C++

HS2021

Marco Agostini

Computer Science
University of Applied Sciences of Eastern Switzerland
September 2021

Contents

1 Introduction

1.1 Why C++?

- Work al almost all platforms from a micro controller to the main frame
- Multi-paradigm language with zero-cost abstraction
- High-level abstraction facilities
- The concepts from C++ can mostly be applied to any other programming language

1.2 Features

- C++ doesnt't has no methods only functions. A function does not have to be a member of an object. If a function belongs to an object it's a member function.
- Please do not write your own loops in C++ try to use the STL (Standard Template Library).
- C++ is compatible with standard C.
- There is no Garbage Collector!
- With a library we can publish functionalities to another program.

1.3 Terminology

Value 42

Type int, char, bool, long, float

Variable int const i{42}

Expression (2+4)*3

Statement while (true);

Declaration int foo();

Definition int j;

Function void bar ( ) { }

1.4 Undefined Behaviour

The undefined behaviour is defined in the C++ standard (funny, isn't it?). Because of the fact, that C++ doesn't have a garbage collection, if in C++ something is written wrong and the compiler doesn't detect it: undefined behaviour can occur.

1.5 C++ Compilation Process

C++ has the advantage of direct compilation into machine code. This eliminates the overhead for a virtual machine in comparison to Java.

***.cpp files for source code**

- Also called "Implementation File"
- Function implementations (can be in .h files as well)
- Source of compilation - aka "Translation Unit"

***.h files for interfaces and templates**

- Called "Header File"

- Declarations and definitions to be used in other implementation files.
- Textual inclusion through a pre-processor (C++20 will incorporate a "Module" mechanism)
- [language=C++]include "header.h"

3 Phases of Compilation

- **Preprocessor** Textual replacement of preprocessor directives, results in (*.i) files. ([language=C++]include)
- **Compiler** Translation of C++ code into machine code (source file (*.i) to object file (*.o))
- **Linker** - Combination of object files (*.o) and libraries into libraries and executables (*.exe).

1.6 Declarations and Definitions

All things with a name that you use in a C++ program must be declared before you can do so!

Defining Functions

$$< return - type >< function - name > (< parameters >)\{/ * body * /\}$$

Tells the compiler that there is a function named $< function - name >$ that takes the parameters $< parameters >$ and returns a value of type $< return - type >$. The Signature of a function is just the combination of name and the parameter types.

One Definition Rule

While a program element can be declared several times without problem there can be only one definition of it. **(ODR = One Definition Rule)**

Include Guard

Include guards ensure that a header file is only included once. Multiple inclusions could violate the One Definition Rule when the header contains definitions. [language=c++] ifndef SAYHELLO$_{HdefineSAYHELLO_{Hinclude\lt iosfwd\gt structGreeter;endif/*SAYHELLO_{H*/}}}$

2 Variables

- Variables always start with a lower case character
- Local variables must always contain a default value (Curly brackets or =).
- A global variable must never be mutable! (Hard to test and can cause problems when multithreading is used)
- Variables are as default value types and therefore declared on the stack.

2.1 Definitions

Defining a variable consists of specifying its ¡type¿, its ¡variable-name¿ and its ¡initial value¿. Empty braces mean default initialisation. Using = for initialisation we can have the compiler determine its type (do not combine with braces!).

$$< type >< variable - name > < initial - value >;$$

Constants
Adding the const keyword in front of the name makes the variable a single-assignment variable, aka a constant. A const must be initialised and is immutable.

When should const be used?

- A lot of code needs names for values, but often does not intend to change it
- It helps to avoid reusing the same variable for different purposes (code smell)
- It creates safer code, because a const variable cannot be inadvertently changed
- It makes reasoning about code easier
- Constness is checked by the compiler
- It improves optimization and parallelization (shared mutable state is dangerous)

Where to place Variable definition?
Do not practice to define all (potentially) needed variables up front (that style is long obsolete!). Every mutable global variable you define is a design error!

A Note on Naming
The C++ convention is to begin variable names with a lower case letter. Spell out what the variable is for and do not abbreviate!

Types for Variables
Are part of the language and don't need an include.

- short, int, long, long long each also available as unsigned version
- bool, char, unsigned char, signed char - are treated as integral numbers as well
- float, double, long double

2.2 Values and Expressions

C++ provides automatic type conversion if values of different types are combined into an expression. Dividing integers by zero is undefined behaviour.

