

. Hence we've $v \in S^*$

$v \in S$
*

a contradiction.

Therefore, the value of the cut (S^*, T^*)

(S
*
,T
*
)

is exactly the same as the value of the flow pushed from s

s
to t
t

by f^*
 f
 $*$
 $:$
 $v(S^*, T^*) = v(f^*).$
 $v(S$
 $*$
 $, T$
 $*$
 $) = v(f$
 $*$
 $).$

As we've seen above, this implies that f^*

f
 $*$

is a maximum flow and (S^*, T^*)

$(S$
 $*$
 $, T$
 $*$
 $)$

is a minimum cut, showing that the Ford-Fulkerson method has indeed computed a flow that is maximum.

Implications of the proof#

The proof above for the Ford-Fulkerson method has both practical and theoretical consequences.

On the practical side, we've seen that by starting from the result flow f^*

f
 $*$

of Ford-Fulkerson, we can directly construct a minimum cut (S^*, T^*)

$(S$
 $*$
 $, T$
 $*$
 $)$

. In other words, by performing a depth-first search from s

s

in the residual graph, we can construct a minimum cut. Thus, we can use the Ford-Fulkerson method to solve the minimum cut problem in flow networks.

We have also seen that

$v(S^*, T^*) = v(f^*).$
 $v(S$
 $*$
 $, T$
 $*$

```
)=v(f  
*  
).
```

Since we can run the Ford-Fulkerson method in every flow network, we get a neat theorem of graph theory:

Max-flow-min-cut Theorem: In every flow network, the value of the maximum flow equals the value of the minimum cut.

```
*/  
// ======  
// ======  
/*
```

Implementation of Ford Fulkerson (Edmonds-Karp)

Learn how to implement the Ford-Fulkerson method.

We'll cover the following

Implementation Notes

Implementing Edmonds-Karp

Runtime of Edmonds-Karp

Trying out the implementation

In this lesson, we'll implement the Ford-Fulkerson method to solve maximum flow problems. We'll select a variant of Ford-Fulkerson called Edmonds-Karp, which has a better worst-case runtime.

Implementation Notes#

In principle, we'll follow the general Ford-Fulkerson method:

Initialize a flow f

f

of value 0

0

.

Construct the residual network G_f

G

f

.

Try to find an s

s

-t

t

-path in G_f

G

f

.

If there is such a path, augment the flow f

f

with it, then go to step 2

2

If there is no such path, return the flow f

f

.
Edmonds-Karp's only addition is that for step 3

3

, a breadth-first search is used to find an s

s

$-t$

t

-path in the residual network.

Our input flow network will be represented using the weighted adjacency list data structure, where the edge weights are the capacities. To simplify the implementation, we won't explicitly construct the residual network. Instead, we'll keep track of the flow $f(u, v)$

$f(u, v)$

going through each edge. This is sufficient to compute residual capacities on the fly using the formula

$$c_f(u, v) = c(u, v) - f(u, v) + f(v, u).$$

c

f

$$(u, v) = c(u, v) - f(u, v) + f(v, u).$$

Another technicality is that the residual network $G_f = (V, E_f, c_f)$

G

f

$$= (V, E_f, c_f)$$

)
might contain edges that are not in the original graph. Here is a minimal example:
In the above network, the edge (a, b)
(a, b)

is partially filled by the current flow. There is no edge (b, a)

(b, a)

.

The residual network G_f

G

f

looks like this:

It contains an edge (b, a)

(b,a)
of weight 2
2
which is not in the original graph.

In our implementation, we do not want to deal with the problem of adding or removing edges during the execution. Thus, we'll use the following trick: whenever there is an edge (a, b)

(a,b)
but no reverse edge (b, a)

(b,a)
, we insert an edge (b, a)
(b,a)
of zero capacity.

*/
// ======
// ======
/*

Disjoint Set Union Data Structure

Study the implementation of a DSU data structure.

The following code snippet contains an implementation of a Disjoint Set Union data structure as it's used in the Implementation of Kruskal's algorithm.

```
*/  
// ======  
// #include <vector>  
// #include <numeric>  
// using namespace std;  
  
// class DisjointSetUnion {  
// private:  
//   vector<int> parents;  
//   vector<int> ranks;  
// public:  
//   DisjointSetUnion(int numberOfSets) {  
//     this->parents = vector<int>(numberOfSets);  
//     this->ranks = vector<int>(numberOfSets);  
//     // every elements starts as its own parent -- and representative  
//     iota(this->parents.begin(), this->parents.end(), 0);  
//   }  
  
//   int find(int u) {  
//     // find the root  
//     int root = u;  
//     while (this->parents[root] != root) { root = this->parents[root]; }  
//     // compress paths  
//     while (this->parents[u] != root) {  
//       int tmp = this->parents[u];  
//       this->parents[u] = root;  
//       u = tmp;  
//     }  
//     return root;
```


What's the Reason for my Perception?

What Makes Embedded Programming Special?

The inventor of C++, Bjarne Stroustrup, FAQ makes it crystal clear that one of the design goals for C++11 was to make C++ even better for the embedded world.

According to Bjarne Stroustrup himself,

"Improved performance and ability to work directly with hardware – make C++ even better for embedded systems programming and high-performance computation."

As a programmer, I resonate a lot with this statement, since I changed from software development in the middleware area into the embedded world. I began my search for the features in C++11 that are well suited for the embedded programming. I presented the results of my search at conferences in Berlin, Munich, or Sindelfingen.

My perception of the new features of Modern C++, such as user-defined literals, the type-trait library, or constant expressions has changed a lot. They are such great features that they belong in the toolbox of each professional C++ developer. This statement is even more true for the embedded world because they have to deal with stronger requirements. Therefore, this course is particularly valuable in embedded programming.

What's the Reason for my Perception?#

The answer is simple as there is no typical embedded system and the diversity is large. The diversity is large. There are RFID transponders about the size of a few millimeters, pacemakers that should work reliably for a longer period, defibrillators consisting of a few boards and cars having more than 100 electronic control units (ECUs). The borders between the embedded systems and other systems as game development or low-level systems are blurred.

What Makes Embedded Programming Special?#

Embedded programming is more intensive than other forms of programming because it must handle specific requirements:

High-performance requirements up to real-time requirements

Safety-critical systems

Reduced resources as memory and CPU power

More Tasks that should be accomplished in parallel

I had a long fight on how to present the features in a structured way. In addition, the features are often dependent on the C++ standard. Here is my plan. The features are the answers to typical requirements in the embedded programming. These requirements are the key points of my structure.

High safety requirements

Performance matters

Careful handling of resources

Doing more jobs in parallel

Additionally, I will provide the information about the C++ standard which has these features.

Before we start with the lessons about embedded programming, we have to make a short detour. In my profession as a software developer, I, Rainer Grimm, heard a lot of myths about C++. I will present them in the next lesson and provide the facts in the lesson after the next.

Myths

An overview of myths and prejudices about C++ in embedded programming and what are the reasons behind them.

We'll cover the following

Some Common Myths

Prejudices

Reasons for Prejudices

As I started working in an embedded environment, I was astonished that there was so much prejudice against the usage of C++ in embedded programming. Most of them are based on the wrong understanding of the C++ programming language.

Some Common Myths#

Templates cause code bloat

Objects have to be created on the heap

Exceptions are expensive

C++ is slow and needs too much memory

C++ is too dangerous in safety-critical systems

You must write object-oriented code in C++

C++ can only be used for applications

The iostream library is too big; the STL library too slow

In summation:

C++ is a cute toy but it cannot handle the challenging tasks.

Prejudices#

The list of prejudice is long consisting partially of half-truth and untruth statements often stated by experienced C programmers. I will only refer to the untruth statements. The half-truth statements are, to a large extent, questions due to the right usage of C++ and, to a small extent, questions of the implementation of the core and the libraries of C++.

Objects must live on the heap.

Objects can be created on the stack or at an arbitrary position with the help of placement new.

C++ is too dangerous in safety-critical systems

Of course, it depends on the experience of the developer. But whoever uses C strings instead of C++ strings; uses C arrays instead of C++ arrays; uses macros instead of constant expressions or templates, can not argue that C++ is not well suited for safety-critical systems. Honestly, the contrary holds. C++ has a lot to offer in safety-critical systems.

You must program object-oriented in C++

C++ is a multi-paradigm language, meaning you can solve your problem in an object-oriented, structured, functional, generic, or generative style.

C++ can only be used for applications

C++ can be used for many different kinds of products for example, fire extinguishers, defibrillators, and cars.

Reasons for Prejudices#

What are the reasons for these half-truths? I think, there are more than one reasons.

Old C++ compilers

Some programmers have knowledge that is based on old C++ compilers of the last millennium. These compilers implement the C++98 standard, but they have a large potential for optimization.

Training deficit

Many embedded programmers have only learned C. Similarly, there is little time for programmers to experiment with new technologies, leading to a deficit in practice or knowledge about C++.

Transition from expert to novice

You have to be brave enough to leave your position as a C expert and continue as a C++ novice.

Legacy codebase in C

The existing codebase is in C, meaning that most programmers use C to fix a bug or implement a solution. There are a lot of standards that you have to fulfill. The courage to use new technologies seems to be inversely proportional to the pressure of the standards.

Many C experts

There are many C experts who continue training the novices in C, making C++ a less respected programming tool.

Curse of the monoculture

Embedded world is often a monoculture. I worked for 15 years as a consultant in the automobile area and used about 10 languages. On the contrary, I used only 3 languages in the embedded area.

Insufficient knowledge about C++

Many developers do not have sufficient knowledge of the classical C++ and have no knowledge at all of modern C++.

Maybe, we will polarize with this lesson but if it helps to make the great features of modern C++ better known in the embedded world than we want to do it voluntarily. In the next lesson, we compare the myths with the facts.

Facts

Here are some facts regarding embedded programming with C++, as well as MISRA C++ and AUTOSAR C++14 guidelines.

We'll cover the following

MISRA C++

MISRA C++ Rules

Conclusion

AUTOSAR C++14 Guideline

C++ Core Guidelines

MISRA C++#

The current MISRA C++:2008 guidelines were published by the Motor Industry Software Reliability Association. They are based on the MISRA C guidelines from 1998. Originally designed for the automotive industry, MISRA C++ became the standard for the implementation of critical software in the aviation, military, and medical sector. Just like MISRA C, MISRA C++ also describes guidelines for a safe subset of C++.

This subset consists of more than 200 rules classified as a document, required, or advisory.

Document:

Mandatory requirements on the developer

Derivations are not permitted

Required:

Mandatory requirements for the developer

Formal derivation must be raised

Advisory:

Should be followed as closely as possible

Formal derivation is not necessary but may be considered

MISRA C++ Rules#

Lets's look at some of the important rules regarding the C++ core language and libraries. To make it clearer, we will present a few rules from MISRA C++.

Unnecessary construct

The project shall not contain unreachable code. (required)

The project shall not contain unused variables. (required)

Assembler

All usage of assembler shall be documented. (document)

Arithmetic

Use of floating-point arithmetic shall be documented. (document)

Language

The code shall conform to the C++03 standard (Remark: Small addition to C++98). (required)

Comments

No C comments shall be used to "comment out" code. (required)

No C++ comments shall be used to "comment out" code. (advisory)

Pointer conversions

NULL shall not be used as an integer value. (required)

Multiple base classes

Classes should not be derived from virtual bases. (advisory)

Virtual functions

Each overriding virtual function shall be declared with the virtual keyword. (required)

Exception handling

Exceptions shall only be used for error handling. (document)

Templates

All partial and explicit specializations for a template shall be declared in the same file as the declarations of their primary template. (required)

Macro replacements

The # and ## operators should not be used. (advisory)

Library

The C library shall not be used. (required)

All library code shall conform to MISRA C++. (document)

You can verify these and all the other MISRA C++ rules with static code analysis tools.

Conclusion#

Which conclusions can we draw from the MISRA C++ rules for the usage of C++ in critical systems? Neither one feature nor the whole language is excluded by MISRA C++.

MISRA C++ also emphasizes why C++ in critical systems becomes more important. (1.1 The use of C++ in critical systems):

C++ offers support for high-speed, low-level, input/output operations, which are essential to many embedded systems.

The increased complexity of applications makes the use of a high-level language more appropriate than assembly language.

C++ compilers generate code with similar size and RAM requirements to those of C.

One small issue remains, however. MISRA C++ is based on classical C++, while Modern C++ has more to offer for embedded systems. Sadly, MISRA C++ cannot keep in lockstep with the C++ standardization but there are efforts being made to fill the gap.

AUTOSAR C++14 Guideline#

AUTOSAR C++14 (AUTomotive Open System ARchitecture) consists of guidelines for the usage of C++ in safety-critical systems. AUTOSAR C++14 is based on a more updated MISRA C++ ,and it is used more frequently in the automotive domain. AUTOSAR C++14 is under active development.

In contrast to MISRA C++, AUTOSAR C++14 allows the usage of dynamic memory allocation and operator overloading.

AUTOSAR C++14 disallows the following points:

malloc, free, and C-casts

const_cast, dynamic_cast, and reinterpret_cast

typedef name

Unions

Multiple inheritances

friend declarations

Dynamic exception specifications (throw)

AUTOSAR C++14 overcomes the biggest weakness of MISRA C++. In contrast to MISRA C++14, AUTOSAR C++14 is based on a more updated C+14. However, it shares one significant weakness with MISRA C++: both are the result of formal standardization, meaning it is challenging to keep it standardized with C++.

The driving force behind MISRA C++ and AUTOSAR C++14 is the automotive industry. This does not hold for the C++ core guidelines, which are a C++ community-driven project.

C++ Core Guidelines#

The C++ core guidelines are a community-driven project, which is hosted on Github. The editors are Bjarne Stroustrup and Herb Sutter.

The abstract to the C++ core guidelines describe their goals:

"This document is a set of guidelines for using C++ well. The aim of this document is to help people to use modern C++ effectively. By "Modern C++" we mean C++11 and C++14 (and soon C++17)."

The C++ core guidelines have a similar scope as MISRA C++ and AUTOSAR C++14. The additional guideline support library (GSL) helps to verify the rules of this set of guidelines.

The rules of the C++ core guidelines deal primarily with the following concerns.

Interfaces

Functions

Classes and class hierarchies

Enumerations

Resource management
Expressions and statements
Error handling
Constants and immutability
Templates and generic programming
Concurrency
The Standard library
Source files
C-style programming

More posts to the C++ core guidelines can be found here: Modernes C++.

You can find more info regarding the C++ Core Guidelines here:

<https://www.modernescpp.com/index.php/what-is-modern-c>
<https://www.modernescpp.com/index.php/c-core-guidelines-the-philosophy>

Technical Report on C++ Performance

In this lesson, we will discuss the C++ features, their overhead, and usage in light of Technical Report on C++ Performance.

We'll cover the following

C++ Features, Overhead, and Usage

The Working Group WG 21 published the ISO/IEC TR 18015, a document that is the ultimate source on the performance numbers of the C++ features. The document expresses its concerns directly.

To give the reader a model of time and space overheads implied by use of various C++ language and library features,

To debunk widespread myths about performance problems,

To present techniques for use of C++ in performance applications, and

To present techniques for implementing C++ Standard language and library facilities to yield efficient code.

