EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

C++ Programming I

Petr Gajdoš

VSB – Technical University of Ostrava

petr.gajdos@vsb.cz

Important Notes

This document is a compilation of several resources that are published under various licenses. Therefore, only links to other resources are provided with correspondings citations, and you are free to use this document as a guide in your studies. Used resources of other authors are available on internet in a form of e-books, etc.

All references are mentioned at the end of this document. Moreover, this document contains some interactive links to such resources that can be downloaded in form of PDF document. E.g. [1] – This resource is published under Creative Commons license, and can be downloaded to your local storage. Important references are highlighted in the "MUST READ" block.

This is a "MUST READ" block. Mentioned resources are important!

See the next slide to read how to prepare required resources and enable linkage between documents.

Linking Required Resources

1. Download [1] (version 2020-02-29) from here: http://www.ece.uvic.ca/~mdadams/cppbook
2. Save the file in the same location (presentation location).
3. Change the filename to "Adams2020-02-29.pdf" (case sensitive).
4. Click here to open the document and verify correct linkage.
5. Use "Alt+Tab" to swap between documents in your PDF reader (depending on type and version of your reader).

C++ Basics

The basics of C, C++, and C# or Java languages were introduced in the previous courses (Introduction to Programming, Object Oriented Programming, Programming in C# I, or Programming in Java I). Moreover, students of this course should have completed the courses Algorithms I, and Algorithms II.

That is why this document is focused only on the main aspects of C++ language and important features that can help you in your further programming.

C++

 See a brief history in [1] page(s) 28, and for more details go to [1] page(s) 36–56.

Pros:

- standardized language with history
- powerful language, low-level access to hardware
- multi-platform
- supported by many vendors → compilers, tools and libraries, optimizations, . . .
- wide application area → games, embedded systems, AI, HPC, telecommunications, databases, . . .

Cons:

- managed code → but smart pointers can help us
- types, memory management → nowadays, programming languages tries to not care

Software Development Environment

Generally, there exist many development environments that can be used, e.g. Dev C++, VSCode, CLion, Code::Blocks, Eclipse, CodeLite, Qt Creator, Brackets, Atom, Visual Studio, ...

Recommended SDE for Linux:

- VSCode with required extensions (depending on the selected toolchain)

Recommended SDE for Windows:

- Visual Studio in case you want MS VC++ Compiler and MS Build Tool (but Clang and CMake are also supported)
- VSCode with required extensions (depending on the selected toolchain) in case you want to use primarily other toolchains and build tools than MS Build

From Windows to Linux:
Using Windows Subsystem for Linux (WSL2), you can work in an "embedded Linux" within your Windows. VSCode installed on windows can run linux build tools, GNU toolchain, ... $\rightarrow$ dual boot is not needed anymore!

- VSCode with required extensions (depending on the selected toolchain) in case you want to use other toolchains and building tools

Thanks to WSL2, there is no need to install forks of linux tools for windows, e.g. Cygwin, MinGW, MSYS2.

Simplified Toolchain

Generally, toolchain is a collection of programming tools used for developing software applications. From perspective of developing C++ application on a selected platform, the set of tools contains:

- Preprocessor: source files (SRC) $\rightarrow$ translation units (TU)
- Compiler: TU $\rightarrow \ldots \rightarrow$ Assembly Code (AC)
  - Lexical Analyzer: TU $\rightarrow$ tokens
  - Syntax Analyzer: tokens $\rightarrow$ Syntax Tree (ST)
  - Semantic Analyzer: ST $\rightarrow$ Semantic Structures (SS), check for type mismatches, incompatible operands, function calls with improper arguments, undeclared variables, ...
  - Intermediate Code Generator: $\ldots \rightarrow$ Intermediate Code(IC) for abstract machine
  - Code Optimizer: IC $\rightarrow$ Optimized Code (OC), unnecessary code is removed
  - Code Generator: OC $\rightarrow$ Assembly Code (AC), allocate storage and generate relocable machine code
- Assembler: AC $\rightarrow$ Object Files (OF)
- Linker: OF $\rightarrow$ Executables

Preprocessor - Building Translation Units

- C++ source files generally have the .cpp, .cxx or .cc extension suffixes.
- A C++ source file can include other files, known as header files, with the #include directive. Header files have extensions like .h, .hpp, or .hxx, or have no extension at all.
- Only source files are passed to the compiler (to preprocess and compile it). Header files aren't passed to the compiler, they are included from source files. Thus each header file can be opened multiple times during the preprocessing phase of all source files.
- A translation unit (TU) is build for each source file. Remember TU when we will speak about scope of some variables.
- Preprocessor handles #include, #define, #ifndef, #... in the source codes.

Listing 1: helloVSB.cpp

```cpp
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char** argv)
{
    cout << "Hello VSB" << endl;
    return 0;
}
```

Listing 2: Getting TU from the source file using GCC

```
g++ -E helloVSB.cpp -o helloVSB.ii
```

See the number of lines in the generated output file helloVSB.ii !!!

Get more technical notes on preprocessor in [1] page(s) 81–86.

Compiler - Building Assembly Code

- The compiler parses the pure C++ source code (now without any preprocessor directives) and converts it into assembly code.
- Performs code optimizations on several levels.
- Finally invokes underlying back-end (assembler in toolchain).

Listing 3: Getting assembly code from the source file using GCC

```
g++ -S helloVSB.cpp
# g++ -S helloVSB.cpp -o helloVSB.s
```

Listing 4: Getting assembly code for TU file using GCC

```
g++ -S helloVSB.ii
# g++ -S helloVSB.ii -o helloVSB.s
```

Assembler - Building Object File

- The assembler converts the file that's generated by compiler into an object code file.
- Produces binary file in some format (ELF, COFF, a.out,...).
- Final object file contains the compiled code (in binary form) of the symbols defined in the input.
- Symbols in object files are referred to by name.
- Object files can refer to symbols that are not defined. Linker should solve this.

Listing 5: Getting object file using GCC

```
g++ -c helloVSB.cpp
# g++ -c helloVSB.cpp -o helloVSB.o
```

Linker - Building Executable

- Links together object files.
- Checks referenced symbols within object files.
- Links all the object files by replacing the references to undefined symbols with the correct addresses.
- This output can be either a shared or dynamic library, or an executable.
- The most common errors: “Unresolved external symbol . . . ”, or “duplicate definitions”

Listing 6: Getting executable using GCC

```
g++ helloVSB.o -o helloVSB
# g++ helloVSB.o aaa.o bbb.o ccc.o -o helloVSB
```

Summary on Compilation

- Knowing the difference between the compilation phase and the link phase can make it easier to hunt for bugs.
- Compiler errors are usually syntactic in nature – a missing semicolon, an extra parenthesis.
- Linking errors usually have to do with missing or multiple definitions.
- Usually, you want to run the whole toolchain at once with all supplementary options, e.g. include directory (-I), include file (-i), link directory (-L), link library (-l), set optimization level (-O*), and much more.
- Setting of different compilers may differ in notion, but the main concept is the same.

See the summary of GCC options.
See the summary of MS VC options.
See the summary of Clang options (LLVM).

Build Tools

Build tools simplify process of compilation, provide configuration wizards, and store all custom settings in some user-friendly format. They provide some kind of scripting or automating the process of compiling source code into binary code, optionally the have GUI. There exist several build tools, e.g. MS Build, Ninja, GNU Make, Qt Build System, boost.build, ...

Moreover, there exist so called Build-script generators (CMake, Mason, GNU Build System, ...) whose purpose is to generate files to be used by a native build tool. They represent a layer over build tools and simplify configuration in some way, e.g. multi-platform or cross-platform compilation.

CMake

- Does not create any output in the source directory, i.e. bin, obj, etc. It performs an out-of-source build and performs the build there.
- It generate files required by suitable buildsystem, and then can invoke a build tool to process the generated buildsystem file.
- Usually, script files are written; otherwise CMake Gui can help with settings.

Typical simplified scenario:

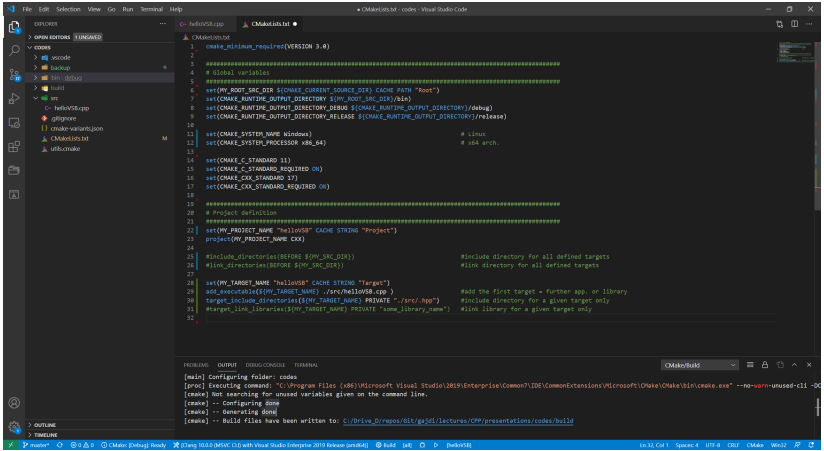
1. Create a CMakeList.txt for your project. This is an entry point for the CMake tool.
2. Configure your project: CMakeList.txt and global definitions are processed, CMake cache is created, project targets are created.
3. Build your project: files required by a buildsystem are generated (e.g. *.sln, *.ninja, ...), after than the selected buildsystem can be called $\rightarrow$ source code is compiled

Listing 7: A minimalistic CMake example

```
cmake_minimum_required(VERSION 3.0)

## Global variables
set(MY_ROOT_SRC_DIR ${CMAKE_CURRENT_SOURCE_DIR} CACHE PATH "Root")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${MY_ROOT_SRC_DIR}/bin)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY_DEBUG ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/debug)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY_RELEASE ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/release)

## Project definition
set(MY_PROJECT_NAME "helloVSB" CACHE STRING "Project")
project(MY_PROJECT_NAME CXX)                                #start a new project
set(CMAKE_SYSTEM_NAME Linux)                                # Windows/Linux/Darwin/Android/FreeBSD/MSYS
set(CMAKE_SYSTEM_PROCESSOR x86_64)                          # x64 arch.

## Compiler definition
set(CMAKE_C_STANDARD 11)
set(CMAKE_C_STANDARD_REQUIRED ON)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

## Targets definitions
#include_directories(BEFORE ${MY_SRC_DIR})                   #include directory for all defined targets
#link_directories(BEFORE ${MY_SRC_DIR})                     #link directory for all defined targets

set(MY_TARGET_NAME "helloVSB" CACHE STRING "Target")
add_executable(${MY_TARGET_NAME} ./src/helloVSB.cpp )       #add the first target = further app. or library
target_include_directories(${MY_TARGET_NAME} PRIVATE "./src/.hpp")      #inc. dir. for a given target only
#target_link_libraries(${MY_TARGET_NAME} PRIVATE "some_library_name")   #link library for a given target only
```

Some definitions can be changed when calling CMake from GUI, e.g. calling CMake commands via command bar in VSCode. See CMake documentation for more details.

Figure 1: VSCode with CMake command bar on bottom (extensions: ms-vscode.cmake-tools and twxs.cmake)



C++ Types, and Literals

 See the list of fundamental c++ types in [1] page(s) 88–89.

 Read more about literals in [1] page(s) 90–99.

There are also type aliases of fundamental types defined in other header files that can be included, e.g. <bits/types.h>, <stdint.h>, <cstdint>, etc. However, there can be some uncertainties when including header files. Header files prior to C++11 should be avoided.

E.g. including C++ header file <cstdint> imports the symbol names in std namespace and possibly in global namespace, on the other hand including C header <stdint.h> imports the symbol names in global namespace and possibly in std namespace.

Type Aliases

Sometimes, it is useful to simplify type names to increase code readability.

C style: typedef creates an alias that can be used anywhere in place of a (possibly complex) type name. The same can be used in C++. typedef is a kind of init-statement.

C++ style: using statement is a preferable way to create alias for an existing type. It is also called alias-declarations. Alias-declaration is not an init-statement, and thus may not be used in contexts which allows initialization statements.

Listing 8: Type aliasing

```cpp
#include <cstdint>
#include <iostream>

using std::cout;
using std::endl;

// typedef unsigned long long ULL;          // C-style with typedef
// typedef unsigned long long* pULL;        // C-style with typedef

using ULL = unsigned long long;             // C++ style
using pULL = unsigned long long*;           // C++ style

int main(int  /*argc*/, char**  /*argv*/)
{
    ULL x = 123ull;
    pULL px = &x;
    cout << typeid(x).name() << endl;
    cout << typeid(px).name() << endl;
    return 0;
}
```

Listing 9: "Minor" differences between typedef and using

```
void seeTheDifference()
{
    for(typedef int Foo; Foo{} != 0;) {}    // OK
    //for(using Foo = int; Foo{} != 0;) {}  // Error: using can not be used in the init-statement of the FOR

    if (typedef int Foo; true) { cout << Foo{} << endl; }   // OK
    //if (using Foo = int; true) { cout << Foo{} << endl; } // Error: using can not be used in the init-statement of ↩
          the IF

    //C++20 is needed
    // for(typedef struct { int x; int y;} S; auto [x, y] : {S{1, 1}, {1, 2}, {3, 5}})
    // {
    //  cout << x << " " << y << endl;
    // }
    // Can not use ‘‘using’’ here at all
}
```

There are more cases when using keyword is more usable for the programmers, e.g. in alias templates, that will be discussed later.

Getting Type Info

C++ has several operators dealing with types. They get more details on types, e.g. typeid, or they retrieve a complete type from some variable, e.g. decltype.

typeid

- Returns a std::type_info object, then e.g. name(), hash_code() can be called.
- "<typeinfo>" header must be included in advance.
- typeid expression is resolved at compile time when applied to a non-polymorphic type; otherwise is resolved at runtime.

Listing 10: Example on typeid

```
int x = 123;
cout << typeid(x).name() << endl;           // using a variable name
cout << typeid(x).hash_code() << endl;
int *y = &x;
cout << typeid(y).name() << endl;
cout << typeid(y).hash_code() << endl;

cout << typeid(int).name() << endl;         // using a type name
```

decltype

- Inspects the declared type of an entity or the type and value category of an expression.
- Can be used to determine "auto" type.
- Everywhere you want to get the type of an expression or declaration, you use decltype.

Listing 11: Example on decltype

```
int x = 123;
cout << typeid(x).name() << endl;           // using a variable name

decltype(x) y = 123;                        // retrieving type from a variable
cout << typeid(y).name() << endl;           // float

auto z = x + 0.5f;
decltype(z) w;
cout << typeid(w).name() << endl;           // float
```

decltype will be discussed in more detail with template classes and methods, e.g.

Listing 12: Example on decltype -> trailing type

```
template<typename A, typename B>
auto add(A const& a, B const& b) -> decltype(a + b) { return a + b; }
```

auto: Consequences of Deduction Rules

auto mostly follows the same type deduction rules as template argument deduction. The only difference is that auto will deduce std::initializer_list from a braced-init-list in some cases, while template argument deduction doesn't do this. This will be discussed later.

However, the behavior in some cases is the same as what template argument deduction would do when deducing types from a function call. See the next example, where const is expected, but it is omitted. auto by itself means that you want a new, locally-owned variable with a copy

of the given value. const-ness is not part of value. An int is an int whether it's specified using a literal, a named constant, an expression, or a non-const variable.

Listing 13: Simple examples on type deduction

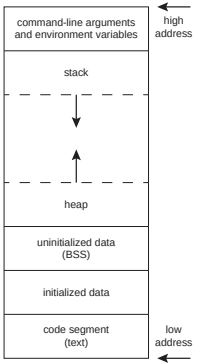
```
const int x0 = 123;                     //It might actually be initialized to zero. Depends on the context.

auto x1 = x0;
cout << typeid(x1).name() << endl;      //int -> If some T is a cv-qualified type, the top level
                                        //cv-qualifiers of T's type are ignored for type deduction
auto& x2 = x0;
cout << typeid(x2).name() << endl;      //const int -> the const qualifier is retained. x2 receives
                                        //type const int& and aliases x0.
const auto x3 = x0;
cout << typeid(x3).name() << endl;      //const int

decltype(auto) x4 = x0;                 //since C++14: const int

auto&& x5 = x0;                         //x5 receives type const int& and aliases x0.
```

Memory Schema

See the memory organization in [1] page(s) 643–646.

Generally, there can be more segments, but this is the simplified schema. For more details, try to search for Executable and Linkable Format (ELF).

- Environment variables and CL arguments "sit" at highest addresses.
- Stack: Growing but limited amount of memory containing stack frames (variables, function arguments, function return values, and caller return addresses)
- Heap: All dynamic allocations take place here.
- BSS: Uninitialized data, typically set to zero when program loaded
- Initialized data: Contain variables initialized to a concrete value upon program loading, prior to execution.
- Code segment: Read-only place where the program "sits" in memory.



Storage Duration

It is important to know how long objects stay in memory to avoid errors that are hard to discover without help of address sanitizer; e.g. a function returns a pointer to a variable allocated within the function.
There are four types of duration:

- automatic: duration is defined by an enclosing code block

  e.g. `{ int i = 123; }`

- static: duration is defined by lifetime of the program, from the start to the end

  e.g. all objects declared at namespace scope, or declared as static or extern

- dynamic duration is defined for all objects allocated dynamically. It is defined by allocation and deallocation. NOTICE: Pointers to these objects have own duration!!!

  e.g. `MyClass* ptr = new MyClass(); ... delete ptr;`

- thread duration is related to thread lifetime, objects must be declared as thread_local.

Initialization

C++ has several ways how to initialize objects: Initialization is a process of providing initial value to object at time of its construction. Initialization $\neq$ declaration.