[language=C++] (5 + 10 * 3 - 6 / 2) // precedence as in normal mathematics = 32 auto x = 3; / 3 // Fractions results of int operations always rundet down! 1 auto y = x [width=0.75]images/literalexamples

Figure 1: C++ Variable Types

2.3 Const

- Const should be used as often as possible, because it optimises the code.
- Const is comparable with the final from Java, although it has a higher guarantee that the variable is not changed.
- Const variables must be initiaized!
- To set const vars at compile time the keyword "constexpr" must be used.

2.4 Auto

The keyword auto can be used to deduct the type of a variable automatically at the declaration. [language=C++] auto const yearOfBirth = 2049; // int auto const name = "Rick Deckard" // std::string

2.5 Strings

std::string is C++'s type for representing sequences of char (which is often only 8 bit). This Strings are mutable in C++ in contrast to Java. Literals like "ab" are not of type std::string they consist of const chars in a null terminated array.

To have a std::string we need to append an s. This requires using namespace std::literals;.

[language=C++] void printName(std::string name) using namespace std::literals; std::cout ¡¡ "my name is: "s ¡¡ name;

String Capabilites

You can iterate over the contents of a string. [language=C++] void toUpper(std::string value) for (char c : value) c = toupper(c);

3 Streams

- In the header files the inclusion of [language=C++]include ¡iosfwd¿ forward declaration header. This is sufficient for function declarations.
- In a source file for "std::cin" and "std::cout" the [language=C++]include ¡iostream¿ should be used. This containts all the definitions needed for "std::cin" etc. If just one of the two stream objects is need use either [language=C++]include ¡ostream¿ or [language=C++]include ¡istream¿. The last two dont include the "std::cin" and "std::cout".
- In the main function "std::cin" and "std::cout" is used with the corresponding shift operators "¡¡", "¿¿".
- "std::istream" objects do return false if we are in an invalid stream state.
- "std::endl" flushes a buffered out stream. Better use "\n".

3.1 Input and Output Streams

Functions taking a stream object must take it as a reference, because they provide a side-effect to the stream (i.e., output characters).

Simple I/O
Stream objects provide C++'s I/O mechanism with the help of the pre-defined globals: std::cin std::cout. Streams have a state that denotes if I/O was successful or not.

- Only .good() streams actually do I/O
- You need to .clear() the state in case of an error
- Reading a std::string can not go wrong, unless the stream is already [language=C++]!good().

Reading a std::strting Value [language=C++] include ¡iostream¿ include ¡string¿ std::string inputName(std::istream in) std::string name; in ¿¿ name; return name; **Reading an int Value** [language=C++] int inputAge(std::istream in) int age-1; if (in ¿¿ age) // Boolean conversion return age; return -1; **Chaining Input Operations**

- Multiple subsequent reads are possible
- If a previous read already failed, subsequent reads fail as well

[language=C++] std::string readSymbols(std::istream in) char symbol; int count-1; if (in ¿¿ symbol ¿¿ count) return std::string(count, symbol); return "error";

3.2 Stream States

Formatted input on stream is must check for is.fail() and is.bad(). If failed, is.clear() the stream and consume invalid input characters before continue.

[width=0.7]images/streamstates

Figure 2: Stream States in C++

3.3 Manipulators

For the formatting of the output a vide variety of manipulator can be used.

4 Iterators

There are always two iterators used (begin() und end()). There is also the possibility to traverse a list from front to back (rbegin() and rend()). If the members are only read the const version (cbegin() and cend()) can be used.

4.1 Iteration

Its possible to index a vector like an array but there is no bounds check. Accessing an element outside the valid range is Undefined Behavior.

Bad Style Iteration!

[language=C++] for (size$_t i = 0; i < v.size(); ++i)//Indexis"unsigned"0 - 1 = MAX_INTstd :: cout << "v[" <$

Element Iteration (Range-Based for)

- Advantage: No index error possible
- Works with all containers, even value lists 1, 2, 3

[width=0.75]images/elementiteration

Iteration with Iterators [language=C++] for (auto it = std::begin(v); it != std::end(v); ++it) std::cout ¡¡ (*it)++ ¡¡ ", "; // Guarantee to just have read-only access with std::cbegin() and std::cend() for (auto it = std::cbegin(v); it != std::cend(v); ++it) std::cout ¡¡ *it ¡¡ ", ";

4.2 Using Iterators with Algorithms

Each algorithm takes iterator arguments. The algorithm does what its name tells us.