The paper documents the work of experts like Dave Abrahams, Howard Hinnand, Dietmar Kühl, Dan Saks, Bill Seymour, Bjarne Stroustrup, and Detlef Vollmann.

C++ Features, Overhead, and Usage#

The authors of the Technical Report on C++ Performance used three computer architectures with five different compilers for their analysis. They used compilers with different optimization options. Let's go over why the results of the analysis were quite remarkable.

Namespaces

They have no significant overhead in size and performance.

Type converting operator

The C++ casts `const_cast`, `static_cast`, and `reinterpret_cast` differ neither in size nor in performance from their C predecessor.

However, `dynamic_cast`, which is executed at run time, has some overhead. (Remark: The conversion has no C predecessor.).

Inheritance

Class

A class without virtual functions is as big as a struct.

A class with virtual functions has the overhead of a pointer and a virtual function table. These are about 2 to 4 bytes.

Function calls

The call of a non-virtual, non-static, and non-inline function is as expensive as the call of a free function.

The call of a virtual function is as expensive as the call of a free function with the help of a pointer that is stored in a table.

Virtual functions of a class template can cause overhead in size. (Remark: Functions that do not depend on template parameters should be extracted in a base class. Therefore, the functionality - independent of template parameters - can be shared between all derived class templates.)

The inlining of a function causes significant performance benefits and is close to the performance of a C macro.

Multiple Inheritance

Can cause time and/or space overhead. Virtual base classes have overhead compared to non-virtual base classes.

Run-time type information (RTTI)

There are about 40 additional bytes required for each class necessarily.

The typeid call is quite slow due to the quality of the implementation.

The conversion during runtime with dynamic_cast is slow, likely due to the quality of the implementation.

Exception handling

There are two strategies for dealing with exceptions: the code and the table strategy. The coding strategy has to move and manage additional data structures for dealing with exceptions. The table strategy has the execution context in a table.

The coding strategy has a size overhead for the stack and the runtime. The runtime overhead is about 6%. This overhead exists even without the throwing of an exception.

The table strategy has neither overhead in program size nor in runtime. (Remarks: That statements hold only if no exceptions were thrown.). The table strategy is more difficult to implement.

Templates

For each template instantiation, you get a new class template or function template. Therefore, the naive use of temples can cause code bloat. Modern C++ compilers can massively reduce the number of template instantiations. The usage of partial or full specialization helps to reduce the template instantiation.

You can read other details of the report here [TR18015.pdf](#).

It is important to note that the “Technical Report on C++ Performance” is from 2006. In the case of modern C++, there are many features for writing faster code that are not accounted for in the report. Author Detlef Vollmann states that they plan to update the report to modern C++, though there are some hurdles to overcome.

Now that we have a solid background in C++, let’s go over the specifics of this coding language. In the next chapter, we will talk about the Safety-Critical Systems in Embedded Programming.

```
*/  
// ======  
// ======  
/*
```

Uniform Initialization with {}

In this lesson, we will learn about initialization with {} and how it prevents narrowing.

We'll cover the following

Two Forms of Initialization

Direct Initialization

Copy Initialisation

Preventing Narrowing

The initialization of variables was uniform in C++11.

Two Forms of Initialization#

Rule: A {}-Initialization is always applicable.

Direct Initialization#

```
string str{"my String"};
```

Copy Initialisation#

```
string str = {"my String"};
```

Preventing Narrowing#

The initialization with {} prohibits narrowing conversion.

Narrowing, or more precisely narrowing conversion is an implicit conversion of arithmetic values. This can lead to a less accurate result which is extremely dangerous.

The following example outlines the issue with the classical initialization for fundamental types. It doesn't matter whether we use direct initialization or assignment.

```
/*
// =====
// #include <iostream>

// int main(){

// char c1(999);
// char c2= 999;
// std::cout << "c1: " << c1 << std::endl;
// std::cout << "c2: " << c2 << std::endl;

// int i1(3.14);
// int i2= 3.14;
// std::cout << "i1: " << i1 << std::endl;
// std::cout << "i2: " << i2 << std::endl;

// }
// =====
/*
```

The output of the program shows that there are two warnings in the code. First, the int literal 999 does not fit into the type char. Second, the double literal does not fit into the int type.

That is not possible with {}-initialization.

```
/*
// =====
// narrowingSolved.cpp

// #include <iostream>
```

```

// int main(){

// char c1{999};
// char c2 = {999};
// std::cout << "c1: " << c1 << std::endl;
// std::cout << "c2: " << c2 << std::endl;

// int i1{3.14};
// int i2 = {3.14};
// std::cout << "i1: " << i1 << std::endl;
// std::cout << "i2: " << i2 << std::endl;

// char c3{8};
// std::cout << "c3: " << c3 << std::endl;
//}
//=====
/*

```

There can be confusion while using different versions of GCC compilers. It makes a difference as to which compiler version we are using. With GCC 6.1 and the versions above, we get an error. With the versions below GCC 6.1, we get a warning. To generate an error rather than a warning in the versions below GCC 6.1, we use the `-Werror=narrowing` flag and the program will generate an error instead. You can try this out with this compiler.

In comparison, the clang++ compiler is much more predictable than GCC.

Compile your program in such a way that narrowing is an error.

Look at another example below:

```

*/
//=====
// #include <iostream>
// using namespace std;

// int main()
//{
// char c3{97};
// std::cout << "c3: " << c3 << std::endl;
//}
//=====
/*
copy:::
*/
//=====
// #include <iostream>
// using namespace std;

// int main() {
// char c3 = {97};
// std::cout << "c3: " << c3 << std::endl;
//}
```

```
// =====  
/*
```

In the code above, if we used the expression `char c3{8}`, it will be indeed not narrow since 8 fits in the type `char`. The same holds for `char c3 = {8}`.

Example 1#

```
*/  
// =====  
// uniformInitialization.cpp
```

```
// #include <map>  
// #include <vector>  
// #include <string>
```

```
/// Initialization of a C-Array as attribute of a class
```

```
// class Array{  
// public:  
//   Array(): myData{1,2,3,4,5}{}  
// private:  
//   int myData[5];  
// };
```

```
// class MyClass{  
// public:  
//   int x;  
//   double y;  
// };
```

```
// class MyClass2{  
// public:  
//   MyClass2(int fir, double sec):x{fir},y{sec} {};  
// private:  
//   int x;  
//   double y;  
// };
```

```
// int main(){
```

```
// // Direct Initialization of a standard container  
// int intArray[] = {1,2,3,4,5};  
// std::vector<int> intArray1{1,2,3,4,5};  
// std::map<std::string,int> myMap{{"Scott",1976}, {"Dijkstra",1972}};
```

```
// // Initialization of a const heap array  
// const float* pData= new const float[3]{1.1,2.2,3.3};
```

```
// Array arr;
```

```
// // Default Initialization of a arbitrary object  
// int i{};           // i becomes 0  
// std::string s{};    // s becomes ""
```

```

// std::vector<float> v{}; // v becomes an empty vector
// double d{};           // d becomes 0.0

// // Initializations of an arbitrary object using public attributes
// MyClass myClass{2011,3.14};
// MyClass myClass1 = {2011,3.14};

// // Initializations of an arbitrary object using the constructor
// MyClass2 myClass2{2011,3.14};
// MyClass2 myClass3 = {2011,3.14};

// }
// =====
/*
Explanation#

```

Firstly, the direct initialization of the C array, the std::vector, and the std::map (lines 32 - 34) is quite easy. In the case of the std::map, the inner {}-pairs are the key and value pairs.

The next special use case is the direct initialization of a const C array on the heap (line 36). The special thing about the array arr in line 39 is that C arrays can be directly initialized in the constructor initializer (line 10).

The default initialization in lines 42 - 45 looks quite simple. That does not sound good. Why? Wait for the next section. We directly initialize, in lines 48 and 49, the public attributes of the objects. It is also possible to call the constructor with curly braces (lines 52 and 53).

Example 2#

```

*/
// =====
// initializerList.cpp

// #include <initializer_list>
// #include <iostream>
// #include <string>

// class MyData{
// public:

//   MyData(std::string, int){
//     std::cout << "MyData(std::string, int)" << std::endl;
//   }

//   MyData(int, int){
//     std::cout << "MyData(int, int)" << std::endl;
//   }

//   MyData(std::initializer_list<int>){
//     std::cout << "MyData(std::initializer_list<int>)" << std::endl;
//   }
// };

// template<typename T>

```

```

// void printInitializerList(std::initializer_list<T> inList){
//   for (auto& e: inList) std::cout << e << " ";
// }

// int main(){

// std::cout << std::endl;

// // sequence constructor has a higher priority
// MyData{1, 2};

// // invoke the classical constructor explicitly
// MyData(1, 2);

// // use the classical constructor
// MyData{"dummy", 2};

// std::cout << std::endl;

// // print the initializer list of ints
// printInitializerList({1, 2, 3, 4, 5, 6, 7, 8, 9});

// std::cout << std::endl;

// // print the initializer list of strings
// printInitializerList({"Only", "for", "testing", "purpose."});

// std::cout << "\n\n";
// }
// =====
/*

```

Special Rule: The {} initialization is always applicable. We must remember that if we use automatic type deduction with auto in combination with a {}-initialization, we will get an std::initializer_list in C++14.

Explanation#

When you invoke the constructor with curly braces such as in line 33, the sequence constructor (line 18) is used first. The classical constructor in line 14 serves as a fallback, but this fallback does not work the other way around.

When we invoke the constructor with round braces such as in line 36, the sequence constructor does no fallback for the classical constructor in line 18. The sequence constructor takes a std::initializer_list.

In C++14, auto with {} always generates initializer_list.

```

auto a = {42}; // std::initializer_list<int>
auto b {42}; // std::initializer_list<int>
auto c = {1, 2}; // std::initializer_list<int>
auto d {1, 2}; // std::initializer_list<int>

```

With C++17, the rules are more complicated than intuitive.

```
auto a = {42}; // std::initializer_list<int>
```

```
auto b {42}; // int
auto c = {1, 2}; // std::initializer_list<int>
auto d {1, 2}; // error, too many
You can read the details here.
```

Let's test your understanding with an exercise in the next lesson.

```
/*
// =====
// initializerList.cpp
// #include <array>
// #include <iostream>
// #include <set>
// #include <unordered_set>
// #include <vector>

// int main(){

// std::cout << std::endl;
// std::array<int, 5> myArray = {-10, 5, 1, 4, 5};
// for (auto i: myArray) std::cout << i << " ";
// std::cout << "\n\n";

// std::vector<int> myVector = {-10, 5, 1, 4, 5};
// for (auto i: myVector) std::cout << i << " ";
// std::cout << "\n\n";

// std::set<int> mySet = {-10, 5, 1, 4, 5};
// for (auto i: mySet) std::cout << i << " ";
// std::cout << "\n\n";

// std::unordered_multiset<int> myUnorderedMultiSet = {-10, 5, 1, 4, 5};
// for (auto i: myUnorderedMultiSet) std::cout << i << " ";
// std::cout << "\n";

// std::cout << std::endl;

// }
// =====
/*
```

Automatic Type Deduction: auto

In this lesson, we will discuss the automatic type deduction using auto.

We'll cover the following

The Facts of auto

Key Features

auto-matically Initialized

Sample Code 1

Explanation 1

Sample Code 2

Explanation 2

Refactorization

Sample Code 3

Explanation 3

The Facts of auto#

Automatic type deduction with auto is extremely convenient. Firstly, we save unnecessary typing, in particular with challenging template expressions. Secondly, the compiler does not make human errors.

The compiler automatically deduces the type from the initializer:

```
auto myDoub = 3.14;
```

Key Features#

The techniques for automatic function template argument deductions are used.

It is very helpful in complicated template expressions.

It empowers us to work with unknown types.

It must be used with care in combination with initializer lists.

The following code compares the definition of explicit and deduced types:

```
/*
// =====
// #include <vector>
```

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

```
// int main(){
```

```
// // define an int-value
```

```
// int i= 5;           // explicit
```

```
// auto i1= 5;         // auto
```

```
// // define a reference to an int
```

```
// int& b= i;          // explicit
```

```
// auto& b1= i;         // auto
```

```
// // define a pointer to a function
```

```
// int (*add)(int,int)= myAdd;      // explicit
```

```
// auto add1= myAdd;            // auto
```

```
// // iterate through a vector
```

```
// std::vector<int> vec;
```

```
// for (std::vector<int>::iterator it= vec.begin(); it != vec.end(); ++it){}
```

```
// for (auto it1= vec.begin(); it1 != vec.end(); ++it1) {}
```

```
// }
```

```
// =====
```

```
/*
```

C++ Insights helps us to visualize of the types that the compiler deduces. Andreas Fertig, author of this tool, wrote a few [blog] entries (<https://www.modernescpp.com/index.php/c-insights-type-deduction>) about auto as well.

auto-matically Initialized#

auto determines its type from an initializer, meaning that without an initializer, there is no type and nor variable. Simply put, the compiler takes care of each type that is initialized. This is a nice side effect of auto that is rarely mentioned.

It makes no difference if we forgot to initialize a variable or did not make it because we failed to understand the language. The result is the same: undefined behavior. With auto, we can overcome these errors.

Moving on from that overview, let's implement auto in a few examples. Before moving on, do you know all the rules for the initialization of a variable? If yes, congratulations! Let's move forward. If not, read the article default initialization and all referenced articles in this article before continuing with the examples.

The aforementioned article states that "objects with automatic storage duration (and their sub-objects) are initialized to indeterminate values". This formulation causes more harm than good. Local variables that are not user-defined will not be initialized by default.

In the following samples, we modified the second program of default initialization to make the undefined behavior clearer.

```
/*
// =====
// init.cpp

// #include <iostream>
// #include <string>

// struct T1 {};

// struct T2{
//     int mem;    // Not ok: indeterminate value
// public:
//     T2() {}
// };

// int n;      // ok: initialized to 0

// int main(){

//     std::cout << std::endl;

//     int n;          // Not ok: indeterminate value
//     std::string s;  // ok: Invocation of the default constructor; initialized to ""
//     T1 t1;         // ok: Invocation of the default constructor
//     T2 t2;         // ok: Invocation of the default constructor

//     std::cout << "::n " << n << std::endl;
//     std::cout << "n: " << n << std::endl;
//     std::cout << "s: " << s << std::endl;
//     std::cout << "T2().mem: " << T2().mem << std::endl;

//     std::cout << std::endl;

// }
```

```
// =====
/*
```

Explanation 1#

First, let us discuss the scope resolutions operator :: is used in line 25. :: addresses the global scope. In our case, it is the variable n in line 14.

Curiously enough, the automatic variable n in line 25 has the value 0. n has an undefined value; therefore, the program has undefined behavior. This is also true for the variable mem of the struct T2 since T2().mem returns an undefined value.

