

# Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 3

25 października 2018

- `const` - uzupełnienie
- `constexpr` - stałe wyrażenia w C++
- `static` - pola statyczne
- `static` - metody statyczne
- singleton
- operatory i sposoby ich przeciążania
- `std::string` - napisy w stylu C++
- przykład

## const - przykład uzupełniający

Co powinny zwracać metody stałe (z modyfikatorem `const`) ?

```
double* data_ = nullptr;
size_t size_ = 0;
public:
    SimpleVector(size_t s) // uwaga na rozmiar s
        : size_(s), data_(new double[s]) {}

    ~SimpleVector(void) {
        delete [] data_; } // uwaga na nullptr

    double& operator [] (unsigned int i) {
        return data_[i]; } // uwaga na zakres

    /*const*/ double& operator [] (unsigned int i) const {
        return data_[i]; } // uwaga na zakres

    double* data(void) { return data_; }

    /*const*/ double* data(void) const { return data_; }
};
```

## constexpr - stałe wyrażenia w C++

Słowo kluczowe `constexpr` gwarantuje, że dane wyrażenie jest stałe podczas procesu kompilacji.

Dotyczy:

- zmiennych
- funkcji
- metod

```
static constexpr int const& tab_size = 42;

int main(void)
{
    double tab0[tab_size];
    return 0;
}
```

Skąd wiadomo, że wartość liczby niezerowych bitów liczby naturalnej obliczana jest w trakcie kompilacji?

```
constexpr int number_of_ones(int n)
{
    return n == 0 ? 0 : ((n&1) + number_of_ones(n >> 1));
}

int main(void)
{
    double tab1[number_of_ones(9)];
    return 0;
}
```

Ograniczenia funkcji zadeklarowanych jako stałe:

- brak definicji zmiennych oraz nowych typów
- pojedyncza instrukcja `return`
- gwarancja że po podstawieniu wartości parametrów wynikiem jest wyrażenie o stałej wartości

<https://en.cppreference.com/w/cpp/language/constexpr>

Wyrażenia stałe z wykorzystaniem klas i obiektów.

```
class Factorial {
    int value_;
    constexpr int factorial(int n) {
        return n <= 1 ? 1 : (n * factorial(n - 1));
    }
public:
    constexpr Factorial(int n) : value_(factorial(n)) {}
    constexpr int value(void) { return value_; }
};

int main(void)
{
    double tab2[Factorial(2).value()];
    Factorial f(2);
    // double tab3[f.value()]; // Uwaga blad

    return 0;
}
```

## static - statyczne pola klas

W języku C++ istnieje możliwość oznaczenia pola wspólnego dla wszystkich obiektów danej klasy. Pola takie określane są mianem statycznych. Każdy obiekt klasy współdzieli wartości pól statycznych. Odwołanie do pól statycznych danej klasy może odbywać się nawet wtedy gdy nie istnieją, żadne obiekty tego typu.

Pola statyczne (`static`) o ile nie są stałe (`const`) muszą być inicjowane poza definicją klasy. Pola statyczne dekladowane w miejscu (`inline`) mogą być inicjowane w ciele klasy. Nie mogą być oznaczone jako `mutable`

Do pól statycznych, o ile są w zakresie publicznym, można się odwoływać używając nazwy klasy:

```
auto x = nazwaKlasy::nazwaZmiennejStatycznej;
```



## static - statyczne pola klas

```
class MaxNumberFilesExceed {};  
  
class File{  
    const static unsigned int max_opened_files_ = 2;  
    static unsigned int opened_files_;  
    //inline static unsigned int opened_files_ = 0; // C++17  
    /**/  
public:  
    File(**/) /*:*/ /**/ {  
        if (max_opened_files_ == opened_files_)  
            throw MaxNumberFilesExceed();  
        ++opened_files_;  
    }  
  
    ~File(void) {  
        --opened_files_;  
    }  
};  
  
unsigned int File::opened_files_ = 0;
```

## static - statyczne pola klas

```
int main(void)
{
    File *p_f1(nullptr), *p_f2(nullptr), *p_f3(nullptr);

    try {
        p_f1 = new File(); // OK
        p_f2 = new File(); // OK
        p_f3 = new File(); // Przekroczona dopuszczalna
                           // ilosc obiektow
    }
    catch (MaxNumberFilesExceed&)
    {
        // reakcja na zaistniały bład
        delete p_f1;
    }

    p_f3 = new File(); // OK
    delete p_f2;
    delete p_f3;

    return 0;
}
```

## static - statyczne metody klas

W języku C++ istnieją również metody statyczne. Nie są one powiązane z żadnym obiektem danej klasy - wewnątrz metody statycznej niedostępny jest wskaźnik `this`. Metody statyczne bezpośrednio mogą się odwoływać jedynie do pól statycznych. Metody statyczne nie mogą być wirtualne (`virtual`), stałe (`const`) oraz ulotne (`volatile`).