[language=C++] // Counting blanks in a string size$_t count_blanks(std :: strings)size_t count0; for(size_t = 0; i <$

// Counting blanks in a string with algorithms size$_t count_blanks(std :: strings)returnstd :: count(s.begin(), s.e$

// Summing up all values in a vector std::vector¡int¿ v5, 4, 3, 2, 1; std::cout ¡¡ std::accumulate(std::begin(v), std::end(v), 0)¡¡ " = sum";

// Number of elements in range void printDistanceAndLength(std::string s) std::cout ¡¡ "distance: "¡¡ std::distance(s.begin(), s.end()) ¡¡"; std::cout ¡¡ "in a string of length: "¡¡ s.size()¡¡";

// Printing all values of a vector void printAll(std::vector¡int¿ v) std::for$_each(std :: crbegin(v), std :: crend(v), print);$

// For each with a Lambda void printAll(std::vector¡int¿ v, std::ostream out) std::for$_each(std :: crbegin(v), std :: crend(v), [out](autox)out << "print : " << x <<' ';);$

4.3 Iterators for I/O

Iterators connect streams and algorithms. Streams (std::istream and std::ostream) cannot be used with algorithms directly.

- std::ostream$_iterator < T > outputsvaluesoftypeTtothegivenstd :: ostream$
    - No end() marker needed for output, it ends when the input range ends.

—std::istream$_iterator < T > |readsvaluesoftypeTfromthegivenstd :: istream$

End iterator is the default constructed —std::istream$_iterator < T > |Itendswhenthestreamisnolongergoo$

4.4 Types

There are five different types of iterators in C++. [language=C++] struct input$_iterator_tag; structoutput_iterator_tag; structforward_iterator_tag : publicinputer_iterator_tag; structbidi$ $publicforward_iterator_tag; structrandom_access_iterator_tag : publicbidirectional_iterator_tag;$

4.4.1 Input Iterator

- The element can be read only once and after that the iterator has to be incremeneted.

- Used for [language=C++]—std::istream$_iterator|and[language = C++]|std :: istreambuf_iterator|$

4.4.2 Forward Iterator

- Element can be read in and changed (Except element or container is const).
- Only allows forward iteration
- Sequenz can be iterated over multiple times

4.4.3 Bidirectional Iterator

- Element can be read in and changed (Except element or container is const).
- Allows forward and backwards iteration
- Sequenz can be iterated over multiple times
- The random access iterator behaves as the bidirectional iterator with the addition that the can access elements over the index

4.4.4 Output Iterator

- Current element can be changed once, after that the iterator has to be incremented.
- There is no end for this iterator (example console prints)

- Used for —std::ostream$_iterator|Writestheresultwithoutknowingtheresult.$

5 Functions

- Functions are always written in lower Camel Case
- A function must be declared always in a header file before the function is used
- A good function has a maximum of five parameters and does exactly one thing
- The call of the function parameters is not defined.
- The main function does implicit return a "0".
- Auto should not be used as a return type, exceptions are: inline, template or constexpr functions in header files.
- Void should not be used as a function parameter
- NEVER return a ref to a local variable since it produces a dangling Reference, because the value lives in the stack frame.

5.1 Default Arguments

A function declaration can provide default arguments for its parameters *from the right.* [language=C++] void incr(int var, unsigned delta = 1); // Default arguments can be omitted calling int counter 0; incr(counter); // uses default for delta

5.2 Function Overloading

The same function name can be used for different functions if parameter number or types differ. Function can not be overloaded just by their return type! If only the parameter type is different there might be ambiguities. The resolution fo overloads happens at compile-time = Ad hoc polymorphism. [language=C++] void incr(int var); int incr(int var); // doesn't compile because of same signature void incr(int var, unsigned delta);

5.3 Reference / Value Arguments

Parameter Declarations

[width=0.75]images/functionparameters

- Value Parameter - Default void f(type par);
- Reference Parameter - side-effect void f(type & par);
- Const-Reference Parameter - optimisation void f(type const & par);
- Const Value Parameter - Prevent changing the para void f(type const par);

Function Return Type

- By (Const) Value - default type f(); or type const f();

- By Reference - Only return a reference parameter (or a call member variable from a member function) type & f(); or type const & f();

Functions as Parameters
Functions are "first class" objects in C++. You can pass them as augment and you can keep them in reference variables.