Sample Code 2#

Now, we will rewrite the program with the help of auto.

```
/*
// =====
// initAuto.cpp
```

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

// struct T1 {};

// struct T2{
//     int mem = 0; // auto mem= 0 is an error
// public:
//     T2() {}
// };

// auto n = 0;

// int main(){

//     std::cout << std::endl;

//     using namespace std::string_literals;

//     auto n = 0;
//     auto s = ""s;
//     auto t1= T1();
//     auto t2= T2();

//     std::cout << "::n " << ::n << std::endl;
//     std::cout << "n: " << n << std::endl;
//     std::cout << "s: " << s << std::endl;
//     std::cout << "T2().mem: " << T2().mem << std::endl;

//     std::cout << std::endl;
```

```
// }
```

```
// =====
```

```
/*
```

Explanation 2#

Two lines in the source code are especially interesting. Firstly, in line 9, the current standard forbids the code to initialize non-constant members of a struct with auto. Therefore, we must use an explicit type. For more on the C++ standardization committee regarding this issue, read this: article.

Secondly, in line 23, C++14 gets C++ string literals. We build them by using a C string literal ("") and adding the suffix s ("s). For convenience, we already imported that in line 20: using namespace std::string_literals.

The output of the program is not as thrilling as it is only for completeness. T2().mem has the value 0.

Refactorization#

auto supports the refactoring of our code. Firstly, it is very easy to restructure our code when there is no type information. Secondly, the compiler automatically takes care of the right types. What does that mean? We will give an answer in the form of a code snippet. Firstly, examine the code without auto:

```
int a = 5;
int b = 10;
int sum = a * b * 3;
int res = sum + 10;
```

When we replace the variable b of type int by a double 10.5, we must adjust all dependent types, which is laborious and dangerous. We must use the right types to handle the issue of narrowing and other intelligent phenomenons in C++.

```
int a2 = 5;
double b2 = 10.5;
double sum2 = a2 * b2 * 3;
double res2 = sum2 * 10.5;
```

Sample Code 3#

This danger is not present in case of auto. Everything happens automatically. Let us an example of this:

```
/*
// =====
// refactorAuto.cpp
```

```
// #include <typeinfo>
// #include <iostream>

// int main()
//{
//    std::cout << std::endl;

//    auto a = 5;
//    auto b = 10;
//    auto sum = a * b * 3;
//    auto res = sum + 10;
//    std::cout << "typeid(res).name(): " << typeid(res).name() << std::endl;

//    auto a2 = 5;
//    auto b2 = 10.5;
//    auto sum2 = a2 * b2 * 3;
```

```

// auto res2 = sum2 * 10;
// std::cout << typeid(res2).name() : " << typeid(res2).name() << std::endl;

// auto a3 = 5;
// auto b3 = 10;
// auto sum3 = a3 * b3 * 3.1f;
// auto res3 = sum3 * 10;
// std::cout << typeid(res3).name() : " << typeid(res3).name() << std::endl;

// std::cout << std::endl;
//}
//=====
/*
Explanation 3#

```

The small variations of the code snippet always determine the right type of res, res2, or res3 which is the job of the compiler. The variable b2 in line 17 is of type double and therefore, res2 is also double.

The variable sum3 in line 24 becomes a float due to multiplication with the literal float 3.1f (a float type). Therefore, the final result, res3, is also a float type. To access the data type from the compiler, we have used the typeid operator which is defined in the header typeinfo.

Often, developers in the embedded domain do not need the correct type but rather a concrete type, such as int. In the case of this example, this is a nice trick. When we switch from the implicit type with auto to the concrete type within, we must make the assignment with the help of {} braces (int res3 = {sum3 * 10});. Thanks to {} the compiler checks if a narrowing conversion has taken place. If we get an error, we know the current type is not what we expected.

Example 1#

```

*/
//=====

// auto.cpp

// #include <iostream>
// #include <vector>

// int func(int) { return 2011; }

// int main()
// {

//     auto i = 5;          // int
//     auto &intRef = i;    // int&
//     auto *intPoint = &i;  // int*
//     const auto constInt = i; // const int
//     static auto staticInt = 10; // static int

//     std::vector<int> myVec;
//     auto vec = myVec; // std::vector<int>
//     auto &vecRef = vec; // std::vector<int>&

```

```

// int myData[10];
// auto v1 = myData; // int*
// auto &v2 = myData; // int (&)[10]

// auto myFunc = func; // (int)()(int)
// auto &myFuncRef = func; // (int)(&)(int)

// // define a function pointer
// int (*myAdd1)(int, int) = [](int a, int b)
// { return a + b; };

// // use type inference of the C++11 compiler
// auto myAdd2 = [](int a, int b)
// { return a + b; };

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

// // use the function pointer
// std::cout << "myAdd1(1, 2) = " << myAdd1(1, 2) << std::endl;

// // use the auto variable
// std::cout << "myAdd2(1, 2) = " << myAdd2(1, 2) << std::endl;

// std::cout << "\n";
// }
// =====
/*
Explanation#

```

In the example above, the compiler automatically deduces the types depending on the value stored in the variable. The corresponding types of variables are mentioned in the in-line comments.

In line 10, we have defined a variable, i and its type is deduced to be int because of the value 5 stored in it.

In lines 11 - 14, we have copied the values into different variables and their type is deduced automatically depending on the value stored in it.

Similarly, in lines 17 - 18, we copy a vector and its reference by using the assignment operator =. auto keyword takes care of vec and vecRef types.

In lines 24 - 25, auto determines the type of myFunc as function pointer and myFuncRef as a reference to that function.

In line 31, we have defined a lambda expression whose return type is inferred by the C++ compiler since we have used the auto keyword.

Example 2#

```

*/
// =====
// autoExplicit.cpp
// #include <iostream>

```

```

// #include <chrono>
// #include <future>
// #include <map>
// #include <string>
// #include <thread>
// #include <tuple>

// int main(){

// auto myInts = {1, 2, 3};
// auto myIntBegin = myInts.begin();

// std::map<int, std::string> myMap = {{1, std::string("one")}, {2, std::string("two")}};
// auto myMapBegin = myMap.begin();

// auto func = [] (const std::string& a){ return a;};

// auto futureLambda = std::async([]{ return std::string("Hello"); });

// auto begin = std::chrono::system_clock::now();

// auto pa = std::make_pair(1, std::string("second"));

// auto tup = std::make_tuple(std::string("second"), 4, 1.1, true, 'a');

// }
// =====
/*

```

Explanation#

In the example above, we used the `auto` keyword in the highlighted lines and left it for the compiler to infer the type during the run time. Since we are handling different C++ libraries when writing extensive codes, it becomes difficult to keep track of each type. `auto` helps by bypassing this problem.

```

*/
// =====
// C++ 14
// #include <chrono>
// #include <functional>

// #include <future>
// #include <initializer_list>
// #include <map>
// #include <string>
// #include <thread>
// #include <tuple>

// int main(){

// std::initializer_list<int> myInts = {1, 2, 3};
// std::initializer_list<int>::iterator myIntBegin = myInts.begin();

// std::map<int, std::string> myMap = {{1, std::string("one")}, {2, std::string("two")}};
```

```
// std::map<int, std::string>::iterator myMapBegin = myMap.begin();  
  
// std::function<std::string(const std::string&)> func = [](const std::string& a){ return a; };  
  
// std::future<std::string> futureLambda = std::async([]{ return std::string("Hello"); });  
  
// std::chrono::time_point<std::chrono::system_clock> begin = std::chrono::system_clock::now();  
  
// std::pair<int, std::string> pa = std::make_pair(1, std::string("second"));  
  
// std::tuple<std::string, int, double, bool, char> tup = std::make_tuple(std::string("second"), 4, 1.1, true, 'a');  
  
//}  
// ======  
/*
```

Explanation#

We have changed the code in the exercise and replaced the auto keywords with their explicit deduction types.

In line 14, we have defined an `std::initializer_list<int>` and in line 15, we have defined its iterator `std::initializer_list<int>::iterator`.

In line 17, we have defined a map `std::map<int, std::string>` and in line 18, we have defined its iterator `std::map<int, std::string>::iterator`

In line 20, we have defined a `std::function<std::string(const std::string&)>` for the lambda expression.

In line 22, we have defined a `std::future<std::string>`.

In line 24, we have defined `std::chrono::time_point<std::chrono::system_clock>`.

In line 26, we have defined `std::pair<int, std::string>`.

In line 28, we have defined a tuple with `std::tuple<std::string, int, double, bool, char>`.

Replacing auto with explicit types can be a difficult task since the user must remember the relevant details for the libraries in use. As an embedded systems programmer, you must work with features from multiple libraries, and the auto helps you handle those features.

For further information, read here [auto](#).

```
*/  
// ======  
// ======  
/*
```

Scoped Enumerations

In this lesson, we will learn about Enumerations or Enums.

We'll cover the following

Drawbacks of Enumerations in Classical C++

Strongly-typed Enumerations

Enumerations conveniently define integer constants with names. These integer constants are called enumerators. Let's go over some of the drawbacks of using classic enums.

Drawbacks of Enumerations in Classical C++#

A short reminder. Three drawbacks of enumerations:

The enumerators implicitly convert to int.

They introduce the enumerators in the enclosing scope.

The type of the enumeration cannot be specified.

Regarding point 3, enumerations cannot be forward declared since their type unknown. We can only guarantee for the enumerators in classical C++. The type must be integral and large enough to hold the enumerators.

Points 1 and 2 are more surprising.

```
/*
// =====
// enumClassic.cpp

// #include <iostream>

// int main(){

// std::cout << std::endl;

// enum Colour{red= 0,green= 2,blue};

// std::cout << "red: " << red << std::endl;
// std::cout << "green: " << green << std::endl;
// std::cout << "blue: " << blue << std::endl;

// int red2= red;

// std::cout << "red2: " << red2 << std::endl;

// // int red= 5; ERROR

// }
```

On the one hand, the enumerators red, green, and blue are known in the enclosing scope, meaning that the definition of the variable red in line 19 is not possible. On the other hand, red can be implicitly converted to int.

If we do not use a name for an enumeration like enum{red, green, blue}, the enumerators will be introduced in the enclosing scope.

But that surprise ends with C++11. Now that we have gone over the drawbacks, let's move on to strongly-typed enumerations.

Strongly-typed Enumerations#

Scoped enumerations are often called strongly-typed enumerations. The strongly-typed enumerations have to follow stronger rules:

The enumerators can only be accessed in the scope of the enumeration.

The enumerators don't implicitly convert to int.

The enumerators aren't imported in the enclosing scope.

The type of the enumerators is by default int. Therefore, we can forward the enumeration.

The syntactical difference between the classic enumerations and the strongly-typed enumerations is minimal.

The strongly-typed enumerations additionally get the keyword class or struct.

If we want to use an enumerator as an int, we have to explicitly convert it with static_cast.

```
/*
// =====
// enumCast.cpp
```

```
// #include <iostream>
```

```
// enum OldEnum{
//   one= 1,
//   ten=10,
//   hundred=100,
//   thousand= 1000
//};
```

```
// enum struct NewEnum{
//   one= 1,
//   ten=10,
//   hundred=100,
//   thousand= 1000
//};
```

```
// int main(){
```

```
// std::cout << std::endl;
```

```
// std::cout << "C++11= " << 2*thousand + 0*hundred + 1*ten + 1*one << std::endl;
// std::cout << "C++11= " << 2*static_cast<int>(NewEnum::thousand) +
//           0*static_cast<int>(NewEnum::hundred) +
//           1*static_cast<int>(NewEnum::ten) +
//           1*static_cast<int>(NewEnum::one) << std::endl;
```

```
// }
```

```
// =====
```

```
/*
```

In order to calculate or output the enumerators, we must convert the enumerators into integral types.
Neither the addition nor the output of strongly-typed enumerations is defined.

In this lesson, we have discussed classical versus strongly-typed enumerations. Commonly, these are known as scoped and unscoped enumerations.

```
/*
// =====
```

```
// =====  
/*
```

Example

In the following example, we will go over the implementation of Enumerations.

We'll cover the following

Example

Example#

We can explicitly specify the type of enumerators. By default, it's int however, we can change the type being used. We can use integral types such bool, char, short int, long int, or, long long int. Read msdn.microsoft.com for the details. Read the post Check types to see as to how we can check at compile time if a type is integral.

We can independently use the scoped property and the explicit type specification of an enumeration. Dependent on the base types, the enumerations have different sizes.

Below is an example of the implementation of scoped enumerations using struct and class.

```
*/  
// =====
```

```
// enum.cpp
```

```
// #include <iostream>
```

```
// enum class Color1{
```

```
// red,
```

```
// blue,
```

```
// green
```

```
};
```

```
// enum struct Color2: char{
```

```
// red = 100,
```

```
// blue, // 101
```

```
// green // 102
```

```
};
```

```
// void useMe(Color2 color2){
```

```
// switch(color2){
```

```
// case Color2::red:
```

```
// std::cout << "Color2::red" << std::endl;
```

```
// break;
```

```
// case Color2::blue:
```

```
// std::cout << "Color2::blue" << std::endl;
```

```
// break;
```

```
// case Color2::green:
```

```
// std::cout << "Color2::green" << std::endl;
```

```
// break;
```

```
// }
```

```
// }
```

```
// int main(){
```

```

// std::cout << std::endl;

// std::cout << "static_cast<int>(Color1::red): " << static_cast<int>(Color1::red) << std::endl;
// std::cout << "static_cast<int>(Color2::red): " << static_cast<int>(Color2::red) << std::endl;

// std::cout << std::endl;

// std::cout << "sizeof(Color1)= " << sizeof(Color1) << std::endl; //int
// std::cout << "sizeof(Color2)= " << sizeof(Color2) << std::endl; //char

// std::cout << std::endl;

// Color2 color2Red{Color2::red};
// useMe(color2Red);

// std::cout << std::endl;

//}

// =====
/*
nullptr Instead of 0 or NULL
In this lesson, we will learn how new null pointer nullptr cleans up in C++ with the ambiguity of the number 0
and the macro NULL.

```

We'll cover the following

The Number 0

The macro NULL

nullptr - The Null Pointer Constant

Generic Code

The Number 0#

The issue with the literal 0 is that it can be either the null pointer (`void* 0`) or the number 0 depending on the context of the problem in question.

Therefore, programs using the number 0 should be confusing.

```

*/
// =====
// null.cpp

// #include <iostream>
// #include <typeinfo>

// int main(){

// std::cout << std::endl;

// int a = 0;
// int* b = 0;
// auto c = 0;
// std::cout << typeid(c).name() << std::endl;

```

```

// auto res = a + b + c;
// std::cout << "res: " << res << std::endl;
// std::cout << typeid(res).name() << std::endl;

// std::cout << std::endl;

//}
// =====
/*

```

The question remains, what is the data type of variable c in line 12 and of variable res in line 15?

The variable c is of type int and the variable res is of type pointer to int: int*. Pretty simple, right? The expression a + b + c in line 15 is pointer arithmetic.

The macro NULL#

The issue with the null pointer NULL is that it implicitly converts to int.

According to en.cppreference.com the macro NULL is an implementation-defined null pointer constant. A possible implementation is as follows:

```

#define NULL 0
//since C++11
#define NULL nullptr

```

But that will not apply to our platform. NULL seems to be of the type long int. We will return to this point later. The usage of the macro NULL raises some questions.

