Course Introduction

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

autumn 2020

Course Introduction

Course language: English

- all study materials in English
- lectures in English
- seminars/consultations Czech or English (depends on the students)
- questions or one-to-one discussions can be in Czech on both seminars
- documentation/comments in English
  (but you should do that always)

Course organisation

- video lectures
- seminars/consultations are online on Discord
  - in the times specified in the timetable
  - more info on the course web

https://www.fi.muni.cz/pv264

Course Introduction

Topic: Advanced usage of modern C++

- ISO C++17, parts of ISO C++20 (as implemented in current compilers)
- known concepts of C++ in more detail
- move semantics (rvalue references)
- functional programming in C++, ranges
- generic programming and metaprogramming in C++ (templates, concepts)
- resource management (smart pointers, RAII)
- modern C++ idioms
- parallel programming (threads, atomic)
- optimisation, profiling, debugging
- interesting C++ libraries
- future of C++

Course Introduction

Prerequisites: PB071, PB161 or equivalent knowledge

- basic syntax and semantics of C and C++
- some programming skills
- compilation workflow
- pointer arithmetic, working with strings and arrays (C and C++)
- value semantics of C++, references
- C++ standard library (containers, algorithms)
- constructors/destructors, copying, basic resource management
- input/output
- basic OOP principles, virtual methods (late binding), inheritance
- exceptions
- basic understanding of templates
- `unique_ptr`

Course Organisation

- **Lectures**
- **Seminars**
  - exercises related to the current lecture's topic
  - (in the 2nd half of the semester) partially project consultation
- **Homework**
  - two assignments, evaluation: pass/fail
  - need to pass both assignments
  - automatic testing + checked by tutor
  - some tests available to students: need to pass all of those!
- **Projects**
  - groups of at most 3 students, evaluation: score
  - topic chosen by you; details later
  - project presentation (last two weeks of the semester)
  - code review of another group's project (twice)
  - main submission before the end of semester; resubmission during the exam period (if you do not gain enough points)

Course Organisation

Evaluation

- the whole course is pass/fail only, no grades
- homeworks pass/fail
- project: 6 points in total, need at least 4 to pass
  - checkpoint (8th week of semester): pass/fail
  - functionality: 3 points
  - code/design quality: 3 points
  - presentation: -1 point if not done or done badly
  - code review: -1 point if not done (twice per semester)

Course Organisation

Standards, Compilers

- we use C++20 (`-std=c++20` for current *gcc*/*clang*; `-std=c++2a` for slightly older *gcc*/*clang*)
- see the web for information about compilers and other tools: `https://www.fi.muni.cz/pv264/tools`

Documentation

- recommended source: `http://en.cppreference.com/w/cpp`

Outline for Today's Lecture

Part 1

- the C++ build process, *cmake*

Part 2

- useful tools: *clang-tidy*, sanitizers, valgrind
- basic C++ knowledge review + extension
  - pointers, references
  - initialization, `initializer_list`
  - C++11: **`auto`**, range-based **`for`**, **`nullptr`**, **`using`**, **`final`**, **`override`**, **`default`**/**`delete`**
  - C++11 standard library: tuples, hashtables
  - `unique_ptr` (basic usage)

Part 3

- some notes on testing and debugging

The Build Process

Header Files (`.h` or `.hpp`)

- contain (function, class) declarations
- may also contain function definitions

The Build Process

Header Files (`.h` or `.hpp`)

- contain (function, class) declarations
- may also contain function definitions
  - inline free functions (`inline` specifier)
  - inline member functions
    (`inline` not needed, if they are inside class declarations)
  - inline variables
- also contain full definitions of templated functions, classes, and variables

Source Files (`.cpp`)

- contain function definitions

Note: various other extension are used (`.cc`, `.cxx`, `.C`, `.c++`), we are going to use `.cpp` in this course.