Metody statyczne, o ile są w zakresie publicznym, mogą być wywoływane poprzez nazwę klasy:

```
auto x = nazwaKlasy::nazwaMetodyStatycznej(/* ... */);
```

## static - "Nazwane konstruktory"

Technika na bardziej intuicyjną i bezpieczniejszą konstrukcję obiektów.

### Problem

```
class Point {
    double x, y;
public:
    Point(double x, double y)
        : x(x), y(y) {}

    //Point(double r, double alpha)
    //    : x(r*std::cos(alpha)), y(r*std::sin(alpha)) {}
};
```

Dlaczego po usunięciu znaku komentarza kod klasy Point się nie skompiluje? W jaki sposób można rozwiązać problem?

## static - "Nazwane konstruktory"

```
class Point {
    double x, y;
    Point(double x, double y) : x(x), y(y) {}

public:
    static Point cartesian(double x, double y) {
        return Point(x, y); }
    static Point polar(double r, double alpha) {
        return Point(r*std::cos(alpha), r*std::sin(alpha)); }
};

int main(void)
{
    auto p1 = Point::cartesian(1, 1);
    auto p2 = Point::polar(1, 3.14);

    return 0;
}
```

Dlaczego konstruktor parametryczny zadeklarowany został w zakresie prywatnym?

## Problem

Projektowana klasa musi spełniać następujące warunki:

- w każdej chwili działania programu istnieje maksymalnie jedna instancja danej klasy,
- klasa umożliwia możliwie prosty dostęp do tej instancji,
- dana klasa jest odpowiedzialna za tworzenie i niszczenie tej instancji

Do czego można wykorzystać klasę o takich właściwościach?

Rozwiązanie tego problemu podają:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:  
Inżynieria oprogramowania: Wzorce projektowe (Wyd. II).  
Warszawa: WNT, 2008

```
class MeasurementDevice {
    MeasurementDevice(void) {}
    /* ... */
public:
    static MeasurementDevice& instance() {
        static MeasurementDevice instance_;
        return instance_;
    }
    /* ... */
    void turnOn(void) { /* ... */ }
    void turnOff(void) { /* ... */ }
};

int main(void)
{
    MeasurementDevice::instance().turnOn();
    MeasurementDevice::instance().turnOff();

    return 0;
}
```

# operatory i sposoby ich przeciążania

Język C++ umożliwia przeciążanie większości operatorów. Przeciążanie operatorów nie jest mechanizmem stricte obiektowym i swobodnie można się bez niego obejść. Jest to tak zwane “lukrowanie składni”, które na celu jedynie zwiększenie czytelności pisanego kodu.

```
class SimpleVector{ /* ... */
public:
    double& operator[](unsigned int i);
    const double& operator[](unsigned int i) const;
    double& at(unsigned int i);
    const double& at(unsigned int i) const;
    /* ... */
};
```

```
SimpleVector sv(N);
sv[1] = 1;
sv.at(1) = 1;
```



## Ograniczenia w przeciążaniu operatorów

- nie wszystkie operatory dostępne w języku C++ mogą być przeciążywane
- nie można tworzyć nowych operatorów
- nie można przeciążać operatorów dla typów podstawowych
- przynajmniej jeden z parametrów operatora musi być zdefiniowany przez użytkownika
- przeciążane operatory nie mogą mieć parametrów domniemanych
- niektóre operatory mogą być zdefiniowane zarówno w ciele klasy jako jej metoda jak i poza klasą jako zwykła funkcja

[https://en.wikipedia.org/wiki/Operators\\_in\\_C\\_and\\_C%2B%2B](https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B)

Operatory wywoływane są na rzecz lewego argumentu. Jeżeli programista nie ma dostępu do klasy dla której chce zdefiniować operator, musi go zdefiniować jako wolną funkcję, np:

```
std::ostream& operator<<(std::ostream& out,  
                        const SimpleVector& v);
```

Następujące operatory muszą być zdefiniowane jako metody klasy:

- przypisania =, R& K::operator =(S b);
- indeksowania [], R& K::operator [] (S b);
- wywołania (), R K::operator () (S a, T b, ...);
- selekcji składowej ->, R\* K::operator ->();

Operatory należy definiować wtedy gdy ich definicja jest naturalna. Jeżeli implementowana klasa reprezentuje jakiś byt matematyczny to zazwyczaj dobrze jest zdefiniować operatory charakterystyczne dla danego bytu. Na przykład dla klasy reprezentującej liczby zespolone wszystkie operatory arytmetyczne mają sens i są jak najbardziej pożądane. Natomiast operator indeksowania, mimo, że można sobie wyobrazić jego zastosowanie, nie jest w tym przypadku intuicyjny i należy go raczej nie definiować.