```

*/
// =====
// nullMacro.cpp

// #include <iostream>
// #include <typeinfo>

// std::string overloadTest(int){
//   return "int";
// }

// std::string overloadTest(long int){
//   return "long int";
// }

// int main(){

//   std::cout << std::endl;

//   int a = NULL;
//   int* b = NULL;
//   auto c = NULL;
//   // std::cout << typeid(c).name() << std::endl;
//   // std::cout << typeid(NULL).name() << std::endl;

```

```

// std::cout << "overloadTest(NULL) = " << overloadTest(NULL) << std::endl;
// std::cout << std::endl;

//}
//=====
/*

```

The compiler complains about the implicit conversion to int in line 19. The warning in line 21 is confusing. The compiler automatically deduces the type of the variable c to long int. At the same time, it complains that the expression NULL must be converted. Our observation is in accordance with the call overloadTest(NULL) in line 26. The compiler uses the version for the type long int (line 10). If the implementation uses NULL of type int, the compiler will choose overloadTest for the parameter type int (line 6), which is according to the C++ standard.

Now, we want to know the current type of the null pointer constant NULL. Therefore, we will comment out lines 22 and 23 of the program.

```

*/
//=====
// nullMacro.cpp

// #include <iostream>
// #include <typeinfo>

// std::string overloadTest(int){
//   return "int";
// }

// std::string overloadTest(long int){
//   return "long int";
// }

// int main(){

//   std::cout << std::endl;

//   int a = NULL;
//   int* b = NULL;
//   auto c = NULL;
//   std::cout << typeid(c).name() << std::endl;
//   std::cout << typeid(NULL).name() << std::endl;

//   std::cout << "overloadTest(NULL) = " << overloadTest(NULL) << std::endl;

//   std::cout << std::endl;

//}

//=====
/*

```

One one hand, NULL seems to be of type long int but on the other hand, it seems to be a constant pointer. This behavior shows the compilation of the above program.

So, we can conclude the following: Don't use the macro NULL.

The new null pointer constant nullptr can solve this problem.

nullptr - The Null Pointer Constant#

The new null pointer nullptr cleans up in C++, with the ambiguity of the number 0 and the macro NULL. nullptr is and remains of type std::nullptr_t.

We can assign arbitrary pointers to a nullptr. The pointer becomes a null pointer and points to no data. We cannot dereference a nullptr. However, pointers of this type can be compared with all pointers and be converted to all pointers. This also holds true for pointers to class members, but we cannot compare and convert a nullptr to an integral type. There is one exception to this rule: we can implicitly compare and convert a bool value with a nullptr, meaning that we can use a nullptr in a logical expression.

Generic Code#

In the generic code, the literal 0 and NULL reveal their true nature. Due to template argument deduction, both literals are integral types in the function template. There is no indication that both literals were null pointer constants.

```
/*
// =====
// generic.cpp

// #include <cstddef>
// #include <iostream>

// template <class P>
// void functionTemplate(P p)
//{
// int *a = p;
//}

// int main()
//{
// int *a = 0;
// int *b = NULL;
// int *c = nullptr;

// functionTemplate(0);
// functionTemplate(NULL);
// functionTemplate(nullptr);
//}
// =====
/*  
We can use 0 and NULL to initialize the int pointer in line 12 and 13. However, if we use the values 0 and NULL as arguments of the function template, the compiler will loudly complain. The compiler deduces 0 in the function template to type int, and it deduces NULL to the type long int. These observations will not hold true for the nullptr, however.  
*/
// =====
// =====
```

```
/*
Example#
*/
// =====
// nullptr.cpp
// #include <iostream>
// #include <string>

// std::string overloadTest(char*){
//   return "char*";
// }

// std::string overloadTest(int){
//   return "int";
// }

// std::string overloadTest(long int){
//   return "long int";
// }

// std::string test2(char*){
//   return "char*";
// }

// int main(){

//   std::cout << std::endl;

//   int* pi = nullptr; // OK
//   // int i= nullptr; // cannot convert 'std::nullptr_t' to 'int'
//   bool b{nullptr}; // OK. b is false.

//   std::cout << std::boolalpha << "b: " << b << std::endl;

//   // // calls int
//   std::cout << "overloadTest(0) = " << overloadTest(0) << std::endl;

//   // // calls char*
//   std::cout << "overloadTest(static_cast<char*>(0))= " << overloadTest(static_cast<char*>(0)) << std::endl;
//   std::cout << "test2(0)= " << test2(0) << std::endl;

//   // // calls char*
//   std::cout << "overloadTest(nullptr)= " << overloadTest(nullptr)<< std::endl;

//   // // call of overloaded 'overloadTest(NULL)' is ambiguous
//   std::cout << "overloadTest(NULL)= " << overloadTest(NULL) << std::endl;

//   std::cout << std::endl;

// }
// =====
```

```
/*
Explanation#
```

The nullptr can be used to initialize a pointer of type int (line 25). However, the nullptr cannot be used to initialize a variable of type int (line 26).

The null pointer constant behaves like a boolean value, which is initialized with false (line 27). If the nullptr has to decide between a long int and a pointer, it will result in a pointer (line 39).

Simple rule to remember: Use nullptr instead of 0 or NULL.

```
/*
// =====
// =====
/*
```

User-Defined Literals: Introduction

In this lesson, we will introduce user-defined literals.

We'll cover the following

User-Defined Literals

Syntax

Examples

The Magic

User-Defined Literals#

User-defined literals are a unique feature in all mainstream programming languages, since they enable us to combine values with units.

Syntax#

Literals are explicit values in a program, including a boolean like true, the number 3 or 4.15; the char a, or the C string "hello". The lambda function [](int a, int b){ return a+b; } is also a function literal.

With C++11, it is possible to generate user-defined literals by adding a suffix to a built-in literal for the int, float, char, and C strings.

User-defined literals must obey the following syntax:

```
<built_in-Literal> _ <Suffix>
```

Usually, we use the suffix for a unit:

Examples#

```
101000101_b
63_s
10345.5_dm
123.45_km
100_m
131094_cm
33_cent
"Hello"_i18n
```

What is the key benefit of user-defined literals? The C++ compiler maps the user-defined literals to the corresponding literal operator, and this literal operator must be implemented by the programmer.

The Magic#

Let's take a look at the user-defined literal 0101001000_b which represents a binary value. The compiler maps the user-defined literal 0101001000_b to the literal operator operator""_b (long long int bin).

A few special rules are important to follow:

There must be a space between the quotation marks and the suffix.

The binary value (0101001000) is in the variable bin.

If the compiler doesn't find the corresponding literal operator, the compilation will fail.

With C++14, we get an alternative syntax for user-defined types. They differ from the C++11 syntax because user-defined types in require no space. Therefore, it is possible to use reserved keywords like _C as a suffix and use a user-defined literal of the form. This can be seen with 11_C. The compiler will map 11_C to the literal operator""_C (unsigned long long int). The simple rule states that we can use suffixes starting with an upper-case letter.

User-defined literals are a very helpful feature in modern C++ if we want to write safety-critical software. Why? Due to the automatic mapping of the user-defined literal to the literal operator, we can implement type-safe arithmetic.

```
/*
// =====
// =====
/*
Example#
*/
// =====
// userDefinedLiteral.cpp
// #include <iostream>
// #include <ostream>

// namespace Distance{
// class MyDistance{
// public:
//     MyDistance(double i):m(i){}

//     friend MyDistance operator +(const MyDistance& a, const MyDistance& b){
//         return MyDistance(a.m + b.m);
//     }
//     friend MyDistance operator -(const MyDistance& a, const MyDistance& b){
//         return MyDistance(a.m - b.m);
//     }

//     friend std::ostream& operator<< (std::ostream &out, const MyDistance& myDist){
//         out << myDist.m << " m";
//         return out;
//     }
// private:
//     double m;

// };

// namespace Unit{
```

```

// MyDistance operator "" _km(long double d){
//   return MyDistance(1000*d);
// }
// MyDistance operator "" _m(long double m){
//   return MyDistance(m);
// }
// MyDistance operator "" _dm(long double d){
//   return MyDistance(d/10);
// }
// MyDistance operator "" _cm(long double c){
//   return MyDistance(c/100);
// }
// }

// using namespace Distance::Unit;

// int main(){

// std::cout << std::endl;

// std::cout << "1.0_km: " << 1.0_km << std::endl;
// std::cout << "1.0_m: " << 1.0_m << std::endl;
// std::cout << "1.0_dm: " << 1.0_dm << std::endl;
// std::cout << "1.0_cm: " << 1.0_cm << std::endl;

// std::cout << std::endl;
// std::cout << "1.0_km + 2.0_dm + 3.0_dm - 4.0_cm: " << 1.0_km + 2.0_dm + 3.0_dm - 4.0_cm << std::endl;
// std::cout << std::endl;

// }
// =====
/*

```

Explanation#

The literal operators are implemented in the namespace `Distance::unit`. We should use namespaces for user-defined literals due to two reasons. First, the suffixes are typically quite short. Second, the suffixes typically stand for units that are already established abbreviations. In the program, we used the suffixes km, m, dm, and cm.

For the class `MyDistance`, we overloaded the basic arithmetic operators (lines 10 - 15) and the output operator (lines 17 - 19). The operators are global functions, and can use - thanks to their friendship - the internals of the class. We store the output distance in the private variable `m`.

We display, in lines 48 - 51, the various distances. In lines 27 to 37, we calculate the meter in various resolutions. The literal operators take as argument a long double and return a `MyDistance` object. `MyDistance` is automatically normalized to meters.

The last test looks quite promising: `1.0_km + 2.0_dm + 3.0_dm - 4.0_cm` is `1000.54 m` (line 55). The compiler takes care of the calculations with all units.

```

*/
// =====

```

```
// =====
/*
Solution#
*/
// =====
// userDefinedLiteralExtended.cpp
// #include <iostream>
// #include <ostream>

// namespace Distance
//{
// class MyDistance
// {
// public:
// explicit MyDistance(double i) : m(i) {}

// friend MyDistance operator+(const MyDistance &a, const MyDistance &b)
// {
// return MyDistance(a.m + b.m);
// }
// friend MyDistance operator-(const MyDistance &a, const MyDistance &b)
// {
// return MyDistance(a.m - b.m);
// }

// friend MyDistance operator*(double m, const MyDistance &a)
// {
// return MyDistance(m * a.m);
// }

// friend std::ostream &operator<<(std::ostream &out, const MyDistance &myDist)
// {
// out << myDist.m << " m";
// return out;
// }

// private:
// double m;
// };

// namespace Unit
// {
// MyDistance operator"" _mi(long double d)
// {
// return MyDistance(1609.344 * d);
// }
// MyDistance operator"" _km(long double d)
// {
// return MyDistance(1000 * d);
// }
// MyDistance operator"" _m(long double m)
```

```

// {
//   return MyDistance(m);
// }
// MyDistance operator"" _ft(long double d)
// {
//   return MyDistance(0.348 * d);
// }
// MyDistance operator"" _dm(long double d)
// {
//   return MyDistance(d / 10);
// }
// MyDistance operator"" _cm(long double c)
// {
//   return MyDistance(c / 100);
// }
// }

// using namespace Distance::Unit;

// int main()
//{
// std::cout << std::endl;
// std::cout << "1.0_mi: " << 1.0_mi << std::endl;
// std::cout << "1.0_km: " << 1.0_km << std::endl;
// std::cout << "1.0_m: " << 1.0_m << std::endl;
// std::cout << "1.0_ft: " << 1.0_ft << std::endl;
// std::cout << "1.0_dm: " << 1.0_dm << std::endl;
// std::cout << "1.0_cm: " << 1.0_cm << std::endl;
//
// std::cout << std::endl;
// std::cout << "0.001 * 1.0_km: " << 0.001 * 1.0_km << std::endl;
// std::cout << "10 * 1_dm: " << 10 * 1.0_dm << std::endl;
// std::cout << "100 * 1.0cm: " << 100 * 1.0_cm << std::endl;
//
// std::cout << std::endl;
// std::cout << "1.0_km + 2.0_dm + 3.0_dm + 4.0_cm: " << 1.0_km + 2.0_dm + 3.0_dm + 4.0_cm << std::endl;
// std::cout << std::endl;
//
// Distance::MyDistance work = 63.0_km;
// Distance::MyDistance workPerDay = 2 * work;
// Distance::MyDistance abbreviationToWork = 5400.0_m;
// Distance::MyDistance workout = 2 * 1600.0_m;
// Distance::MyDistance shopping = 2 * 1200.0_m;
//
// Distance::MyDistance myDistancePerWeek = 4 * workPerDay - 3 * abbreviationToWork + workout +
// shopping;

```

```
// std::cout << "4 * workPerDay - 3 * abbreviationToWork + workout + shopping: " << myDistancePerWeek;  
// std::cout << "\n\n";  
//  
// ======  
/*
```

Explanation#

The mile unit is implemented in lines 31-32.

The feet unit is implemented in lines 40-41.

The components work, workPerDay, workout, abbreviationToWork, and shopping are defined in lines 75-79.

In line 81, we have computed myDistancePerWeek using the components we have defined above.

In the next lesson, we will learn about the raw and cooked forms of user-defined literals.

User-Defined Literals: Raw and Cooked

In this lesson, we will study raw and cooked forms of user-defined literals.

We'll cover the following

The Four User-Defined Literals

Raw and Cooked Forms

The Four User-Defined Literals#

C++11 has user-defined literals for characters, C strings, integers, and floating point numbers. For integers and floating point numbers, they are available in raw and cooked form. Due to C++14, we have built-in literals for binary numbers, C++ strings, complex numbers, and time units.

To clarify, below are the raw and cooked variations of the literal types.

How should we read the table? The data type character has the form of character_suffix, such as the example at 's'_c. The compiler tries to invoke the literal operator operator""_c('s'). The character, in this case, is of the type char. C++ also supports the data type char, the data type wchar_t, char16_t, and char32_t. We can use these types as a base for our C string. In the table, we used a char. The table shows that the compiler maps the C string "hi"_i18 to the literal operator operator""_i18n("hi",2). The length of the C string is 2.

Raw and Cooked Forms#

The compiler can map integers or floating point numbers to integers (unsigned long long int) or floating point numbers (long double), but the compiler can also map them to C strings.

The first variant is called cooked form, and the second variant is called raw form. The compiler will use the raw form if the literal operator wants its arguments as C string. If not, the compiler uses the cooked form. If we implement both versions, the compiler will choose the cooked form since it has higher priority.

Admittedly, the last lines have a lot of potential for confusion. To solve this, we sum up all the information from the perspective of the signatures in the following table. The first column has the signature of the literal

operator. The second column has the type of the user-defined literal, and the last column is an example of a user-defined literal that fits the signature of the literal operator.

Let's look at an example of this topic in the next lesson.