The Build Process

1. **Preprocessing** `g++ -E example.cpp`
   - source file + header files → expanded source file
2. **Compilation** `g++` *compile options* `-S example.cpp` (does 1, 2)
   - source file → assembler file (`.s`)
3. **Assembly** `g++` *compile options* `-c example.cpp` (does 1, 2, 3)
   - assembler file → object file (`.o`)
4. **Linking** `g++ example.o main.o -lm -o example` (does 4)
   - object files + library files → executable file

`g++` *compile options* `example.cpp main.cpp -lm -o example` (does 1–4)

The Build Process

Build Automation

- basic: *make*, `Makefile`

```
program: main.o example.o
    $(CXX) $< -o $@

%.o: %.cpp example.h
    $(CXX) -std=c++20 -Wall -Wextra -pedantic -g $< -o $@
```

- rules: target, dependencies, command
- checks if a dependency is newer than target and only runs those rules
- quite powerful (see documentation)
- `$(CXX)` – variable, refers to the C++ compiler (`$(CC)` for C compiler)
  - defaults to g++ on Linux, or to the value of the `CXX` environment variable
  - `make CXX=g++-10 CC=gcc-10`

The Build Process

Build Script Generation using *CMake*

- cross-platform tool
- generates Makefiles (and also files for other build systems)
  - you may want to look at *ninja* (`cmake -GNinja`)
- main file `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.5)
project(example)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 \
                     -Wall -Wextra -pedantic")
set(SOURCE_FILES example.cpp main.cpp)
add_executable(example ${SOURCE_FILES})
# target_link_libraries(example ...libraries...)
```

The Build Process

Using `cmake`

- create a *separate build directory*

```
mkdir build
```

- from the build directory run `cmake` *path to source directory*

```
cd build
cmake ..
```

- run `make` in the build directory

Useful tricks

- change default compiler
  - `CC=clang CXX=clang++ cmake ..` or

```
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ ..
```

- change build configuration: `ccmake`
  - `CMAKE_BUILD_TYPE`: Release / Debug / RelWithDebInfo

Dependency Management

- C++ does not come with a standard package manager
- popular package managers for C++:
  - **Conan**: becoming de-facto standard, suitable for complex dependencies, distributed repositories, precompiled packages
  - **vcpkg**: multi-platform manager maintained by Microsoft, always builds everything from source
  - **Hunter**: completely integrated within CMake (no extra file for dependencies)

Dependency Management

- C++ does not come with a standard package manager
- popular package managers for C++:
  - **Conan**: becoming de-facto standard, suitable for complex dependencies, distributed repositories, precompiled packages
  - **vcpkg**: multi-platform manager maintained by Microsoft, always builds everything from source
  - **Hunter**: completely integrated within CMake (no extra file for dependencies)
- if you feel the need for package manager, try Conan first
- `FetchContent`: much simpler alternative to package managers – often best solution in most cases

FetchContent

- available since CMake 3.11
- specify dependency from various source (zip, git, SVN, ...)
- dependency is fetched at the **configuration** step
- can work with both; CMake-based and other build system's dependencies

CMake-based dependency example:

```
FetchContent_Declare(
  fmt
  GIT_REPOSITORY https://github.com/fmtlib/fmt.git
  GIT_TAG        7.0.3 # Can be tag/commit/branch
)
FetchContent_MakeAvailable(fmt) # Available since CMake 3.14

# Specify your project here
# Link the fmt library
target_link_libraries(myProject PRIVATE fmt)
```

FetchContent – Older CMake

- `FetchContent_MakeAvailable` is not available in version prior 3.14
- you have to write `FetchContent_MakeAvailable` by yourself:

```
FetchContent_GetProperties(fmt)
if(NOT fmt_POPULATED)
  FetchContent_Populate(fmt)
  add_subdirectory(${fmt_SOURCE_DIR} ${fmt_BINARY_DIR})
endif()
```

- Note that `if(NOT fmt_POPULATED)` is recommended, so parent CMakes can override (thus unify) dependencies

FetchContent – Non-CMake Dependency

- write custom alternative to FetchContent_MakeAvailable
  - either use ExternalProject_Add to preserve original build process
  - write custom CMake build process

