Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 5

5 listopada 2020

treść dzisiejszego wykładu

- proces budowania programu
- przykład - wielomian w ciele liczb rzeczywistych
- zarządzanie pamięcią na stosie - zmienne automatyczne
- zarządzanie pamięcią na stercie - operatory `new` oraz `delete`
- zarządzanie zasobami - RAII
- kopiowanie i przenoszenie - przypomnienie
- szablony funkcji
- szablony klas
- szablony metod
- przykład - implementacja metody Newtona - Raphsona.

budowanie programu - organizacja kodu w plikach

W języku C++ klasy implementowane są zazwyczaj w dwóch rodzajach plików:

- pliki nagłówkowe (ang. *header*) z rozszerzeniem `.hpp` lub `.h`
- pliki źródłowe (ang. *source*) z rozszerzeniem `.cpp`, `.c` lub `.cc`

Plik nagłówkowy stanowi interfejs danego modułu (moduł może zawierać więcej niż jedną klasę), który jest dołączany (dyrektywa `#include`) do innych modułów. Wszystko co zawiera plik nagłówkowy jest widoczne dla innych użytkowników i jest niejako publiczne (nawet jeśli zadeklarowane w obszarze prywatnym - `private`).
Plik źródłowy może być kompilowany do pliku obiektowego (zawierającego kod binarny) a następnie dołączany do innych jednostek translacji w trakcie konsolidacji w celu utworzenia pliku wykonywalnego. Pliki obiektowe, biblioteki statyczne i dynamiczne nie są czytelne dla ich użytkowników - zawierają kod maszynowy.

budowanie programu - kompilacja i linkowanie

Budowanie plików wykonywalnych, tj. takich, które mogą być uruchamiane bezpośrednio w środowisku systemu operacyjnego, składa się z dwóch głównych etapów:

- kompilacji, w trakcie której powstają pliki zawierające kod obiektowy
- konsolidacji, w trakcie której kod zawarty w plikach obiektowych i bibliotekach statycznych jest łączony w plik wykonywalny.

Proces budowania może być wykonywany ręcznie lub automatycznie. Ręczne budowanie dużych projektów może być niewygodne bądź też niemożliwe. W tym celu powstały narzędzia automatycznej budowy oprogramowania. Wiele środowisk programistycznych (IDE), implementuje swój własny sposób automatycznej budowy oprogramowania.

budowanie programu - kompilacja i linkowanie

Obecnie proces kompilacji zajmuje znacznie więcej czasu procesora niż proces linkowania (uwaga: nie zawsze tak było i niekoniecznie tak musi być w przyszłości).
Narzędzia automatycznej budowy optymalizują ten czas i dla każdej jednostki translacji zbierają informacje czy uległa ona zmianie od ostatniego uruchomienia kompilacji. Jeżeli dany moduł oraz moduły, od których zależy nie uległy zmianie od ostatniej kompilacji proces ten jest pomijany dla danej jednostki translacji.

przykład - wielomian w ciele liczb rzeczywistych

Problem

Należy przygotować klasę reprezentującą wielomian w ciele liczb rzeczywistych. Należy umożliwić:

- ustawienie współczynnika dowolnego stopnia dla danego wielomianu,
- pobieranie wartości współczynnika zadanego stopnia,
- wyznaczenie aktualnego stopnia wielomianu,
- obliczenie wartości wielomiany dla danego argumentu.

Dodatkowo należy uwzględnić:

- prawidłowe zarządzanie pamięcią,
- prawidłowe operacje kopiowania oraz przenoszenia.

przykład - wielomian w ciele liczb rzeczywistych

Implementacja naiwna

```
class Polynomial
{
    static const size_t MAX_DEGREE = 5;

public:
    Polynomial();
    void setCoeff(size_t n, float coeff);

    float value(float arg) const;
    float coeff(size_t n) const;
    size_t degree() const;
private:
    float coeffs_[MAX_DEGREE+1];
};
```

Jakie są zalety a jakie wady tej implementacji ?