Listing 14: Possible Initializations

```
struct S { int a; double b; };

int a = 123;                            // CONSTANT initialization -> static storage duration
const int b = 123;                      // CONSTANT initialization -> static storage duration
static int c = 123;                     // CONSTANT initialization -> static storage duration

int main(int argc, char** argv)
{
    static int d = 123;                 // CONSTANT initialization -> static storage duration

    static int e;                       // CONSTANT initialization -> ZERO initialization

    int f;                              // DEFAULT initialization -> automatic, static, or thread-local storage;
                                        //INDETERMINATE value.
    int *g = new int;                   // DEFAULT initialization -> dynamic storage; INDETERMINATE value.
                                        // DEFAULT initialization is performed als for all non-static data members
                                        // not mentioned in called constructor initializer list


    //int h();                          // MOST VEXING PARSE: Variable creation vs. function call ???
                                        // But in case of this, it is a valid VALUE INITIALIZATION
                                        // Foo::Foo() : x() {} where x is a data member of Foo class
```

```
    int i{};                                // VALUE initialization to the default value -> ZERO initialization
    int *j = new int();                     // VALUE initialization -> what happen is based on concrete type
    int *k = new int{};                     // VALUE initialization -> what happen is based on concrete type

    int l(123);                             // DIRECT initialization; there can be more arguments
    int m{123};                             // DIRECT initialization; there can be more arguments
    int *n = new int(123);                  // DIRECT initialization; there can be more arguments

    int o = 123;                            // COPY initialization
    int p = o;                              // COPY initialization
    int q = {p};                            // COPY initialization

                                            // LIST initializations
    S r{ 123, 1.23 };                       // DIRECT LIST initialization
    // ... = S{ 123, 1.23 };                // DIRECT LIST initialization
    S *s = new S{123, 1.123};               // DIRECT LIST initialization
    S t = {123, 1.123};                     // COPY LIST initialization
                                            // List initialization has more forms, e.g. aggregate or reference

    //WHAT ABOUT ARRAYS?
    char u[10];                             // u has INDETERMINATE value
    int v[10] = {1,2,3};                    // COPY LIST initialization -> AGGREGATE initialization, arrays
    char w[10]{"Hello VSB"};                // DIRECT LIST initialization -> AGGREGATE initialization, arrays
    char x[10]{};                           // DIRECT LIST initialization
    char *z = new char[10]{"Hello VSB"};    // DIRECT LIST initialization -> AGGREGATE initialization, arrays
    return 0;
}
```

Initialization vs. Assignment

The following class *Foo* illustrates the calling sequence only. A new macro *__PRINT__* was defined just to make the notation more readable. Generally, macros should be avoided in modern C++.

Listing 15: Foo Class with COUTs

```
#include <iostream>

using std::cout;
using std::endl;

#define __PRINT__  cout <<  __PRETTY_FUNCTION__ <<  endl;       //use __FUNCSIG__ on MVCS

class Foo
{
public:
    Foo() { __PRINT__ }
    Foo(int _i) { __PRINT__ }
    ~Foo() { __PRINT__ }
    Foo& operator=(const Foo &a) noexcept {  __PRINT__ return *this; }  //copy assignment
    Foo& operator=(Foo &&a) noexcept { __PRINT__ return *this; };       //move assignment
};
```

```cpp
void initialization()
{
    __PRINT__
    Foo x{123};
}

void assignment()
{
    __PRINT__
    Foo x;
    x = 123;
}

int main(int argc, char** argv)
{
    initialization();
    cout << endl;
    assignment();
    return 0;
}
```

```
void initialization()
Foo::Foo(int)
Foo::~Foo()

void assignment()
Foo::Foo()
Foo::Foo(int)
Foo &Foo::operator=(Foo &&)
Foo::~Foo()
Foo::~Foo()
```

Init vs. Assignment: See the difference in the calling sequences on the right.

If possible, always prefer direct initialization over assignment (copy initialization). Assignments create temporary objects that are later discarded.

Uniform Initialization

Use the brace-initialization form {} to uniformly initialize objects regardless of their type. This can be used for both direct initialization and copy initialization. This initialization is preferred in modern C++.

Initialization of standard containers (such as the vector and the map, . . .) is possible because all standard containers have an additional constructor that takes an argument of the type std::initializer_list<T>.

The way initialization using *std::initializer_list<T>* works is as follows:

- The compiler resolves the types of the elements in the initialization list (all the elements must have the same type).
- The compiler creates an array with the elements in the initializer list.
- Compiler creates an *std::initializer_list<T>* object to wrap the previously created array.
- The *std::initializer_list<T>* object is passed as an argument to the constructor

For an empty braced initializer list, the default constructor is called; otherwise constructor taking initializer list takes precedence over other constructors.

Listing 16: List Initialization (Brace Initialization)

```cpp
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
using std::initializer_list;

#define __PRINT__ cout << __PRETTY_FUNCTION__ << endl;

class Foo
{
    vector<int> m_data;
public:
    Foo() : m_data{} { __PRINT__ }
    Foo(initializer_list<int> list) : m_data{list} { __PRINT__ }
};

int main(int argc, char** argv)
{
    Foo f0{};                   //
    Foo f1{123, 321};           //OK
    Foo f2 = {123, 321};        //OK

    vector<int> v0{1,2,3};      //OK
    vector<int> v1 = {1,2,3};   //OK
}
```

```
Foo::Foo()
Foo::Foo(initializer_list<int>)
Foo::Foo(initializer_list<int>)
```

Initializing Non-static Members

See the recommended initialization of non-static members in the code.

Listing 17: Where to Init

```
struct Bar { int a; double b; };
class Foo
{
    const int x = 123;      // [1] both static and non-static constants
    Bar b{1, 1.23};         // [2] default values for members of classes with multiple constructors

    int y;
public:
    Foo() {}
    Foo(int _y) : y{_y} {}  // [3] use direct-list initialization for members that don't have default values
    Foo(int _y, Bar _b) : y{_y}, b{_b} {}
    Foo(int _y, Bar _b)
    {
        y = _y;             // [4] use assignment in constructors when the other options are not possible.
        b = _b;
    }
};
```

The Keyword auto

auto prevents correctness and performance issues that can bedevil manual type declarations, but some of auto's type deduction results may be wrong from the perspective of a programmer, and thus it is important to know how to guide *auto* to the right answer.

The placeholder auto may be accompanied by modifiers, such as *const* or *&*, which will participate in the type deduction.

Listing 18: Simple examples on type deduction

```
auto getInt() -> int { return 123; }        // Trailing type int
void usingAuto()
{
    auto i = 123;
    auto f = 123.0f;
    auto s = "Hello VSB";
    auto& j = i;
    auto a = { 1, 2 };
    auto b{4};
    for (auto x : a) {}
    auto r = getInt();
}
```

Reference vs. Pointer

Pointer is an object whose value is an address in memory where another object (instance, variable, function, ...) is stored.

 Read more basics about pointers in [1] page(s) 105–106.

Reference is an alias for already existing object → a reference can not be null and its value must always exist.

 Read more basics about pointers in [1] page(s) 107–109.

Const vs. Constexpr Keywords

const

- Creates a variable whose value can not be changed.
- Attempt to modify this value causes compilation error.
- Needs an initializer.
- Commonly used when the same "magic value" appears in code multiple times → avoiding mistakes when changing the value.
- Can be applied to member methods.
- Use const to indicate that the value can not be modified, or that the member function is constant.
- Common syntax: const <type> <variable>{initializer}, e.g. `const int c{123};`

Listing 19: Example on const

```
int getA() { return 123; }

void t_const()
{
    const int c0 = 123;          // this can be evaluated at compile time because its address is never read
    cout << c0 << endl;
    int p[c0];                   // OK

    const int c1 = getA();       // return type of getInt is not const -> this constant is evaluated at runtime
    //int p1[c1];                // Compiler error
}
```

constexpr

- Represents a constant expression.
- Such expressions can be evaluated at compile time $\rightarrow$ performance increasing.
- Is applied on variable declarations or functions.
- Can be applied to member methods.
- Functions can be constexpr if they return a value that can be evaluated at compile time and return a literal type (void, scalar types, references, ...)
- In C++11, a constexpr function should contain only one return statement. From C++14, they can have more than one statements.
- All constexpr functions are implicitly inline.
- Constexpr is always const, but const is not constexpr.
- Use constexpr to indicate expressions that can be evaluated at compile time.

Listing 20: Example on constexpr

```
constexpr int getB() { return 123; }

void t_constexpr()
{
    constexpr int c0 = 123;          // evaluated at compile time
    cout << c0 << endl;
    int p[c0];                       // OK

    constexpr int c1 = getB();       // getInt() is evaluated at compile time as well
    int p1[c1];                      // OK -> evaluated at compile time
}
```

Listing 21: Behavior of constexpr function

```
constexpr int sum(int a, int b) { return a+b; }

void t_mix()
{
    constexpr int c0 = sum(1,2);     // behaves as a constexpr function, the value is evaluated at compile time.
    const int c1 = sum(1,2);         // behaves as a constexpr function, the value is evaluated at compile time.
    int c2 = sum(1,2);               // behaves as a standard function, the value is evaluated at runtime.

    //constexpr int c3 = sum(c2,1); // Compiler error, c2 is not a compile time constant -> can not be used
                                     // to evaluate compile time expression
}
```

The Statement constexpr-if

It is possible to use constexpr after if keyword to indicate constexpr-if statement.
This constexpr-if statement is evaluated at compile time and all branches of if statement that are not taken are discarded.

- This is often used in templated code when it is necessary to do certain things differently, e.g. depending on the type. This will be discussed later.

Generally, it can be used in if statement for those expressions that can be evaluated at compile time.

Listing 22: Using constrexp-if

```
template <typename T>
bool isFloatValue(T x)
{
    if constexpr (std::is_same_v<T, float>)
    {
        return true;
    }
    return false;
}
```

```
void t_constexpr_if()
{
    constexpr int x = 123;
    if constexpr (x < 100)
        cout << "x < 100" << endl;
    else
        cout << "x >= 100" << endl;
}
```

Helper Functions, Definitions

Helper Functions, Definitions

Before you read the following slides, you should be familiar with some definitions that will be used. These common definitions simplify code examples.

using statement will be used to make symbol name from the given namespace accessible for unqualified lookup as if declared in the same class scope, block scope, or namespace as where this using-declaration appears.

Listing 23: Included headers + usings

```
#include <iostream>
#include <iomanip>

using std::cout;
using std::endl;
using std::left;
using std::setw;
```

Some helper template functions are defined. Template functions and classes will be discussed later. For now, this is just a notice that this functions can be used in below mentioned examples.

Listing 24: Printing size of variables and types, + alignments

```
template<class T>
constexpr void printSizeInfo(const T value)
{
    cout << "size: " << left << setw(5) << sizeof(T) << endl;
}

template<typename T>
constexpr void printSizeInfo()
{
    cout << "size: " << left << setw(5) << sizeof(T) << "alignment: " << alignof(T) << endl;
}
```

Other template functions provide information on data member offset. The standard C++ keyword offsetof can be used instead, but it works on 100% for standard-layout classes only, otherwise it is conditionally supported.
On the other hand, this can not be used at "compile-time" because of implicit casting leading to forbidden reinterpret_cast.

Listing 25: Getting data memeber offset

```
template <typename T, typename U>
size_t offset_of(T const U::* member)
{
    const U object {};
    return size_t(&(object.*member)) - size_t(&object);
}

template <typename T, typename U> void printOffsetInfo(T const U::* member)
{
    const size_t offset = offset_of(member);
    cout << "class size: " << left << setw(5) << sizeof(U) << " member size: " << left << setw(5) << sizeof(T) << "↩
         offset: " << offset << endl;
}
```

Finally, a helper function to print value of type T in its binary form is provided. 2^0 bit is on the right, $2^{sizeof(T)*8-1}$ bit is on the left, i.e. least significant bit is on the right in the output.

Listing 26: Print bits

```
template<typename T>
bool printBits(const T& value)
{
    const unsigned char* ptr = reinterpret_cast<const unsigned char*>(&value) + sizeof(T) - 1 ;
    for (int i=sizeof(T); i>0; i--, ptr--)
    {
        for (int j=7; j>=0; j--)
        {
            cout << ((*ptr & (1 << j)) > 0 ? "1" : "0");
        }
    }
    cout << endl;
    return true;
}
```

Listing 27: Use this in C++20 and above

```
#include <format>
...
const unsigned char* ptr = reinterpret_cast<const unsigned char*>(&some_variable);
std::format("{:b}", *ptr);
```

Structures

Introduction

See the basics for c++ structs/unions/classes in [1] page(s) 234–348.

- struct is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths.
- Struct is a class where members are public by default.
- Other access modifiers can be used as well; However, it is a best-practice to make structures as small as possible without extra definitions that brings structures closer to standard classes.
- Ordering of members can play an important role in size and alignment of the structure.

Structure Examples

Listing 28: Simple structures

```
struct S0       //Everything is public by default !!!
{
    int a;
    char b;
};

struct S1
{
    int a;
private:        // Access modifier other than public
    char b;
};

struct S2
{
    int a = 123; //Default values
    char b ;
};
```

Listing 29: ... with constructors

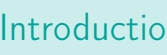
```
struct S3
{
    int a;
    char b;

    S3() : a(123), b('a') {}
    S3(int _a) : a(_a), b('a') {}

    // This will cause an error because of default value
    // -> ambiguous call of constructor
    // -> remove default value
    S3(int _a, char _b = 'a') : a(_a), b(_b) {}

    // Constructor Delegation
    S3(char _b) : S3(123, _b) {}

    // More on constructors and operators will be ↩
        explained on classes !!!
};
```

Listing 30: Possible initializations (some of them can not be used when members are of complex types)

```
// If a structure if reduced to its bare minium, with no constructor,
// no method, no inheritance, no private method or data, no member initializer,
// if a structure only defines public data members, then an special initialization
// feature of C++ (aggregate initialization) can be used.
S0 s0{4,'x'};

//S1 s1{4};                     // Can not use aggregate initialization because of private data member

S2 a{321,'a'};
S2 b{'x'};
S2 c{321};

S2 d = {321,'a'};
S2 e{a: 123, b: 'x' };          // old GNU style
S2 f{b: 'x', a: 123};           // old GNU style
S2 g{.a = 123, .b = 'x'};       // new
S2 h = {.a = 123, .b = 'x'};    // new

S3 i;
//S3 j{123};                    // Error: call to constructor of 'S3' is ambiguous => compiler does not know what ↩
     constructor to call !!!
S3 k{123, 'c'};
```

Unions

Introduction



See the basics for c++ structs/unions/classes in [1] page(s) 234–348.

- union is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths.
- Unions is a class where members are public by default.
- Other access modifiers can be used as well; However, it is a best-practice to make structures as small as possible without extra definitions that brings structures closer to standard classes.
- Ordering of members can play an important role in size and alignment of the structure.
- Union is a special class type that can hold only one of its non-static data members at a time.

- Unions cannot contain a non-static data member with a non-trivial special member function (copy constructor, copy-assignment operator, or destructor).
- If a union contains a non-static data member with a non-trivial special member function (copy/move constructor, copy/move assignment, or destructor), that function is deleted by default in the union and needs to be defined explicitly by the programmer.

Unions Examples

Listing 31: Simple unions

```
union U0                    // size: 4  alignment: 4
{
    int b;
    bool a;
};

union MIX                   // size: 4    alignment: 4
{
    struct {
        char a;
        char b;
        char c;
        char d;
    };
    int m_data;
};
```

Listing 32: ... and their usage

```
    printSizeInfo<U0>();

    U0 u{true};
    cout << "u.a = " << u.a << endl;
    cout << "u.b = " << u.b << endl;
    u.a = false;
    cout << "u.a = " << u.a << endl;
    cout << "u.b = " << u.b << endl;


    U0 v{513};
    cout << "v.a = " << v.a << endl;
    cout << "v.b = " << v.b << endl;
    v.a = false;
    cout << "v.a = " << v.a << endl;
    cout << "v.b = " << v.b << endl;

    MIX m{1,3,7,15};
    printSizeInfo<MIX>();
    printBits(m.m_data);

    // 0x000000ff = 0b00000000000000000000000011111111;
    m.m_data = 0x000000ff;
    printBits(m.m_data);
    printBits(m.a);
```

Alignment, Padding

Alignment

Processors do not access memory one byte at a time, but in larger chunks of powers of two (2, 4, 8, 16, 32, and so on). This means, that it is important that compilers align data in memory so that it can be easily accessed by the processor. In case of misalignment, data must be read in multiple chunks, shift, unnecessary bytes must be discarded, and the rest combined.

Compilers align variables based on the size of their data type. Typically, these are 1 byte for bool and char, 2 bytes for short, 4 bytes for int, long, and float, 8 bytes for double and long long, and so on.

When it comes to structures or unions, the alignment must match the size of the largest member in order to avoid performance issues.

Listing 33: Simple alignment, see also orderings of bytes

```
struct S0                // size=2, alignment=1, bytes=|a|b|
{
    bool a;
    char b;
};

struct S1                // size=8, alignment=4, bytes=|aaaa|b...|
{
    int a;
    bool b;
};

struct S2                // size=8, alignment=4, bytes=|a...|bbbb|
{
    bool a;
    int b;
};
```

When it comes to structures or unions, the alignment must match the size of the largest member in order to avoid performance issues.

The some padding must be added int the inner representation to match this alignment.

Also it is a good practice to place members within the structure according to its size in decreasing order (S1 vs. S2).

alignas keyword

Listing 34: Aligning data members, see also orderings of bytes

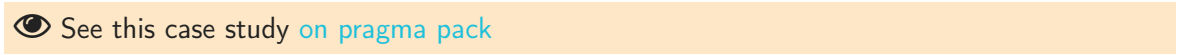
```
struct S3                   // size=8, alignment=4, bytes=|axxx|bbbb|
{
    bool a;
    char x[3];              // manual padding
    int b;
};

struct alignas(4) S4        // size=4, alignment=4, bytes=|ab..|
{
    bool a;
    char b;
};

struct alignas(8) S5        // size=16, alignemnt=8
{
    alignas(2) bool a;
    alignas(8) int b;
};
```

The alignas specifier can be applied both to the type declaration and the member data declarations.

"Manual" aligning/padding can be done with including extra data members (see S3).

#pragma pack(1)

#pragma pack(1) instructs the compiler to pack structure members with particular alignment. Most compilers, when you declare a struct, will insert padding between members to ensure that they are aligned to appropriate addresses in memory. This avoids the performance penalty and #pragma pack(1) can lead to worse performance.



bitfields

Bitfields

Data structure that declares data member with explicit size in bits. Adjacent bitfield members may be packed to share and straddle the individual bytes.