```
/*
// =====
// =====
/*
- Example
We'll take a look at an example of raw and cooked forms of user-defined literals.
*/
// =====
// average.cpp

// #include "distance.h"
// #include "unit.h"

// using namespace Distance::Unit;

// int main()
//{
// std::cout << std::endl;

// std::cout << "1.0_km: " << 1.0_km << std::endl;
// std::cout << "1.0_m: " << 1.0_m << std::endl;
// std::cout << "1.0_dm: " << 1.0_dm << std::endl;
// std::cout << "1.0_cm: " << 1.0_cm << std::endl;

// std::cout << std::endl;

// std::cout << "0.001 * 1.0_km: " << 0.001 * 1.0_km << std::endl;
// std::cout << "10 * 1_dm: " << 10 * 1.0_dm << std::endl;
// std::cout << "100 * 1.0cm: " << 100 * 1.0_cm << std::endl;
// std::cout << "1_.0km / 1000: " << 1.0_km / 1000 << std::endl;

// std::cout << std::endl;
// std::cout << "1.0_km + 2.0_dm + 3.0_dm + 4.0_cm: " << 1.0_km + 2.0_dm + 3.0_dm + 4.0_cm << std::endl;
// std::cout << std::endl;

// auto work = 63.0_km;
// auto workPerDay = 2 * work;
// auto abbreviationToWork = 5400.0_m;
// auto workout = 2 * 1600.0_m;
// auto shopping = 2 * 1200.0_m;

// auto distPerWeek1 = 4 * workPerDay - 3 * abbreviationToWork + workout + shopping;
// auto distPerWeek2 = 4 * workPerDay - 3 * abbreviationToWork + 2 * workout;
// auto distPerWeek3 = 4 * workout + 2 * shopping;
// auto distPerWeek4 = 5 * workout + shopping;
```

```
// std::cout << "distPerWeek1: " << distPerWeek1 << std::endl;

// auto averageDistance = getAverageDistance({distPerWeek1, distPerWeek2, distPerWeek3,
distPerWeek4});
// std::cout << "averageDistance: " << averageDistance << std::endl;

// std::cout << std::endl;
//}
/// =====
/// distance.h

// #ifndef DISTANCE_H
// #define DISTANCE_H

// #include <iostream>
// #include <ostream>

// namespace Distance
//{
// class MyDistance
// {
// public:
// MyDistance(double i) : m(i) {}

// friend MyDistance operator+(const MyDistance &a, const MyDistance &b)
// {
// return MyDistance(a.m + b.m);
// }
// friend MyDistance operator-(const MyDistance &a, const MyDistance &b)
// {
// return MyDistance(a.m - b.m);
// }

// friend MyDistance operator*(double m, const MyDistance &a)
// {
// return MyDistance(m * a.m);
// }

// friend MyDistance operator/(const MyDistance &a, int n)
// {
// return MyDistance(a.m / n);
// }

// friend std::ostream &operator<<(std::ostream &out, const MyDistance &myDist)
// {
// out << myDist.m << " m";
// return out;
// }

// private:
// double m;
```

```

// };

// }

// Distance::MyDistance getAverageDistance(std::initializer_list<Distance::MyDistance> inList)
//{
// auto sum = Distance::MyDistance{0.0};
// for (auto i : inList)
//   sum = sum + i;
// return sum / inList.size();
//}

// #endif
/// unit.h

// #ifndef UNIT_H
// #define UNIT_H

// #include "distance.h"

// namespace Distance{

//   namespace Unit{
//     MyDistance operator "" _km(const char* k){
//       return MyDistance(1000* std::stold(k));
//     }
//     MyDistance operator "" _m(const char* m){
//       return MyDistance(std::stold(m));
//     }
//     MyDistance operator "" _dm(const char* d){
//       return MyDistance(std::stold(d)/10);
//     }
//     MyDistance operator "" _cm(const char* c){
//       return MyDistance(std::stold(c)/100);
//     }
//   }
// }

// #endif
// =====
/*
Explanation#
We did our computations based on user-defined literals of the type long double in the cooked form. To make our calculation in the raw form, we must only adjust the literal operators.

```

It is only necessary to convert the arguments of the literal operator from type C string to long double. That is quite easy to do with the new function std::stold.

```

*/
// =====
// =====
/*

```

Built-in Literals

A shorthand introduction to the new built-in literals in C++ 14.

We'll cover the following

New Built-in Literals with C++14

New Built-in Literals with C++14#

In C++ 14, there are a few new built-in literals. These are built-in literals for binary numbers, C++ strings, complex numbers, and time units. At first, let go over an overview of this new concept.

image.png

You must keep a few special rules in mind. There is a main different between the built-in literals and the user-defined literals: the built-in literals have no underscore. For the first time, C++ supports (with C++14) a C++ string literal. So far, C++ only supported C-string literals, meaning that we must always use a C-string literal to initialize a C++ string. The time literals are also very convenient since they implicitly know their unit and support basic arithmetic. They are of the type std::chrono::duration.

```
/*
// =====
// =====
/*
```

Example

In this lesson, we will look at an example for built-in literals.

We'll cover the following

Example

Explanation

Example#

Children often complain that their school day is exhausting. The question arises, how many seconds does a child need for a typical school day? The program provides the answer.

The base unit for time is second.

```
/*
// =====
// #include <iostream>
// #include <chrono>

// using namespace std::literals::chrono_literals;

// int main()
// {

//   std::cout << std::endl;

//   // typedef std::chrono::duration<long long, std::ratio<2700>> hour;
//   auto schoolHour = hour(1);
//   // // auto schoolHour= 45min;

//   auto shortBreak = 300s;
//   auto longBreak = 0.25h;
```

```

// auto schoolWay = 15min;
// auto homework = 2h;

// auto schoolDayInSeconds = 2 * schoolWay + 6 * schoolHour + 4 * shortBreak + longBreak + homework;

// std::cout << "School day in seconds: " << schoolDayInSeconds.count() << std::endl;

// std::chrono::duration<double, std::ratio<3600>> schoolDayInHours = schoolDayInSeconds;
// std::chrono::duration<double, std::ratio<60>> schoolDayInMinutes = schoolDayInSeconds;
// std::chrono::duration<double, std::ratio<1, 1000>> schoolDayInMilliseconds = schoolDayInSeconds;

// std::cout << "School day in hours: " << schoolDayInHours.count() << std::endl;
// std::cout << "School day in minutes: " << schoolDayInMinutes.count() << std::endl;
// std::cout << "School day in milliseconds: " << schoolDayInMilliseconds.count() << std::endl;

// std::cout << std::endl;
//}
// =====
/*
Explanation#
The program is entirely self-explanatory. The suffixes are expressive enough. Making the correct additions is the job of the compiler. The time literals support the base arithmetic addition, subtraction, multiplication, division, and modulo operation.
*/
// =====
// =====
/*

```

Assertions at Compile Time

In this lesson, we will learn about assertions at compile time in modern C++.

We'll cover the following

`static_assert`

`static_assert` is the tool in modern C++ used to make our code safe.

`static_assert#`

The usage of `static_assert` is quite easy. `static_assert` requires an expression and a string. The expression must be predicate that can be evaluated at compile time. Predicate means the expression returns true or false. If the expression evaluates to false, we will get an error message at compile-time with the string as a message. Of course, we get no executable.

There are a few points we must consider.

The `static_assert` expression will be evaluated at compile-time, and we have no runtime overhead.

Expressions that can be evaluated at compile time are called constant expressions.

We can use `static_assert` expressions in all parts of our program. Therefore, it is a good idea to put general requirements on our source code in a separate header. As a result, the `static_assert` expression will be

automatically verified at compile time if we include the header. This helps to port our code to a new platform since we can easily check if the new platform supports the type requirements.

The example in the next lesson will build on your understanding of this concept.

Example#

```
/*
// =====
// staticAssert.cpp
// #include <iostream>
// #include <type_traits>

// template <class T>
// struct Add
//{
//   // check the assertion
//   static_assert(std::is_arithmetic<T>::value, "Argument T must be an arithmetic type");
// };

// int main()
//{
//   // will work
//   static_assert(sizeof(void *) >= 8, "64-bit addressing is required for this program");

//   // int is arithmetic
//   Add<int> addInt = Add<int>();

//   // double is arithmetic
//   Add<double> addDouble = Add<double>();

//   // char is arithmetic
//   Add<char> addChar = Add<char>();

//   // std::string is not arithmetic
//   Add<std::string> addString = Add<std::string>(); // if you comment this line, the code will run fine
//}

/*
Explanation#
The program uses static_assert in the class scope (line 9) and the local scope (line 16). The assertions in the class definition guarantee that the structure is initialized with an arithmetic type, explaining why the template instantiation in line 28 is not valid.
```

```
/*
// =====
// =====
```

```

/*
Exercise
In this lesson, you will use assertions to ensure that the algorithm only accepts natural numbers.

We'll cover the following

Task
Task#
Ensure that the algorithm gcd only accepts natural numbers. Test your algorithm.
*/
// =====
// staticAssertGcd.cpp

// #include <iostream>
// #include <type_traits>

// template <typename T>
// T gcd(T a, T b)
// {
//     static_assert(std::is_integral<T>::value, "T should be an integral type!");
//     if (b == 0)
//     {
//         return a;
//     }
//     else
//     {
//         return gcd(b, a % b);
//     }
// }

// int main()
// {
//     std::cout << std::endl;

//     std::cout << gcd(3.5, 4.0) << std::endl; // should be gcd(3, 4)
//     std::cout << gcd("100", "10") << std::endl; // should be gcd(100, 10)

//     std::cout << std::endl;
// }
// =====
/*

```

Explanation#
The `static_assert` operator and the predicate `std::is_integral<T>::value` enable us to check at compile time whether or not `T` is an integral type. A predicate always returns a boolean value.

The compilation will not fail by accident since the modulo operator is not defined for a double value and a C string, but the compilation fails since the assertion in line 8 will not hold true. Now, we get an exact error message rather than a cryptic output of a failed template instantiation.

The rule is quite simple: **If a compilation must fail, we will get an unambiguous error message. **

For further information, see static_assert.

The power of static_assert can be evaluated at compile time. We have the new type-trait library in C++11. The powerful type-trait library empowers you to check, compare, and change types at compile time. We will discuss this concept in the next section.

```
*/  
// ======  
// ======  
/*
```

Rvalue and Lvalue References

Rvalues and Lvalues#

Rvalues are

Temporary objects

Objects without a name

Objects from which we can not get the address

If one of these characteristics holds for an object, it is an rvalue. On the other hand, values with a name and an address are lvalues.

Lvalues can be on the left side of an assignment operator.

Rvalues can only be on the right side of an assignment operator.

A few examples of rvalues:

```
int five= 5;  
std::string a= std::string("Rvalue");  
std::string b= std::string("R") + std::string("value");  
std::string c= a + b;  
std::string d= std::move(b);
```

As previously stated, Rvalues are on the right side of an assignment. The value 5 and the constructor call std::string("Rvalue") are rvalues since we cannot determine either the address or the name of the value 5. The same holds for the addition of the rvalues in the expression std::string("R") + std::string("value").

The addition of the two strings a + b is notable. Both strings are lvalues, but the addition creates a temporary object. A special use case is std::move(b). The new C++11 function converts the type of the lvalue b into an rvalue reference.

A Few Exceptions#

Earlier, we stated that rvalues are on the right side of an assignment, and lvalues can be on the left side of an assignment. It is important to note that this is not always true:

```
const int five= 5;  
five= 6;
```

Even though the variable five is an lvalue, it is also a constant, so it cannot be used on the left side of an assignment operator.

Lvalue references are declared by one & symbol. Rvalue references are declared by two && symbols. Lvalues can be bound to lvalue references, and rvalues can be bound to rvalue references or constant lvalue references.

```
MyData myData;  
  
MyData& lvalueRef(myData);  
  
MyData&& rvalueRef(MyData());  
const MyData& constLValueRef(MyData());  
The binding of rvalues to rvalues references has higher priority.
```

Rvalue References: Applications#
Move Semantic#
Cheap moving of objects instead of expensive copying.
No memory allocation and deallocation.
Non-copyable but moveable objects can be transferred by value.
We will discuss this concept in more detail in a later lesson.

Perfect Forwarding#
Forward an object without changing the value categories.
We will discuss this concept in more detail in a later lesson.

```
*/  
// ======  
// rvalueReference.cpp  
// #include <algorithm>  
// #include <iostream>  
// #include <string>  
  
// struct MyData{};  
  
// std::string function( const MyData & ) {  
//   return "lvalue reference";  
// }  
  
// std::string function( MyData && ) {  
//   return "rvalue reference";  
// }  
  
// int main(){  
  
//   std::cout << std::endl;  
  
//   MyData myData;  
  
//   std::cout << "function(myData): " << function(myData) << std::endl;  
//   std::cout << "function(MyData()): " << function(MyData()) << std::endl;  
//   std::cout << "function(std::move(myData)): " << function(std::move(myData)) << std::endl;  
  
//   std::cout << std::endl;  
  
// }  
// ======  
/*
```

Explanation#

The code above is a simple example of rvalue and lvalue references.

In line 22, myData is an lvalue since it has a name and address.

In line 23, MyData() is an rvalue since it has neither a name nor a reference. Rather, this rvalue is a call to the default constructor of the struct MyData.

In line 24, std::move(myData) creates an rvalue reference as well since you can neither determine the address of the myData nor the created object's name.

In the next lesson, we will learn about the differences between the copy semantic and the move semantic.

```
/*
// =====
// =====
/*
```

Copy versus Move Semantic

In this lesson, we will compare the performance of the copy and move semantic for the containers in the Standard Template Library (STL).

We'll cover the following

Copy vs. Move

Some Important Points to Remember

std::swap

Explanation

A lot has been written on the advantages of the move semantic over the copy semantic. Rather than an expensive copy operation, we can use a cheap move operation. Let's break that down further.

There is one subtle difference between copy and move semantic: if we create a new object based on an existing one, the copy semantic will copy the elements of the resource, while the move semantic will move the elements of the resource. Of course, copying is expensive, and moving is cheap, but there are additional serious consequences to this technique:

With copy semantic, a std::bad_alloc will be thrown because the program is out of memory.

The resource of the move operation is afterward in a "valid but unspecified state".

The second point is demonstrated clearly with std::string.

Copy vs. Move#

Copy

```
string str1("ABCDEF");
string str2;
str2 = str1;
```

Move

```
string str1("ABCDEF");
string str3;
str3 = std::move(str1);
```

In copy semantic, both strings str1 and str2 have the same content “ABCDEF” after the copy operation. So, what is the difference between copy and move semantic?

The string str1 is in opposition with the copy semantic afterward empty. This is not guaranteed but is often the case. We explicitly requested the move semantic with the function std::move. The compiler will automatically perform the move semantic if it ensures that the source of the move semantic is no longer needed.

We explicitly request the move semantic in our program by using std::move. Although it is called std::move, we should have a different picture in mind. When we move, we transfer ownership. By moving, the object is given to someone else

Some Important Points to Remember#

A class supports copy semantic if the class has both a copy constructor and a copy assignment operator.

A class supports move semantic if the class has both a move constructor and a move assignment operator.

If a class has a copy constructor, it should also have a copy assignment operator. The same holds for the move constructor and move assignment operator.

std::swap#

Below is an example of the process of using the move semantic and the copy semantic to swap two variables. The copy version resembles the way that we performed this function with C++11. It shows how the move semantic is more efficient and saves memory.

```
std::vector<int> a, b;
```

```
swap(a, b);
```

```
template <typename T>
```

```
void swap(T& a, T& b){
```

```
    T tmp(a);
```

```
    a = b;
```

```
    b = tmp;
```

```
}
```

```
template <typename T>
```

```
void swap(T& a, T& b){
```

```
    T tmp(std::move(a));
```

```
    a = std::move(b);
```

```
    b = std::move(tmp);
```

```
}
```

Explanation#

This is what the T tmp(a); command essentially performs:

Allocates tmp in stack and elements of tmp in the heap.

Copies each element from a to tmp.

The T tmp(std::move(a)); command

Redirects the pointer from tmp to a.

