

KIV/CPP – Programming in C++

03. Objects, virtual methods, const, constexpr, final, function and operator overloading

Martin Úbl

KIV ZČU

2025/2026

Class, object

- class – declaration, implementation
- object – instance of a class
 - it already has a place in memory

```
class Rect
{
    // ...
};
```

```
Rect obdelnik;
Rect jinyObdelnik(2, 5);
```

Virtual methods

- virtual method – polymorphism
- keyword `virtual`
- virtual method table
 - at runtime, the program chooses which method to call
 - dynamic polymorphism
- overriding methods in descendants
 - the parent must have `virtual`
 - the descendant may have `override` (it should)
- the overridden method must have an identical signature
 - return value
 - name
 - parameters
 - qualifiers (`const`, `volatile`, `register`)
 - calling convention

```
class Parent
{
    public:
        virtual std::string GetName();
};

class Child : public Parent
{
    public:
        virtual std::string GetName() override;
};
```

- keyword `final`
- declares that the entity must not be overridden further
 - class – must not have a descendant
 - method – must not be overridden by a descendant
- semantic check
- optimization – so-called devirtualization
 - the compiler can “get rid of” dynamic polymorphism if it is sure that this is possible
 - small speedup

Virtual methods

```
class Parent {
    public:
        virtual std::string GetName();
};

class Child : public Parent {
    public:
        virtual std::string GetName()
            override final;
};

class Grandchild final : public Child {
    // ...
};
```

Pure virtual methods

- pure virtual methods
- intentionally unimplemented methods
- declared by writing `= 0` after the method signature
- their presence makes the class abstract
- they impose an obligation on some descendant to override and implement the method
 - a class with a pure virtual method cannot be instantiated
- classes that contain only pure virtual methods are effectively interfaces
 - C++ does not have the concept of an *interface* (as Java or C# does, for example)

```
class IDrawable {  
    public:  
        virtual void Draw() const = 0;  
};
```

- defaulted implementations
 - default, move, copy constructors
 - move-assignment and copy-assignment operators
 - destructor
 - introduced by `= default` after the signature
 - the programmer “allows” the compiler to generate the default implementation (indicates intent)
- deleted implementations
 - move, copy constructors
 - move-assignment and copy-assignment operators
 - type conversions
 - introduced by `= delete` after the signature
 - prevents the compiler from generating these constructs for us

- defaulted implementation

```
class Object {  
    public:  
        Object() = default;  
        Object(int parametr);  
        virtual ~Object() = default;  
};
```

Deleted implementation

- deleted implementation
- we can, for example, prohibit copying

```
class Object {  
    public:  
        Object();  
  
        // prohibit copying by constructor  
        Object(const Object& obj) = delete;  
        // prohibit copying by assignment  
        Object& operator=(const Object&) = delete;  
};
```

- note: operators later; this is only for illustration for now

- in C++, a constructor with one parameter is considered converting
- analogically, constructor with multiple parameters could be considered converting, if we pass an initializer list instance as input
- explicit exclusion
 - introduced by `explicit` before the declaration
 - prevents automatic use of the converting constructor

Explicit exclusion

```
class Object {  
    public:  
        Object(int a);  
};
```

- versus

```
class Object {  
    public:  
        explicit Object(int a);  
};
```

- in the first case this passes; in the second it does not:

```
void SomeFunc(Object a);  
SomeFunc(42);
```

Default parameter

- function or method parameters can have a default value
- they must not be followed by parameters that do not have one
- they are no longer written in a separate definition

```
class TextObject {  
    public:  
        TextObject(Color color, int size = 12) {  
            // ...  
        }  
        void Mod(Color color = Color::White,  
                  int size = 12);  
};  
  
void TextObject::Mod(Color color, int size) {  
}
```

- static methods

```
class Rectangle
{
    public:
        static double Calc_Surface(double a,
                                   double b);
};

...
double s = Rectangle::Calc_Surface(2.0, 3.0);
```

Static attributes

- static attributes
 - defined in the class
 - declared in the implementation

```
// somewhere in .h or .hpp file
class Rectangle
{
    public:
        static size_t Instance_Counter;
};
```

```
// somewhere in .cpp file
size_t Rectangle::Instance_Counter = 0;
```