```
FetchContent_GetProperties(nonCmakeLib)
if(NOT nonCmakeLib_POPULATED)
  FetchContent_Populate(nonCmakeLib)
  ## Specify CMake build process
  file(GLOB_RECURSE src "${nonCmakeLib_SOURCE_DIR}/*cpp")
  add_library(extLib STATIC ${src})
  target_include_directories(extLib PUBLIC
      "${nonCmakeLib_SOURCE_DIR}/include")
endif()

## Use your library
target_link_library(myProject PRIVATE extLib)
```

Useful Tools

`valgrind` tool suite

- (you should already know about this)
- *memcheck* – checks memory-related errors
  - memory leaks
  - uninitialized memory
  - wrong memory access, invalid `free`/`delete`
- *other tools* – heap/cache profiling, call graph analysis, ...

- currently works on Linux and OS X only
  - there are some alternatives for Windows (Dr. Memory)
- cannot detect some stack-related memory errors

Useful Tools

`clang-tidy` (static analysis)

- diagnose and fix typical programming errors and style violations
- also runs clang static analyser
- *Note*: We use clang-tidy to check your code, the list of enabled checks is given on the web.

`clang`/`gcc` sanitizers (at runtime)

- AddressSanitizer `-fsanitize=address`
  - out-of-bound accesses, memory leaks, . . .
- MemorySanitizer `-fsanitize=memory`
  - uninitialized memory access, `clang` only
- UndefinedBehaviourSanitizer `-fsanitize=undefined`
  - detect undefined behaviour

- see documentation on the web

Pointers, references

Main differences between pointers and references in C++

Pointers, references

Main differences between pointers and references in C++

- pointers may be uninitialized
  - references are always initialized
- pointers may point to *null* (`nullptr`)
  - references always point to an object[1]
- pointers may be redirected (if not `const`)
  - reference may never be redirected

[1] Really?

Pointers, references

Main differences between pointers and references in C++

- pointers may be uninitialized
  - references are always initialized
- pointers may point to *null* (`nullptr`)
  - references always point to an object[1]
- pointers may be redirected (if not `const`)
  - reference may never be redirected

[1] Really? Yes!

Pointers, references

Main differences between pointers and references in C++

- pointers may be uninitialized
  - references are always initialized
- pointers may point to *null* (`nullptr`)
  - references always point to an object[1]
- pointers may be redirected (if not `const`)
  - reference may never be redirected

```
int* ptr = &x;
*ptr = 3;
ptr = &y;
```

```
int& ref = x;
ref = 3;
// cannot do this
```

[1] Really? Yes!

Pointers, references + `const`

	modify object	redirect pointer
`int*`		
`const int*`		
`int* const`		
`const int* const`		
`int&`		
`const int&`		

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`		
`int* const`		
`const int* const`		
`int&`		
`const int&`		

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`	No	Yes
`int* const`		
`const int* const`		
`int&`		
`const int&`		

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`	No	Yes
`int* const`	Yes	No
`const int* const`		
`int&`		
`const int&`		

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`	No	Yes
`int* const`	Yes	No
`const int* const`	No	No
`int&`		
`const int&`		

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`	No	Yes
`int* const`	Yes	No
`const int* const`	No	No
`int&`	Yes	No
`const int&`		

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`	No	Yes
`int* const`	Yes	No
`const int* const`	No	No
`int&`	Yes	No
`const int&`	No	No

Pointers, references + `const`

	modify object	redirect pointer
`int*`	Yes	Yes
`const int*`	No	Yes
`int* const`	Yes	No
`const int* const`	No	No
`int&`	Yes	No
`const int&`	No	No

- `int&` is *almost* like `int* const`
  - with different syntax

Initialization

Initialization in C++(14)

Initialization

Initialization in C++(14)

```
Object x;
Object x = y;
Object x(y);
Object x(a, b, c);
Object x(); // NOT initialization! "most vexing parse"
Object x{}; // C++11
Object x{a, b, c}; // C++11
Object x = {a, b, c}; // not only C++11
```

- is that all?

Initialization

Initialization in C++(14)

