

KIV/CPP – Programming in C++

01. Introduction to Modern C++

Martin Úbl

KIV ZČU

2025/2026

- multi-paradigm compiled language
 - procedural
 - object-oriented
 - generic programming
 - elements of functional programming
- Bjarne Stroustrup et al.
- standards:
 - C++98, C++03
 - C++11, C++14, C++17
 - C++20, C++23
 - C++26, C++2a?

- safety
 - working with memory
 - type safety
 - working with resources
- readability
- performance
- efficient development
 - object-oriented paradigm
 - generic programming
- standard library (`libc` + STL)

- deterministic object lifetime
- statically typed
- templates + static polymorphism

- memory allocation
 - static
 - dynamic – new and delete (vs. malloc and free in C)
- prefer static allocation where possible
 - or containers that allocate dynamically internally
 - memory and resource management – RAI
- support for compile-time evaluation (constexpr, consteval, constexpr)
- copy and move semantics
- idioms
 - RAI, Pimpl, CRTP, SFINAE, scope guard
 - and more...

<https://en.cppreference.com/w/cpp/language>
- exceptions
- namespaces

- Hello world!

```
#include <iostream>
```

```
int main(int argc, char** argv)
```

```
{
```

```
    std::cout << "Hello_world" << std::endl;
```

```
    return 0;
```

```
}
```

- Hello world!
- using namespace - can be bad practice (name conflicts)

```
#include <iostream>
```

```
using namespace std; // ugh!
```

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

- data types are the same as in C
 - here they fall into the POD (Plain Old Data) category
 - since C++20 officially "trivial and standard layout"
- additionally, for example:
 - classes (`class`)
 - strongly typed enumerations (`enum class`)

- class
 - the `class` keyword
 - visibility sections (`public`, `private`, `protected`)
 - constructors
 - initializer lists
 - destructor
 - virtual methods (polymorphism)
 - separation of definition and implementation
- simplified example on the following slides
 - for now without some details that will be important later
 - for example `const` and references, virtual destructor, see later

```
class Player {                                     // Player.h
public:
    Player();
    Player(std::string name);
    ~Player();

    virtual double Get_X();
    virtual double Get_Y();
    std::string Get_Name();

protected:
    std::string mName;

private:
    double mX, mY;
};
```

```
#include "Player.h"                                // Player.cpp

Player::Player() : mName("Unknown_player") {
    // initialize
}

Player::Player(std::string name)
    : mName(name) {
    // initialize
}

Player::~~Player() {
    // deinitialize
}

double Player::Get_X() {
    return mX;
}

...
```

- implementation can be written in the class declaration
- virtual destructor

```
class Player {                                     // Player.h
public:
    Player() : mName("Unknown_player") {
        //
    }
    virtual ~Player() {
        // ensure proper destructor calls
        // in a polymorphic scheme
    }
    ...
};
```

- reference
 - “safe pointer”
 - always refers to a valid object/POD
 - always initialized
 - cannot be “rebound” to another object/POD
 - the & symbol (ambiguity!)
 - cannot be null
 - can be bound to an existing reference

```
int a = 1;
int &b = a;
int &c = b;
int *d = &a; // possible, but not safe
```

```
b = 5;           // a == 5
c = 10;          // a == 10
*d = 15;         // a == 15
```

- static allocation
- dynamic allocation
- encapsulated dynamic allocation

```
Player player("Adam");
```

```
Player* player = new Player("Adam");
```

```
std::unique_ptr<Player> player  
    = std::make_unique<Player>("Adam");
```

- scope
- delimits the validity of a name (and object)

```
void someFunction()  
{  
    Player adam("Adam"); // (1)  
    Player* josef = new Player("Josef"); // (2)  
    ...  
} // (3)
```

- (1) calls the Player constructor (adam)
- (2) calls the Player constructor (josef)
- (3) calls the Player destructor (adam),
does NOT call the Player destructor (josef)