5.4 Variadic Arguments

Variadic functions take a variable number of arguments. This example is even a template function with variadic arguments. [language=C++] template¡typename First, typename...Types¿ void printAll(First const first, Types const ...rest) std::cout ¡¡ first; if (sizeof...(Types)) std::cout ¡¡ ", "; printAll(rest...);

5.5 Lambdas

- Can be written into variables [language=C++]—auto l = [](); l();—
- The smallest lambda is [language=C++]—[]()— the first two brackets are the function object and the round brackets the call.

Defining Inline functions. Auto const for function variable for Lambda. [] introduces a Lambda function. Can contain captures: [=] or [&] to access variables from scope. [language=C++] auto const g = [](char c) -¿ return std::toupper(c)M ; g('a');

5.5.1 Captures

Captured variables are imutable default. To change them they have to be declared as —mutable—.

- —[=]— - default implicit capture variables used in body by value
- —[]— - default capture variable used in body by reference
- —[var = value]— - introduce new capture variable with value
- —[=, out]— - capture all by copy, out by reference
- —[, = x]— - capture all by reference, but x by copy/value

[language=C++] // Capturing by value int x = 5; auto l = [x]() mutable std::cout ¡¡ ++x; ; // Capuring by reference auto const l = [x]() std::cout ¡¡ ++x; ;

5.6 Functor

Functors are types which provide an operation. Functors have an overloaded call operator. Lambdas internally work with functors. The —operator()— function can theoretically be overload as often as needed. [language=C++] struct Accumulator int count0; int accumulated$_v alue0; voidoperator()(intvalue)count++; accumulated_v alue+=val$

int average(std::vector¡int¿ values) Accumulator acc; for(auto v : values) acc(v); return acc.average(); int main(int argc, char **argv) std::vector¡int¿ values 1, 2, 6, 4, 5, 3 ; std::cout ¡¡ average(values);

6 Exceptions

An exception can throw any copyable type. No means to specify what could be thrown. No check if you catch an exception that might be thrown at call-site. No meta-information is available as part of the exception. Exception thrown while exception is propagated results in a program abort (not while caught).

6.1 Failing Functions

What should we do, if a function cannot fulfil its purpose?

1. Ignore the error and provide potentially undefined behaviour
2. Return a standard result to cover the error
3. Return an error code or error value
4. Provide an error status as a side-effect
5. Throw an Exception

Ignore the Error

- Relies on the caller to satisfy all preconditions.
- Viable only if not dependent on other resources.
- Most efficient implementation.
- Simple for the implementer but hard for the caller.

Cover the Error with a Standard Result

- Reliefs the caller from the need to care if it can continue with the default value
- Can hide underlying problems.
- Often better if caller can specify its own default value.

Error Value

- Only feasible if result domains is smaller than return type
- POSIX defines -1 to mark failure of system calls
- Burden on the caller to check the result

Cover the Error with a Standard Result

- Requires reference parameter
- (Bad!) Alternative: global variable (POSIX: errno)
- E.g: std::istreamss states (good(), fail()) is chan- ged as a side-effect of input

6.2 Catching Exceptions

Principle: Throw by value, catch by const reference. This avoids unnecessary copying and allows dynamic polymorphism for class types. [language=C++] try throwingCall(); catch (type const e) //Handle type exception catch (type2 const e) //Handle type2 exception catch (...) //Handle other exception types The Standard Library has some pre-defined exception types that you can also use in ¡stdexcept¿. All have a constructor parameter for the "reason" of type std::string. It provides the what() member function to obtain the "reason"

[width=0.75]images/exceptions

Keyword noexcept
Functions can be declared to explicitly not throw an exception with the noexcept keyword. The compiler does not need to check it. If an exception is thrown (directly or indirectly) from a noexcept function the program will terminate.

7 Classes and Operators

Are defined in header files and not in *.cpp files! The implementation can then be done in a suitable file.