```
Object x;
Object x = y;
Object x(y);
Object x(a, b, c);
Object x(); // NOT initialization! "most vexing parse"
Object x{}; // C++11
Object x{a, b, c}; // C++11
Object x = {a, b, c}; // not only C++11
```

- is that all?
  - we forgot about temporary objects and dynamically allocated objects
  - temporary: `Object()`, `Object(a, b)`, `Object{}`, `Object{a, b}`, `{a, b}` (if the type can be inferred), `y` (what does this mean?)
  - dynamic: `new Object`, ...

Initialization (simplified!)

Default initialization

- variable declared without initializer
- `new` expression without initializer
- base class or member variable not included in member initializer list of a constructor

What happens?

Initialization (simplified!)

Default initialization

- variable declared without initializer
- `new` expression without initializer
- base class or member variable not included in member initializer list of a constructor

What happens?

- class type: default constructor is called
- arrays: all elements default-initialized
- otherwise: **nothing**
  - `static`/global variables: value is zero
  - other variables: **undefined value**

Initialization (simplified!)

Value initialization

- variable declared with `{}`
- `new`, temporary object, member variable created with `()` or `{}`

What happens?

Initialization (simplified!)

Value initialization

- variable declared with `{}`
- **`new`**, temporary object, member variable created with `()` or `{}`

What happens?

- class type: default constructor is called (if it exists[2])
- otherwise: zero-initialized

Note: This is very simplified. If you want the full story, go read cppreference.com.

[2] otherwise, initializer list constructor may be called, see next slide

Initializer List

Motivation: Arrays can be initialized by lists.

```
int array[] = { 0, 0, 7, 42 };
```

Since C++11 we can do the same with user-defined types:

- `std::initializer_list`
- constructor with one parameter of type `std::initializer_list<T>`
  - used if initializing using braces and
  - all elements of the list have type `T` (can be converted to `T`)
    - priority over any other constructor (except for default)

```
struct MyVec {
    MyVec( std::initializer_list<int> );
};
int main() {
    MyVec vec{ 1, 2, 3 };
}
```

Initializer List

- `std::initializer_list` is a lightweight immutable object
  - copying it does not copy the elements
  - typical use: pass-by-value
- defines methods `begin`, `end`, `size`
- can be also created when binding a brace-enclosed list to `auto`
  - sometimes useful in range-based `for`

```
auto x = { 1, 2, 3, 4 };
// x is std::initializer_list<int>

for (int n : { 1, 1, 2, 3, 5 }) { /* ... */ }
```

auto

- (you should know this already)
- `auto`matic type deduction:
  - same rules as for template types deduction
  - except for `std::initializer_list` (see previous slide)
  - does not create references! for that, use `auto&`

When should we use `auto`?

auto

- (you should know this already)
- `auto`matic type deduction:
  - same rules as for template types deduction
  - except for `std::initializer_list` (see previous slide)
  - does not create references! for that, use `auto&`

When should we use `auto`?

- we don't know the real type (but the compiler knows)
  - how can this happen?
- we know the real type but it is either:
  - too ugly (typical use: iterators), or
  - it can be clearly inferred (by humans) from context

```
auto printer = printerFactory.createInkPrinter();
auto ptr = std::make_unique<ListNode>();
for (const auto& person: people) { /* ... */ }
```

range-based `for`

- (you should know this already)

```
for (type var : container) { do_something(var); }
```

- what does this translate to?

range-based `for`

- (you should know this already)

```
for (type var : container) { do_something(var); }
```

- what does this translate to?

```
{
    auto&& __l = container;
    auto __i = std::begin(__l),
         __e = std::end(__l);
    for (; __i != __e; ++__i) {
        type var = *__i;
        do_something(var);
    }
}
```

range-based `for`

- (you should know this already)

```
for (type var : container) { do_something(var); }
```

- what does this translate to?