- scope
- delimits the validity of a name (and object)

```
void someOtherFunction()  
{  
    Player* josef = new Player("Josef"); // (1)  
    ...  
    delete josef;                        // (2)  
}
```

- (1) calls the Player constructor (josef)
- (2) calls the Player destructor (josef)
- error-prone
 - allocate statically if possible
 - or use smart pointers (later lectures)

- explicit scope
- delimited by braces
- combined with some control structure or declaration (loop, function, ...)

```
void someFunction()  
{  
    //  
}
```

- or standalone

```
// some outer code  
  
{  
    //  
}
```

- both work the same way (identifier validity and object lifetime)

- inheritance
- inheritance “visibility”

```
class Object {
public:
    Object(std::string name) {
        mName = name;
    }
    ...
};

class Player : public Object {
public:
    Player() : Object("Unknown_player") {
    }
    Player(std::string name) : Object(name) {
    }
};
```

- inheritance and virtual methods
- the override keyword

```
class Base {  
    public:  
        virtual void Hello();  
        void Goodbye();  
};
```

```
class Derived : public Base {  
    public:  
        virtual void Hello() override;  
        void Goodbye(); // (1)  
};
```

- (1) override would cause a compilation error –
Base::Goodbye is not virtual, so it cannot be overridden, only
hidden

- the auto keyword
- automatic type deduction
- there must always be something to deduce from
 - variable initialization
 - function return value

```
auto i = 15; // (1)
auto p = new Player(); // (2)
auto itr = playerMap.find(key); // (3)
auto ptr = std::make_unique<Player>(); // (4)
```

- (1) deduces int
- (2) deduces Player*
- (3) deduces std::map<int, Player*>::iterator
- (4) deduces std::unique_ptr<Player>

- STL containers
- higher abstraction than arrays (for example in C)
- the standard defines, for example, computational and memory complexities
- it usually does not prescribe a specific implementation
 - for example, the requirement is sequential access to a constant number of elements in constant time
 - the standard library programmer then implements it using a static array

- STL containers
- `std::array`
- generic type (templates), representation of a fixed-length array

```
#include <array>
```

```
std::array<int, 5> fiveNumbers  
    = { 1, 2, 3, 4, 5 };
```

```
std::cout << fiveNumbers[1]; // "2"
```

- C-style analogy:

```
int fiveNumbers[5] = { 1, 2, 3, 4, 5 };
```

- STL containers
- `std::vector`
- generic type (templates), representation of a variable-length array

```
#include <vector>
```

```
std::vector<int> someNumbers  
    = { 1, 2, 3 };
```

```
someNumbers.push_back(15);
```

```
std::cout << someNumbers[3]; // "15"
```

- more STL containers in the next lecture
- `std::list` – list
- `std::map` – associative storage
- `std::set` – set
- `std::queue`, `deque` – queue, double-ended queue
- and many more...

- header files
 - in STL (standard library) usually without an extension
 - otherwise often the .hpp or .h extension
 - the preprocessor does not care
- C++ variant of standard-library C header files
 - wrappers are defined to avoid conflicts
 - the extension is removed from the file name and the letter c is prepended
 - for example, `math.h` changes to `cmath`
 - functions should then be prefixed with `std::`
 - for example, `std::max`, `std::pow`, ...
- the idea is to drop header files in favor of modules (C++20)
 - but there is still a long way to go

- compilers
 - gcc/g++ (GNU)
 - clang/clang++ (LLVM)
 - icc (Intel)
 - MSVS cl (Microsoft)
 - and more...
- important criterion: support for C++11 and later standards
 - today we often require at least C++17
- development environments
 - Visual Studio, VSCode (Microsoft)
 - Dev-Cpp (Bloodshed)
 - Xcode (Apple)
 - CLion (JetBrains)
 - and more...

- established coding standards
- for example, C++ Core Guidelines
 - <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
 - worth reading when needed during coding

- lab
- getting familiar with the environment
 - Microsoft Visual Studio
- simple “Hello world” and examples of working with classes and containers