Forward declaration

- forward declaration (similarly in C)
 - we know that the class is defined somewhere
 - we cannot include its definition first
 - for example, because of a circular dependency

```
class Map;
```

```
class GameObject {  
    protected:  
        std::shared_ptr<Map> mMap;  
};
```

```
class Map {  
    protected:  
        std::vector<std::unique_ptr<GameObject>  
            mObjects;  
};
```

- keyword `const`
- immutable value, immutable object
 - `const` variable – immutable value of the variable
 - `const` method – does not change the internal state of the object
- a `const` value always physically has a place in memory
 - unlike `constexpr`; see below
- semantic check
- room for compiler optimizations
- for methods, this can be partially bypassed with the keyword `mutable`
 - typically for attributes that change temporarily during a call
 - for example, `std::mutex` and locking for the duration of the call
 - use only exceptionally and only where it fits

- constant value

```
const size_t size = vec.size();
```

- const method – GetName() does not change the object's state; it only returns some name

```
class Person {  
    public:  
        std::string GetName() const;  
};
```

- const parameter

```
void print_var(const int value);
```

- const variable – pointer
- the address it points to can be changed; the memory it points to cannot be changed

```
const int* values;
```

- const pointer to mutable data
- the address it points to cannot be changed; the memory it points to can be changed

```
int* const values;
```

- reference to const memory

```
const int& cr_value;
```

- const reference to mutable memory is meaningless
- the “target” of a reference is already constant by nature
 - warning C4227: anachronism used: qualifiers on reference are ignored

```
int& const rc_value;
```

- constexpr
- an expression or function that can be evaluated at compile time
- constants
- calculation of “magic” constants
- deterministic calculations with constant parameters

```
constexpr double PI = 3.1415926535897932384;  
constexpr double Inv_PI = 1.0 / PI;
```

```
constexpr uint64_t faktorial(uint64_t n) {  
    return (n > 1) ? n * faktorial(n - 1) : 1;  
}
```

- note: factorial by recursion is still bad; it only demonstrates a possible use of constexpr

- since C++17, a constexpr function no longer has to be just a one-liner with a single return
- an expression/function is evaluated as constexpr only when
 - the result is assigned to another constexpr variable
 - the parameters are constexpr
- otherwise, fallback to runtime evaluation
- replacement for #define
 - constexpr is type-safe
 - a constexpr function is evaluated at compile time whenever possible
 - always prefer it over macros
- since C++17, there is also `if constexpr` – more on that later

- since C++20, the keyword `constexpr` for functions
- a variant of `constexpr`, but the function **must** be evaluable at compile time

```
constexpr int magic(int n) {  
    return n*n*2 + 15;  
}
```

- since C++20, the keyword `constexpr` for variables and constants
- a variant of `constexpr`, for constants
- the variable/constant **must** be initialized with a value that is computed at compile time

// OK:

```
constexpr double MyMagic = magic(16) + 8;
```

// Fail:

// (time() is neither `constexpr` nor `constexpr`)

```
constexpr double MyMagic = time(nullptr) + 8;
```

Function and method overloading

- a principle known from many languages
- a function/method has the same name, but always differs in its parameters
- the return value may also differ (but not exclusively)
- naturally, each declaration has its own implementation

```
class RGBColor {  
    public:  
        void Set(int r, int g, int b);  
        void Set(const RGBColor& color);  
        bool Set(const std::string& str);  
}
```

- note: watch out for similar types (for example, signed and unsigned variants)

- classic mathematical operators
 - +, -, *, /, %, ++, --
- bitwise
 - &, |, ^, ~, <<, >>
- comparison
 - ==, <=, >=, <, >, !=
- assignment (+ compound)
 - =, +=, *=, /=, ...
- access
 - [], *, &, ->, .., ...
- others...
 - fnc(...), comma, ternary operator, casts, new, delete, sizeof, ...
- <https://en.cppreference.com/w/cpp/language/operators>

- a class can define what an operator will do
- the number of parameters is given by the operator's arity
- we choose
 - the return value
 - the parameter type
 - `const`
 - reference
- do not overload operators just because we can
 - (bad) production example: operator `+` sent data over a network socket