- Class members are implicitly inline.
- A class does one thing well and is named after that.
- A class consists of member functions with only a few lines.
- Has a class invariant: provides guarantee about its state (values of the member variables).
- Don't make member variables const as it prevents copy assignment. Don't add members to communicate between member function calls.
- Member functions should when possible be const, as long as they don't change the this object

[language=C++] class ¡GoodClassName¿ ¡member variables¿ ¡constructor¿ ¡member function¿ ;

Class type in a header file. [language=C++] ifndef DATE$_{HdefineDATE_{H_c}lassDate}$

int year, month, day;

public: Date() = default; Date(int year, int month, int day) : yearyear, monthmonth, dayday /*...*/ static bool isLeapYear(int year) /*...*/

private: bool isValidDate() const /*...*/ ;

endif /* DATE$_{H*}$/

Implementation of the class. [language=C++] include "Date.h" Date::Date(int year, int month, int day) : yearyear, monthmonth, dayday if (!isValidDate()) throw std::out$_of_range"invaliddate"$;

Date::Date() : Date1980, 1, 1 // Default constructor

Date::Date(Date const other) : Dateother.year, other.month, other.day // copy constructor

bool Date::isLeapYear(int year) /* ... */

7.1 Access Specifier

- private: visible only inside the class (and friends); for hidden data members
- protected: also visible in subclasses
- public: visible from everywhere; for the interface of the class

Static Member Functions and Variables
No *static* in *.cpp file only in *.h file!

7.2 Constructors

Function with name of the class and no return type.

- Default Constructor - No parameters. Implicitly available if there are no other explicit constructors. Has to initialize member variables with default values.
- Copy Constructor - Has one <own-type> const & parameter. Implicitly available (unless there is an explicit move constructor or assignment operator). Copies all member variables.
- Move Constructor - Has one <own-type> && parameter. Implicitly available (unless there is an explicit copy constructor or assignment operator). Moves all members
- Typeconversation Constructor - Has one <other-type> const & parameter. Converts the input type if possible. Declare explicit to avoid unexpected conversions.
- Initializer List Constructor - Has one std::initializer_list parameter. Does not need to be explicit, implicit conversion is usually desired. Initializer List constructors are preferred if a variable is initialized with { }
- Destructor - Named like the default constructor but with a ∼. Must release all resources. Implicitly available. Must not throw an exception. Called automatically at the end of the block for local instances.

[language=C++] class Date public: Date(int year, int month, int day); Date(); // Default-Constructor Date() = default; // explizit Default-Constructor Date(Date const); // Copy-Constructor Date(Date); // Move-Constructor explicit Date(std::string const); // Typeconversation-Constructor Date(std::initializer$_list < Element > elements); //InitializerList{-}Constructor\ Date(); //Destructor Date(Date const) = delete; //deleteimplicitCopy - Constructor$;

7.3 Defaulted Constructor

In order no to state the default constructor explicitly in the cpp file it can be defined in the header file of the class. This is also possible for the move and the copy constructor. [language=C++] ifndef DATE$_{HdefineDATE_HclassDateintyear9999,month12,day31;//...Date()=default}$

7.4 Inheritance

Base classes are specified after the name: $class < name >:< base1 >, ..., < baseN >$. Multiple inheritance is possible and inheritance can specify visibility. If no visibility is specified the default of the inheriting class is used.

[language=C++] class Base private: int onlyInBase; protected: int baseAndInSubclasses; public: int everyoneCanFiddleWithMe ; class Sub : public Base //Can see baseAndInSubclasses and //everyoneCanFiddleWithMe ;

8 Operator Overloading

Custom operators can be overloaded for user-defined types. Declared like a function, with a special name: ¡returntype¿ operator op(¡parameters¿);. Unary operators -¿ one parameters and binary operators -¿ two parameters.

Free Operator
Free operator< uses two parameters of Date each *const* & return type *bool*. Is inline when defined in header. The only problem we have is that we don't have access to private members.

Inline Keyword
Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small (All the functions defined inside the class are implicitly inline).

Friend keyword
A friend function can be given a special grant to access private and protected members.

[language=C++] // File Any.cpp include "Date.h" Any.cpp include ¡iostream¿ void foo() std::cout ¡¡ Date::myBirthday; Date d; std::cin ¿¿ d; std::cout ¡¡ "is d older? " ¡¡ (d ¡ Date::myBirthday);

// File Date.h class Date int year, month, day; // private :-(; inline bool operator¡(Date const lhs, Date const rhs) return lhs.year ¡ rhs.year —— // Does not WOKR! (lhs.year == rhs.year (lhs.month ¡ rhs.month —— (lhs.month == rhs.month lhs.day == rhs.day)));

Member Operator
Member operator< uses one parameter of type *Date*, which is *const*&, return type *bool* and Right-hand side of operation. Implicit this object: *const* due to qualifier, left-hand side of operation. [language=C++] // File Any.cpp include "Date.h" include ¡iostream¿ void foo() std::cout ¡¡ Date::myBirthday; Date d; std::cin ¿¿ d; std::cout ¡¡ "is d older? " ¡¡ (d ¡ Date::myBirthday); // File Date.h class Date int year, month, day; // private :-) bool operator¡(Date const rhs) const return year ¡ rhs.year —— (year == rhs.year (month ¡ rhs.month —— (month == rhs.month day == rhs.day))); ;