zarządzanie pamięcią - zmienne automatyczne

```
Typ zmienna;
```

- dotyczą kontekstu
- po opuszczeniu danego kontekstu są usuwane
- stosunkowo łatwo jest nimi zarządzać
- umieszczane są na stosie

```
struct A {
    A(void) { std::cout << "A␣c-tor" << std::endl; }
    ~A(void) { std::cout << "A␣d-tor" << std::endl; }
};

int main(void){
    A f1;
    return 0;
}
```

zarządzanie pamięcią - zmienne automatyczne

```
struct B {
    B(void) { std::cout << "B␣c-tor" << std::endl; }
    ~B(void) { std::cout << "B␣d-tor" << std::endl; }
};
```

```
int main(void){
    A a1;
    B b1;
    {
        A a2;
        {
            B b2;
        }
        A a3;
    }
    return 0;
}
```

Ile będzie wywołań konstruktorów i destruktorów? W jakiej kolejności będą się one wywoływane?

zarządzanie pamięcią - operatory new oraz delete

```
Typ* w_zmienna = new Typ;
```

```
auto w_zmienna = new Typ;
```

- zmienne dynamiczne tworzone są na stercie
- nie są związane z kontekstem
- po opuszczeniu danego kontekstu usuwane są **jedynie** wskaźniki odnoszące się do danej zmiennej
- stosunkowo łatwo jest doprowadzić do wycieku pamięci
- używamy wtedy gdy nie nie da się utworzyć zmiennej automatycznej

zarządzanie pamięcią - operatory new oraz delete

Język C++ dostarcza operatory przydzielania pamięci na stercie (new) oraz jej zwalniania (delete) w wersji skalarnej oraz tablicowej

```
auto w_zmienna = new Typ(parametr1, parametr2, ...);
delete w_zmienna;

auto w_zmienna_tablicowa = new Typ[wielkoscTablicy];
delete[] w_zmienna_tablicowa;
```

- w przypadku niepowodzenia alokacji zgłaszany jest wyjątek **bad_alloc** - o wyjątkach w dalszej części wykładu
- wielkość tworzonej tablicy nie musi być znana w czasie kompilacji
- bardzo ważne aby używać poprawnego operatora zwalniającego pamięć - adekwatnego do sposoby jej alokowania
- nie można utworzyć tablicy obiektów, które nie posiadają domyślnego konstruktora

zarządzanie pamięcią - operatory new oraz delete

```
int main(void){
    auto a1 = new A;
    auto b1 = new B;
    {
        auto a2 = new A;
        {
            auto b2 = new B;
        }
    }
    return 0;
}
```

Ile będzie wywołań konstruktorów i destruktorów? W jakiej kolejności będą się one wywoływane?

Delikatnie poprawiona wersja kodu.

```
int main(void){
    auto a1 = new A;
    auto b1 = new B;
    {
        auto a2 = new A;
        {
            auto b2 = new B;
            \*...*\
            delete b2;
        }
        delete a2;
    }

    delete b1;
    delete a1;
    return 0;
}
```

Poprawna wersja kodu.

```
int main(void){
    A_ptr a1;
    B_ptr b1;
    {
        A_ptr a2;
        {
            B_ptr b2;
            \*...*\
        }
    }

    return 0;
}
```

RAII

```
class A_ptr
{
public:
    A_ptr() : ptr_(new A) {}
    ~A_ptr() { delete ptr_; }
private:
    A * ptr_;
};
```

Inicjowanie przy pozyskaniu zasobu (ang. *Resource acquisition is initialization - RAII*) jest techniką pozwalającą na unikanie błędów związanych z zarządzaniem zasobami. Przydzielenie zasobu jest związane z konstrukcją obiektu, zaś jego zwolnienie z niszczeniem obiektu. Ponieważ standard gwarantuje, że obiekt jest niszczony gdy wychodzi poza swój zakres to wywoływany w tym momencie destruktor prawidłowo zwalnia zasób.

przykład - wielomian w ciele liczb rzeczywistych, c.d.

Implementacja (mniej)naiwna

```
class ConstSizePolynomial
{
public:
    ConstSizePolynomial(size_t degree = 5);

    void setCoeff(size_t n, float coeff);

    float value(float arg) const;
    float coeff(size_t n) const;
    size_t degree() const;

    ~ConstSizePolynomial();
private:
    size_t degree_;
    float * coeffs_;
};
```

Jakie są zalety a jakie wady tej implementacji ?