- The name of the bitfield (member) that is being declared is optional -> nameless bitfield introduces the specified number of bits of padding
- Because bitfields do not necessarily begin at the beginning of a byte, address of a bitfield member cannot be taken. In other words, we cannot have pointers to bitfield members as they may not start at a byte boundary. Pointers and non-const references to bitfield members are not possible.
- A bitfield member cannot be static

Bitfields Examples

Listing 35: Bitfields examples

```
struct S0
{
    uint8_t  a : 3;     // 3 bits
    uint8_t  : 2;       // 2 bits unused            => ↩
          COMPILER DO NOT OPTIMIZE, AND PREFERES SMALLER↩
           SIZEOF THE STRUCTURE !!!
    uint8_t  b : 2;     // 2 bits
};

struct S1
{
    uint8_t  a : 3;     // 3 bits
    uint8_t  : 2;       // 2 bits unused            => ↩
          BUT COMPILER OPTIMIZE THIS AND ALIGN NEXT DATA↩
           TO NEXT BYTE !!!
    uint8_t  b : 6;     // 6 bits
    uint8_t  c : 2;     // 2 bits
};
```

Listing 36: Usage

```
    printSizeInfo<S0>();
    S0 s0{.a = 0b111, .b = 1};
    printBits(s0);

    printSizeInfo<S1>();
    S1 s1{.a = 0b111, .b = 1, .c = 1};
    printBits(s1);
```

```
size: 1    alignment: 1
00100111
size: 2    alignment: 1
0100000100000111
```

Listing 37: Bitfields examples

```
struct S2
{
    uint8_t  a : 3;     // 3 bits
    uint8_t  b : 2;     // 2 bits unused
    uint8_t  c : 6;     // 6 bits
    uint8_t  d : 2;     // 2 bits
};

struct S3
{
    uint8_t  a : 3;     // 3 bits: value of a
    uint8_t  : 0;       // 5 bits: unused, start at new ↩
         byte
    uint8_t  b : 6;     // 6 bits: value of b
    uint8_t  c : 2;     // 2 bits: value of
};
```

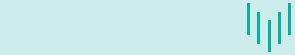
Listing 38: Usage

```
    printSizeInfo<S2>();
    S2 s2{.a = 0b111, .b = 1, .c = 1};
    printBits(s2);

    printSizeInfo<S3>();
    S3 s3{.a = 0b111, .b = 1, .c = 1};
    printBits(s3);
```

```
size: 2    alignment: 1
0000000100001111
size: 2    alignment: 1
0100000100000111
```

Classes

Introduction



See the basics for c++ classes in [1] page(s) 234–348.

In the following slides, some key features of classes will be highlighted. It is important to read above mentioned slides, or remember thinks from the previous subjects.

Size of a Class

Now, you should know that class can contain data members, function members, and type members. However, it can consist of zero or more members.

Only non-static data declarations in a class definition add anything to the size of class objects! But the *sizeof* an empty class in not zero!

Until C++20, the minimum size for all objects is at least 1 byte => also instances of an empty class must occupy at least 1 byte used to unique address identification.

Listing 39: Example on empty classes

```
class EmptyClass{};

class ClassWithEmpties0{
    EmptyClass e0;
    EmptyClass e1;
    EmptyClass e2;
};

//Since c++20, there is [[no_unique_addess]] attribute
class ClassWithEpmties1
{
    int x;
    [[no_unique_address]] EmptyClass e;
};

void t_emptyClass()
{
    printSizeInfo<EmptyClass>();            // probably 1
    EmptyClass e0;
    printSizeInfo(e0);                      // probably 1

    printSizeInfo<ClassWithEmpties0>();     // probably 3
    printSizeInfo<ClassWithEmpties1>();     // probably 8 in c++ < 20,
                                            // 4 in c++ >= 20 with compiler support
}
```

EBO

- The "Empty Base Class Optimization", also known as the EBCO or EBO is a common compiler optimization.
- This is a real useful in case of implementation of containers as there is always a set of stateless features (e.g. hast, allocator, ...).
- With respect to multiple inheritance, it often makes sense to rearrange the order of base classes so that empty base classes appear first on the base class list. However, sometimes it is better to make some shuffle of base classes to force compiler optimizations.
- Compilers are allowed to (and some compilers do) perform aggressive EBO optimization by permuting the layout order of base classes.

Class Layout

In component design, base classes are often simply collections of typedefs, static members, enumerators, and other class members that do not occupy storage inside the class object.

Class members that don't affect layout

- static data members -> they have static lifetime or thread storage duration
- type members (included nested classes)
- non-virtual member functions

What about virtual member functions?

Only the first virtual function in a class increases its size (compiler-dependent, but on most it's like this). All subsequent methods do not because a class instance doesn't hold pointers to methods themselves, but to a virtual function table, which is one per class!
Remember also padding that can be added by compiler to achieve memory alignment.

Listing 40: Example on some class

```
class SomeClass
{
public:
    void method0(int, float, double);                           // 0
    void method1() { cout << "some output" << endl; }           // 0
    virtual void method2() { cout << "some output" << endl; }   // + 8 = sizeof(void*) on x64

private:
    enum { min = 0, max = 123};                                 // 0
    static int someStaticMember;                                // 0
    int m_data;                                                 // + 4
};

void t_someClass()
{
    printSizeInfo<SomeClass>();         // probably 16 = | 4b m_data | 4b padding | 8b pointer to vftable |
    SomeClass a;
    printSizeInfo(a);                   // probably 16
}
```

Non-Static Data Member Layout

C++ compilers did not (or very rarely) reorder member layout. Ordering should be modified to state that members with the same access lie in the same relative order. "Non-static data members of a (non-union) class with the same access control are allocated so that later members have higher addresses within a class object."

- Usually, public members are defined before protected and private.
- It is typical that storage for a base class subobject precedes storage for derived class data members.
- However, a compiler may elect to optimize storage use by permuting the base class subobject order.

Listing 41: Interleaved Layout

```
class A {
public:     // access specifier
    uint8_t a = 0b1;
    uint8_t b = 0b11;
private:    // access specifier
    uint8_t c = 0b111;
    uint8_t d = 0b1111;
public:     // access specifier
    uint8_t e = 0b11111;
private:    // access specifier
    uint8_t f = 0b111111;
};

void t_interleavedlayout()
{
    printSizeInfo<A>();     // size: 6    alignment: 1
    A a;
    printSizeInfo(a);       // size: 6

    printBits(a);
    // 001111110001111100001111000011100000011000000001
}
```

Listing 42: Ordered Layout

```
class B {
public:     // access specifier
    uint8_t a = 0b1;
    uint8_t b = 0b11;
    uint8_t e = 0b11111;
private:    // access specifier
    uint8_t c = 0b111;
    uint8_t d = 0b1111;
    uint8_t f = 0b111111;
};

void t_orderedlayout()
{
    printSizeInfo<B>();     // size: 6    alignment: 1
    B b;
    printSizeInfo(b);       // size: 6

    printBits(b);
    // 001111110000111100000111000111110000001100000001
}
```

Standard Layout

Plain Old Data layout (POD layout) is a standard-layout class that also lacks:

- private or protected non-static data members
- user-declared constructors
- base classes

Regarding the layout of standard-layout classes, C++ guarantees only that:

- The first non-static data member is at offset zero. This means that it's reasonable to reinterpret_cast a pointer to a standard-layout class object to a pointer to its first non-static data member.
- Each subsequent non-static data member has an offset greater than the offset of the non-static data member declared before it.
- The storage for objects of the class are in contiguous memory.
- Any class, even a standard-layout class, may have padding bytes after any non-static data member.

Checking Standard Layout

Calling offsetof(C, M) returns the offset in bytes of non-static data member M from the beginning of class type C. This works for standard types. For other types id is "conditionally supported feature".

static-assert

Use static-assert instead of *assert* if possible. Static assertions don't cost anything at runtime, it is evaluated at compile-time!

Listing 43: Checking offset at compile time - see Class B above

```
void t_testOffset()
{
    printOffsetInfo(&B::b);     // class size: 6     member size: 1    offset: 1

    // B has private members -> not a standard layout
    //static_assert(std::is_standard_layout_v<B>, "B does not respect the standard layout.");
    static_assert(offsetof(B,b)==1, "Error: offset of B::b is not equal to 1.");
}
```

Operators

 See how to access class members in [1] page(s) 239–246.

scope-resolution operator

To refer to function member of class outside of class body must use ::, e.g.

```
ClassName::memberName
```

This operator is used also to access static members of the class, or to refer to type member outside of class body.

member-selection operator

Use ., to access members of a class (data members, function members), e.g.

```
class ClassName { public: int x;};
ClassName c;
c.x = 1;
```

Implicit parameter

this

All member functions always have an implicit parameter referring to class object.

- Implicit parameter is accessible inside member function via this keyword.
- this is pointer to object for which member function is being invoked.
- All data members can be accessed through this pointer.
- Since data members can also be referred to directly by their names, explicit use of this often not needed and normally avoided.

References

References in C !!!

This is what you can remember from C language.

- l-value refers to memory location which identifies an object. l-value may appear as either left hand or right hand side of an assignment operator(=). l-value often represents as identifier.
- r-value refers to data value that is stored at some address in memory. A r-value is an expression that can't have a value assigned to it which means r-value can appear on right but not on left hand side of an assignment operator(=).

If C++ would follow the same concept of l-values/r-value, what about this code?

Listing 44: Test

```
void t_test()
{
    string s = "test";
    s + '1' = s;
    s + '2' = s + '3';
}
```

l-value

- can appears on the left side of assignment
- has a name
- has an address

Listing 45: lvalue examples

```
int x;
x = 123;

const string s0 = "this is const string";
string s1 = "this is string";

int* ptr = nullptr;
```

r-value

- can be a temporary object (without name)
- can be a literal constant
- can be a function return value that is not lvalue reference
- can be a result of built-in operators that is not lvalue reference

References in C++ !!!

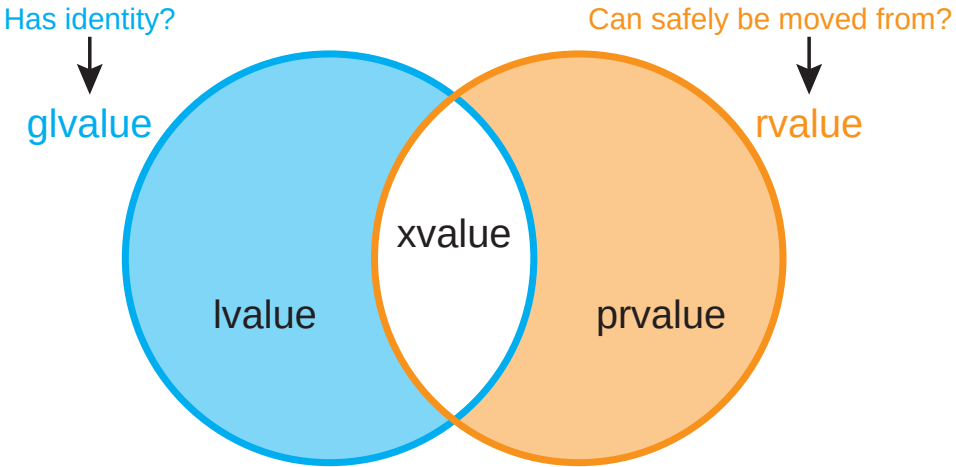
C++98 reference types are lvalue references. C++11 introduces rvalue reference type.

GOOD TO KNOW

- `int&` is an lvalue reference.
- `int&&` is an rvalue reference.
- Initializing a reference is called binding.
- Non-const lvalue reference can only bind to an lvalue.
- Const lvalue reference can bind to an lvalue, or an rvalue. This has some backward compatibility reasons.
- rvalue reference can only bind to an rvalue.

In fact, C++ is more complex in value definitions.

- xvalue (expiring value) - expression, which identifies a non-temporary object, which can be moved
- lvalue (left value) - expression, which identifies a non-temporary object of function, which cannot be moved
- prvalue (pure rvalue) - expression, which initializes an object
- glvalue (generalized lvalue) groups lvalue and xvalue. glvalue has address in memory (has identity) and thus usually can be assigned a value (if it is not const)
- rvalue groups prvalue and xvalue. rvalue can be either moved (xvalue) or does not belong to an existing object at all (prvalue). rvalue can be passed to move constructors, move assignment operators or move functions


Has identity?
glvalue
Can safely be moved from?
rvalue
xvalue
lvalue
prvalue

glvalue	rvalue	Value Category	Meaning	Example	Code
YES	NO	lvalue	Has identity and cannot be moved	variable or data member	`T x; class C {T x;}; x=T();`
				lvalue reference	`T& x=y; x=T();`
				named rvalue reference	`T&& x=T(); x=T();`
				function call or operator with T return type	`T& f(); f()=T();`
				assignment and compound assignment expr.	`(x=1)=2; (x+=1)=2;`
				pre-increment, pre-decrement operator	`(++x)=1; (--x)=1;`
				indirection operator	`(*x)=1;`
				subscript operator (except if x is array rvalue)	`x[1]=2;`
				cast to lvalue reference	`static_cast<T&>(x)=T();`
	YES	xvalue	Has identity and can be moved from	function call or operator with T return type	`T&& f(); std::move(...);`
				Subscript operator or array rvalue	`x[1];`
				cast to rvalue reference	`static_cast<T&&>(x);`
				non-static data member of type T of rvalue object	
NO		prvalue	Has no identity	literal	`13, false, nullptr, 'a'`
				function call or operator with T return type	`T f();`
				post-increment, post-decrement operator	`x++;`
				arithmetic expression	`x+y; x%y; x&y; x<<y;`
				logical expression	`x&&y; x||y; !x;`
				comparison expression	`x<y; x==y; x>=y;`
				address of expression	`&x;`
				lambda expression	`[](int a){return a*a;}`

l-value

- has a name
- all variables are l-values
- can be assigned values
- some expressions return l-value
- l-value persists beyond expressions
- functions that return by reference return l-value
- reference to l-value is call lvalue reference
- has an address
- can appears on the left side of assignment

r-value

- does not have a name
- is a temporary value without name
- can not be assigned values
- some expressions return r-value
- does not persists beyond expressions
- functions that return by value return r-value
- reference to r-value is called rvalue reference
- can be a literal constant

Named rvalue reference is lvalue (see the previous table)?

Listing 46: rvalue example

```
#define __PRINT__ cout << __PRETTY_FUNCTION__ << endl;

class Foo
{
public:
    Foo() { __PRINT__ }
    Foo(const Foo& other) { __PRINT__ }
    Foo(Foo&& other) noexcept { __PRINT__ }
    ~Foo() { __PRINT__ }
};

// Foo(const Foo& other) is called, because x is lvalue.
void t_m0(Foo&& x) { Foo y = x; }
Foo&& t_m1() { return std::move(Foo()); }

int main(int,  char**)
{
    Foo tmp;
    t_m0(std::move(tmp));
    cout << endl;
    // Finally, Foo(Foo&& other) is called
    Foo b = t_m1();
    return 0;
}
```

IF-IT-HAS-A-NAME rule

Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: if it has a name, then it is an lvalue. Otherwise, it is an rvalue.

```
Foo::Foo()
Foo::Foo(const Foo &)
Foo::~Foo()

Foo::Foo()
Foo::~Foo()
Foo::Foo(Foo &&)
Foo::~Foo()
Foo::~Foo()
```

std::move(...)

std::move(...) does not move anything somewhere!

It is declared as an rvalue reference and does not have a name. Hence, it is an rvalue. Thus, std::move "turns its argument into an rvalue even if it isn't," and it achieves that by "hiding the name."

Listing 47: std::move() example

```
class Bar : public Foo
{
public:
    //Bar(Bar&& other) noexcept : Foo(other) { } // WRONG: other is lvalue -> Foo(const Foo& other) is called
    Bar(Bar&& other) noexcept : Foo(std::move(other)) { }       // OK: Foo(Foo&& other) is called
};
```

Working with Resources

RAII - "Resource Acquisition Is Initialization."

HOWEVER, the slogan is about initialization, but its real meaning is about cleanup → freeing all resources, preventing memory leaks, and providing exception safety.

Listing 48: Unsafe code

```
void someMethod()
{
    try
    {
        int *ptr = new int[123];
        throw std::runtime_error("Some error");
        delete[] ptr;       // this will never happen
    }
    catch (const std::exception& e)
    {
        cout << "Exception: " << e.what() << endl;
    }
}
```

Stack unwinding: For every local scope between throw and try, the runtime invokes destructors of all local variables in that scope, see the cppreference for more details ———>

Listing 49: Safe code

```
struct RAII
{
    RAII(int* _ptr) : ptr(_ptr) {}
    ~RAII() { delete[] ptr; ptr=nullptr; }
    int *ptr;
};

void someMethod()
{
    try
    {
        RAII x = new int[123];
        throw std::runtime_error("Some error");
        // Stack unwinding in runtime calls ~RAII()
    }
    catch (const std::exception& e)
    {
        cout << "Exception: " << e.what() << endl;
    }
}
```

RAII - Foo class

Listing 50: Foo - variant A

```
class Foo
{
public:
    Foo() : size{0}, ptr{nullptr} {}
    void pushBack(int value)
    {
        int *tmp = new int[size+1];
        std::copy(ptr, ptr+size, tmp);
        delete[] ptr;
        ptr = tmp;
        ptr[size++] = value;
    }

private:
    int *ptr;
    size_t size;
};

void t_Foo()
{
    Foo x;
    x.pushBack(1);
}   // default destructor is called -> memory leak
```

NEW PROBLEM:

Inner data remains in memory after default destructor is called → memory leak

Listing 51: Foo - variant B

```
class Foo
{
public:
    Foo() : size{0}, ptr{nullptr} {}
    void pushBack(int value)
    {
        int *tmp = new int[size+1];
        std::copy(ptr, ptr+size, tmp);
        delete[] ptr;
        ptr = tmp;
        ptr[size++] = value;
    }
    ~Foo() { delete[] ptr; }

private:
    int *ptr;
    size_t size;
};

void t_Foo()
{
    Foo x;
    x.pushBack(1);
    {
        Foo y = x;  // default copy constructor is called -> ptr is also coppied
    }               // destructor is called on y -> ptr is deleted -> x.ptr data
                    // was deleted as well
}                   // Error in destructing, x.ptr already deleted !!!
```

NEW PROBLEM:

Double freeing

Listing 52: Foo - variant C

```
class Foo
{
public:
    Foo() : size{0}, ptr{nullptr} {}
    Foo(const Foo& other)
    {
        ptr = new int[other.size];
        size = other.size;
        std::copy(other.ptr, other.ptr + other.size, ptr);
    }
    void pushBack(int value)
    {
        int *tmp = new int[size+1];
        std::copy(ptr, ptr+size, tmp);
        delete[] ptr;
        ptr = tmp;
        ptr[size++] = value;
    }
    ~Foo() { delete[] ptr; }

private:
    int *ptr;
    size_t size;
};
```

NEW PROBLEM:

Copy constructor solves previous problem, but copy assignment remains unresolved.

```
void t_Foo()
{
    Foo x;
    x.pushBack(1);
    {
        Foo y = x;  // OK because of Copy ↩
            constructor
    }
    {
        Foo y;
        y = x;      // PROBLEM: Default copy ↩
            assignment operator is called!!!
    }
}
```

Listing 53: Foo - variant D - implementing copy assignment operator

```
// Foo& operator=(const Foo& other)
// {
//  if (this == &other) return *this;   // Checks self-assignment
//  delete[] ptr;                       // delete possible previous data
//  ptr = new int[other.size];
//  size = other.size;
//  std::copy(other.ptr, other.ptr + other.size, ptr);
//  return *this;
// }

Foo& operator=(const Foo& other)
{
    Foo tmp{other};             // Initialization, copy constructor is called. Self-copy can not occure
    std::swap(this->size, tmp.size);
    std::swap(this->ptr, tmp.ptr);
    return *this;
}
```

The first copy assignment operator is not recommended because of:

- Self-assignment test must be done! This prevent self-copy assignment leading to deletion of inner data (ptr).

