Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 6

12 listopada 2020

Treść dzisiejszego wykładu

- sposoby przekazywania argumentów do funkcji oraz metod
- sposoby zwracania wyników przez funkcje oraz metody
- wskaźnik `this`
- wyrażenia stałe – `constexpr`
- pola i metody statyczne – `static`
- singleton
- przeładwowywanie operatorów
- zaprzyjaźnianie - – `friend`
- przypomnienie

sposoby przekazywania obiektów do funkcji

przez wartość

```
void funkcja(Typ1 zmienna1, Typ2 zmienna2);
```

przez adres zmienne

```
void funkcja(Typ1* w_zmienna1, Typ2* w_zmienna2);
```

przez referencję do zmiennej

```
void funkcja(Typ1& zmienna1, Typ2& w_zmienna2);
```

sposoby przekazywania obiektów do funkcji

Który ze sposobów przekazywania zmiennej do funkcji będzie najlepszy dla poszczególnych typów?

typy proste

```
char, int, float, double
```

typ zdefiniowany przez użytkownika

```
struct F {
    double data[1000];
};
```

sposoby zwracania wyniku przez funkcję

przez wartość

```
Typ0 funkcja(Typ1 zmienna1, ...) {
    Typ0 zmienna;
    /**/
    return zmienna; }
```

sposoby zwracania wyniku przez funkcję

przez wskaźnik do dynamicznie alokowanego obiektu

```
Typ0* funkcja(Typ1 zmienna1, ...) {
    auto wsk = new Typ0;
    /**/
    return wsk;
}
```

Uwaga

Zwracanie adresu dynamicznie alokowanego obiektu przy zwiększonej uwadze może czasami być stosowane, jeżeli rzeczywiście jest niezbędne. Często natomiast zwracany jest adres zmiennej globalnej. Zwracanie adresu zmiennej automatycznej jest **zawsze** złym rozwiązaniem.

sposoby zwracania wyniku przez funkcję

Metoda klasy może zwracać wskaźnik należący do tej klasy, lub adres jakiegoś innego obiektu składowego. Należy uważać, żeby obiekt nie został zniszczony wcześniej niż wskazanie na jego składową.

```
struct F {
    double* data(void) { return data_; }
private:
    double data_[1000];
};
int main(void){
    double* wsk;
    {
        F f;
        wsk = f.data();
    }
    wsk[0]; //?
    return 0;
}
```

sposoby zwracania wyniku przez funkcję

przez przekazanie adresu obiektu do wypełnienia

```
void funkcja(Typ0* zmienna0, Typ1 zmienna1, ...) {
    zmienna0->wykonajOperacje1(zmienna1, ...);
}
```

przez przekazanie obiektu do wypełnienia

```
void funkcja(Typ0& zmienna0, Typ1 zmienna1, ...) {
    zmienna0.wykonajOperacje1(zmienna1, ...);
}
```

Uwaga

Ze względu na nieczytelności stosowanie argumentów wyjściowych nie jest zalecanym sposobem zwracania wyniku przez funkcję. Konieczność stosowania tego typu konstrukcji zazwyczaj wynika ze złego projektu klasy / modułu i należy spróbować go poprawić.

specjalny wskaźnik this

Wskaźnik this wskazuje na obiekt, na rzecz którego wywołana została metoda danej klasy. Dostępny jest jedynie w zasięgu lokalnym metod niestatycznych.

```
struct F {
    F copy(void) { return *this; }
};

struct B {
    B copy(void) { return *this; }
};
```

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;
}
```

constexpr - stałe wyrażenia w C++

```
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;
}
```

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

`constexpr` - stałe wyrażenia w C++

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

constexpr - stałe wyrażenia w C++

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 deklarowane w miejscu (`inline`) modą 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;
    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_;
    }
};
```

static - statyczne pola klas

```
int main(void)
{
    try
    {
        File f1;      // OK
        File f2;      // OK
        File f3;      // Przekroczona dopuszczalna ilosc obiektow
    }
    catch (MaxNumberFilesExceed&)
    {
        // reakcja na zaistnialy blad nie jest konieczna !
    }

    File f3;          // OK
    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 pozwalająca 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::cos(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?

rozwiązanie z wykorzystaniem typów

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

public:
    Point(XCoordinate&, YCoordinate&);
    Point(Radius&, Angle&);

    /* ... */
};

int main(void){
    auto p1 = Point(XCoordinate(1),YCoordinate(1));
    auto p2 = Point(Radius(1), Angle(45));
    return 0;
}
```

W jakich jednostach wyrażona jest wartość kąta ?