Operator overloading

```
class Number {  
    private:  
        int number = 0;  
  
    public:  
        Number& operator=(int rhs) {  
            number = rhs;    // assignment  
            return *this;  
        }  
  
        Number& operator+=(int rhs) {  
            number += rhs;    // compound addition  
            return *this;  
        }  
  
    ...  
}
```

```
...  
    Number& operator=(const Number& rhs) {  
        number = rhs.getNumber();  
        return *this;  
    }
```

```
    Number& operator+=(const Number& rhs) {  
        number += rhs.getNumber();  
        return *this;  
    }
```

```
...
```

...

```
// type conversion operators
```

```
operator int() const {
```

```
    return number;
```

```
}
```

```
operator bool() const {
```

```
    return (number != 0);
```

```
}
```

```
operator std::string() const {
```

```
    return std::to_string(number);
```

```
}
```

...

```
...  
    friend std::ostream& operator<<  
        (std::ostream& os, const Number& c);  
}  
  
std::ostream& operator<<(std::ostream& os,  
                        const Number& c)  
{  
    os << "Number is " << c.number;  
    return os;  
}
```

- note: the keyword `friend` lets the given class “look into” the private/protected part of the class

Operator overloading

- an operator can also be “defined additionally” outside the class

```
Number operator+(const Number& a, const Number&
{
    Number ret;
    ret = (int)a + (int)b;
    return ret;
}
```

- note: this can be done better, for example by additionally defining a converting constructor for int; then this is enough

```
return (int)a + (int)b;
```

- return value = “result” of the operation
 - it may be expected
 - for example, `Number + Number` will certainly return an instance of `Number`
 - or it may be established, for example, by convention
 - the result of assignment should also be `Number`
 - then it follows the conventions of the surrounding code
 - (the result of assigning an integer is the assigned integer)
 - or strongly dependent on the usage context
 - functors

Functor

- a functor is a class whose instances “can be called like a function” – thanks to an overloaded function call operator ()

```
class NasobFunc
{
    public:
        int operator()(const int a) {
            return a*2;
        }
};

...
NasobFunc mul;

int a = 5;
int b = mul(a);
```

Overloaded operators in the STL

- `istream` and `ostream`, operators `<<`, `>>`
 - `std::cout << number << string;`
- iterators, operators `++`, `-`, `+`, `-`, `==`, `!=` dereference
 - `++itr`
- `string`, operators `==`, `!=`, `+`
 - `if (str == "hello") ...`
- and many others

- C++20 added a new operator for comparison
- `<=>`, the so-called *spaceship operator*
- three-way comparison
- it is not binary (yes/no), but returns an ordering constant
- implicitly `std::strong_ordering`
 - `less`, `equal`, `equivalent`, `greater`
 - one expression therefore returns the result of several different comparisons

Spaceship operator

- example – let the compiler deduce

```
struct IntWrapper {  
    int value;  
    auto operator<=>(const IntWrapper& a) const  
        = default;  
};
```

- example – define ourselves what the compiler would deduce

```
struct IntWrapper {  
    int value;  
    auto operator<=>(const IntWrapper& a) const {  
        return value <=> a.value;  
    }  
};
```

Spaceship operator

- example – define our own

```
struct IntWrapper {  
    int value;  
    auto operator<=>(const IntWrapper& a) const  
    {  
        if (value < a.value)  
            return std::strong_ordering::less;  
        else if (value == a.value)  
            return std::strong_ordering::equal;  
        else  
            return std::strong_ordering::greater;  
    }  
};
```

- *rewritten expressions*
- the compiler can replace some expressions with equivalents in a given context
- the spaceship operator is one example where this can happen

```
IntWrapper a{4}, b{5};
```

$$a < b \quad \text{-->} \quad (a <=> b) < 0$$

- then `a <=> b` returns `std::strong_ordering::less`
 - this expands to `-1`, which is `< 0`, so the condition is satisfied

- `<=>`
- useful as a replacement for listing all operators
- essentially, it can replace `<`, `>`, `<=`, `>=`, `==` and `!=`
- it can pay off when
 - we are interested in the result of multiple kinds of comparison
 - we do not want to implement the whole set of operators

- `<=>`
- for numeric values, the processor would do something like this anyway
- the `CMP` instruction (and similar ones) effectively performs “all comparisons at once”
 - in fact, on most platforms, they execute the `SUB` instruction (subtraction) without write-back to the register
 - they set the corresponding flags in the `FLAGS` register
- so this is not overhead, but rather an approximation of how the processor actually compares values
- for strings, an interface identical to the C `strcmp` function is defined
 - ideal for lexicographic ordering