- There are better ways how to write this. Sometimes it can be useful move data rather than to make a copy.

The Rule of Three

If your class directly manages some kind of resource (such as a new'ed pointer), then you almost certainly need to hand-write three special member functions:

- A destructor to free the resource.
- A copy constructor to copy the resource.
- A copy assignment operator to free the left-hand resource and copy the right-hand one

The Rule of Zero

If your class does not directly manage any resource (uses library components such as vector, string, ...) then you should strive to write no special member functions. Default them all!

- Let the compiler implicitly generate a defaulted destructor.
- Let the compiler generate the copy constructor.
- Let the compiler generate the copy assignment operator.

Explicitly defaulting your special members can help your code to be self-documenting.

 See more details on "delete" and "default" keywords in [1] page(s) 293.

 See more details on RAII in [1] page(s) 874.

RAII - Foo class - Let's Continue

Listing 54: Foo - variant E

```
Foo(const Foo& other)
{
    //This will be improved later!!!
    ptr = new int[other.size];
    size = other.size;
    std::copy(other.ptr, other.ptr + other.size, ptr);
}
Foo(Foo&& other) noexcept
{
    ptr = std::exchange(other.ptr, nullptr);
    size = std::exchange(other.size, 0);
}
Foo& operator=(const Foo& other)
{
    Foo tmp{other};
    std::swap(this->size, tmp.size);
    std::swap(this->ptr, tmp.ptr);
    return *this;
}
Foo& operator=(Foo&& other) noexcept
{
    Foo tmp{std::move(other)};
    std::swap(this->size, tmp.size);
    std::swap(this->ptr, tmp.ptr);
    return *this;
}
```

WHAT'S NEW:

Move constructor was created that can "replace" slow copy constructor in some cases.

When you write your own move constructor, write move assignment operator as well.

Remember rvalue references and std::move()

The Rule of Five

If your class directly manages some kind of resource (such as a new'ed pointer), then you may need to hand-write five special member functions for correctness and performance:

- A destructor to free the resource.
- A copy constructor to copy the resource.
- A copy assignment operator to free the left-hand resource and copy the right-hand one
- A moveconstructor to transfer ownership of the resource
- A move assignment operator to free the left-hand resource and transfer ownership of the right-hand one

But sometimes, you must write a **by-value assignment** operator to free the left-hand resource and transfer ownership of the right-hand one (aka some STL containers.).

Our Foo class will be improved by smart pointers in some upcoming lectures!!!

More on Classes

What should we already know?

In the following slides, more details on classes will be provided. Till now, you should be familiar with class layouts, operators needed to access members of classes (::, .), implicit member functions parameter (this), basic principles of references (lvalue ref., rvalue ref.) and their usage in copy/move constructors and assignment operators.

Constructors, Destructors

Constructors

- have a class name and no return type.
- are used to instantiation.
- are invoked automatically during instantiation.
- can be overloaded.
- can be created by compiler.
- can have some other features: default, deleted, parameter/parameterless, copy/move, explicit
- can be delegated.
- can be called with respect to inheritance.

Destructors

- have a class name prefixed by tilde ($\sim$).
- are parameterless
- are used to "destroy" instance.
- are automatically called when program finished execution, when a scope (the $\{\ldots\}$ parenthesis) containing local variable ends, or when delete operator is called.

Default Constructor

is a constructor with no arguments. This constructor can be generated by compiler if no other user-defined constructor exists!

```
class Foo{};                    // Empty class, compiler generates public default constructor
```

```
class Foo
{
public:
    Foo() = default;            // Explicitly defaulted constructor
};
```

```
class Foo
{
public:
    Foo(int _x = 0) {};         // User-defined default constructor
};
```

Sometimes it is useful to mark a constructor as deleted (by using delete keyword) to prevent calling the constructor, or to guide programmer to use another constructor.

```
class Foo
{
public:
    Foo() = delete;         // Explicitly deleted constructor
    Foo(int _x) {}
};
```

If possible, initialize class data members before constructor's body is called. Remember member initialization (Lecture Notes 02).

```
class Foo
{
public:
    Foo() : x{0}, s{""} {};     // Default constructor that initializes data memebrs
private:
    int x;
    std::string s;
};
```

Parametrized Constructors, Delegating Constructors

Parametrized constructor accepts one or more arguments/parameters. It is never generated by compiler and blocks generating of the default constructor by compiler.

Delegating is a process when a constructor calls another constructor of the class.

Listing 55: Delegating constructors Layout

```
class Foo
{
public:
    Foo(int _x, int _y) : x{_x}, y{_y} { __PRINT__ }

    // Delegating constructor
    Foo(int _x) : Foo(_x,0) { __PRINT__ }

    // Delegating constructor
    Foo() : Foo(0) { __PRINT__ }

    ~Foo() { __PRINT__ }
private:
    int x;
    int y;
};
```

Listing 56: ... and usage

```
void t_delegating()
{
    // Stack, automatic storage, constructor is called
    Foo a;
    // Stack, automatic storage, constructor is called
    Foo b{1,2};
}   // two destructors are called (for a and b)
```

```
Foo::Foo(int, int)
Foo::Foo(int)
Foo::Foo()
Foo::Foo(int, int)
Foo::~Foo()
Foo::~Foo()
```

Converting Constructors, Explicit Constructors

Converting Constructors

- Constructor that is not declared with explicit specifier is called converting constructor.
- If a class has a constructor which can be called with argument, then this constructor becomes conversion constructor because such a constructor allows conversion of the single argument to the class being constructed.
- converting constructor can be used for implicit conversions.

Explicit Constructors

- Explicit constructor is constructor that cannot be used for performing implicit conversions or copy initialization.
- Prefixing constructor declaration with explicit keyword makes constructor explicit.
- The explicit keyword is an optional decoration for constructors that take exactly one argument. It only applies to single-argument constructors since those are the only constructors that can be used in type casting.

Listing 57: Example on Converting constructors - see explicit in Bar class

```
class Foo
{
public:
    Foo(int x) {}
};

class Bar
{
public:
    explicit Bar(int x) {}              // explicit constructor -> implicit conversion is not allowed
};

void fnFoo(Foo x) {}
void fnBar(Bar x) {}
```

Listing 58: . . . calling constructors

```
void t_explicit()
{
    Foo a(123);                         // OK: Normal call of Foo::Foo(int)
    Foo b = 123;                        // OK: Implicit conversion calls Foo::Foo(int)
    fnFoo(123);                         // OK: Implicit call of Foo::Foo(int)
    fnFoo((Foo)123);                    // OK: Explicit call of Foo::Foo(int)

    Bar c(123);                         // OK: Normal call of Bar::Bar(int)
    //Bar d = 123;                      // Compile-time error, implicit conversion is not allowed
    //fnBar(123);                       // Compile-time error, implicit conversion is not allowed
    fnBar((Bar)123);                    // OK: Explicit call of Bar::Bar(int)
    fnBar(static_cast<Bar>(123));       // OK: Explicit call of Bar::Bar(int)
}
```

Copying vs Moving

 See more details on Copy and Move constructors in [1] page(s) 248–266.

Copying propagates the value of the source object to the destination object without modifying the source object→ "other" is const!

```
Foo(const Foo& other){}
```

Moving propagates the value of the source object to the destination object and is permitted to modify the source object → "other" is not const!

```
Foo(Foo&& other){}
```

Moving is always at least as efficient as copying, and for many types, moving is more efficient than copying.

Constructors

Listing 59: Copy and Move Constructors

```
//Copy constructor
Foo(const Foo& other)
{
    //This will be improved later using smart ↩
         pointers!!!
    ptr = new int[other.size];
    size = other.size;
    std::copy(other.ptr,
              other.ptr + other.size,
              ptr);
}

//Move constructor
Foo(Foo&& other) noexcept :
    ptr{std::exchange(other.ptr, nullptr)},
    size{std::exchange(other.size, 0)}
{}
```

Assignment Operators

Listing 60: Copy and Move Assignment Operators

```
//Copy assignment operator
Foo& operator=(const Foo& other)
{
    Foo tmp{other};
    std::swap(this->size, tmp.size);
    std::swap(this->ptr, tmp.ptr);
    return *this;
}

//Move assignment operator
Foo& operator=(Foo&& other) noexcept
{
    Foo tmp{std::move(other)};
    std::swap(this->size, tmp.size);
    std::swap(this->ptr, tmp.ptr);
    return *this;
}
```

The use of tmp variable prevents from self-assignment !!!

Remember the Rule of Zero

Listing 61: "Full" class implementation

```
class Foo
{
public:
    Foo() : s{"test"} {}
    Foo(const Foo& other) : s{other.s} {}
    Foo(Foo&& other) noexcept : s{std::exchange(other.s,↩
          "")} {}
    ~Foo() {}
    Foo& operator=(const Foo& other)
    {
        //if (this == &other) return *this;
        Foo tmp{other};
        std::swap(s, tmp.s);
        return *this;
    }
    Foo& operator=(Foo&& other) noexcept
    {
        //if (this == &other) return *this;
        Foo tmp{std::move(other)};
        std::swap(s, tmp.s);
        return *this;
    }
    static Foo getFoo() { __PRINT__ return Foo(); }
private:
    std::string s;
};
```

Listing 62: "RoZ" class implementation

```
class Bar
{
public:
    Bar() : s{"test"} {}
    Bar(const Bar& other) = default;
    Bar(Bar&& other) noexcept = default;
    ~Bar() = default;

    Bar& operator=(const Bar& other) = default;
    Bar& operator=(Bar&& other) noexcept = default;

    static Bar getBar() { return Bar(); }
private:
    std::string s;
};
```

Defaulting (using default keyword) was used to make code more "self-documented". All rows containing the "default" keyword can be omitted, but this behavior can be compiler dependent.

Listing 63: Using Bar class (the same for Foo) - see the comments

```
void t_Bar()
{
    Bar a;                              // initialization, default constructor is called
    Bar b = a;                          // initialization, copy constructor is called

    Bar c;                              // initialization, default constructor is called
    c = a;                              // assignment, copy assignment operator is called
    c = std::move(b);                   // assignment, move assignment operator is called,
                                        // b lost ownership of the data

    Bar d(a);                           // initialization, copy constructor is called
    Bar e = std::move(a);               // initialization, move constructor is called
                                        // a lost ownership of the data

    Bar f(std::move(e));                // initialization, move constructor is called
                                        // e lost ownership of the data

    Bar g = Bar::getBar();              // copy elision -> only one call of default constructor is called
    Bar h = Bar(Bar(Bar::getBar()));    // copy elision -> only one call of default constructor is called

    Bar *i= new Bar();                  // Creating instance on heap
    delete i;
}
```

Dynamic allocation is here → see new and delete keywords. Smart pointer will be used later instead of this.

New term Copy elision will be discussed on following slides!

Copy Elision

Copy Elision

The compiler is allowed to elide copies where results are "as if" copies were made.

Return Value Optimization (RVO) is one such instance.

- Caller allocates space on stack for return value, and passes the address to callee.
- Callee constructs result directly in that space.

Listing 64: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

How many parameters are passed to the foo function? → ONE !!!
The foo function is passed the address where the result should be written!

Returning Values → Unoptimized Version

stack frames

Listing 65: Example on Copy Elision

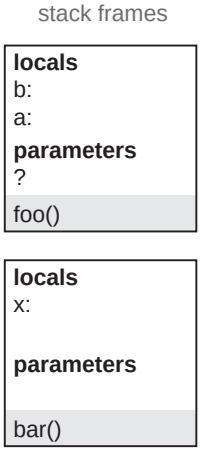
```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

- Calling bar() creates a new stact frame.
- Calling foo() creates a new stack frame.
- Address of x is passed as a parameter to foo().
- foo() is populating a and b local values.
- a is copied after return statement -> x is set!
- foo() stack frame is destroyed.

Returning Values → Unoptimized Version

Listing 65: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

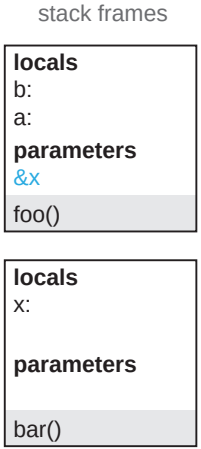


- Calling bar() creates a new stact frame.
- Calling foo() creates a new stack frame.
- Address of x is passed as a parameter to foo().
- foo() is populating a and b local values.
- a is copied after return statement -> x is set!
- foo() stack frame is destroyed.

Returning Values → Unoptimized Version

Listing 65: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

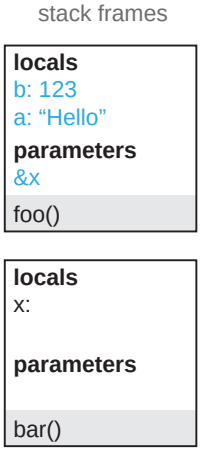


- Calling bar() creates a new stact frame.
- Calling foo() creates a new stack frame.
- Address of x is passed as a parameter to foo().
- foo() is populating a and b local values.
- a is copied after return statement -> x is set!
- foo() stack frame is destroyed.

Returning Values → Unoptimized Version

Listing 65: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

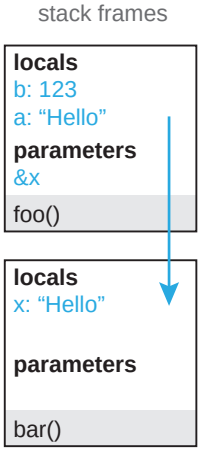


- Calling bar() creates a new stact frame.
- Calling foo() creates a new stack frame.
- Address of x is passed as a parameter to foo().
- foo() is populating a and b local values.
- a is copied after return statement -> x is set!
- foo() stack frame is destroyed.

Returning Values → Unoptimized Version

Listing 65: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```



- Calling bar() creates a new stact frame.
- Calling foo() creates a new stack frame.
- Address of x is passed as a parameter to foo().
- foo() is populating a and b local values.
- a is copied after return statement -> x is set!
- foo() stack frame is destroyed.

Returning Values → Unoptimized Version

stack frames

Listing 65: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

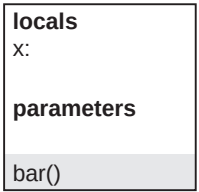
- Calling bar() creates a new stact frame.
- Calling foo() creates a new stack frame.
- Address of x is passed as a parameter to foo().
- foo() is populating a and b local values.
- a is copied after return statement -> x is set!
- foo() stack frame is destroyed.

Copy Elision - Return Value Optimization

stack frames

Listing 66: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

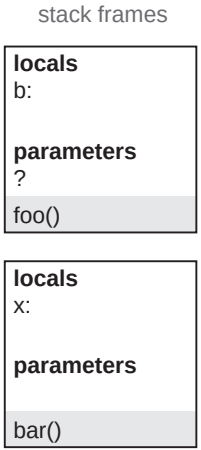


- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. But, no memory is allocated for a!
- Address of x is passed as a parameter to foo().
- foo() is populating a and b values. But the value of a is written directly in &x address.
- foo() stack frame is destroyed.

Copy Elision - Return Value Optimization

Listing 66: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

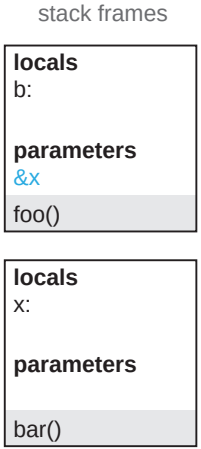


- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. But, no memory is allocated for a!
- Address of x is passed as a parameter to foo().
- foo() is populating a and b values. But the value of a is written directly in &x address.
- foo() stack frame is destroyed.

Copy Elision - Return Value Optimization

Listing 66: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```

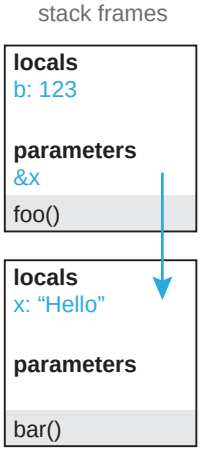


- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. But, no memory is allocated for a!
- Address of x is passed as a parameter to foo().
- foo() is populating a and b values. But the value of a is written directly in &x address.
- foo() stack frame is destroyed.

Copy Elision - Return Value Optimization

Listing 66: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```



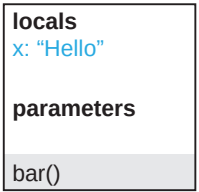
- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. But, no memory is allocated for a!
- Address of x is passed as a parameter to foo().
- foo() is populating a and b values. But the value of a is written directly in &x address.
- foo() stack frame is destroyed.

Copy Elision - Return Value Optimization

stack frames

Listing 66: Example on Copy Elision

```
using std::string;

string foo()
{
    string a{"Hello"};
    int b{123};
    return a;
}

void bar()
{
    string x{foo()};
}
```



- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. But, no memory is allocated for a!
- Address of x is passed as a parameter to foo().
- foo() is populating a and b values. But the value of a is written directly in &x address.
- foo() stack frame is destroyed.

Copy Elision - NO Return Value Optimization

Listing 67: Here RVO can not be applied.