```
*/
```

```
// =====
```

```
// =====
```

/*
Move Semantic

We will talk about some important, often overlooked properties of the move semantic in this lesson.

We'll cover the following

std::move

STL

Example

User-Defined Data Types

Example:

The Strategy of the Move Constructor

Automatically Generated Methods

Rule of Zero or Rule of Six

Containers of the standard template library (STL) can have non-copyable elements. The copy semantic is the fallback for the move semantic. Let's learn more about the move semantic.

std::move#

The function std::move moves its resource.

The function needs the header <utility>.

The function converts the type of its argument into a rvalue reference.

The compiler applies move semantic to the rvalue reference.

std::move is under the hood a static_cast to an rvalue reference.

static_cast<std::remove_reference<decltype(arg)>::type&&>(arg);

What is happening here?

decltype(arg): deduces the type of the argument

std::remove_reference<....> removes all references from the type of the argument

static_cast<....&&> adds two references to the type

Copy semantic is a fallback for move semantic. This means if we invoke std::move with a non-moveable type, copy-semantic is used. This is due to the fact that an rvalue can be bound to an rvalue reference and a constant lvalue reference.

STL#

Each container of the STL and std::string gets two new methods:

Move constructor

Move assignment operator

These new methods get their arguments as non-constant rvalue references.

Example#

```
vector{  
    vector(vector&& vec);           //move constructor  
    vector& operator = (vector&& vec); //move assignment  
    vector(const vector& vec);       //copy constructor  
    vector& operator = (const vector& vec); // copy assignment
```

The classical copy constructor and copy assignment operator get their arguments as constant lvalue references.

User-Defined Data Types#

User-defined data types can support the move and copy semantics as well.

Example:#

```
class MyData{  
    MyData(MyData&& m) = default;  
    MyData& operator = (MyData&& m) = default;  
    MyData(const MyData& m) = default;  
    MyData& operator = (const myData& m) = default;  
};
```

The move semantic has priority over the copy semantic.

The Strategy of the Move Constructor#

Set the attributes of the new object.

Move the content of the old object.

Set the old object in a valid state.

The move constructor is automatically created if all attributes of the class and all base classes have one move constructor. The automatically-created move constructor delegates its job to the attributes of the class and all base classes.

This rule holds for the big six:

Default constructor

Destructor

Move and copy constructor

Move and copy assignment operator

Automatically Generated Methods#

The following table shows the dependency generated when you create one of the big six or a constructor.

The key idea is it to start simple when you create a class. The compiler automatically creates the big six if your class and the base classes use

built-in data-types

Containers of the STL or std::string

When your class has a pointer, put it into a std::unique_ptr or std::shared_ptr depending on the semantic you want to express

std::shared_ptr can be copied

std::unique_ptr cannot be copied

*/

// =====

// =====

/*

user-declared: a method which is used (defined, defaulted, or deleted).

defaulted: a method that the compiler generates or a method that is requested via the default.

How should you read this table?

Start with the first column and move upwards.

When you create Nothing, you get all from the constructor to the move assignment operator to the right.

When you create Any constructor, you get no default constructor but the remaining five special methods.

When you create a default constructor, you get all five remaining specials methods.

And so on ...

We must explicitly mention that the compiler can only create special methods if our class is simple enough. If you want to have move details to the automatically created methods, watch Howard Hinnands presentation titles, Everything you need to know about move semantic.

Rule of Zero or Rule of Six#

The rule of zero or the rule of six means that you must implement one or all of the six special methods.

Do not implement one of the six special methods if not necessary. When the compiler complains because one of the six special methods is not available, think about the semantic of your user-defined type, and implement the missing methods.

```
/*
// =====
// =====
/*
```

- Examples

Some examples of copy and move semantics will be discussed in this lesson.

We'll cover the following

Example 1

Explanation

Example 2

Explanation

Example 3

Explanation

Example 1#

```
/*
// =====
// copyMoveSemantic.cpp
// #include <iostream>
// #include <string>
// #include <utility>

// int main(){

// std::string str1{"ABCDEF"};
// std::string str2;

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

// // initial value
// std::cout << "str1= " << str1 << std::endl;
// std::cout << "str2= " << str2 << std::endl;

// // copy semantic
// str2= str1;
// std::cout << "str2= str1;\n";
// std::cout << "str1= " << str1 << std::endl;
// std::cout << "str2= " << str2 << std::endl;

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

// std::string str3;

// // initial value
// std::cout << "str1= " << str1 << std::endl;
```

```

// std::cout << "str3= " << str3 << std::endl;
// // move semantic
// str3= std::move(str1);
// std::cout << "str3= std::move(str1);\n";
// std::cout << "str1= " << str1 << std::endl;
// std::cout << "str3= " << str3 << std::endl;

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

//}
//=====
/*
Explanation#

```

In the above example, we demonstrated how the value of str1 can be transferred to strings using two different methods: copy semantic and move semantic.

In line 18, we used the copy semantic, and the string "ABCDEF" is present in both str1 and str2. We can, therefore, say that the value has been copied from str1 to str2.

In line 32, we used the move semantic, and now the string "ABCDEF" is present only in str3 but not in str1. We can therefore say the value has moved from str1 to str3.

Example 2#

```

*/
// =====
// swap.cpp
// #include <algorithm>
// #include <iostream>
// #include <vector>

// template <typename T>
// void swap(T &a, T &b)
//{
// T tmp(std::move(a));
// a = std::move(b);
// b = std::move(tmp);
//}

// struct MyData
//{
// std::vector<int> myData;

// MyData() : myData({1, 2, 3, 4, 5}) {}

// // copy semantic
// MyData(const MyData &m) : myData(m.myData)
// {
// std::cout << "copy constructor" << std::endl;
// }

```

```

// MyData &operator=(const MyData &m)
// {
//   myData = m.myData;
//   std::cout << "copy assignment operator" << std::endl;
//   return *this;
// }
//};

// int main()
//{
// std::cout << std::endl;

// MyData a, b;
// swap(a, b);

// std::cout << std::endl;
// };
// =====
/*

```

Explanation#

The example shows the workings of a simple swap function that uses the move semantic internally. MyData does not support move semantic.

Line 9 invokes the move constructor in line 20.

Lines 10 and 11 invoke the move assignment operator defined in line 24.

When you invoke move on a copyable type, copy-semantic will begin. This is due to the fact that an rvalue is first bound to an rvalue reference, and the second is bound to a const lvalue reference.

Copy semantic is a fallback for move semantic.

Example 3#

```

*/
// =====
// bigArray.cpp
// #include <algorithm>
// #include <chrono>
// #include <iostream>
// #include <vector>

// using std::cout;
// using std::endl;

// using std::chrono::duration;
// using std::chrono::system_clock;

// using std::vector;

// class BigArray

```

```
// {  
  
// public:  
// BigArray(size_t len) : len_(len), data_(new int[len]) {}  
  
// BigArray(const BigArray &other) : len_(other.len_), data_(new int[other.len_])  
// {  
//   cout << "Copy construction of " << other.len_ << " elements " << endl;  
//   std::copy(other.data_, other.data_ + len_, data_);  
// }  
  
// BigArray &operator=(const BigArray &other)  
// {  
//   cout << "Copy assignment of " << other.len_ << " elements " << endl;  
//   if (this != &other)  
//   {  
//     delete[] data_;  
  
//     len_ = other.len_;  
//     data_ = new int[len_];  
//     std::copy(other.data_, other.data_ + len_, data_);  
//   }  
//   return *this;  
// }  
  
// ~BigArray()  
// {  
//   if (data_ != nullptr)  
//     delete[] data_;  
// }  
  
// private:  
// size_t len_;  
// int *data_;  
// };  
  
// int main()  
// {  
  
//   cout << endl;  
  
//   vector<BigArray> myVec;  
  
//   auto begin = system_clock::now();  
  
//   myVec.push_back(BigArray(1000000000));  
  
//   auto end = system_clock::now() - begin;  
//   auto timeInSeconds = duration<double>(end).count();  
  
//   cout << endl;
```

```
// cout << "time in seconds: " << timeInSeconds << endl;
// cout << endl;
//}
// =====
/*
```

Explanation#

BigArray only supports copy semantic. This is a performance issue in line 54. The containers of the standard template library have copy-semantic.

This means that the containers want to copy all elements. If BigData had implemented move semantic implemented, it would have been used automatically in line 54 since the constructor call BigArray(1000000000) creates an rvalue.

Copy semantic is a fallback for move semantic.

```
/*
// =====
// =====
/*
```

Task 1#

In the program below, a BigArray with 10 billion entries will be pushed to an std::vector. Compile the program and measure its performance. The program requires that you compile this program for 64-bit.

Task 2#

Extend BigArray with move semantic and measure the performance once more.

How big is the performance gain?

```
/*
// =====
/// //bigArray.cpp
// #include <algorithm>
// #include <chrono>
// #include <iostream>
// #include <vector>

// using std::cout;
// using std::endl;

// using std::chrono::system_clock;
// using std::chrono::duration;

// using std::vector;

// class BigArray{

// public:
//   BigArray(size_t len): len_(len), data_(new int[len]){}

//   BigArray(const BigArray& other): len_(other.len_), data_(new int[other.len_]){
//     cout << "Copy construction of " << other.len_ << " elements " << endl;
//     std::copy(other.data_, other.data_ + len_, data_);
//   }
```

```

// BigArray& operator=(const BigArray& other){
//   cout << "Copy assignment of " << other.len_ << " elements " << endl;
//   if (this != &other){
//     delete[] data_;

//     len_ = other.len_;
//     data_ = new int[len_];
//     std::copy(other.data_, other.data_ + len_, data_);
//   }
//   return *this;
// }

// ~BigArray(){
//   if (data_ != nullptr) delete[] data_;
// }

// private:
// size_t len_;
// int* data_;
// };

// int main(){

// cout << endl;

// vector<BigArray> myVec;

// auto begin= system_clock::now();

// myVec.push_back(BigArray(1000000000));

// auto end= system_clock::now() - begin;
// auto timeInSeconds= duration<double>(end).count();

// cout << endl;
// cout << "time in seconds: " << timeInSeconds << endl;
// cout << endl;

// }
// =====
/*
Explanation#
In lines 37-40, we defined the move constructor for BigArray. Note that in line 40, we explicitly set other.data_ to nullptr after the elements have been moved into the new object.

In lines 43-55, we defined the move assignment operator = for BigArray. Note that in line 51, we explicitly set other.data_ to nullptr after the elements have been moved.

```

As you can see, move semantic is way quicker than the copy semantic since we are only redirecting the pointer to an already stored data and assigning the correct size. Thus, the costs of move semantics are

independent of the size of the data structure. This does not hold for the copy semantic. Memory allocation becomes more and more expensive the bigger the size of the user-defined type.

```
/*
// =====
// =====
/*
```

Perfect Forwarding

We will learn about perfect forwarding in this lesson.

We'll cover the following

A Perfect Factory Method

First Iteration

Second Iteration

Third Iteration - std::forward

Fourth Iteration - The Perfect Factory Method

If a function template forwards its arguments without changing their lvalue or rvalue characteristics, we call it perfect forwarding.

A Perfect Factory Method#

Firstly, a short disclaimer: the expression a perfect factory method is not a formal term.

A perfect factory method is actually a generic factory method, meaning that the function should have the following characteristics:

It can take an arbitrary number of arguments

It can accept lvalues and rvalues as an argument

It forwards its arguments identical to the underlying constructor

In other words, a perfect factory method should be able to create each arbitrary object.

Perfect forwarding enables it to write a function that can identically forward its arguments. In doing so, the lvalue and rvalue properties are respected.

Let's start with the first iteration and move towards implementing the perfect factory method.

First Iteration#

For efficiency reasons, the function template should take its arguments by reference. To say it exactly, it should take it as a non-constant lvalue reference. The following is the function template create in our first iteration.

```
/*
// =====
// perfectForwarding1.cpp

// #include <iostream>

// template <typename T, typename Arg>
// T create(Arg &a)
// {
//     return T(a);
// }
```

```

// int main()
//{
// std::cout << std::endl;

// // Lvalues
// int five = 5;
// int myFive = create<int>(five);
// std::cout << "myFive: " << myFive << std::endl;

// // Rvalues
// int myFive2 = create<int>(5);
// std::cout << "myFive2: " << myFive2 << std::endl;

// std::cout << std::endl;
//}
//=====
/*

```

If we compile the program, we will get a compiler error, due to the fact that the rvalue (line 21) cannot be bound to a non-constant lvalue reference.

Now, we have two ways to solve the issue.

Change the non-constant lvalue reference (line 6) in a constant lvalue reference. We can bind an rvalue to a constant lvalue reference. But that is not perfect since the function argument is constant and we can therefore not change it.

Overload the function template for a constant lvalue reference and a non-const lvalue reference. This is the easiest and better technique to implement.

Second Iteration#

Here is the factory method create overloaded for a constant lvalue reference and a non-constant lvalue reference

```

*/
//=====
// perfectForwarding2.cpp

// #include <iostream>

// template <typename T,typename Arg>
// T create(Arg& a){
// return T(a);
// }

// template <typename T,typename Arg>
// T create(const Arg& a){
// return T(a);
// }

// int main(){

// std::cout << std::endl;
```

```

// // Lvalues
// int five=5;
// int myFive= create<int>(five);
// std::cout << "myFive: " << myFive << std::endl;

// // Rvalues
// int myFive2= create<int>(5);
// std::cout << "myFive2: " << myFive2 << std::endl;

// std::cout << std::endl;

// }
// =====
/*

```

The solution has two conceptional issues.

To support n different arguments, we must overload $2^n + 1$

```

2
n
+1
variations of the function template create.  $2^n + 1$ 
2
n
+1
because the function create without an argument is part of the perfect factory method.

```

The function argument materializes in the function body of create to an lvalue since it has a name. a is not movable anymore. Therefore, we have to perform an expensive copy rather than a cheap move. If the constructor of T (line 12) needs an rvalue, it will not work anymore.

Now, we have the solution in the shape of the C++ function std::forward.

Third Iteration - std::forward#

With std::forward, the solution looks promising:

```

*/
// =====
// perfectForwarding3.cpp

// #include <iostream>

// template <typename T,typename Arg>
// T create(Arg&& a){
//   return T(std::forward<Arg>(a));
// }

// int main(){

// std::cout << std::endl;

// // // Lvalues
// int five=5;

```

```
// int myFive= create<int>(five);
// std::cout << "myFive: " << myFive << std::endl;

// // Rvalues
// int myFive2= create<int>(5);
// std::cout << "myFive2: " << myFive2 << std::endl;

// std::cout << std::endl;

// }
// =====
/*
```

Example#

```
/*
// =====
// perfectForwarding.cpp
// #include <iostream>
// #include <string>
// #include <utility>

// template <typename T, typename T1>
// T create(T1 &&t1)
// {
//   return T(std::forward<T1>(t1));
// }

// int main()
// {

// std::cout << std::endl;

// // Lvalues
// int five = 5;
// int myFive = create<int>(five);
// std::cout << "myFive: " << myFive << std::endl;

// std::string str{"Lvalue"};
// std::string str2 = create<std::string>(str);
// std::cout << "str2: " << str2 << std::endl;

// // Rvalues
// int myFive2 = create<int>(5);
// std::cout << "myFive2: " << myFive2 << std::endl;

// std::string str3 = create<std::string>(std::string("Rvalue"));
// std::cout << "str3: " << str3 << std::endl;

// std::string str4 = create<std::string>(std::move(str3));
// std::cout << "str4: " << str4 << std::endl;
```

```
// std::cout << std::endl;
// };
// =====
/*
```

Explanation#

We used the universal reference in line 7 of the code so it can bind rvalues or lvalues.

In lines 17 and 21, we called the function create using lvalues five and str.

In lines 25 and 28, we called the function create using the rvalues 5 and Rvalue.

We implemented an interesting technique in line 31. We called the function create with an rvalue reference of str3 generated by using the function std::move.

Let's test your understanding of this topic with an exercise in the next lesson.