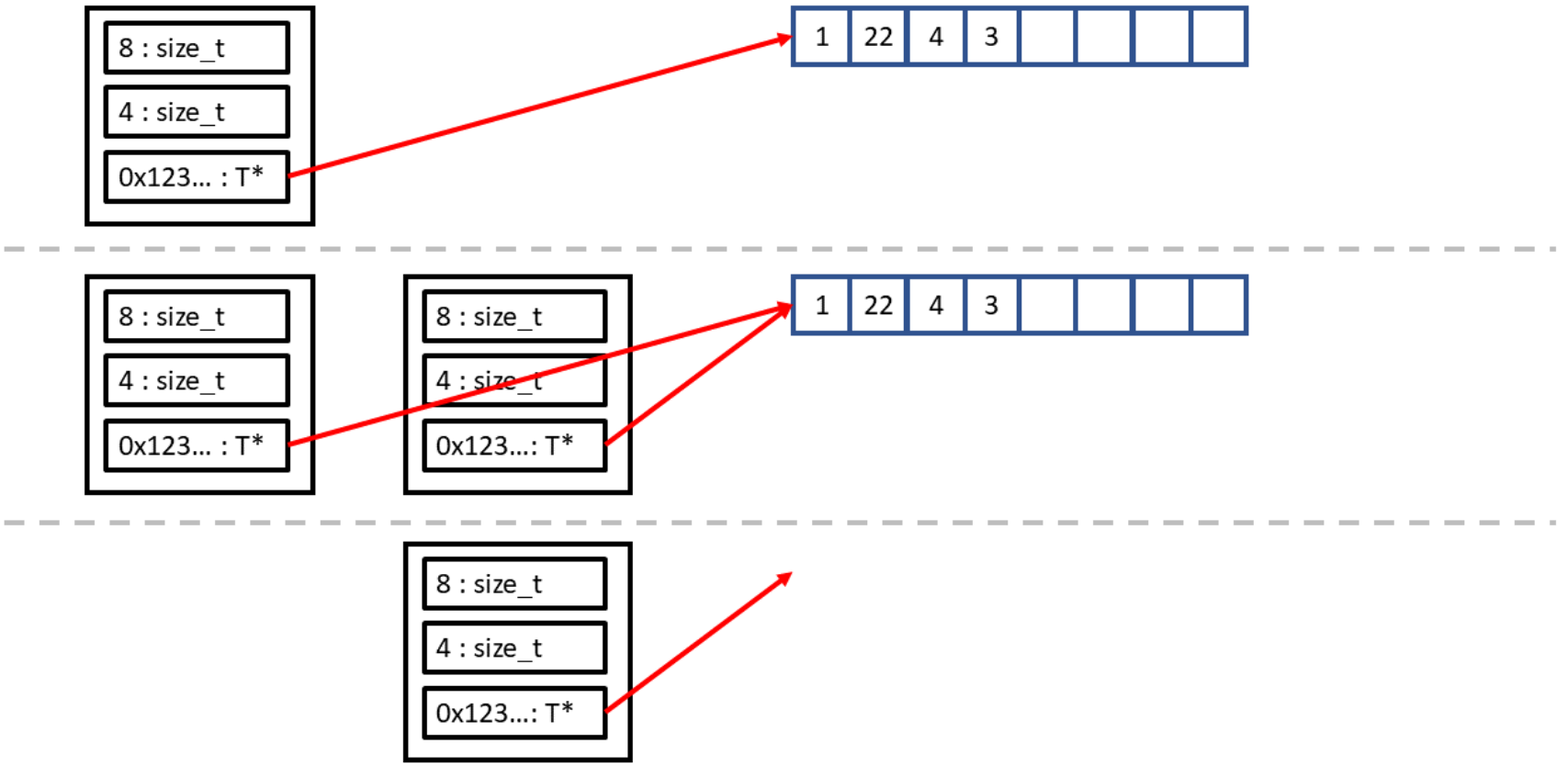
przykład - wielomian w ciele liczb rzeczywistych, c.d.