```
using std::string;

string foo()
{
    string a{"Hello"};
    string aa{"Grrrr"};

    return (std::rand() > 123) ? a : aa;
}

void bar()
{
    string x{foo()};
}
```

In some cases, RVO can not be used because the return address can not be assigned for sure.

Pass-By-Value Copy Elision

Passing temporaries by value is another opportunity to apply copy elision. Pass-by-value implies callee can change its copy of the argument without being observed by caller.

- Caller allocates space for callee's by-value parameter on stack.
- Any lvalue arguments get copied into that space ($\rightarrow$ no elision happens).
- Any rvalue arguments are simply constructed in that space to begin with.

Listing 68: Example on Pass-by-Value Copy Elision

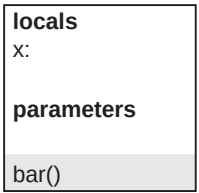
```
void foo(string a)
{
    int b{123};
    return;
}

void bar()
{
    foo(string("Hello"));
    int x;
}
```

Here, `string{"Hello"}` is rvalue.

stack frames

Listing 69: Example on Pass-by-Value Copy Elision

```
void foo(string a)
{
    int b{123};
    return;
}

void bar()
{
    foo(string("Hello"));
    int x;
}
```



- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. Memory for a temporary parameter is allocated on that stack frame! Moreover, no memory for "a" is allocated.
- foo() is populating b value.
- foo() stack frame is destroyed.

Listing 69: Example on Pass-by-Value Copy Elision

```
void foo(string a)
{
    int b{123};
    return;
}

void bar()
{
    foo(string("Hello"));
    int x;
}
```

- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. Memory for a temporary parameter is allocated on that stack frame! Moreover, no memory for “a” is allocated.
- foo() is populating b value.
- foo() stack frame is destroyed.

Listing 69: Example on Pass-by-Value Copy Elision

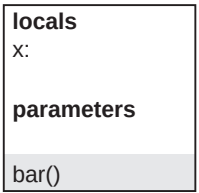
```
void foo(string a)
{
    int b{123};
    return;
}

void bar()
{
    foo(string("Hello"));
    int x;
}
```

- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. Memory for a temporary parameter is allocated on that stack frame! Moreover, no memory for “a” is allocated.
- foo() is populating b value.
- foo() stack frame is destroyed.

stack frames

Listing 69: Example on Pass-by-Value Copy Elision

```
void foo(string a)
{
    int b{123};
    return;
}

void bar()
{
    foo(string("Hello"));
    int x;
}
```



- Calling bar() creates a new stack frame.
- Calling foo() creates a new stack frame. Memory for a temporary parameter is allocated on that stack frame! Moreover, no memory for "a" is allocated.
- foo() is populating b value.
- foo() stack frame is destroyed.

Copy Elision - Back to Class Initializations

Listing 70: Example on Pass-by-Value Copy Elision

```
class Bar
{
public:
    Bar() : s{"test"} {}
    Bar(const Bar& other) = default;
    Bar(Bar&& other) noexcept = default;
    ~Bar() = default;

    Bar& operator=(const Bar& other) = default;
    Bar& operator=(Bar&& other) noexcept = default;

    static Bar getBar() { return Bar(); }
private:
    std::string s;
};

void t_test()
{
    Bar g = Bar::getBar();              // copy elision ↩
        -> only one call to default constructor is ↩
        called
    Bar h = Bar(Bar(Bar::getBar()));    // copy elision ↩
        -> only one call to default constructor is ↩
        called
}
```

Copy Elision ensures, that constructor is called only once on each row of t_test() method.

Inheritance

What should we already know?

Inheritance in C++ has similar features as in C# or Java. But there are also some differences that must be taken into account

Inheritance and all its aspects are well described in our reference presentation. That is why we will focus on key features only.

 See the basic description of inheritance in [1] page(s) 463–491.

Quiz

After reading referenced slides, you should be able to answer at least the following questions?

1. What is the syntax to define a derived class from a base class?
2. What is the difference between hiding and overriding?
3. Does interface have the same meaning as in C#?
4. Does C++ support multiple inheritance?
5. What is called class hierarchy and how can be represented?

Public Inheritance

Listing 71: Base Class Example

```
class BaseClass
{
public:
    BaseClass() : m_data{1} {}

public:
    void f() {}
protected:
    void g() {}
private:
    int m_data;
};
```

: public

Public inheritance is the most common form of inheritance in OOP.

Listing 72: Example of Derived Class - see the comments

```
class DerivedClassA : public BaseClass
{
    // f remains PUCLIC
    // g remains PROTECTED
    // m_data is not accessible
public:
    void m()
    {
        f();                    // OK
        g();                    // OK
        //m_data = 123;         // Not accessible
    }
};

void t_publicInheritance()
{
    DerivedClassA a;
    a.f();                      // OK
    //a.g();                    // Not accessible, g is PROTECTED
    a.m();
}
```

Protected Inheritance

Listing 73: Base Class Example

```
class BaseClass
{
public:
    BaseClass() : m_data{1} {}

public:
    void f() {}
protected:
    void g() {}
private:
    int m_data;
};
```

: protected

Inheritance relationship only seen by derived classes and their friends and class itself and its friends.

Listing 74: Example of Derived Class - see the comments

```
class DerivedClassB : protected BaseClass
{
    // f became PROTECTED
    // g remains PROTECTED
    // m_data is not accessible
public:
    void m()
    {
        f();                    // OK
        g();                    // OK
        //m_data = 123;         // Not accessible
    }
};

void t_protectedInheritance()
{
    DerivedClassB b;
    // b.f();                   // Not accessible, it is PROTECTED
    // b.g();                   // Not accessible, it is PROTECTED
    b.m();
}
```

Private Inheritance

Listing 75: Base Class Example

```
class BaseClass
{
public:
    BaseClass() : m_data{1} {}

public:
    void f() {}
protected:
    void g() {}
private:
    int m_data;
};
```

: private

Inheritance relationship only seen by class itself and its friends (not derived classes and their friends).

Listing 76: Example of Derived Class - see the comments

```
class DerivedClassC : private BaseClass
{
    // f became PRIVATE
    // g became PRIVATE
    // m_data is not accessible
public:
    void m()
    {
        f();                    // OK
        g();                    // OK
        //m_data = 123;         // Not accessible
    }
};

void t_privateInheritance()
{
    DerivedClassC c;
    // c.f();                   // Not accessible, it is PRIVATE
    // c.g();                   // Not accessible, it is PRIVATE
    c.m();
}
```

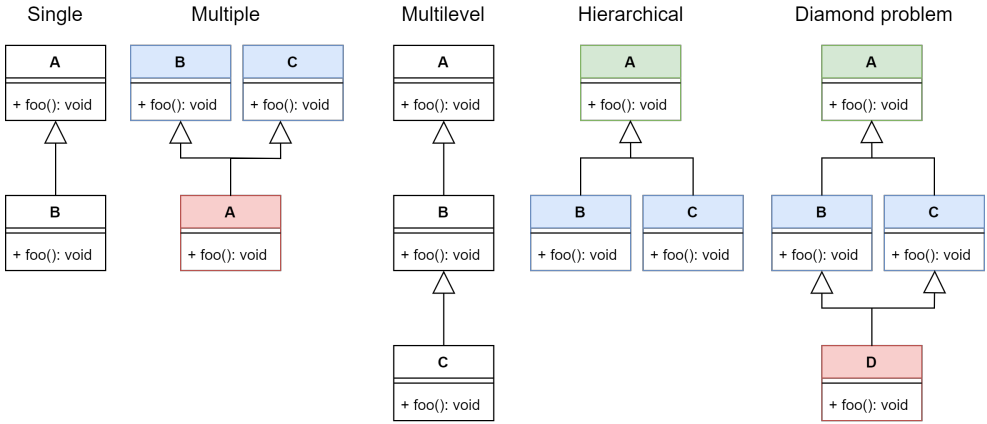
Inheritance and Constructors

- By default, constructors not inherited.
- Special constructors (i.e., default, copy, and move constructors) cannot be inherited, as well as copy/move assignment operators
- In special cases, base class constructors/operators can be reused in derived classes. This can be done by using statement. They may still be hidden in derived class.

Listing 77: Inheritance and Constructors - see the comments

```
class Base
{
public:
    Base() : x{0}, y{0} {}
    Base(int _x, int _y) : x{_x}, y{_y} {}
private:
    int x, y;
};
class Derived : public Base
{
    // default Base() constructor is not inherited !!!
    // inherit non-special constructors from Base
    using Base::Base;
    // default public constructor Derived() is implicitly declared
    using Base::operator=;
public:
    void m0();
    void m1();
};

void t_test()
{
    Derived d0;          // calls Derived::Derived()
    Derived d1{1,2};     // calls Base::Base(int,int)
}
```

Inheritance Issues



To solve problems related to inheritance, it is necessary to become acquainted with upcasting, downcasting, hiding, overloading, overriding, and virtual calling.

Upcasting and Downcasting

Casting

Generally, C++ provides C-style casting, static_cast, dynamic_cast, reinterpret_cast, and const_cast. This will be discussed later. For now, the following facts help us to understand upcasting and downcasting.

- static_cast is performed at compile-time. The compiler perform a check: "Could the input be cast to the output?" This is can be used for cases where you are casting up or down an inheritance hierarchy of pointers (or references). But the check is only at compile time, and the compiler assumes you know what you are doing.
- dynamic_cast is performed at run-time. It can only be used in the case of a pointer or reference cast, and in addition to the compile time check, it does an additional run time check that the cast is legal. It requires that the class in question have at least one virtual method, which allows the compiler (if it supports run-time type information - RTTI) to perform this additional check. However, if the type in question does not have any virtual methods, then it cannot be used! You need at least one virtual method in a class for RTTI to successfully apply dynamic_cast operator!

Upcasting

Upcasting is converting derived-class pointer or reference to base-class pointer or reference.

- Is always safe → always ends in a valid type. Derived-class instance is always a base-class instance, in other words, derived-class object has always a base-class subobject.
- No explicit type-cast is needed.

Listing 78: Base and Derived Class

```
class BaseClass
{
public:
    int x = 0;
};

class DerivedClass : public BaseClass
{
public:
    int y = 0;
};
```

Listing 79: Upcasting examples - see the comments

```
void t_upcasting()
{
    BaseClass b;
    b.x = 123;
    DerivedClass d;
    d.y = 456;                                  // d.x remains 0

    BaseClass* pb = nullptr;
    DerivedClass* pd = &d;                      // pointer to Derivedclass

    pb = &d;                                    // OK, no cast is required
    pb = pd;                                    // OK, no cast is required
    pb = static_cast<BaseClass*>(pd);           // OK, upcasting pointer

    BaseClass&  rB = d;                         // OK, getting l-value reference, no cast is required
    //BaseClass&& rrB = d;                      // ERROR, this is not working
    BaseClass&& rrB = std::move(d);             // OK, getting r-value reference, no cast is required

    rrB = BaseClass();                          // OK, getting r-value reference
    rrB = DerivedClass();                       // OK, getting r-value reference to derived class

    rrB = static_cast<BaseClass&&>(b);          // OK, casting l-value to r-value reference
    rrB = static_cast<BaseClass&&>(d);          // OK, upcasting l-value to r-value reference
}
```

Downcasting

Downcasting is converting base-class pointer or reference to derived-class pointer or reference.

- It forces base-class object to be treated as derived-class object.
- Is not always safe → not every base-class object is also derived-class object
- It requires explicit cast → static_cast or dynamic_cast, or C-style casting.

Listing 80: Base and Derived Class

```
class BaseClass
{
public:
    virtual void bcMethod() {}          // This makes Baseclass polymorphic !!!
    int x = 0;
};

class DerivedClass : public BaseClass
{
public:
    void dcMethodOnly();
    int y = 0;
};
```

Listing 81: Downcasting

```
void t_downcasting()
{
    BaseClass b;
    b.x = 123;
    DerivedClass d;
    d.y = 456;                                          // d.x remains 0

    BaseClass* pb = &b;                                 // pointer to Baseclass
    DerivedClass* pd = nullptr;

    // OK in minor cases. Generally, Baseclass may lack Derivedclass methods.
    // It means that those methods can not be called on pd->... !!!
    pd = static_cast<DerivedClass*>(pb);
    //cout << pd->y << endl;                            // What is the value?
    //pd->dcMethodOnly();                               // Uncomment code -> undefined symbol error -> SLICING

    pd = dynamic_cast<DerivedClass*>(pb);               // OK on polymorphic classes only!!!
    if (pd == nullptr)                                  // But realtime casting leads to nullptr!
        cout << "Whops, something is wrong" << endl;

    // pd = &b;                                         // ERROR
    pd = (DerivedClass*)pb;                             // OK, C-style downcast

    pb = &d;                                            // CHANGE. pointer to Derivedclass
    pd = static_cast<DerivedClass*>(pb);                // OK, but you must know what are you doing
    pd = dynamic_cast<DerivedClass*>(pb);               // OK on polymorphic classes only!!!
    if (pd == nullptr)                                  // pd != nullptr
        cout << "Whops, something is wrong" << endl;    // NO OUTPUT pd != nullptr
```

```
    // DerivedClass& rD = b;                          // NOT VALID
    // DerivedClass&& rrD = std::move(b);             // NOT VALID

    DerivedClass& rD = static_cast<DerivedClass&>(b);  // OK, but you must know what are you doing
    rD = dynamic_cast<DerivedClass&>(b);               // ERROR: runtime check fails because of BAD_CAST!
}
```

See more details on slicing in [1] page(s) 497.

See more details on bad_cast on GeeksForGeeks.

Hiding and Overloading

Hiding Members in Derived Classes

hiding is providing new versions of member functions in a derived class to "hide" original functions of the base class.

Listing 82: Using, overloading

```
//use __FUNCSIG__ on MVCS
#define __PRINT__  cout <<  __PRETTY_FUNCTION__ <<  endl;

class BaseClass
{
public:
    void foo(int i) const { __PRINT__ }
};

class DerivedClass : public BaseClass
{
public:
    void foo(int i) const { __PRINT__ }
};
```

Listing 83: test - see the comments

```
void t_test()
{
    BaseClass b;
    b.foo(1);           // BaseClass::foo(int)
    DerivedClass d;
    d.foo(1);           // DerivedClass::foo(int)
}
```

Bringing Members into "Derived" Scope

using statement can be used to bring base members into scope of derived class. Thus overloaded methods of the base class can be called.

This is typically used when overloading function members of the base class. E.g., foo() is overloaded → is can accept int or float.

Function and operator overloading is a kind of compile-time polymorphism.

Listing 84: Using, overloading

```
//use __FUNCSIG__ on MVCS
#define __PRINT__  cout <<  __PRETTY_FUNCTION__ <<  endl;

class BaseClass
{
public:
    void foo(int i) const { __PRINT__ }
};

class DerivedClass : public BaseClass
{
public:
    using BaseClass::foo;
    void foo(float i) const { __PRINT__ }
};
```

Listing 85: test - see the comments

```
void t_test()
{
    BaseClass b;
    b.foo(1);           // BaseClass::foo(int)
    DerivedClass d;
    d.foo(1);           // BaseClass::foo(int) !!!
    d.foo(1.0f);        // DerivedClass::foo(float)
}
```

Run-time polymorphism, Virtuals

Run-time polymorphism

- Polymorphism is a characteristic of being able to assign different meaning to something in different contexts, e.g. a function of a derived-class can behave differently in comparison to its base-class version.
- Is also called dynamic polymorphism, e.g. reference or pointer to type T refer to "dynamic" object of type D, where D is somehow derived from T.

  ```
  BaseClass *ptr = new DerivedClass();
  ```

  When calling member through such pointer or reference, may want actual function invoked to be determined by dynamic type of object referenced by pointer or reference, i.e. may want to call `ptr->m();`, where m() is member of Derived class, not a BaseClass.

Virtuals

- This is about virtual keyword applied on destructors, member functions.
- In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.
- Virtual function is member function with polymorphic behavior. When calling such function, actual function invoked will be determined by dynamic type of referenced object.
- Virtual function is automatically virtual in all derived classes $\rightarrow$ not necessary to repeat virtual qualifier
- Making destructor virtual ensures hierarchical calling of destructors $\rightarrow$ releasing resources.

override

override keyword can be used to "change" behavior of virtual function in derived class.

final

Specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

Listing 86: Overriding function

```
class A
{
public:
    virtual void foo() { __PRINT__ };
    virtual void bar() { __PRINT__ };
};
class B : public A
{
public:
    void foo() override { __PRINT__ };          // OK: B::foo overrides A::foo
    void bar() final { __PRINT__ };             // OK: B::bar overrides A::bar
};
class C : public B
{
public:
    void foo() override { __PRINT__ };          // OK: C::foo overrides B::foo
    //void bar() override { __PRINT__ };        // ERROR: B::bar is final and can not be overriden
};
```

Listing 87: Overriding function - test

```
void t_test()
{
    A a;
    a.foo();                                    // virtual void A::foo()
    a.bar();                                    // virtual void A::bar()

    B b;
    b.foo();                                    // virtual void B::foo()
    b.bar();                                    // virtual void B::bar()

    C c;
    c.foo();                                    // virtual void C::foo()
    c.bar();                                    // virtual void B::bar() !!!!!!!!

    A* pA = &b;
    pA->foo();                                  // virtual void B::foo() !!!!!!!!
    pA->bar();                                  // virtual void B::bar() !!!!!!!!

    pA = &c;
    pA->foo();                                  // virtual void C::foo() !!!!!!!!
    pA->bar();                                  // virtual void B::bar() !!!!!!!!
}
```

Pure virtual function

A pure virtual function is specified by placing "= 0" in its declaration as follows:
`virtual void someFunctionName()= 0;` The "= 0" tells the compiler that the function has no body.

Abstract class

Class with one or more pure virtual functions called abstract class.

- Derived classes need not override all of its pure virtual methods.
- Class that does not override all pure virtual methods of abstract base class will also be abstract.

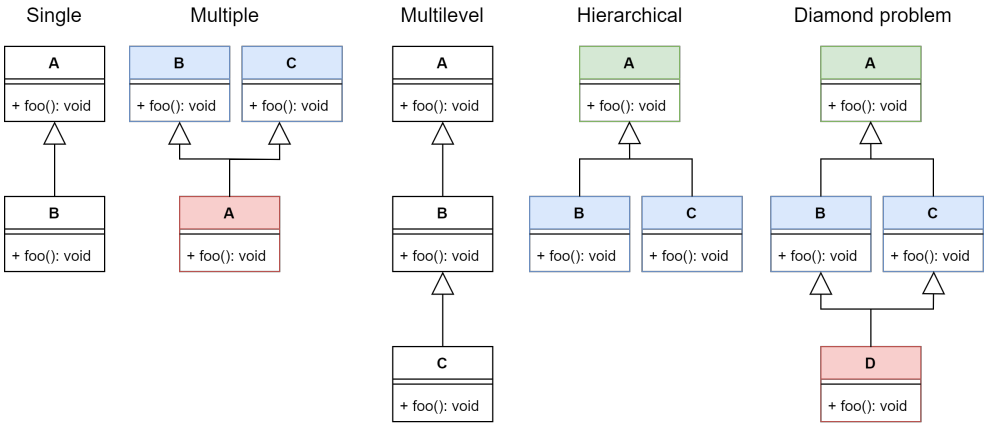
Listing 88: Pure virtual function and abstract class

```
class A
{
public:
    virtual void foo() = 0;
};

class B : public A
{
    void foo() override;            // OK: B::foo overrides A::foo
};

void t_test() { A a; }              // ERROR: Can not create instance of abstract class.
```

See more on abstract classes in [1] page(s) 518–521.

Back to Inheritance

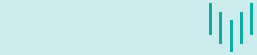
Back to Inheritance



See more on multiple inheritance and dreaded diamond problem in [1] page(s) 531–537.

Smart Pointers

Introduction



 See the basics for Smart Pointers in [1] page(s) 890–971.

In the following slides, some key features will be highlighted.

Pointer recap.

A pointer is a low-level construct that represents the address of an object, in memory. For instance a pointer to X, noted X*, represents the address of an object of type X. The value of an X* therefore looks like a memory address, like 0x06af34c2.

The pointer is itself an object, and you can manipulate it in code. In particular, you can retrieve the value of the object it points to, by dereferencing it with *, e.g. (*p)

For example, if p is a pointer of type X*, and say that p equals 0x06af34c2, then *p gives the object of type X that is stored at 0x06af34c2. And p->abc gives the member (data or function) abc in the object X.

Pros and Cons of Pointers

Pros

- Basic idea behind pointers: Since a pointer is typically much smaller than an object (a pointer contains only a memory address, which is only 32 or 64 bits tops), it is usually cheaper to copy a pointer than to copy an object.
- Useful for dynamic memory allocation: you ask the OS for a chunk of memory to store an object, and the OS would give an available memory address, which maps well with the concept of a pointer.

Cons

- They can contain an invalid address.
- To make sure you don't accidentally dereference this sort of pointer, you need to check for the nullity of pointers.
- Even if you do test for null pointers, you not completely safe, e.g. pointing to 0x00000001
- Managing the life cycle of this object is required.

Smart Pointers in C++11 and up

All the standard smart pointers arrived together in C++11.
They basically provide automatic memory management: when a smart pointer is no longer in use, that is when it goes out of scope, the memory it points to is deallocated automatically.

- auto_ptr C++98. Deprecated since in C++11, and finally removed in C++17.
- unique_ptr C++11 replacement for auto_ptr, C++14 adds make_unique.
- shared_ptr adds reference-counting, C++17 adds shared_ptr<T[]>.
- weak_ptr for "weak" references that do not increment reference counters.

A smart pointer syntactically behaves like a pointer in many way: It can be dereferenced with operator* or operator->, that is to say you can call *sp or sp->member on it. And it is also convertible to bool, so that it can be used in an if statement.
Traditional pointers are now also known as raw pointers. Raw pointers (*) play still an important role in C++.

What is “smart”? Do you remeber RAII?

The principle of RAII is simple: wrap a resource (a pointer for instance) into an object, and dispose of the resource in its destructor. And this is exactly what smart pointers do:

Listing 89: SmartPointer class

```
template <typename T>
class SmartPointer
{
public:
    explicit SmartPointer(T* _ptr) : ptr{_ptr} {}
    ~SmartPointer() { delete ptr; }                 // BUT IS THIS REALY SMART? WHAT ABOUT COPY AND TWICE DELETION!!!