Odwrotnie jest natomiast w przypadku obiektu reprezentującego kolekcję innych obiektów. Warto się zastanowić nad implementowaniem operatora indeksowania, zwłaszcza jeżeli dany typ przypomina działaniem tablicę. Argumentem operatora indeksowania może być dowolny typ, który w tym przypadku ma sens.

Operatory przypisania i przeniesienia, jeśli nie są zdefiniowane przez użytkownika, podobnie jak konstruktor kopiujący i przenoszący, zostaną wygenerowane przez kompilator.

```
class SimpleVector{
/* ... */
public:
/* ... */
    SimpleVector& operator=(const SimpleVector& v);
    SimpleVector& operator=(SimpleVector&& v);
};
```

# operatory i sposoby ich przeciążania

```
SimpleVector& SimpleVector::operator=(const SimpleVector& v) {  
    delete [] data_ ;  
    data_ = new double[v.size_] ;  
    size_ = v.size_ ;  
    for (unsigned int i = 0; i < size_ ; ++i) {  
        data_[i] = v.data_[i] ;  
    }  
    return *this ;  
}  
  
SimpleVector& SimpleVector::operator=(SimpleVector&& v) {  
    delete [] data_ ;  
    data_ = v.data_ ;  
    v.data_ = nullptr ;  
    size_ = v.size_ ;  
    v.size_ = 0 ;  
    return *this ;  
}
```

```
int main(void){  
    SimpleVector v1(5);  
    SimpleVector v2(3);  
    v2 = v1;  
    v1 = std::move(v2);  
    SimpleVector v3 = v1;  
    return 0;  
}
```

## Wskazówki

- zdefiniowanie własnego operatora logicznego wyłącza mechanizm “short-circuit evaluation”
- przeciążony operator (->) musi zwracać wskaźnik lub obiekt (przez wartość albo referencję) który także implementuje operator (->)
- dobrze jest implementować całe grupy operatorów:
  - (+), (-) (+=), (-=)
  - (\*), (/), (\*=), (/=)
  - (==), (!=), (<), (>), (<=), (>=)
- istnieje możliwość przeciążania operatorów (**new**) i (**delete**) i ich odpowiedników tablicowych, musi to być jednak uzasadnione w kontekście całego projektu

## std::string - napisy w stylu C++

Język C++ udostępnia wygodny typ do obsługi ciągów znakowych  
- `std::string`

[https://en.cppreference.com/w/cpp/string/basic\\_string](https://en.cppreference.com/w/cpp/string/basic_string)

<http://www.cplusplus.com/reference/string/string/>

```
int main(void){
    std::string s1("Napis_1");
    std::string s2 = "Napis_2";
    std::string s3 = s1;
    std::cout << s1 << " " << s2 << " " << s3 << std::endl;

    return 0;
}
```

Klasa `std::string` udostępnia kilka metod pozwalających na wygodne manipulowanie ciągami znaków, m.in.:

- `length` - zwraca długość napisu
- `capacity` - zwraca rozmiar przydzielonej pamięci
- `clear` - czyści napis
- `find` - przeszukuje napis w celu odnalezienia wzorca

```
std::string s_n("to_jest_napis");  
std::string s_f("jest");  
auto i = s_n.find(s_f);
```

- `c_str` - zwraca napis w stylu języka C
- `operator[]` - zwraca znak znajdujący się na wskazanym miejscu
- `operator==` i inne - porównywanie dwóch napisów



Moduł `std::string` udostępnia również kilka przydatnych funkcji, np.:

- `stoi`, `ltoi`, `stof`, `ltod` - konwersja napisu do wartości liczbowej

```
std::string s("11");  
auto x = stoi(s);
```

- `operator+` - łączy dwa napisy (konkatenacja)

```
std::string s1("Napis_1");  
std::string s2 = "Napis_2";  
auto s12 = s1 + s2;
```

- `operator>>` - wczytywanie napisu ze strumienia
- `operator<<` - wypisywanie napisu do strumienia