9 Enums and Namespaces

9.1 Namespaces

- Namespaces are scopes for grouping and preventing name clashes
- Global namespaces has the :: prefix
- Nesting of namespaces is possible
- Nesting of scopes allows hiding of names
- Namespaces can only be defined outside of classes and functions
- The same same namespace can be opened and closed multiple times
- Qualified names are. used to access names in a namespace: demo::subdemo::foo()
- A name with a leading :: is called fully qualified name: ::std::cout.

Using Declarations

- Import a name from a namespace into the current scope That name can be used without a namespace prefix Useful if the name is used very often
- Alternative: using alias for types if name is long
- There are also using directives, which import ALL names of a namespace into the current scope. Use them only in local scope to avoid "pollution" of your namespace.

Anonymous Namespaces

- Special case: omit name after namespace
- Implicit using directive for the chosen stream
- Hides modules internals
- Use them only in source files (*.cpp)

9.2 Name Resolution of Namespace Members

Types and (non-member) functions belonging to that type should be placed in a common namespace. The Advantage is *Argument Dependent Lookup! ADL:* When the compiler encounter an unqualified function or operator call with an argument of a user-defined type it looks into the namespaces in which that type is defined to resolve the functionoperator. E.g. it is not necessary to write std:: in front of for_each when std::vector::begin() is an argument of the function.

9.3 Enumerations

Enumerations are useful to represent types with only a few values. An enumeration creates a new type that can easily be converted to an integral type. The individual values (enumerators) are specified in the type. Unless specified explicitly, the values start with 0 and increase by 1.

9.4 Arithmetic Types

Disclaimer: You usually do not want to implement your own arithmetic types! We will cover the basics.

- Arithmetic types must be equality comparable
- Boost can be used to get != operator → boost::equality_comparable
- It might be convenient to have the output operator
- Result must be in a specific range (Modulo)

10 Standard Containers

There are three main types of standard containers in the C++ language. Containers can be: default-constructed, copy-constructed from another container of the same type, equality compared, emptied with clear().

- Sequence Containers Elements are accessible in order as they were inserted/created. Find in linear time through the algorithm find.
- Associative Containers Elements are accessible in sorted order find as member function in logarithmic time
- Hashed Containers Elements are accessible in unspecified order find as member function in constant time

[width=0.75]container

Figure 3: Member Function of a Container

10.1 Introduction to std::array¡T, N¿ and std::vector¡T¿

Array
C++'s std::array¡T, N¿ is a fixed-size Container. T is a template type parameter (= placeholder for type). N is a positive integer, template non-type parameter (= placeholder for a value). Elements can be accessed with a subscript operator [] or at(). . The size is bound to the array object and can be queried using .size();. Avoid plain C-Array whenever possible: [language=C++]—int arr[]1, 2, 3, 4, 5;—

- at() throws an exception on invalid index access
- [] has undefined behavior on invalid index access Behavior
- The size of an array must be known at compile-time and cannot be changed. Otherwise it contains N default-constructed elements: std::array¡int, 5¿ emptyArray;

[width=0.75]images/array

Vector
C++'s std::vector¡T¿ is a Container = contains its elements of type T (no need to allocate them). java.util.ArrayList¡T¿ is a collection = keeps references to T objects (must be newed). T is a template type parameter (= placeholder for type). std::vector can be initialized with a list of elements. Otherwise it is empty: std::vector¡double¿ vd;.

[width=0.75]images/vector

Append Elements to an std::vector¡T¿

- $v.push\_back(< value >);$
- $v.insert(< iterator - position >, < value >);$

Filling a Vector with Values [language=C++] std::vecor¡int¿ v; v.resize(10); std::fill(std::begin(v), std::end(v), 2);

std::vector¡int¿ v(10); std::fill(std::begin(v), std::end(v), 2);

std::vector v(10, 2);

// Filling increased values with iota std::vector¡int¿ v(100); std:iota(std::begin(v), std::end(v), 1);

Finding and counting elements of a vector
std::find() and std::find_if() return an iterator to the first element that matches the value or condition. [language=C++] auto zero$_it = std :: find(std :: begin(v), std :: end(v), 0); if(zero_it == std :: end(v)) std :: cout << "nozerofound";$

10.2 Common Container Constructors

[language=C++] // Constructor with Initializer List std::vector¡int¿ v1,2,3,5,6,11; // Construction with a number of elements, five times a 42 std::list¡int¿ l(5,42); // Range with a pair of iterators std::deque¡int¿ qbegin(v), end(v);

10.3 Sequence Container

std::vector¡T¿, std::deque¡T¿, std::list¡T¿, std::array¡N, T¿.
Defines order of elements as inserted/appended to the container. Lists are very good for splicing and in the middle insertions. Array and deque are very efficient unless bad usage.