```
*/
// =====
// =====
/*
```

Memory Management: Memory Allocation

In this lesson, we will learn about a subsection of memory management - memory allocation.

We'll cover the following

Introduction

Memory Allocation

new

new[]

Placement new

Typical use-cases

Failed Allocation

New Handler

Introduction#

Explicit memory management in C++ is highly complex, but it also provides us with great functionality.

Unfortunately, this special domain is not as common in C++. For example, you can directly create objects in static memory, in a reserved area, or even in a memory pool. This functionality is often key in the safety-critical applications of the embedded world.

C++ enables the dynamic allocation and deallocation of memory.

Dynamic memory (heap) must be explicitly requested and released by the programmer.

You can use the operators new and new[] to allocate memory and the operators delete and delete[] to deallocate the memory.

The compiler manages its memory automatically on the stack.

Smart pointers manage the memory automatically.

Memory Allocation#

new#

Due to the operator new, you can dynamically allocate memory for the instance of a type.

```
int* i = new int;
double* d = new double(10.0);
Point* p = new Point(1.0, 2.0);
```

new causes the memory allocation and the object initialization.

The arguments in the brackets go directly to the constructor.

new returns the address of the object that has been given memory.

If the class of dynamically created objects is part of a type hierarchy, more constructors are invoked.

new[]#

new[] creates a C array of objects. The newly created objects need a default constructor.

```
double* d = new double[5];
Point* p = new Point[10];
```

The class of the allocated object must have a default constructor.

The default constructor will be invoked for each element of the C.

The STL Containers and the C++ String automatically manage their memory.

Placement new#

Placement new is often used to instantiate an object in a pre-reserved memory area. In addition, you can overload placement new globally or for your own data types. This is a big benefit that C++ offers.

```
char* memory = new char[sizeof(Account)]; // allocate std::size_t
Account* acc = new(memory) Account; // instantiate acc in memory
```

The header <new> is necessary

Placement new can be overload on a class basis or global

Typical use-cases#

Explicit memory allocation

Avoidance of exceptions

Debugging

Failed Allocation#

If the memory allocation fails, new and new[] will raise a std::bad_alloc exception. This is not the desired response. Rather, you can invoke placement new with the constant std::nothrow. This call generates a nullptr in the error case

```
char* c = new(std::nothrow) char[10];
if (c){
    delete c;
}
else{
    // an error occurred
}
```

New Handler#

In the case of a failed allocation, you can use std::set_new_handler with your own handler.

std::set_new_handler returns the older handler and needs a callable unit. A callable unit is typically a function, a function object, or a lambda-function. The callable unit should take no argument and return nothing. We can get the currently-used handler by invoking the function std::get_new_handler.

Your own handler enables you to implement special strategies for failed allocations:

request more memory
terminate the program with std::terminate
throw an exception of type std::bad_alloc
In the next lesson, we will learn how to deallocate memory.

```
*/  
// ======  
// ======  
/*
```

Memory Management: Memory Deallocation

In this lesson, we will learn about the second subsection of memory management - memory deallocation.

We'll cover the following

Memory Deallocation

delete
delete[]
Placement Delete
Memory Deallocation#
delete#

A new with previously allocated memory will be deallocated with delete.

```
Circle* c= new Circle;  
...  
delete c;
```

The destructors of the object and the destructors of all base classes will be automatically called. If the destructor of the base class is virtual, we can destroy the object with a pointer or reference to the base class.

After the memory of the object is deallocated, the access to the object is undefined. We must initialize the pointer of the object to a point it to a different object.

The deallocation of new[] allocated object with delete has undefined behavior.

delete[]#
You must use the operator delete[] for the deallocation of a C array that was allocated with new[].

```
Circle* ca= new Circle[8];  
...  
delete[] ca;
```

By invoking delete[], all destructors of the objects will automatically be invoked.

The deallocation new allocated object with delete[] is undefined behavior.

Placement Delete#

According to placement new, you can implement placement delete. The C++ runtime will not automatically call placement delete. Therefore, it is the programmer's duty to do so.

A commonly used strategy is to invoke the destructor in the first step and to delete it in the second step. The destructor deinitializes the object, and the placement delete deallocates the memory.

```
char* memory= new char[sizeof(Account)];  
Account* a= new(memory) Account; // placement new
```

```
...
a->~Account();           // destructor
operator delete(a, memory); // placement delete
```

In the next lesson, we will learn how to overload the new and delete operators.

```
*/
// =====
// =====
/*
```

Memory Management: Overloading Operator new and delete 1

In this lesson, we will learn how to overload the operators new and delete so we can manage memory in a better way.

We'll cover the following

The Baseline

Operator new

Operator delete

Counting Allocations and Deallocation

Addresses of the Memory Leaks

Comparison of the Memory Addresses

It happens quite often that a C++ application allocates memory but does not deallocate it. This is the job for the operators new and delete. Due to both features, you can explicitly manage the memory management of an application.

Occasionally, we must verify that an application has correctly released its memory. In particular, for programs running for long periods of time, it is a challenge to allocate and deallocate memory from a memory management perspective. Of course, the automatic release of the memory during the shutdown of the program is not an option.

The Baseline#

As a baseline for our analysis, we use a simple program that often allocates and deallocates memory.

```
/*
// =====
// overloadOperatorNewAndDelete.cpp
```

```
// #include "myNew.hpp"
// #include "myNew2.hpp"
// #include "myNew3.hpp"
```

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

```
// class MyClass
//{
// float *p = new float[100];
//};
```

```
// class MyClass2
//{
// int five = 5;
```

```
// std::string s = "hello";
// };

// int main()
//{
// int *myInt = new int(1998);
// double *myDouble = new double(3.14);
// double *myDoubleArray = new double[2]{1.1, 1.2};

// MyClass *myClass = new MyClass;
// MyClass2 *myClass2 = new MyClass2;

// delete myDouble;
// delete[] myDoubleArray;
// delete myClass;
// delete myClass2;

// // getInfo();
//}

// // =====
// // =====

/// myNew.hpp

// #ifndef MY_NEW
// #define MY_NEW

// #include <cstdlib>
// #include <iostream>
// #include <new>

// static std::size_t alloc{0};
// static std::size_t dealloc{0};

// void *operator new(std::size_t sz)
//{
// alloc += 1;
// return std::malloc(sz);
//}

// void operator delete(void *ptr) noexcept
//{
// dealloc += 1;
// std::free(ptr);
//}

// void getInfo()
//{
// std::cout << std::endl;
}
```

```
// std::cout << "Number of allocations: " << alloc << std::endl;
// std::cout << "Number of deallocations: " << dealloc << std::endl;

// std::cout << std::endl;
//}

// #endif // MY_NEW
// =====

// =====
// myNew2.hpp

// #ifndef MY_NEW2
// #define MY_NEW2

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <string>
// #include <array>

// int const MY_SIZE = 10;

// std::array<void *, MY_SIZE> myAlloc{
//     nullptr,
// };

// void *operator new(std::size_t sz)
// {
//     static int counter{};
//     void *ptr = std::malloc(sz);
//     myAlloc.at(counter++) = ptr;
//     return ptr;
// }

// void operator delete(void *ptr) noexcept
// {
//     auto ind = std::distance(myAlloc.begin(), std::find(myAlloc.begin(), myAlloc.end(), ptr));
//     myAlloc[ind] = nullptr;
//     std::free(ptr);
// }

// void getInfo()
// {

//     std::cout << std::endl;

//     std::cout << "Not deallocated: " << std::endl;
//     for (auto i : myAlloc)
```

```
// {
//   if (i != nullptr)
//     std::cout << " " << i << std::endl;
// }

// std::cout << std::endl;
//}

// #endif // MY_NEW2
//=====
/*
 */

//=====
// myNew3.hpp

// #ifndef MY_NEW3
// #define MY_NEW3

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <string>
// #include <array>

// int const MY_SIZE= 10;

// std::array<void* ,MY_SIZE> myAlloc{nullptr,};

// void* operator new(std::size_t sz){
//   static int counter{};
//   void* ptr= std::malloc(sz);
//   myAlloc.at(counter++)= ptr;
//   std::cerr << "Addr.: " << ptr << " size: " << sz << std::endl;
//   return ptr;
// }

// void operator delete(void* ptr) noexcept{
//   auto ind= std::distance(myAlloc.begin(),std::find(myAlloc.begin(),myAlloc.end(),ptr));
//   myAlloc[ind]= nullptr;
//   std::free(ptr);
// }

// void getInfo(){

//   std::cout << std::endl;
//   std::cout << "Not deallocated: " << std::endl;
//   for (auto i: myAlloc){
//     if (i != nullptr ) std::cout << " " << i << std::endl;
//   }
// }
```

```
// }

// std::cout << std::endl;

// }

// #endif // MY_NEW3
// =====
/*
```

The key question is as follow:, is there a corresponding delete to each new call?

Operator new#

C++ offers the operator new in four variations:

```
void* operator new (std::size_t count );
void* operator new[](std::size_t count );
void* operator new (std::size_t count, const std::nothrow_t& tag);
void* operator new[](std::size_t count, const std::nothrow_t& tag);
```

The first two variations will throw a std::bad_alloc exception if they can not provide the memory. The last two variations return a null pointer. It's convenient and sufficient to overload only version 1 since the versions 2 - 4 use version 1:

```
void* operator new(std::size_t count)
```

This statement also holds for the variants 2 and 4, which are designed for C arrays. You can read the details of the global operator new here.

The statements also hold for operator delete.

Operator delete#

C++ offers six variations for operator delete :

```
void operator delete (void* ptr);
void operator delete[](void* ptr);
void operator delete (void* ptr, const std::nothrow_t& tag);
void operator delete[](void* ptr, const std::nothrow_t& tag);
void operator delete (void* ptr, std::size_t sz);
void operator delete[](void* ptr, std::size_t sz);
```

According to the properties of operator new, it is sufficient to overload operator delete for the first variant since the remaining 5 use void operator delete(void* ptr) as a fallback.

Only a word about the two last versions of operator delete. In this version, you have the length of the memory block in the variable sz at your disposal. Read the details here.

Counting Allocations and Deallocation#

Let's use the header myNew.hpp (line 3). The same holds for the lines 34. Here we invoke the function getInfo to get information about our memory management.

```
/*
// =====
// overloadOperatorNewAndDelete.cpp

// #include "myNew.hpp"
```

```
/// #include "myNew2.hpp"
/// #include "myNew3.hpp"

// #include <iostream>
// #include <string>

// class MyClass
//{
// float *p = new float[100];
//}

// class MyClass2
//{
// int five = 5;
// std::string s = "hello";
//}

// int main()
//{
// int *myInt = new int(1998);
// double *myDouble = new double(3.14);
// double *myDoubleArray = new double[2]{1.1, 1.2};

// MyClass *myClass = new MyClass;
// MyClass2 *myClass2 = new MyClass2;

// delete myDouble;
// delete[] myDoubleArray;
// delete myClass;
// delete myClass2;

// getInfo();
//}

//=====
// #include "myNew.hpp"

// #ifndef MY_NEW
// #define MY_NEW

// #include <cstdlib>
// #include <iostream>
// #include <new>

// static std::size_t alloc{0};
// static std::size_t dealloc{0};

// void *operator new(std::size_t sz)
//{
// alloc += 1;
// return std::malloc(sz);
}
```

```
// }

// void operator delete(void *ptr) noexcept
//{
//    dealloc += 1;
//    std::free(ptr);
//}

// void getInfo()
//{
//    std::cout << std::endl;

//    std::cout << "Number of allocations: " << alloc << std::endl;
//    std::cout << "Number of deallocations: " << dealloc << std::endl;

//    std::cout << std::endl;
//}

// #endif // MY_NEW
// // =====
// // myNew2.hpp

// #ifndef MY_NEW2
// #define MY_NEW2

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <string>
// #include <array>

// int const MY_SIZE = 10;

// std::array<void *, MY_SIZE> myAlloc{
//     nullptr,
// };

// void *operator new(std::size_t sz)
//{
//     static int counter{};
//     void *ptr = std::malloc(sz);
//     myAlloc.at(counter++) = ptr;
//     return ptr;
// }

// void operator delete(void *ptr) noexcept
//{
//     auto ind = std::distance(myAlloc.begin(), std::find(myAlloc.begin(), myAlloc.end(), ptr));
//     myAlloc[ind] = nullptr;
```

```
// std::free(ptr);
//}

// void getInfo()
//{
// std::cout << std::endl;

// std::cout << "Not deallocated: " << std::endl;
// for (auto i : myAlloc)
// {
// if (i != nullptr)
// std::cout << " " << i << std::endl;
// }

// std::cout << std::endl;
//}

// #endif // MY_NEW2
//=====
/// myNew3.hpp

// #ifndef MY_NEW3
// #define MY_NEW3

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <string>
// #include <array>

// int const MY_SIZE= 10;

// std::array<void* ,MY_SIZE> myAlloc{nullptr,};

// void* operator new(std::size_t sz){
// static int counter{};
// void* ptr= std::malloc(sz);
// myAlloc.at(counter++)= ptr;
// std::cerr << "Addr.: " << ptr << " size: " << sz << std::endl;
// return ptr;
//}

// void operator delete(void* ptr) noexcept{
// auto ind= std::distance(myAlloc.begin(),std::find(myAlloc.begin(),myAlloc.end(),ptr));
// myAlloc[ind]= nullptr;
// std::free(ptr);
//}

// void getInfo(){
```

```

// std::cout << std::endl;

// std::cout << "Not deallocated: " << std::endl;
// for (auto i: myAlloc){
//     if (i != nullptr) std::cout << " " << i << std::endl;
// }

// std::cout << std::endl;

//}

// #endif // MY_NEW3
// =====
// =====
/*

```

In the header file myNew.hpp, we created two static variables alloc and dealloc (line 10 and 11). They keep track of how often we have used the overloaded operator new (line 13) and operator delete (line 18). In the functions, we delegate the memory allocation to std::malloc and the memory deallocation to std::free. The function getInfo (lines 23 - 31) provides us the numbers and displays them.

The question is as follows: have we cleaned everything properly?

Of course, not! That was the intention of this and the next lesson. Now, we know that we have leaks. Maybe it will be helpful to determine the addresses of the objects which we have forgotten to clean up.

Addresses of the Memory Leaks#

So, we have to put more cleverness into the header myNew2.hpp.