Gotowa implementacja

```
class Polynomial
{
public:
    Polynomial();

    void setCoeff(size_t n, float coeff);

    float value(float arg) const;
    float coeff(size_t n) const;
    size_t degree() const;
private:
    void _resize(size_t degree);

    size_t currentPolynomialDegree_;
    ConstSizePolynomial polynomial_;
};
```

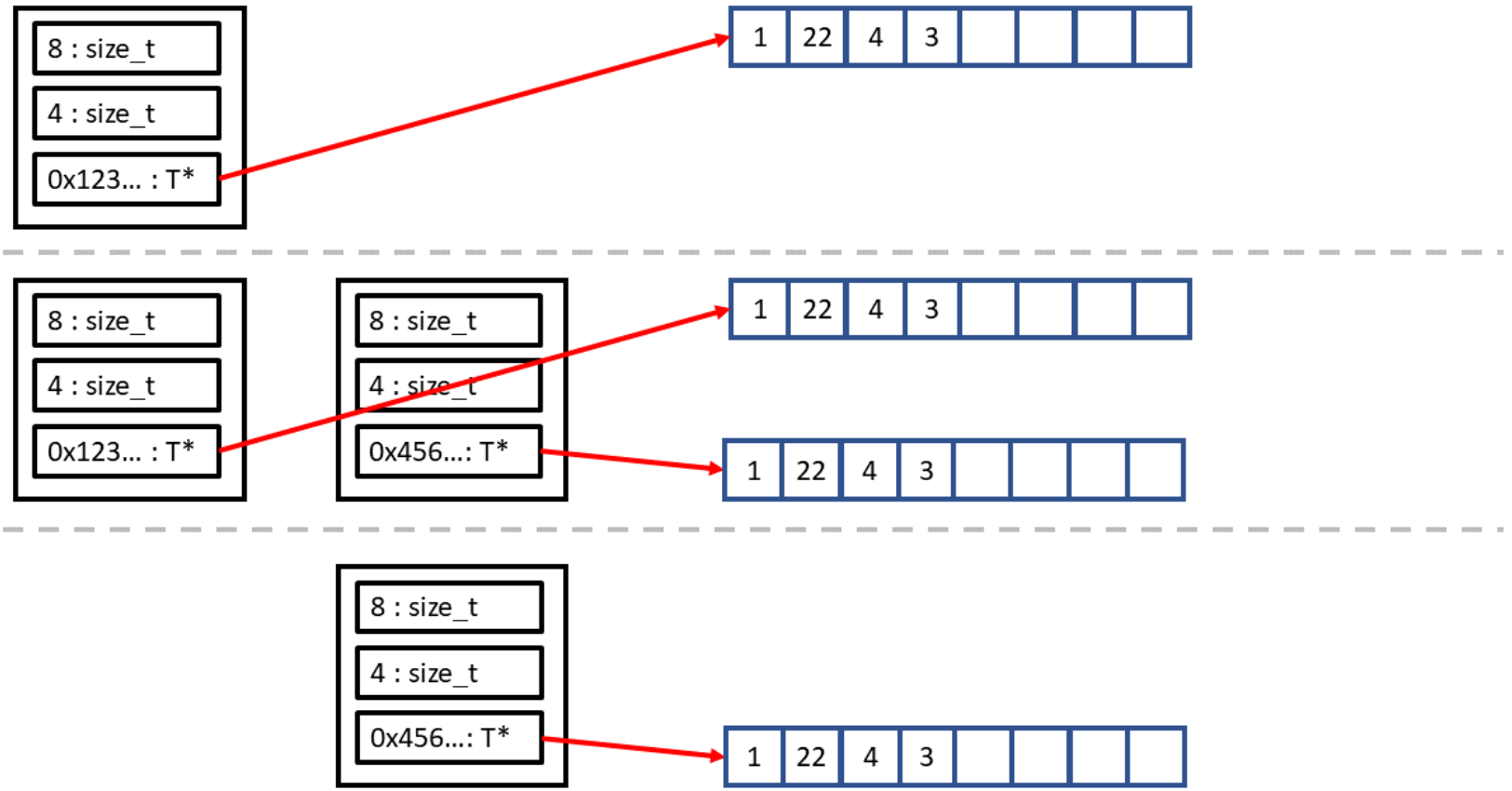
kopiowanie i "przenoszenie" obiektów - kopia "płytka"