[width=0.6]sequencecontainer

Figure 4:

10.4 Associated Containers

Can be searched by content and not by sequence.

[width=0.6]associativecontainer

Figure 5:

10.5 Hashed Containers

Introduced in C++11. Standard lacks feature for creating your own hash functions.

10.6 Iterators

Different containers support iterators of different capabilities. Categories are formed around increasing "power".

Input Iterator
Supports reading the "current" element (of type Element). Allows for one-pass input algorithms. Can be compared with == and !=. Can also be copied.

[language=C++] struct input$_iterator_tag;$

Element operator*(); It operator++(); It operator++(int); bool operator==(It const); bool operator!=(It const); It operator=(It const); It(It const); //copy ctor

Forward Iterator
Can do whatever an input iterator can do plus: supports changing the current element. Still allows only for one-pass input algorithms. [language=C++] struct forward$_iterator_tag; Elementoperator*(); Itoperator++(); Itoperator++(int); booloperator==(Itconst); booloperator!=(Itconst); Itoperator=(Itconst); It(Itconst); //copyc$

Bidirectional Iterator
Can do whatever the forward iterator can do plus going backwards. [language=C++] struct bidirectional$_iterator_tag; Itoperator--(); Itoperator--(int);$

Random Access Iterator
Can do what the bidiretional iterator can do plus: Directly access element at index (offset to current position): distance can be positive or negative, Go n steps forward or backward, "Subtact" two iterators to get the distance, Compare with relational operators (¡, ¡=, ¿, ¿=). Allows also random access in algorithms.

Output Iterators
Can write value to current element, but only once (it = value). Modeled after std::ostream_iterator.

[language=C++] struct output$_iterator_tag; Elementoperator*(); Itoperator++(); Itoperator++(int);$

10.6.1 Iterator Functions

Has two functions std::distance(start, goal); std::advance(itr, n);.

[language=C++] int main() std::vector¡int¿ primes2, 3, 5, 7, 11, 13; auto current = std::begin(primes); auto afterNext = std::next(current); std::cout ¡¡ "current: " ¡¡ *current ¡¡ " afterNext: " ¡¡ *afterNext ¡¡ "; std::advance(current, 1); std::cout ¡¡ "current: " ¡¡ *current ¡¡ " afterNext: " ¡¡ *afterNext ¡¡ ";

11 STL Algorithms

The "algorithm.h" are the algorithms defined for general purpose. and in the "numeric.h" are the general numeric functions.

What is are the advantages of the STL algorithms?

- Correctness It is much easier to use an algorithm correctly than implementing loops correctly.
- Readability Applying the correct algorithm expresses your intention much better than a loop. Someone else will appreciate it when the code is readable and easily understandable.
- Performance Algorithms might perform better than handwritten loops

Iterator for Ranges

- First - Iterator pointing to the first element.
- Last - Iterator pointing to the last element.
- if First == Last the range is empty.

[language=C++] std::vector¡int¿ values54, 23, 17, 95, 85, 57, 12, 9; std::xxx(begin(values), end(values), ...);

11.1 Functor

11.2 Examples

11.3 Pitfalls

12 Function Templates

Can be compared as a Generic in Java. The keyword "template" is used to declare a template. The template parameter list contains one or more templates parameters. The compiler resolves the function template and figures out the template arguments. C++ uses duck-typing. So every type can be used as argument as long as it supports the used operations.