    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
private:
    T* ptr;
};
```

Smart pointers must be safe from the perspective of copying, moving, and “sharing” ownership od data (resources).

Unlike raw pointer, smart pointer owns its pointed-to memory.

RAW Pointers

- They are not “smart” pointers, they are not “dumb” pointers either. There is still a place to use them.
- Share a lot with references but can be null (nullptr).
- raw pointers (and references) represent access to an object, but not ownership!

Listing 90: RAW pointers

```
void t_rawPointer()
{
    int x = 123;
    int *ptrA = &x;
    *ptrA = 456;

    const int* ptrB = &x;
    //*ptrB = 456;                // Error. Target object is const

    int* const ptrC = &x;
    *ptrC = 456;                  // OK, x=456
    //ptrC++;                     // Error. Pointer is const

    const int* const ptrD = &x;
    //*ptrD = 456;                // Error. Target object is const
    //ptrD++;                     // Error. Pointer is const
}
```

RAW Pointers: “Pointing to Managed Object”

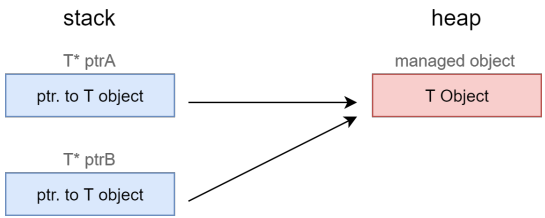
Listing 91: RAW pointers

```
class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} {}
    int x;
};

void t_test()
{
    SomeClass *ptrA = new SomeClass(123);
    SomeClass *ptrB = ptrA;
    delete ptrA;
    ptrA = nullptr;
    cout << ptrB->x << endl;        // Managed object ↩
          does not exist
    delete ptrB;                    // Managed object ↩
          does not exist
    ptrB = nullptr;
}
```

RAW Pointers: “Pointing to Managed Object”

Listing 91: RAW pointers

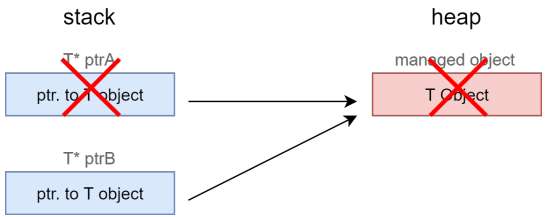
```
class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} {}
    int x;
};

void t_test()
{
    SomeClass *ptrA = new SomeClass(123);
    SomeClass *ptrB = ptrA;
    delete ptrA;
    ptrA = nullptr;
    cout << ptrB->x << endl;        // Managed object ↩
          does not exist
    delete ptrB;                    // Managed object ↩
          does not exist
    ptrB = nullptr;
}
```



RAW Pointers: “Pointing to Managed Object”

Listing 91: RAW pointers

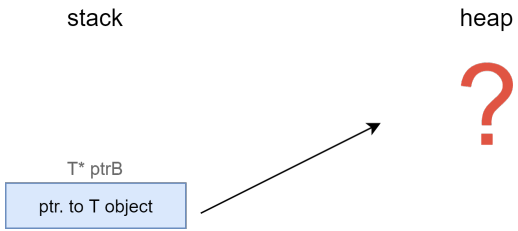
```
class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} {}
    int x;
};

void t_test()
{
    SomeClass *ptrA = new SomeClass(123);
    SomeClass *ptrB = ptrA;
    delete ptrA;
    ptrA = nullptr;
    cout << ptrB->x << endl;         // Managed object ↩
          does not exist
    delete ptrB;                     // Managed object ↩
          does not exist
    ptrB = nullptr;
}
```



RAW Pointers: “Pointing to Managed Object”

Listing 91: RAW pointers

```
class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} {}
    int x;
};

void t_test()
{
    SomeClass *ptrA = new SomeClass(123);
    SomeClass *ptrB = ptrA;
    delete ptrA;
    ptrA = nullptr;
    cout << ptrB->x << endl;         // Managed object ↩
          does not exist
    delete ptrB;                     // Managed object ↩
          does not exist
    ptrB = nullptr;
}
```



Unique Pointer

std::unique_ptr is smart pointer that retains exclusive ownership of object through pointer.

- is movable → move operation transfers ownership → two unique_ptr can not own the same object
- is not copyable → this prevent from creating another unique_ptr that could share object

- make_unique is used to create unique_ptr object because of exception safety reasons.
- Destructor destroys managed object (if any)
- operator= assigns unique_ptr

Unique Pointer: Key Features

DEFERENCING/SUBSCRIPTING

- operator* dereferences pointer to managed object
- operator-> dereferences pointer to managed object
- operator[] provides indexed access to managed array

MODIFIERS

- release returns pointer to managed object and releases ownership.
- reset replaces managed object
- swap swaps managed objects

OBSERVERS

- get returns pointer to managed object
- get_deleter returns deleter used for destruction of managed object
- operator bool checks if there is associated managed object

Unique Pointers: “Pointing to Managed Object”

Listing 92: Test Class

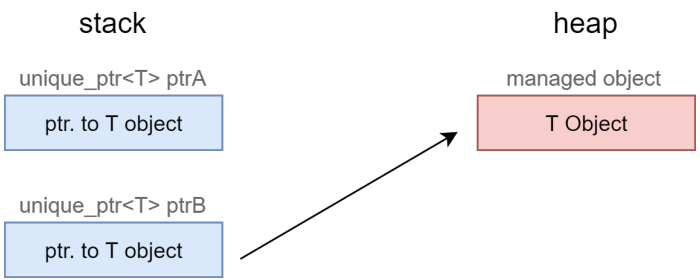
```
#define __PRINT__ cout << __PRETTY_FUNCTION__ << endl;

class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} { __PRINT__ }
    ~SomeClass() {  __PRINT__ }
    int x;
};
std::ostream& operator<<(std::ostream& os, const SomeClass& vec)    // This provides output to stream
{
    os << vec.x;
    return os;
}
```

Listing 93: Unique pointers

```
void t_test()
{
    std::unique_ptr<SomeClass> ptrA = std::make_unique<SomeClass>(123);     // SomeClass::SomeClass(int)
    //std::unique_ptr<SomeClass> ptrB = ptrA;                               // Copy constructor is deleted !!!
    std::unique_ptr<SomeClass> ptrB = std::move(ptrA);                      // Move is required
    //cout << ptrA->x << endl;                                              // Error: Access violation reading ↩
         location
    cout << ptrB->x << endl;    //OK
}                                                                           //SomeClass::~SomeClass()
```

Listing 93: Unique pointers

```
void t_test()
{
    std::unique_ptr<SomeClass> ptrA = std::make_unique<SomeClass>(123);     // SomeClass::SomeClass(int)
    //std::unique_ptr<SomeClass> ptrB = ptrA;                               // Copy constructor is deleted !!!
    std::unique_ptr<SomeClass> ptrB = std::move(ptrA);                      // Move is required
    //cout << ptrA->x << endl;                                              // Error: Access violation reading ↩
          location
    cout << ptrB->x << endl;    //OK
}                                                                           //SomeClass::~SomeClass()
```



Unique Pointers: Passing to Functions

Listing 94: Functions to be called

```
void PassingByValue(std::unique_ptr<SomeClass> p)                // RUNTIME ERROR: COPY IS NOT ALLOWED !!!
{
    cout << __FUNCTION__ << "\t" <<  p << "\t" << *p  << endl;
    (*p).x = 222;
}

void PassingInnerData(SomeClass* p)                              // OK, BUT VERY DANGEROUS !!!
{
    cout << __FUNCTION__ << "\t" <<  p << "\t" << *p  << endl;
    (*p).x=333;
}

void PassingByReference(std::unique_ptr<SomeClass>& p)
{
    cout << __FUNCTION__ << "\t" <<  p << "\t" << *p  << endl;
    (*p).x = 444;
}

void PassingByRValue(std::unique_ptr<SomeClass>&& p)
{
    cout << __FUNCTION__ << "\t" <<  p << "\t" << *p  << endl;
    (*p).x = 555;
}
```

Listing 95: Calling previous functions

```
void t_passingUniquePointers()
{
    auto p0 = std::make_unique<SomeClass>(111);
    cout << p0 << "\t" << *p0 << endl;                          // e.g. 000002260B5F4AB0    111

    //auto p2 = p1;                                             // COPY IS NOT ALLOWED
    auto p = std::move(p0);                                     // MOVE IS OK
    cout << p << "\t" << *p << endl;                            // 000002260B5F4AB0 111

    //PassingByValue(p2)                                        // COPY IS NOT ALLOWED  -> COMPILATION ERROR

    //cout << "BEFORE:\t" << p << "\t" << *p << endl;           // Uncomment these three lines -> error
    //PassingByValue(std::move(p));                             // p BECOMES A HOLLOW OBJECT -> THIS IS CALLED "SINK"
    //cout << "AFTER:\t" << p << "\t" << *p << endl;            // ERROR:   Access violation reading location

    cout << "BEFORE:\t" << p << "\t" << *p << endl;             // BEFORE:  000002260B5F4AB0    111
    PassingInnerData(&(*p));                                    // PassingInnerData 000002260B5F4AB0    111
    cout << "AFTER:\t" << p << "\t" << *p << endl;              // AFTER:   000002260B5F4AB0    333

    cout << "BEFORE:\t" << p << "\t" << *p << endl;             // BEFORE:  000002260B5F4AB0    333
    PassingByReference(p);                                      // PassingByReference   000002260B5F4AB0    333
    cout << "AFTER:\t" << p << "\t" << *p << endl;              // AFTER:   000002260B5F4AB0    444

    cout << "BEFORE:\t" << p << "\t" << *p << endl;             // BEFORE:  000002260B5F4AB0    444
    PassingByRValue(std::move(p));                              // PassingByRValue  000002260B5F4AB0    444
    cout << "AFTER:\t" << p << "\t" << *p << endl;              // AFTER:   000002260B5F4AB0    555
}
```

Unique Pointer: [], and Deleters

unique_ptr has a specialization for array types, i.e. `std::unique_ptr<T[]>`.

unique_ptr is always a template of two parameters. If you provide no second parameter, it is defaulted to std::default_delete<T>, otherwise `std::unique_ptr<T, Deleter>` is used.

- Deleter implementation has zero memory cost in case of default deleter, or deleter of functor/closure type with no state.
- If no memory cost for deleter state, unique_ptr has same memory cost as raw pointer.

Listing 96: Example on deleter

```
struct FileCloser
{
    void operator()(FILE *fp) const
    {
        assert(fp != nullptr);
        fclose(fp);
    }
};

void t_deleter()
{
    FILE *fp = fopen("input.txt", "r");
    std::unique_ptr<FILE, FileCloser> uptr(fp);
}
```

Shared Pointer

std::shared_ptr is smart pointer that retains shared ownership of object through pointer → reference-counting.

- is movable → move operation transfers ownership
- is copyable → this creates additional owner
- It contains two pointer → pointer to managed object and pointer to control block.
- Access to underlying control block is thread safe. No guarantee made for accesses to owned object.

- Use make_shared to create shared_ptr object.
- Destructor destroyes managed object (if any)
- operator= assigns shared_ptr

Shared Pointer: Key Features

DEFERENCING/SUBSCRIPTING

- operator* dereferences pointer to managed object
- operator-> dereferences pointer to managed object
- operator[] provides indexed access to managed array

MODIFIERS

- reset replaces managed object
- swap swaps values of two shared_ptr objects

OBSERVERS

- get returns pointer to managed object
- use_count returns number of shared_ptr objects referring to same managed object
- operator bool checks if there is associated managed object
- owner_before provides owner-based ordering of shared pointers

Control Block

IT CONTAINS:

- pointer to managed object
- use count = the number of shared_ptr instances pointing to object
- weak count = the number of weak_ptr instances pointing to object, plus one if use count is nonzero
- other data, e.g. deleter

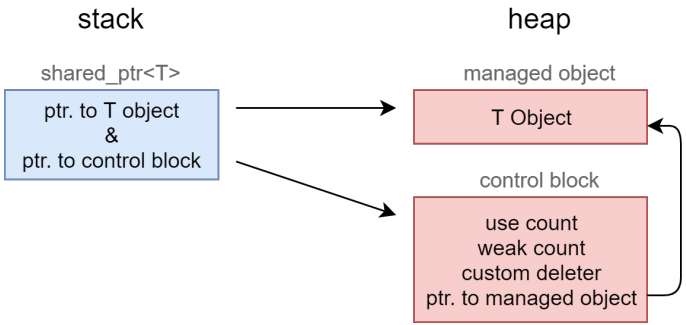
Shared Pointers: “Pointing to Managed Object”

Listing 97: Test Class

```
#define __PRINT__ cout << __PRETTY_FUNCTION__ << endl;

class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} { __PRINT__ }
    ~SomeClass() {  __PRINT__ }
    int x;
};
std::ostream& operator<<(std::ostream& os, const SomeClass& vec)    // This provides output to stream
{
    os << vec.x;
    return os;
}
```

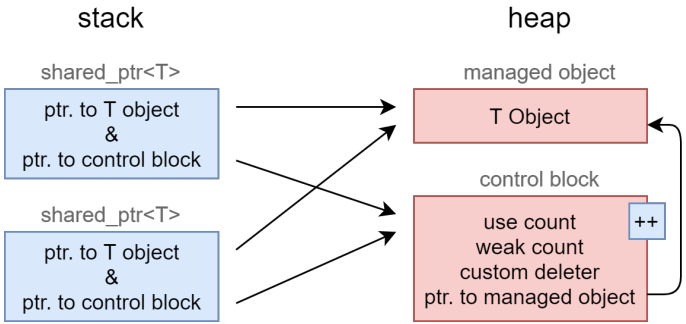
Listing 98: Shared pointers

```
void t_test()
{
    std::shared_ptr<SomeClass> ptrA                 // SomeClass::SomeClass(int)
        = std::make_shared<SomeClass>(123);
    {
        std::shared_ptr<SomeClass> ptrB = ptrA;     // reference-counting ++
        cout << *ptrA << endl;                      // 123
        cout << *ptrB << endl;                      // 123
    }
    cout << *ptrA << endl;
}                                                   // SomeClass::~SomeClass()
```



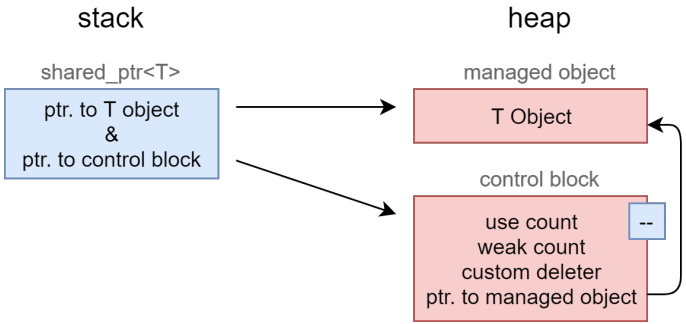
Listing 98: Shared pointers

```
void t_test()
{
    std::shared_ptr<SomeClass> ptrA                 // SomeClass::SomeClass(int)
        = std::make_shared<SomeClass>(123);
    {
        std::shared_ptr<SomeClass> ptrB = ptrA;     // reference-counting ++
        cout << *ptrA << endl;                      // 123
        cout << *ptrB << endl;                      // 123
    }
    cout << *ptrA << endl;
}                                                   // SomeClass::~SomeClass()
```



Listing 98: Shared pointers