- konstruktor kopiujacy i kopiujacy operator przypisania generowany automatycznie przez kompilator
- zły sposób kopiowania gdy obiekt alokuje pamięć na stercie
- niski koszt operacji



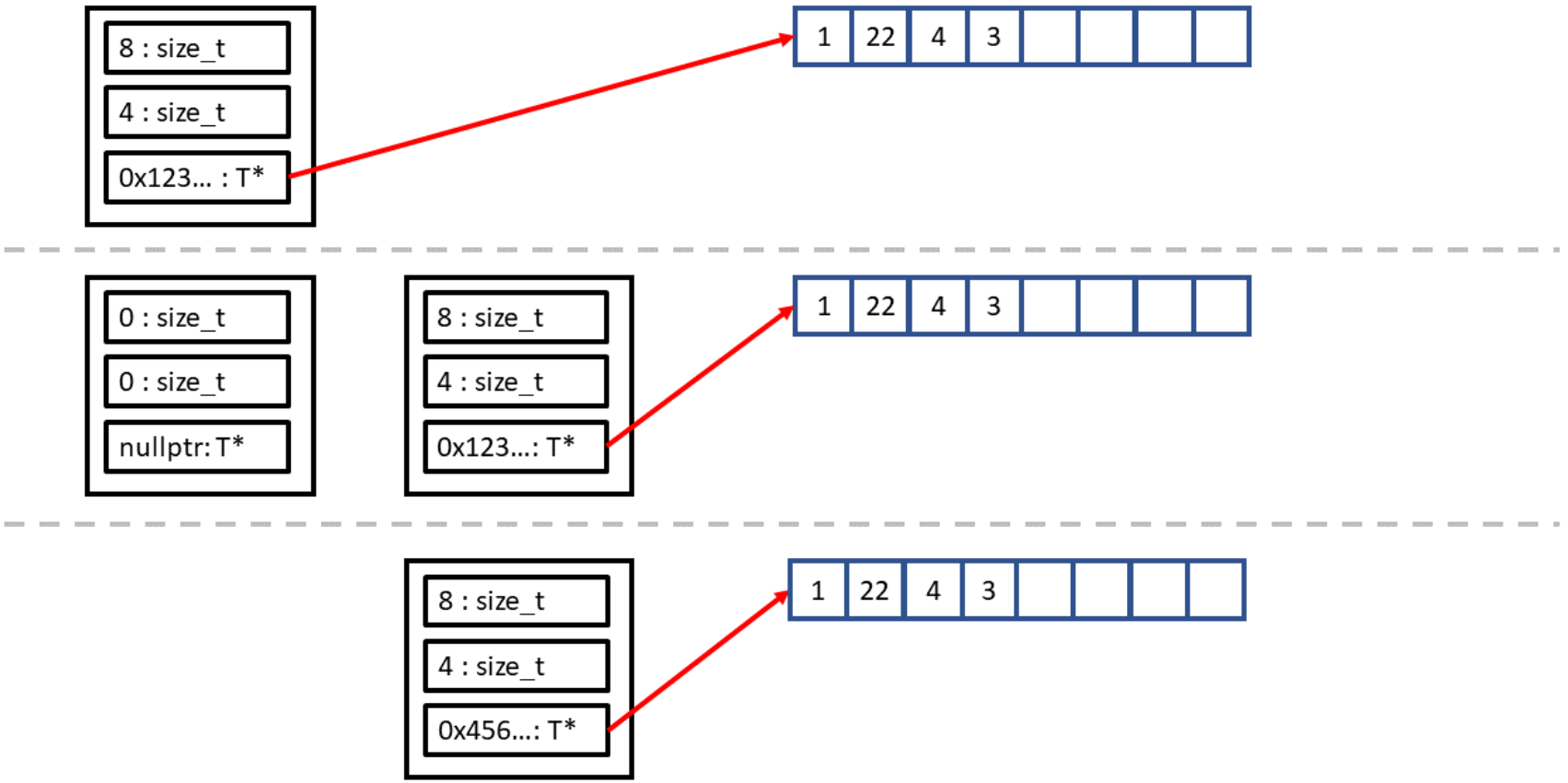
kopiowanie i "przenoszenie" obiektów - kopia "głęboka"