[language=C++] template ¡Template-Parameter-List¿ FunctionDefinition

[language=C++] // file min.h template ¡typename T¿ T min(T left, T right) return left ¡ right ? left : right; // file smaller.cpp include "min.h" smaller.cpp include ¡iostream¿ int main() int first; int second; if (std::cin ¿¿ first ¿¿ second) auto const smaller = min(first, second); std::cout ¡¡ "Smaller of " ¡¡ first ¡¡ " and " ¡¡ second ¡¡ " is: " ¡¡ smaller ¡¡ ";

Template Definition

- Templates are usually defined in a header file A compiler needs to see the whole template definition to create an instance Function template definitions are implicitly inline
- The definition in a source file is possible, but then it can only be used in that translation unit.
- Type checking happens twice When the template is defined: only basic checks are performed: syntax and resolution of names that are independent of the template parameters. When the template is instantiated (used): The compiler checks whether the template arguments can be used as required by the template.

Template Argument Deduction

12.1 Variadic Templates

13 Class Templates

- In addition to functions also class types can habe template parameters
- Since C++17, similar to function templates, the compiler might deduce the template arguments
- Compile-time polymorphism
- Class templates can be specialized

Rules

- Define class templates completely in header files
- Member functions of class templates Either in class template directly Or as inline function templates in the same header file
- When using language elements depending directly or indirectly on a template parameter, you must specify typename when it is naming a type.
- static member variables of a template class can be defined in header without violating ODR, even if included in several compilation units.

[language=C++] std::vector¡int¿ oldValues1, 2, 3;

[language=C++] std::vector newValues1, 2, 3; std::vector¡int¿ emptyValues;

template ¡TemplateParameters¿ class TemplateName /*...*/ ;

template ¡typename T¿ class Stack /*...*/ ;

Example [language=C++] template ¡typename T¿ class Sack using SackType = std::vector¡T¿; using size$_t$*ype* = *typenameSackType* :: *size*$_t$*ype*; *SackTypetheSack*;

public: bool empty() const return theSack.empty(); size$_t$*ypesize*()*constreturntheSack.size*();*voidputInto*
Deffinieren einer FUnktion ausserhalb der Klassendefinition [language=C++] template ¡typename T¿ inline T Sack¡T¿::getOut() if (empty()) throw std::logic$_e$*rror*"*EmptySack*"; *autoin*
static$_c$*ast* < *size*$_t$*ype* > (*rand*()*TretvaltheSack.at*(*index*); *theSack.erase*(*theSack.begin*()+ *index*); *returnretval*;

13.1 Type Aliases (C++11)

It is common for template definitions to define type aliases in order to ease their use. This has the advantage of less typing and reading als also a single point to change the aliased type. Before used typedef.

using StackType = std::vector¡T¿;

14 Heap Memory Management

- Stack memory is scarce
- It might be needed for creating object structures.
- Also needed for polymorphic factory functions to class hierarchies.
- Dont do it yourself! Always rely on library classes for managing the heap.
- Resource Acquisition Initialization (RAII) Idiom Allocation in the constructor Deallocation in the desctructor Use RAII wrapper as value in local scope Destructor will be called when the scoped is exited(}, return or exception).

14.1 Smart Pointers

In the modern C++ world we can use smart pointers, which are C++ templates, to make memory management easier. With these smart pointers we dont have to call ”delete ptr;” by ourselfs. Still: always prefer storing the value locally as value-type variable (Stack-based or member).

14.1.1 std::unique_ptr¡T¿

- defined in ”¡memory¿”
- Used for unshared heap memory Or for local stuff that must be on the heap Can be returned from a factory function
- Only a single owner exists
- It can wrap to-be-freed pointers from C functions when interfacing legacy code.
- Not the best for class hierarchies
- Can not be copied

14.1.2 std::shared_ptr¡T¿

- Works more like a java reference and allows multiple owners.
- The pinter is ”std::shared_ptr” and associates objects of Type T using ”Std::makeshared¡T¿(..).

[language=C++] struct Article Article(std::string title, std::string content); //.. ; Article cppExam”How to pass CPl?”, ”In order to pass the C++ exam, you have to...”; std::shared$_ptr < Article > abcPtr = std :: make_shared < Article > (”Alphabet”, ”ABCDEFGHIJKLMNOPQRSTUVXYZ”)$;

What is it for?

- If you really need heap-allocated objects, because you create your own object networks you can use std::shared_ptr¡T¿
- If you need to support run-time polymorphic container contents or class members that can not be passed as reference, e.g., because of lifetime issues

- Factory functions returning std::shared_ptr for heap allocated objects.
- But first check if alternatives are viable: (const) references as parameter types or class members Plain member objects or containers with plain class instances

The usage is counted on the referenced object to keep track of how many reference currently point to this object on the heap.

14.1.3 std::weak_ptr¡T¿