```
void t_test()
{
    std::shared_ptr<SomeClass> ptrA                 // SomeClass::SomeClass(int)
        = std::make_shared<SomeClass>(123);
    {
        std::shared_ptr<SomeClass> ptrB = ptrA;     // reference-counting ++
        cout << *ptrA << endl;                      // 123
        cout << *ptrB << endl;                      // 123
    }
    cout << *ptrA << endl;
}                                                   // SomeClass::~SomeClass()
```



Shared Pointers: Passing to Functions

Listing 99: Functions to be called

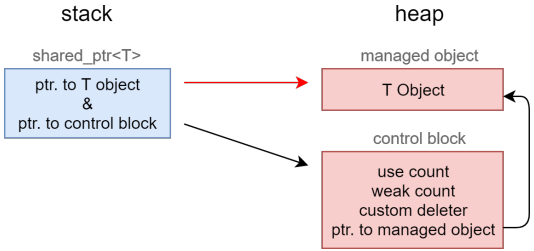
```
void PassingByValue(std::shared_ptr<SomeClass> p)                //OK, COPY IS ALLOWED
{
    cout <<  "\t" <<  p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;
    (*p).x = 222;
}

void PassingInnerData(SomeClass* p)                              // OK, BUT VERY DANGEROUS !!!
{
    cout <<  "\t" <<  p << "\t" << *p << "\t" << endl;
    (*p).x = 333;
}

void PassingByReference(std::shared_ptr<SomeClass>& p)           //OK
{
    cout <<  "\t" <<  p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;
    (*p).x = 444;
}

void PassingByRValue(std::shared_ptr<SomeClass>&& p)             //OK
{
    cout <<  "\t" <<  p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;
    (*p).x = 555;
}
```

Listing 100: Calling previous functions

```
void t_passingSharedPointers()
{
    auto p0 = std::make_shared<SomeClass>(111);                                  // SomeClass::SomeClass(int)
    cout << p0 << "\t" << *p0 << "\t(" << p0.use_count() << ")" << endl;         // 000001D60AF438B0 111 (1)

    auto p = p0;                                                                 // COPY IS OK
    //auto p = std::move(p0);                                                    // MOVE IS OK -> COUNT WILL REMAIN 1
    cout << p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;            // 000001D60AF438B0 111 (2)

    PassingByValue(p);                                                           // 000001D60AF438B0 111 (3)
    //PassingByValue(std::move(p));                                              // p BECOMES A HOLLOW OBJECT -> THIS IS ↩
        CALLED "SINK"
    cout << p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;            // 000001D60AF438B0 222 (2)

    PassingInnerData(&(*p));                                                     // 000001D60AF438B0 222
    cout << p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;            // 000001D60AF438B0 333 (2)

    PassingByReference(p);                                                       // 000001D60AF438B0 333 (2)
    cout << p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;            // 000001D60AF438B0 444 (2)

    PassingByRValue(std::move(p));                                               // 000001D60AF438B0 444 (2)
    cout << p << "\t" << *p << "\t(" << p.use_count() << ")" << endl;            // 000001D60AF438B0 555 (2)
}
```

Pointer in Control Block?

- shared_ptr, unlike unique_ptr, places a layer of indirection between the physical heap-allocated object and the notion of ownership.
- shared_ptr instances primary participate in ref-counted ownership of the control block.
- The control block itself is sole arbiter of what it means to "delete the controlled object."
- ... thus the "inner" pointer (see the red arrow) can refer to certain parts/members of T object, e.g. i-th element of inner array.



Listing 101: Aliasing constructor

```
using vec = std::vector<int>;
auto x = {1,2,3,4,5,6};
auto pvec = std::make_shared<vec>(x);
std::shared_ptr<int> p(pvec, &(*pvec)[3]);
cout << (*pvec)[0] << endl;                    // 1
cout << *p << endl;                            // 4
```

- make_shared and make_unique wrap raw new
- ~shared_ptr and ~unique_ptr wrap raw delete
- unique_ptr<T> is implicitly convertible to shared_ptr<T>

```
std::shared_ptr<T> sptr = std::make_unique<T>();
```

Weak Pointer

std::weak_ptr is smart pointer holding non-owning reference to object managed by shared_ptr.

- can not be deferenced → (*ptr) is forbidden
- It must be converted to shared_ptr in order to access referenced object.
- It has the same physical layout as shared_ptr.
- is movable and copyable

Reference counting nature of shared_ptr causes it to leak memory in case of circular references → such cycles should be broken with weak_ptr.

Think of a weak_ptr as a "ticket for a shared_ptr." The "redeem a ticket" operation can be spelled in two ways: by explicit type-conversion, or by calling wptr.lock().

```
if (auto sharedPtr = weakPtr.lock()){ sharedPtr-> ... ; }
```

Weak Pointer: Key Features

- Use weak_ptr to create weak pointer.
- Destructor destroys weak pointer.
- operator= assigns weak_ptr

MODIFIERS

- reset releases reference to managed object
- swap swaps values of two weak_ptr objects

OBSERVERS

- use_count returns number of shared_ptr objects referring to same managed object
- expired checks if referenced object was already deleted
- lock creates shared_ptr that manages referenced object
- owner_before provides owner-based ordering of weak pointers

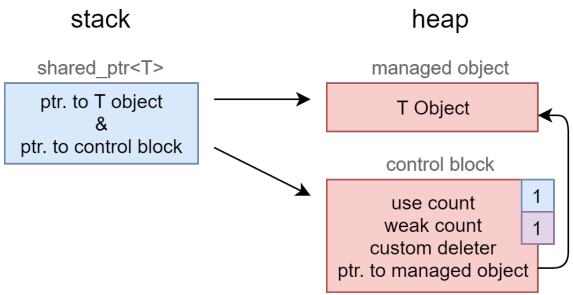
Weak Pointers: "Pointing to Managed Object"

Listing 102: Test Class

```
#define __PRINT__ cout << __PRETTY_FUNCTION__ << endl;

class SomeClass
{
public:
    explicit SomeClass(int _x) : x{_x} { __PRINT__ }
    ~SomeClass() {  __PRINT__ }
    int x;
};
std::ostream& operator<<(std::ostream& os, const SomeClass& vec)    // This provides output to stream
{
    os << vec.x;
    return os;
}
```

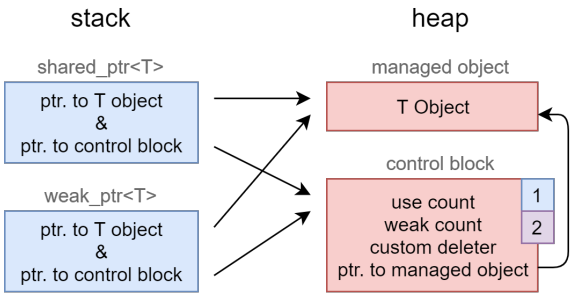
Listing 103: Weak pointers

```
void t_test()
{
    std::weak_ptr<SomeClass> ptrB;                      // Not initialized
    {
        auto ptrA = std::make_shared<SomeClass>(123);   // SomeClass::SomeClass(int)
        ptrB = ptrA;
        cout << ptrB.use_count() << endl;               // 1
        //cout << *ptrB << endl;                        // NOT ALLOWED
    }                                                   // SomeClass::~SomeClass() !!!
    auto tmp = ptrB.lock();                             // tmp is shared_ptr
    cout << *tmp << endl;                               // ERROR! Managed object was destroyed
}
```



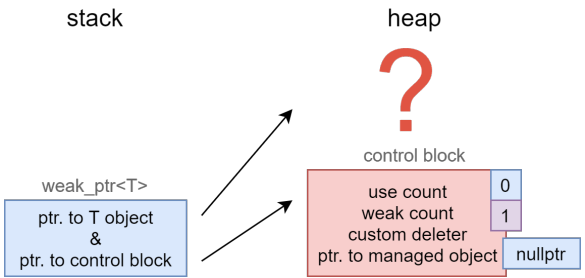
Listing 103: Weak pointers

```
void t_test()
{
    std::weak_ptr<SomeClass> ptrB;                          // Not initialized
    {
        auto ptrA = std::make_shared<SomeClass>(123);       // SomeClass::SomeClass(int)
        ptrB = ptrA;
        cout << ptrB.use_count() << endl;                   // 1
        //cout << *ptrB << endl;                            // NOT ALLOWED
    }                                                       // SomeClass::~SomeClass() !!!
    auto tmp = ptrB.lock();                                 // tmp is shared_ptr
    cout << *tmp << endl;                                   // ERROR! Managed object was destroyed
}
```



Listing 103: Weak pointers

```
void t_test()
{
    std::weak_ptr<SomeClass> ptrB;                          // Not initialized
    {
        auto ptrA = std::make_shared<SomeClass>(123);       // SomeClass::SomeClass(int)
        ptrB = ptrA;
        cout << ptrB.use_count() << endl;                   // 1
        //cout << *ptrB << endl;                            // NOT ALLOWED
    }                                                       // SomeClass::~SomeClass() !!!
    auto tmp = ptrB.lock();                                 // tmp is shared_ptr
    cout << *tmp << endl;                                   // ERROR! Managed object was destroyed
}
```



Smart Pointers: Circular References

Using smart pointers bring some challenges that must be solved in real applications. One of them is related to circular references. Using shared pointers can be dangerous when a complex data structure (e.g. trees, graphs) contain circuits.

 We refer to [1] page(s) 937–954 for more details.

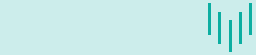
This problem will be discussed in more detail on our practical lessons.

Smart Pointers: Good to Know

1. Use unique_ptr or shared_ptr to represent ownership.
2. Prefer unique_ptr over shared_ptr unless you need to share ownership.
3. Use make_unique() to make unique_ptrs .
4. Use make_shared() to make shared_ptrs .
5. Use std::weak_ptr to break cycles of shared_ptrs .
6. Take smart pointers as parameters only to explicitly express lifetime semantics.
7. If you have non-std smart pointers, follow the basic pattern from std.
8. Take a unique_ptr<T> parameter to express that a function assumes ownership of a T.
9. Take a unique_ptr<T>& parameter to express that a function reseats the T.
10. Take a shared_ptr<T> parameter to express that a function is part owner of T.
11. Take a shared_ptr<T>& parameter to express that a function might reseat the shared pointer.
12. Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object.
13. Do not pass a pointer or reference obtained from an aliased smart pointer.

STL Overview

Introduction to STL



See the basics for Standard Template Library in [1] page(s) 540–641.

In the following slides, only key features will be highlighted.

Standard Template Library (STL)

- "Developed" by Alexander Stepenov & Meng Lee at Hewlett Packard.
- Provides number of functionalities grouped in std namespace.
- Based on templates $\rightarrow$ almost every stl component is a template.

STL Components, sublibraries

- language support library (e.g., exceptions, memory management)
- diagnostics library (e.g., assertions, exceptions, error codes)
- general utilities library (e.g., functors, date/time)
- strings library (e.g., C++ and C-style strings)
- localization library (e.g., date/time formatting and parsing, character classification)
- containers library (e.g., sequence containers and associative containers)
- iterators library (e.g., stream iterators)
- algorithms library (e.g., searching, sorting, merging, set operations, heap operations, minimum/maximum)
- numerics library (e.g., complex numbers, math functions)
- input/output (I/O) library (e.g., streams)
- regular expressions library (e.g., regular expression matching)
- atomic operations library (e.g., atomic types, fences)
- thread support library (e.g., threads, mutexes, condition variables, futures)

STL: What use it?

PROS:

- quicker development → no need to develop commonly used classes
- reliable → C++ standard news are taken into account (memory safety, type safety, ...)
- portable
- efficient & fast (with respect to others features)
- accurate
- smaller and readable code → lower maintenance of the code

CONS:

- performance (was also a pros?) → sometimes things can be done in better way with respect to performance, but usually with lower safety, portability or versatility.

STL Core Components

Containers and adopters represent data and provides variety of methods for accessing data in different time complexity.

- Sequence containers
- Ordered associative containers
- Unordered associative containers (hashed)
- Container adopters

Algorithms represent operations on that data.
Iterators serve as a middle layer between containers and algorithms.

STL Containers

Sequential Containers	Header File
array	<array>
vector	<vector>
deque	<deque>
forward_list	<forward_list>
list	<list>
Associative Containers	**Header File**
set, multiset	<set>
map, multimap	<map>
Unordered Containers	**Header File**
unordered_set, unordered_multiset	<unordered_set>
unordered_map, unordered_multimap	<unordered_map>
Container Adopters	**Header File**
stack	<stack>
queue, priority_queue	<queue>

COMMON FEATURES:

- deafualt constructor
- uniform initialization
- copy constructor
- iterator constructor
- size()
- clear()
- begin(), end()
- default allocator

STL Containers: array

- is a wrapper around normal C-style arrays.
- supports iterators.
- knows about its size.
- provides random access.
- can be used with C functions.
- can not grow, size is fixed at compile time

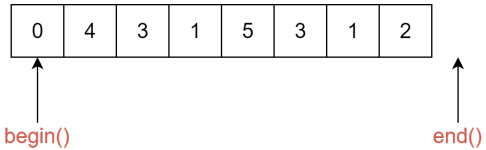
Listing 104: std::array

```
void t_array()
{
    std::array<int, 5> a{0,4,3,1,2};
    for (size_t i=0; i<a.size(); ++i)
    {
        cout << a[i] << " ";
    }
    cout << endl;

    std::array<int, 5> b = {0,4,3,1,2};
    //std::array<int> c{0,4,3,1,2};     // Not allowed
    std::array d{0,4,3,1,2};        // OK since C++17

    //Since C++20
    // auto e { std::to_array<int, 5>({0,4,3,1,2}) };
    // auto f { std::to_array<int>({0,4,3,1,2}) };
    // auto g { std::to_array({0,4,3,1,2}) };
}
```

STL Iterator

- Iterator is an object that allows iteration over collection of elements, where elements are often (but not necessarily) in container.
- Iterators support many of same operations as pointers, e.g. *, ->, ==, !=, ++, - -, . . . , in depending on type.
- There are five different types of iterators: 1) input, 2) output, 3) forward, 4) bidirectional, and 5) random access
- Has one of three possibilities of access order:
  1. forward (i.e., one direction only)
  2. forward and backward
  3. any order (i.e., random access)
- Has one of three possibilities in terms of read/write access:
  1. can only read referenced element (once or multiple times)
  2. can only write referenced element (once or multiple times)
  3. can read and write referenced element (once or multiple times)

An iterator is created through begin() and end() functions called on a container.



Listing 105: Iterator

```
void t_print()
{
    std::array<int, 8> a{0,4,3,1,5,3,1,2};
    for(const auto x : a)
    {
        cout << x << " ";
    }
    cout << endl;

    for (auto it = a.begin(); it != a.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //#include <algorithm>
    std::for_each(a.begin(), a.end(),
        [](const int& x) { std::cout << x << " " ; });
    cout << endl;
}
```

STL Iterator: Good to Know

- Do not dereference iterator unless it is known to validly reference some object.
- Some operations on container can invalidate some or all iterators.
- Incrementing iterator past end of container or decrementing iterator before beginning of container results in undefined behavior
- Input and output iterators can only be dereferenced once at each position.

STL Containers: vector

- represents a dynamic array.
- supports iterators.
- is efficient for adding/removal at the end.
- provides random access.
- grow automatically, init capacity can be set

Listing 106: std::vector - three most used initializations

```
void t_vector()
{
    std::vector<int> v0{0,4,3,1,2};
    cout << "size = " << v0.size() << endl;          // 5
    cout << "capacity = " << v0.capacity() << endl; // 5

    //std::vector<int> v1 = std::vector(10);         // until C++14

    std::vector<int> v1(5, 0);                       // {0,0,0,0,0}
    cout << "size = " << v1.size() << endl;          // 5
    cout << "capacity = " << v1.capacity() << endl; // 5

    std::vector<int> v2;                             // {}
    v2.reserve(5);                                   // { , , , , ,}
    cout << "size = " << v2.size() << endl;          // 0
    cout << "capacity = " << v2.capacity() << endl; // 5
}
```

Take a look at a nice overview on geeksforgeeks.

vector: push_back, emplace_back

This is about adding items to a vector instance. Try to avoid insert methods, especially inserting elements at the beginning of the vector. Adding at the end is O(1).

push_back
Adds a new element at the end of the vector, after its current last element. The content of value is copied (or moved) to the new element. Container size is increased by one if needed → automatic reallocation.

emplace_back
Inserts a new element at the end of the vector, right after its current last element. This new element is constructed in place using args as the arguments for its constructor. Container size is increased by one if needed → automatic reallocation.

If possible, reserve a storage (memory) for vector elements!!! See the benchmark on the next slide.

vector: Preallocating storage

Without "reserve"

Listing 107: std::vector - push or emplace leads to reallocation

```
void t_0()
{
    size_t length = 1 << 27;        //2^27
    std::vector<size_t> v;


    for (size_t i=0; i<length; ++i)
    {
        //v.push_back(i);
        v.emplace_back(i);
    }
}
```

push_back → 0.941 s
emplace_back → 0.899 s

With "reserve"

Listing 108: std::vector - push or emplace with preallocated memory