```

*/
// =====
// overloadOperatorNewAndDelete.cpp

// #include "myNew.hpp"
// #include "myNew2.hpp"
// // #include "myNew3.hpp"

// #include <iostream>
// #include <string>

// class MyClass{
//     float* p= new float[100];
// };

// class MyClass2{
//     int five= 5;
//     std::string s= "hello";
// };

// int main(){


```

```

// int* myInt= new int(1998);
// double* myDouble= new double(3.14);
// double* myDoubleArray= new double[2]{1.1,1.2};

// MyClass* myClass= new MyClass;
// MyClass2* myClass2= new MyClass2;

// delete myDouble;
// delete [] myDoubleArray;
// delete myClass;
// delete myClass2;

// getInfo();

// }
// =====
/*

```

Therefore, we must be clever about the header myNew2.hpp.

The key idea is to use the static array myAlloc (line 15) to track the addresses of all std::malloc (line 19) and std::free (line 27) invocations. On the function operator new, we cannot use a container that needs dynamic memory. This container would invoke the operator new — a recursion that would cause the program to crash. Therefore, we used an std::array in line 15, since std::array gets its memory at compile time. std::array can become too small in this process. Therefore, we invoke myAlloc.at(counter++) in order to check the array boundaries.

Which memory address we have forgotten to release? The output gives the answer.

A simple search for the object having the address is the best approach since it is probable that a new call of std::malloc reuses an already-used address. This suffices, so long as the objects have been deleted in the meantime. But why are the addresses parts of the solution? We must only compare the memory address of the created objects with the memory address of the objects that are not yet deleted.

Comparison of the Memory Addresses#

In addition to the memory address, we also have the size of the reserved memory at our disposal. We will use this information in operator new.

```

*/
// =====
// overloadOperatorNewAndDelete.cpp

// #include "myNew.hpp"
// #include "myNew2.hpp"
// #include "myNew3.hpp"

// #include <iostream>
// #include <string>

// class MyClass
// {
//   float *p = new float[100];
// };

```

```

// class MyClass2
//{
// int five = 5;
// std::string s = "hello";
//};

// int main()
//{
// int *myInt = new int(1998);
// double *myDouble = new double(3.14);
// double *myDoubleArray = new double[2]{1.1, 1.2};

// MyClass *myClass = new MyClass;
// MyClass2 *myClass2 = new MyClass2;

// delete myDouble;
// delete[] myDoubleArray;
// delete myClass;
// delete myClass2;

// getInfo();
//}

//=====
//=====

/*

```

The allocation and deallocation of the application are clearly more transparent.

A simple comparison shows that we forgot to release an object with 4 bytes and an object with 400 bytes. In addition, the sequence of allocation in the source code corresponds to the sequence of outputs in the program. Thus, it should be easy to identify the missing memory release

The program has two main issues. Firstly, we statically allocate the memory for `std::array`. Secondly, we want to know which object was not released. In the next lesson, we will solve both issues.

```

*/
/*
Memory Management: Overloading Operator new and delete 2
In this lesson, we will refine the strategy for overloading operators new and delete.
```

We'll cover the following

Who is the Bad Guy?

First Try

Theory

Understanding the Output

All at Run Time

What were the not-so-nice properties of the previous lesson?

Firstly, we only get a hint of which memory was lost. Secondly, we had to prepare the whole bookkeeping of memory management at compile time. In this lesson, we aim to overcome these shortcomings.

Who is the Bad Guy?#

Special tasks call for special strategies. We must use a small macro for debugging purposes.

Let's take a look at this macro. #define new new(__FILE__, __LINE__)

The macro causes each new call to be mapped onto the overloaded new call. This overloaded new call also receives the name of the file and the line number respectively. That is exactly the information we need to solve this problem.

So, what will happen if we use the macro in line 6?

```
/*
// =====
// overloadOperatorNewAndDelete2.cpp

//#include "myNew4.hpp"
//#include "myNew5.hpp"

// #define new new(__FILE__, __LINE__)

// #include <iostream>
// #include <new>
// #include <string>

// class MyClass{
//   float* p= new float[100];
// };

// class MyClass2{
//   int five= 5;
//   std::string s= "hello";
// };

// int main(){

//   int* myInt= new int(1998);
//   double* myDouble= new double(3.14);
//   double* myDoubleArray= new double[2]{1.1,1.2};

//   MyClass* myClass= new MyClass;
//   MyClass2* myClass2= new MyClass2;

//   delete myDouble;
//   delete [] myDoubleArray;
//   delete myClass;
//   delete myClass2;

//   dummyFunction();

//   //getInfo();
```

```
// }
// =====
// myNew4.hpp

// #ifndef MY_NEW4
// #define MY_NEW4

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <array>

// int const MY_SIZE= 10;

// int counter= 0;

// std::array<void* ,MY_SIZE> myAlloc{nullptr,};

// void* newImpl(std::size_t sz,char const* file, int line){
//   void* ptr= std::malloc(sz);
//   std::cerr << file << ":" << line << " " << ptr << std::endl;
//   myAlloc.at(counter++)= ptr;
//   return ptr;
// }

// void* operator new(std::size_t sz,char const* file, int line){
//   return newImpl(sz,file,line);
// }

// void* operator new [] (std::size_t sz,char const* file, int line){
//   return newImpl(sz,file,line);
// }

// void operator delete(void* ptr) noexcept{
//   auto ind= std::distance(myAlloc.begin(),std::find(myAlloc.begin(),myAlloc.end(),ptr));
//   myAlloc[ind]= nullptr;
//   std::free(ptr);
// }

// #define new new(__FILE__, __LINE__)

// void dummyFunction(){
//   int* dummy= new int;
// }

// void getInfo(){

//   std::cout << std::endl;
//   std::cout << "Allocation: " << std::endl;
```

```

// for (auto i: myAlloc){
//   if (i != nullptr) std::cout << " " << i << std::endl;
// }

// std::cout << std::endl;

//}

// #endif // MY_NEW4
//=====
/*
The preprocessor substitutes all new calls, showing exactly the modified main function.
*/
//=====
// class MyClass{
//   float* p= new("main.cpp", 14) float[100];
// };

// class MyClass2{
//   int five= 5;
//   std::string s= "hello";
// };

// int main(){

//   int* myInt= new("main.cpp", 24) int(1998);
//   double* myDouble= new("main.cpp", 25) double(3.14);
//   double* myDoubleArray= new("main.cpp", 26) double[2]{1.1,1.2};

//   MyClass* myClass= new("main.cpp", 28) MyClass;
//   MyClass2* myClass2= new("main.cpp", 29) MyClass2;

//   delete myDouble;
//   delete [] myDoubleArray;
//   delete myClass;
//   delete myClass2;

//   dummyFunction();

//   getInfo();

//}

//=====
/*
Lines 2 and 12 demonstrate how the preprocessor substitutes the constants __FILE__ and __LINE__ in the
macro. So, how does this technique work? The header myNew4.hpp solves the problem.
*/
//=====
// myNew4.hpp

// #ifndef MY_NEW4

```

```
// #define MY_NEW4

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <array>

// int const MY_SIZE = 10;

// int counter = 0;

// std::array<void *, MY_SIZE> myAlloc{
//   nullptr,
// };

// void *newImpl(std::size_t sz, char const *file, int line)
// {
//   void *ptr = std::malloc(sz);
//   std::cerr << file << ":" << line << " " << ptr << std::endl;
//   myAlloc.at(counter++) = ptr;
//   return ptr;
// }

// void *operator new(std::size_t sz, char const *file, int line)
// {
//   return newImpl(sz, file, line);
// }

// void *operator new[](std::size_t sz, char const *file, int line)
// {
//   return newImpl(sz, file, line);
// }

// void operator delete(void *ptr) noexcept
// {
//   auto ind = std::distance(myAlloc.begin(), std::find(myAlloc.begin(), myAlloc.end(), ptr));
//   myAlloc[ind] = nullptr;
//   std::free(ptr);
// }

// #define new new (__FILE__, __LINE__)

// void dummyFunction()
// {
//   int *dummy = new int;
// }

// void getInfo()
// {
```

```

// std::cout << std::endl;

// std::cout << "Allocation: " << std::endl;
// for (auto i : myAlloc)
// {
//   if (i != nullptr)
//     std::cout << " " << i << std::endl;
// }

// std::cout << std::endl;
// }

// #endif // MY_NEW4
// =====
/*
Theory#

```

In lines 25 and 29, we implement special operators new and new[] to delegate their functionality to the helper function newImpl (line 18 - 23). The function completes two important jobs:

It displays the name of the source file and the line number (line 20) to each new call of the function.

In the static array myAlloc, it keeps track of each used memory address (line 21).

This fits the behavior of the overloaded operator delete which sets all memory addresses to the null pointer nullptr (line 35). The memory addresses stand for the deallocated memory areas. In the end, the function getInfo displays the memory addresses that were not deallocated. We can directly see them together with the file name and line number.

Of course, we can also directly apply the macro in the file myNew4.hpp. Now that we've gone over the theory, what is the output of the program?

Understanding the Output#

The memory areas to three memory addresses were not deallocated. The problems, therefore, are

new calls in line 23 and line 13 in main.cpp and

the new call in line 42 in myNew4.hpp

Impressive, isn't it? It's important to note that the presented technique has two significant drawbacks.

We must overload the simple operator new and the operator new [] for arrays, due to the fact that the overloaded operator new is not a fallback for the three remaining operators new.

We cannot use the special operator new that returns a null pointer in the error case since it will be explicitly called by the operator new with the argument std::nothrow: int* myInt= new (std::nothrow) int(1998);

Now, we must solve the first issue. We must use a data structure for the array myAlloc that manages its memory at run time. Therefore, it is no longer necessary to eagerly allocate the memory at compile time.

All at Run Time#

Why did we not allocate memory in the operator new? The operator new was globally overloaded. Therefore, a call of new would end in never-ending recursion. That will occur if we use a container such as std::vector which dynamically allocates its memory.

This restriction no longer holds since we did not overload the global operator new, which is a fallback for the three remaining new operators. Due to the macro, our own variant of the operator new is now used. Therefore, we can use std::vector in our operator new.

You can see this operation in the program below while using the header myNew5.hpp

```
/*
// =====
// myNew5.hpp

// #ifndef MY_NEW5
// #define MY_NEW5

// #include <algorithm>
// #include <cstdlib>
// #include <iostream>
// #include <new>
// #include <string>
// #include <vector>

// std::vector<void*> myAlloc;

// void* newImpl(std::size_t sz,char const* file, int line){
//   static int counter{};
//   void* ptr= std::malloc(sz);
//   std::cerr << file << ":" << line << " " << ptr << std::endl;
//   myAlloc.push_back(ptr);
//   return ptr;
// }

// void* operator new(std::size_t sz,char const* file, int line){
//   return newImpl(sz,file,line);
// }

// void* operator new [] (std::size_t sz,char const* file, int line){
//   return newImpl(sz,file,line);
// }

// void operator delete(void* ptr) noexcept{
//   auto ind= std::distance(myAlloc.begin(),std::find(myAlloc.begin(),myAlloc.end(),ptr));
//   myAlloc[ind]= nullptr;
//   std::free(ptr);
// }

// #define new new(__FILE__, __LINE__)

// void dummyFunction(){
```

```
// int* dummy= new int;
// }

// void getInfo(){

// std::cout << std::endl;

// std::cout << "Allocation: " << std::endl;
// for (auto i: myAlloc){
//     if (i != nullptr) std::cout << " " << i << std::endl;
// }

// std::cout << std::endl;

// }

// #endif // MY_NEW5
// =====
// overloadOperatorNewAndDelete2.cpp

#ifndef include "myNew4.hpp"
#ifndef include "myNew5.hpp"

// #define new new(__FILE__, __LINE__)

// #include <iostream>
// #include <new>
// #include <string>

// class MyClass{
// float* p= new float[100];
// };

// class MyClass2{
// int five= 5;
// std::string s= "hello";
// };

// int main(){

// int* myInt= new int(1998);
// double* myDouble= new double(3.14);
// double* myDoubleArray= new double[2]{1.1,1.2};

// MyClass* myClass= new MyClass;
// MyClass2* myClass2= new MyClass2;

// delete myDouble;
// delete [] myDoubleArray;
// delete myClass;
// delete myClass2;
```

```
// dummyFunction();

// getInfo();

// }
/*

```

- Example

The example in this lesson shows the deterministic behavior of RAII in C++.

We'll cover the following

Example - RAII

Explanation

Example - RAII#

RAII stands for Resource Acquisition Is Initialization, and it is one of the most important idioms in C++. It states that a resource should be acquired in the constructor of the object and released in the destructor of the object. It is important to remember that the destructor will automatically be called if the object goes out of scope.

Is this not deterministic? In Java or Python (`__del__`), you have a destructor but not the guarantee. Therefore, if you use the destructor to release a critical resource, such as a lock, it can end disastrously. In C++, however, this problem is prevented.

Look at the example below:

```
/*
// =====
// raii.cpp

// #include <iostream>
// #include <new>
// #include <string>

// class ResourceGuard
//{
// private:
// const std::string resource;

// public:
// ResourceGuard(const std::string &res) : resource(res)
// {
// std::cout << "Acquire the " << resource << "." << std::endl;
// }
// ~ResourceGuard()
// {
// std::cout << "Release the " << resource << "." << std::endl;
// }
//};

// int main()
//{
```

```

// std::cout << std::endl;

// ResourceGuard resGuard1{"memoryBlock1"};

// std::cout << "\nBefore local scope" << std::endl;
// {
//   ResourceGuard resGuard2{"memoryBlock2"};
// }
// std::cout << "After local scope" << std::endl;

// std::cout << std::endl;

// std::cout << "\nBefore try-catch block" << std::endl;
// try
// {
//   ResourceGuard resGuard3{"memoryBlock3"};
//   throw std::bad_alloc();
// }
// catch (std::bad_alloc &e)
// {
//   std::cout << e.what();
// }
// std::cout << "\nAfter try-catch block" << std::endl;

// std::cout << std::endl;
// */
/*

```

Explanation#

ResourceGuard is a guard that manages its resource. In this case, the resource is a simple string. ResourceGuard reliably creates its resource and releases the resource in its destructor (line 14 - 16).

The destructor of resGuard1 (line 23) will be called exactly at the end of the main function (line 46).

The lifetime of resGuard2 (line 27) ends in line 28. Therefore, the destructor will automatically be executed. Even the presence of an exception does not alter the reliability of resGuard3 (line 36).

Its destructor will be called at the end of the try block (line 35 - 38).

```

*/
/*

```

Congratulations! You have successfully completed the "Embedded Programming" module.

Summary#

This module started with an overview of embedded programming and a few myths and facts. Then, we learned how embedded programming deals with safety-critical systems. In the end, we learned how we can use embedded programming for better utilization of resources.

Takeaways#

We started the journey by learning the basics of C++, such as data types, loops, and pointers.

We explored classes and object-oriented programming concepts like information hiding, inheritance, and polymorphism.

We learned how classes and templates can help us create simple, efficient, reusable, and secure code.

We learned the use of the latest features that have been introduced in the C++ Standard Library.

We saw how concurrency and multithreading can be used to perform complex tasks easily and time efficiently.

We learned the basic concepts of graph theory and how to represent graphs as data structures in code.

We explored the power of embedded programming and learned how it can be used in safety-critical systems, etc.

```
*/  
// ======  
// ======  
// ======  
// ======  
// ======  
// ======
```