```
{
    auto&& __l = container;
    auto __i = std::begin(__l),
         __e = std::end(__l);
    for (; __i != __e; ++__i) {
        type var = *__i;
        do_something(var);
    }
}
```

- note: this holds for C++11/14, in C++17 `__i` and `__e` do not have to be the same type (why is this useful?)

nullptr

- use instead of NULL; why?

nullptr

- use instead of NULL; why?
  - not a macro
  - distinct type std::nullptr_t
  - convertible to other pointers, bool, but NOT to int

```
void foo(int x); // 1
void foo(const char* ptr); // 2

// What does this call?
foo(nullptr); // calls 2
foo(NULL); // who knows... (maybe 1, maybe 2, maybe error)
```

using

- “Better `typedef`”
- nicer syntax; can be templated

```
using IntVec = std::vector<int>;

template <typename T>
using Matrix = std::vector<std::vector<T>>;

Matrix<int> matrix;
```

`override`, `final`

For member functions:

- written after method signature (like `const`)
- both denote virtual function override
  - prevent signature mistakes
- `final` also prevents further override
  - use with care

For classes:

- `final` makes class non-inheritable

Recommendation:

- write `virtual` only when not overriding
- write `override` when overriding
- write `final` if you really have to
  - hint: you most probably don't

`default`/`delete`

`default`

- force the compiler to automatically generate the given method
- only works for default constructor, copy constructor/assignment, move constructor/assignment and destructor
- when is this needed?

default/delete

default

- force the compiler to automatically generate the given method
- only works for default constructor, copy constructor/assignment, move constructor/assignment and destructor
- when is this needed?
  - something inhibits the automatic generation
  - e.g. parametric constructor inhibits default constructor

delete

- forbid the compiler to call the function
- can be used with all functions

```
struct A {
    A() = default;
    A(const A&) = delete; // forbid copying
    A& operator=(const A&) = delete;
};
```

C++11 standard library additions

`std::tuple`

- fixed-size collection of different types
- useful function `std::make_tuple`, `std::tie`
- since C++17, can be also initialized with `{ a, b, c }`

`std::unordered_set`, `std::unordered_map`

- work like `set` and `map`, but are implemented with a hash table
- the elements need to implement a hash function
  by specializing the `std::hash<T>` template

`std::array`

- wrapper around C-style fixed-size arrays
- has the interface of standard containers, `size` method etc.

`std::forward_list`

- singly linked list

C++17 structured binding

Unpacking `std::tuple`

- in C++11/14: `std::tie(a, b, c) = getTuple();`
  - `a`, `b`, `c` have to be declared first
  - `a`, `b`, `c` are *copies* of the values (even if `getTuple()` returns a tuple of references)
- in C++17: structured binding
  - `auto [a, b, c] = getTuple();`
    (types of `a`, `b`, `c` are automatically deduced)
  - `auto& [a, b, c] = getTuple();` (force references)
    (also `const auto&`, `auto&&`)
- usable with
  - C-like arrays
  - anything tuple-like (`std::tuple`, `std::pair`, `std::array`, anything that specialises `std::tuple_size` and `std::tuple_element`)
  - public data members

unique_ptr

unique_ptr

- header <memory>
- one of C++11 *smart pointers*
- zero-overhead at runtime
- unique_ptr *owns* the allocated memory; once the unique_ptr object goes out of scope, the memory is deallocated
- no other unique_ptr may own the same memory (*unique*)
- we can still have *raw pointers* pointing to the same memory (non-owning pointers)
- ownership may be transferred (uses move semantics – we will talk about this later)

unique_ptr Example

```cpp
// creating new unique_ptr
std::unique_ptr< Object > ptr(new Object(params));
// since C++14, better use this:
auto ptr = std::make_unique< Object >(params);

// operators ->, * work as usual
ptr->method();
function(*ptr);

// get underlying raw pointer
Object* rawPtr = ptr.get();

// transfer ownership
std::unique_ptr< Object > otherPtr = std::move(ptr);
// ptr is now equal to nullptr
// otherPtr owns the allocated memory
```