Problem

Projektowana klasa musi spełniać następujące warunki:

- w każdej chwili działania programu istnieje maksymalnie jedna instancja danej klasy,
- klasa umożliwiała 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

singleton

```
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 przeładowywane
- 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

operatory i sposoby ich przeciążania

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 i sposoby ich przeciążania

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 i sposoby ich przeciążania

Operator przypisania i przenoszący operator przypisania, jeśli nie są jest 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);
};
```

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

zaprzyjaźnianie - `friend`

- Funkcja, która jest przyjacielem klasy, ma dostęp do wszystkich jej prywatnych i chronionych składowych.
- To klasa określa, które funkcje są jej przyjaciółmi.
- Deklaracja przyjaźni może się pojawić w dowolnej sekcji i jest poprzedzona słowem kluczowym `friend`.
- Jedna funkcja może się przyjaźnić z kilkoma klasami. Funkcją zaprzyjaźnioną może być funkcja składowa z innej klasy.
- Możemy w klasie zadeklarować przyjaźń z inną klasą, co oznacza, że każda metoda tej innej klasy jest zaprzyjaźniona z klasą pierwotną.

zaprzyjaźnianie - friend

```
class Complex{
public:
    Complex(double x, double y) : x(x), y(y) {}
    friend std::ostream& operator<<(std::ostream out&,
                                    const Complex& a);
private:
    double x = 0, y = 0;
};

std::ostream& operator<<(std::ostream out&,
                         const Complex& a) {
    out << a.x << " + j" << a.y;
    return out;
}
```

przypomnienie

Istotę programowania obiektowego stanowią obiekty i ich wzajemne relacje w trakcie działania programu.
Klasa stanowi opis tego jak obiekty są tworzone i niszczone, jakie mogą mieć stany wewnętrzne oraz w jaki sposób współpracują z innymi obiektami obecnymi w programie.

Klasa

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

Obiekt

```
int main(void)
{
    Point p1(1, 1);
    Point p2 = p1 + p1;
    /*...*/
    return 0;
}
```

przypomnienie

W języku C++ istnieje podział na trzy główne typy obiektów:

- typ zwykły,
- typ referencyjny,
- typ wskaźnikowy.

Typ referencyjny może być tak zwaną l-referencją lub r-referencją.

```
int main(void)
{
    Point p(1, 1);          // obiekt
    Point* wp = &p;         // wskaznik
    Point& rp = p;          // l-referencja
    std::move(p);           // r-referencja
    return 0;
}
```

przypomnienie

W języku C++ istnieje możliwość definiowania wolnych funkcji, tj. funkcji niezwiązanych z żadnym obiektem bądź klasą.
Funkcje te mogą przyjmować postać zwykłą bądź postać operatora.
Funkcje mogą przyjmować dowolną ilość argumentów dowolnego typu oraz zwracać tylko jedną wartość. Mogą także mieć takie same nazwy (o ile różnią się typem argumentów) - polimorfizm statyczny.

```
void print_info(const char* str)
{
    std::cout << str << std::endl;
}
```

```
Point operator+(const Point& lhs, const Point& rhs)
{
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

przypomnienie

Wszystkie metody niestatyczne są to **funkcje**, wywoływane ma rzecz danego obiektu, w których obiekt ten jest niejawnym argumentem danym przez wskaźnik `this`.
Metody mogą mieć modyfikator `const`, który informuje kompilator (użytkownika), że dana metoda nie zmienia **stanu wewnętrznego** obiektu.
Metody statyczne mogą modyfikować jedynie statyczne pola klasy, nie są związane z żadnym konkretnym obiektem danego typu.

```
class TrainBuilder
{
    Locomotive buildLocomotive(/*...*/) const;
    TrainWagon buildTrainWagon(/*...*/) const;
public:
    TrainBuilder(/*...*/);
    Train build(const TrainSpec& spec) const;
};
```

przypomnienie

Wszystkie pola (inaczej atrybuty) klasy pomijając pola statyczne (`static`) oraz modyfikowalne (`mutable`) opisują stan wewnętrzny obiektu.
Podobnie jak metody, pola również mogą mieć atrybut stałości (`const`)

```
class MemoryChunk
{
private:
    struct MemorySize { size_t size; };

    const double * memory_;
    const MemorySize size_;

public:
    MemoryChunk(const double* mem, const size_t size);
    const double* data(void) const;
    const double& operator[](size_t index) const;
};
```