```
void t_1()
{
    size_t length = 1 << 27;        //2^27
    std::vector<size_t> v;
    v.reserve(length);

    for (size_t i=0; i<length; ++i)
    {
        //v.push_back(i);
        v.emplace_back(i);
    }
}
```

push_back → 0.381 s
emplace_back → 0.357 s

HW used for benchmark: Notebook, CPU: i7-8565U, 16GB RAM, OS Win10 x64

vector: remove, erase, shrink_to_fit

This is about removing items from in-between a vector instance. When an item disappears from somewhere in the middle between other items, then all items right from it must move one slot to the left → runtime cost within O(n).

remove

It works by overwriting "bad" elements with "good" succeeding elements, and return new END of sequence. Beware of pointers! It preserves size and capacity!

erase

Invalidates iterators and references at or after the point of the erase, including the end() iterator. It does not change capacity.

shrink_to_fit

This can be called to reduce allocated capacity to fit size of the container. This make a new allocation.

vector: Removing elements, preserving elements ordering

Listing 109: std::vector - removing elements → preserving ordering

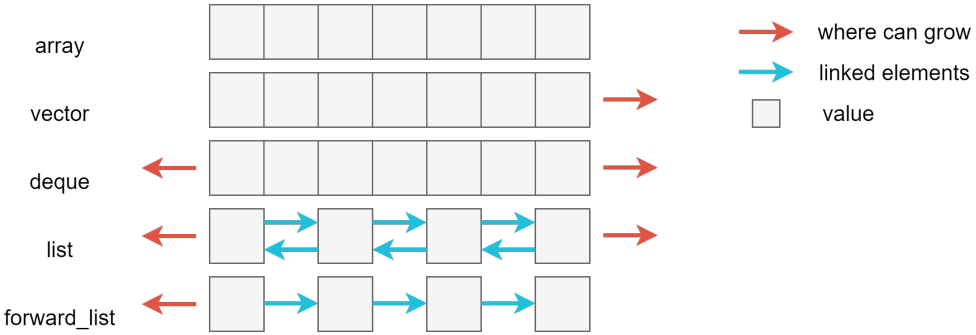
```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& v)
{
    std::for_each(v.begin(), v.end(), [&](const int& x) { os << x << " " ; });
    os << "\tSize:" << v.size() <<  "\tCapacity:" << v.capacity() << endl;
    return os;
}

void t_eraseRemoveA()
{
    std::vector<int> v{0,9,9,9,1,2,3,4};
    cout << v;                                          // 0 9 9 9 1 2 3 4  Size:8  Capacity:8

    const auto newEnd (remove(begin(v), end(v), 9));
    cout << v;                                          // 0 1 2 3 4 2 3 4  Size:8  Capacity:8

    v.erase(newEnd, v.end());
    cout << v;                                          // 0 1 2 3 4        Size:5  Capacity:8

    v.shrink_to_fit();
    cout << v;                                          // 0 1 2 3 4        Size:5  Capacity:5
}
```

vector: Removing elements, breaking elements ordering

Listing 110: std::vector - removing elements → breaking ordering

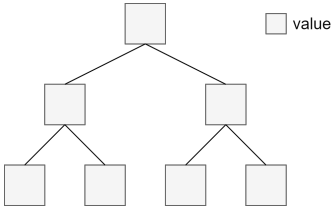
```
void t_eraseRemoveB()
{
    std::vector<int> v{0,9,9,9,1,2,3,4};
    cout << v;                                          // 0 9 9 9 1 2 3 4  Size:8  Capacity:8

    auto it = v.begin();
    for (; it!=v.end(); it++)
    {
        if (*it == 9)
        {
            //std::swap(*it, v.back());
            *it = std::move(v.back());
            v.pop_back();
        }
    }
    cout << v;                                          // 0 4 3 2 1        Size:5  Capacity:8

    v.shrink_to_fit();
    cout << v;                                          // 0 4 3 2 1        Size:5  Capacity:5
}
```

STL vector: Sequential Containers



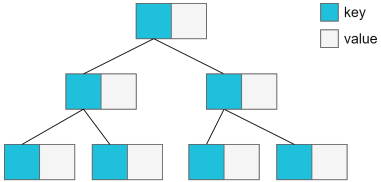
STL Containers: set, multiset

- are implemented as binary tree.
- Elements/Values represent keys.
- Their elements are sorted (<,>).
- Does not provide random access → need to iterate
- Elements can not be modified directly.



STL Containers: map, multimap

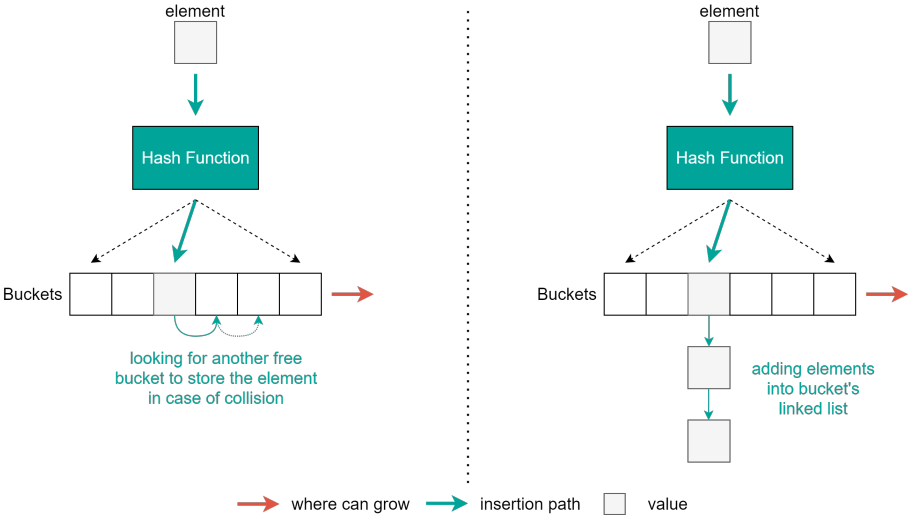
- are implemented as binary tree.
- Their elements are key-value pairs.
- Elements are sorted by keys only (<,>).
- Does not provide random access → need to iterate
- Keys can not be modified directly.



STL Containers: Unordered Containers

- They are associative containers implemented as hash tables.
- Elements are hashed and stored in undefined order.
- Fast search/insertion/deletion $\rightarrow$ this may decrease in performance over time and size of containers $\rightarrow$ rehashing
- Iterators are constant, adding an removal of elements is affected by a hash function.
- In case of unordered_set and unordered_multiset, values act as keys for hashing.
- In case of unordered_map and unordered_multimap, key-values pair are stored, where keys are used for hashing.
- unordered_multiset and unordered_multimap can contain duplicities.

Storing data in a hash table depends on inner implementation. The main difference consists in solving possible collisions. Hash table is rehashed when Load Factor=|entries|/|capacity| $\rightarrow$ 1.



STL Containers: unordered_set

Listing 111: std::unordered_set

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::unordered_set<T>& s)
{
    cout << "Buckets count: " << s.bucket_count() << endl;
    cout << "Load factor: " << s.load_factor() << endl;
    cout << "Number of elements: " << s.size() << endl;
    cout << endl;
    for(const auto& x : s)
        os << "Bucket #: " << s.bucket(x) << "\t" << x << endl;
    os << endl;
    return os;
}
void t_unorderedSet()
{
    std::unordered_set<std::string> data;
    data.insert("This");
    data.insert("is");
    data.insert("an");
    data.insert("unordered set");
    data.insert("example.");
    cout << data;
}
```

```
Buckets count: 8
Load factor: 0.625
Number of elements: 5

Bucket #: 1     This
Bucket #: 5     example.
Bucket #: 5     is
Bucket #: 6     an
Bucket #: 7     unordered set
```

See the ordering of elements. Moreover, two elements are stored with the same bucket id.

STL Containers: unordered_map

Listing 112: std::unordered_map

```
template<typename S, typename T>
std::ostream& operator<<(std::ostream& os, const std::unordered_map<S,T>& m)
{
    // for(const auto& x : m)
    //  os << "first: " << x.first << "\tsecond: " << x.second << endl;
    for(const auto& [key, value] : m)       // structured binding since C++17
        os << "first: " << key << "\tsecond: " << value << endl;
    os << endl;
    return os;
}
void t_unorderedMap()
{
    std::unordered_map<std::string, int> um = {
        {"abc", 123},
        {"def", 456}
    };
    std::unordered_map<std::string, std::string> data;
    data.insert(std::make_pair("Hey", "you"));
    data.insert({"Hello", "VSB"});
    data["Key"] = "Value";
    data["C++"] = "Good";
    data["C#"] = "Good";
    data["VB"] = "Bad";
    cout << data;
}
```

```
first: Hey      second: you
first: C#       second: Good
first: Hello    second: VSB
first: Key      second: Value
first: C++      second: Good
first: VB       second: Bad
```

See the ordering of elements.

NEWs

first, second members structured binding

STL Containers: hash

Listing 113: std::unordered_set - Custom class hashing

```
class Student
{
public:
    Student(const std::string& _name, const std::string& _login) : name{_name}, login{_login} {}
    const std::string& getName() const { return name; }
    const std::string& getLogin() const { return login; }
private:
    std::string name;
    std::string login;
};
struct StudentHash
{
    size_t operator()(const Student& s) const
    {
        auto h0 = std::hash<std::string>{}(s.getName());
        auto h1 = std::hash<std::string>{}(s.getLogin());
        return h0 ^ h1;
    }
};
struct StudentEquality
{
    bool operator()(const Student& s0, const Student& s1) const
    {
        return s0.getLogin() == s1.getLogin();
    }
};
```

See the usage of three custom types:

1. Data storage → Student
2. Type providing hashing function → StudentHash
3. Type checking equality → StudentEquality . . . (The same can be achieved by overriding operators == and != in the Student class.)

Listing 114: ... and its usage

```
void t_hash()
{
    std::unordered_set<Student, StudentHash, StudentEquality> ↩
        students;
    students.insert(Student{"Franta Omacka", "MAL123"});
    students.insert(Student{"Lucka Blahova", "BLA456"});
    students.insert(Student{"Karel Nejedly", "NEJ456"});

    for (const auto& x: students)
    {
        cout << x.getLogin() << "\t" << x.getName() << endl;
    }
}
```

```
MAL123 Franta Omacka
BLA456 Lucka Blahova
NEJ456 Karel Nejedly
```

STL Containers: Other useful data types

std::bitset

is available in the std namespace in the <bitset> header. A bitset represents a fixed-size sequence of bits, with the size defined at compile time.

Listing 115: bitset

```
void t_bitsets()
{
    std::bitset<8> b1; // [0,0,0,0,0,0,0,0]
    std::bitset<8> b2{ 10 }; // [0,0,0,0,1,0,1,0]
    std::bitset<8> b3{"1010"}; // [0,0,0,0,1,0,1,0]
    std::bitset<8> b4{ "ooooxoxo", 8, 'o', 'x' }; // [0,0,0,0,1,0,1,0]
}
```

Contains all expected functions, e.g. count, any, all, none, test, ...
and operators, e.g. [], &, |, ∼, ˆ, «, »

std::pair

is an abstract data structure found in the standard library which can bound two heterogeneous values. Pair is an ordered structure → has tu members → first, second.
Use it instead of creating "artificial" types, e.g. `struct S {int x; int y;};`

Listing 116: pair

```
void t_pairs()
{
    std::pair<int, std::string> p1{123, " Hello VSB"};
    cout << p1.first << "\t" << p1.second << endl;

    std::pair<int, std::string> p2;
    p2.first = 123;
    p2.second = "Hello VSB";

    std::pair<int, std::string> p3 = std::make_pair<int, std::string>(123, " Hello VSB");
    std::pair<int, std::string> p4 = std::make_pair<>(123, " Hello VSB");
    auto p5 = std::make_pair<>(123, " Hello VSB");

    auto [a, b] = p5;       // binding: int a=p5.first; std::string b=p5.second
    cout << a << "\t" << b << endl;
}
```

Pairs are also useful when iterating map, multimaps, ...

std::tuple

is used when a "pair" is not enough because more values are needed. It has unnamed members that can only be retrieved with a function call (std::get<id>(tupple)), or can be copied to named variables.

Listing 117: tuple

```
void t_tuples()
{
    using std::get;

    std::tuple<int, std::string, bool> t1{123, " Hello VSB", true};
    cout << get<0>(t1) << "\t" << get<1>(t1) << "\t" << get<2>(t1) << endl;

    std::tuple<int, std::string, bool> t2;
    get<0>(t2) = 123;   get<1>(t2) = "Hello VSB";   get<2>(t2) = true;

    std::tuple<int, std::string, bool> t3 = std::make_tuple<int, std::string>(123, " Hello VSB", true);
    std::tuple<int, std::string, bool> t4 = std::make_tuple<>(123, " Hello VSB", true);
    auto t5 = std::make_tuple<>(123, " Hello VSB", true);

    // binding: int a=get<0>(t5); std::string b=get<1>(t5); bool c=get<2>(t5);
    auto [a, b, c] = t5;
    cout << a << "\t" << b << "\t" << c << endl;

    if (auto [a, b, c] = t5; c == true)
        cout << "U are a good programmer." << endl;
}
```

STL Algorithms

STL Algorithms

 See the basics for Standard Template Library in [1] page(s) 567–579.

Moreove, we strongly recommend to watch the following presentation or to read its offline PDF version.

 Jonathan Boccara prepared a nice talk on CppCon2018 105 STL Algorithms in Less than an Hour. PDF version can be downloaded here.

STL Snippets

STL Snippets

STL is pretty complex and large to explain in a single presentation. That is why we focus on snippets and code examples that can help you with your daily coding.

This function will be used in all below mentioned examples. It allows us to print std::vector to the standard output.

Listing 118: ... output std::vector

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& v)
{
    std::for_each(v.begin(), v.end(), [&](const T& x) { os << x << " " ; });
    os << endl << "Size: " << v.size() <<  "\tCapacity: " << v.capacity() << endl << endl;
    return os;
}
```

STL Snippets: random

Try to avoid rand() in real applications because of poor quality from the perspective of generating random numbers (range, periodicity, distribution, ...).

Listing 119: generating function

```
std::vector<int> createRandom(const size_t count, const int minV, const int maxV)
{
    std::random_device rd{};        // used for seeding and pseudo-random engine
    auto mt = std::mt19937_64{rd()};    // one of available engines
    auto dist = std::uniform_int_distribution{minV, maxV}; // one of the ↩
        available distributions

    // decltype(dist.param()) newLimits{0,100};     // changing limits at runtime
    // dist.param(newLimits);

    std::vector<int> v;
    v.reserve(count);                               // Preallocate space
    for (size_t i=0; i<v.capacity(); ++i)
    {
        v.emplace_back(dist(mt));
    }
    return v;
}
```

Listing 120: ... and its usage

```
void t_random()
{
    std::vector<int> v =
        createRandom(10,0,2);
    cout << v;
}
```

```
1 2 0 2 2 1 2 0 2 2
Size: 10 Capacity: 10
```

STL Snippets: sets

Listing 121: working with sets

```
void t_sets()
{
    std::vector<int> a{0,1,2,3,4,5,6,7,8,9};
    cout << "a: " << a;
    std::vector<int> b{0,1,2,10,12,13};
    cout << "b: " << b;

    std::vector<int> added;     // b-a
    std::set_difference(b.begin(), b.end(), a.begin(), a.end(),
        std::back_inserter(added));
    cout << "added: " << added;

    std::vector<int> removed;   // a-b
    std::set_difference(a.begin(), a.end(), b.begin(), b.end(),
        std::back_inserter(removed));
    cout << "removed: " << removed;

    std::vector<int> unionAB;
    std::set_union(a.begin(), a.end(), b.begin(), b.end(),
        std::back_inserter(unionAB));
    cout << "unionAB: " << unionAB;

    std::vector<int> intersectionAB;
    std::set_intersection(a.begin(), a.end(), b.begin(), b.end(),
        std::back_inserter(intersectionAB));
    cout << "intersectionAB: " << intersectionAB;
}
```

```
a: 0 1 2 3 4 5 6 7 8 9
Size: 10 Capacity: 10

b: 0 1 2 10 12 13
Size: 6  Capacity: 6

added: 10 12 13
Size: 3  Capacity: 3

removed: 3 4 5 6 7 8 9
Size: 7  Capacity: 9

unionAB: 0 1 2 3 4 5 6 7 8 9 10 ↩
     12 13
Size: 13 Capacity: 13

itersectionAB: 0 1 2
Size: 3  Capacity: 3
```

STL Snippets: binary_search, find, find_if, count, count_if

Listing 122: searching for elements and counting

```
void t_findingElements()
{
    std::vector<int> v{0,1,2,3,4,5,6,7,8,9,10,11};

    int x = 5;
    bool r0 = std::binary_search(v.begin(), v.end(), x); // v must be sorted !
    cout << x << " is in the vector: " << std::boolalpha << r0 << endl;

    auto rIter = std::find(v.begin(), v.end(), x);       // v must not be sorted
    cout << x << " is in the vector: " << std::boolalpha
        << (rIter!=v.end()) << endl;

    rIter = std::find_if(v.begin(), v.end(), [&x](int i){return (i==x);});
    cout << x << " is in the vector: " << std::boolalpha
        << (rIter!=v.end()) << endl;

    auto r1 = std::count(v.begin(), v.end(), x);
    cout << "Number of occurences of: " << x << ": " << r1 << endl;

    r1 = std::count_if(v.begin(), v.end(), [](int i){return (i%2 == 0);});
    cout << "Number of even numbers: " << r1 << endl;
}
```

```
5 is in the vector: true
5 is in the vector: true
5 is in the vector: true
Number of occurences of 5: 1
Number of even numbers: 6
```

STL Snippets: transform

Listing 123: transofrming elements

```
void t_transform()
{
    std::vector<int> v{0,1,2,3,4,5,6,7,8,9};
    cout << "v: " << v;

    std::vector<int> r;
    std::transform(v.begin(), v.end(), std::back_inserter(r),
        [](int i){ return i*2;});
    cout << "r: " << r;

    // Reuse r when it is already initialized and has required number of elements
    std::transform(v.begin(), v.end(), r.begin(), [](int i){ return i*3;});
    cout << "r: " << r;

    // make a sum of two vecotrs.
    // The second range needs to be at least as long as the first one.
    std::vector<int> sumVR;
    std::transform(v.begin(), v.end(), r.begin(), std::back_inserter(sumVR),
        [](int i, int j){ return i+j;});
    cout << "sumVR: " << sumVR;
}
```

```
v: 0 1 2 3 4 5 6 7 8 9
Size: 10 Capacity: 10

r: 0 2 4 6 8 10 12 14 16 18
Size: 10 Capacity: 13

r: 0 3 6 9 12 15 18 21 24 27
Size: 10 Capacity: 13

sumVR: 0 4 8 12 16 20 24 28 32 ↩
     36
Size: 10 Capacity: 13
```

STL Snippets: sort, partition

Sort: Reorder the range of elements.
Partition: All elements for which the predicate P returns true precede the elements for which predicate P returns false. Relative order of the elements is not preserved.

Listing 124: sorting and partitioning

```
void t_sorting()
{
    std::vector v = createRandom(10,0,10);
    cout << "Before: " << v;
    std::sort(v.begin(), v.end());
    cout << "After: " << v;
    std::sort(v.begin(), v.end(), std::greater<int>());
    cout << "Descresing: " << v;
    std::sort(v.begin(), v.end(), [](int a, int b) { return a%2 < b%2; });
    cout << "Custom: " << v;
}

void t_partitioning()
{
    std::vector v = createRandom(10,0,10);
    cout << "Before: " << v;
    auto pPoint = std::partition(v.begin(), v.end(), [](int i){ return i<5;});
    cout << "After: " << v;
    cout << "Partition Point: " << *pPoint << endl;
}
```

```
Before: 10 1 5 0 7 8 3 2 7 7
Size: 10        Capacity: 10

After: 0 1 2 3 5 7 7 7 8 10
Size: 10        Capacity: 10

Descresing: 10 8 7 7 7 5 3 2 1 0
Size: 10        Capacity: 10

Custom: 10 8 2 0 7 7 7 5 3 1
Size: 10        Capacity: 10

Before: 9 4 8 10 8 6 2 7 0 3
Size: 10        Capacity: 10

After: 3 4 0 2 8 6 10 7 8 9
Size: 10        Capacity: 10

Partition Point: 8
```

References

References

[1] M. Adams.
Lecture slides for programming in c++ (version 2020-02-29), 2020.

[2] J. Galowicz.
C++ 17 STL Cookbook.
Packt Publishing Ltd, 2017.

[3] G. O'Regan.
C and c++ programming languages.
In *The Innovation in Computing Companion*, pages 63–68. Springer, 2018.

[4] B. Stroustrup.
The C++ programming language.
Addison-Wesley, Upper Saddle River, NJ, fourth edition edition, 2013.

[5] D. Zak.
Introduction to Programming with C++: 8th Edition.
Cengage Learning, 2016.

Thank you for your attention