- konstruktor kopiujacy i kopiujacy operator należy napisać własnoręcznie
- właściwy sposób kopiowania gdy obiekt alokuje pamięć na stercie
- wysoki koszt operacji



kopiowanie i "przenoszenie" obiektów - przeniesienie

- konstruktor przenoszący oraz przenoszący operator przypisania należy napisać własnoręcznie
- właściwy sposób "kopiowania" gdy obiekt alokuje pamięć na stercie
- niski koszt operacji



przykład - wielomian w ciele liczb rzeczywistych, c.d.

```
class ConstSizePolynomial
{
public:
    ConstSizePolynomial(size_t degree = 5);
    ConstSizePolynomial(const ConstSizePolynomial&);
    ConstSizePolynomial(ConstSizePolynomial&&);

    ConstSizePolynomial& operator=(ConstSizePolynomial&&);

    ConstSizePolynomial&
        operator=(const ConstSizePolynomial&) = delete;


    /* ... */
    ~ConstSizePolynomial();
private:
    size_t degree_;
    float * coeffs_;
};
```

programowanie uogólnione

- jeden z paradygmatów programowania
- możliwe jest tworzenie kodu (algorytmu) bez znajomości typów danych na jakich będzie operował - o ile algorytm na to pozwala.
- możliwe jest opracowanie wzorców typów danych, które będą się zachowywać w różny sposób w zależności od lokalnej potrzeby użytkownika.

Paradygmat programowania to inaczej sposób w jaki programista patrzy na program i w jak steruje jego przepływami sterowania. W programowaniu obiektowym program traktowany jest jako współpracujące ze sobą obiekty.
Jakie są przykłady algorytmów, które są poprawne niezależnie od typu danych?

problem obliczania potęgi o wykładniku naturalnym

Potęgowanie za pomocą mnożenia

$$x^k = x \cdot x^{k-1} = \underbrace{x \cdot x \cdot x \cdot \ldots \cdot x}_{k} \tag{1}$$

Algorytm szybkiego potęgowania

$$x^n = \begin{cases} x \cdot (x^2)^{\frac{n-1}{2}}, & \text{jeśli } n \text{ jest nieparzyste} \\ (x^2)^{\frac{n}{2}}, & \text{jeśli } n \text{ jest parzyste} \end{cases} \tag{2}$$

problem obliczania potęgi o wykładniku naturalnym

```
double power(double x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}

int main(void)
{
    for( unsigned int i = 0; i < 10; ++i)
        std::cout << power(2, i) << std::endl;
    return 0;
}
```

Co musimy zrobić gdy chcemy podnosić do potęgi tylko liczby całkowite, reprezentowane na przykład przez typ `long long` ? Dlaczego chcielibyśmy móc to robić?

problem obliczania potęgi o wykładniku naturalnym

```
long long power(long long x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

problem obliczania potęgi o wykładniku naturalnym

```
long long power(long long x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

```
template<typename T>
T power(T x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

szablony w języku C++

Szablony (ang. *template*) w języku C++ oznaczane są przy pomocy słowa kluczowego `template`. Deklaracja lub definicja funkcji (lub klasy, metody) szablonowej poprzedza konstrukcja:

```
template<typename T>
```

lub:

```
template<class T>
```

W trakcie kompilacji, kod odpowiedniej funkcji zostaje wygenerowany na podstawie przekazanych argumentów szablonu. Każda funkcja wygenerowana w ten sposób posiada osobny kod binarny w programie i osobny adres w pamięci.

szablony funkcji

- Argumenty szablonu mogą być podawane w sposób jawny za nazwą wołanej funkcji, argument ten określa typ dla którego konkretyzowany jest szablon:

```
auto r1 = power<int>(2, 4);
```

- Jeśli kompilator na podstawie wywołania jest wstanie wydedukować typy argumentów szablonu, wtedy argumenty te mogą być pominięte w wywołaniu; Automatyczna dedukcja typów argumentów szablonu wyłącza automatyczne konwersje:

```
auto r2 = power(2, 4);
auto r3 = power(2.1, 4);
```

- Podanie argumentów w sposób jawny, z kolei umożliwia automatyczną konwersję typów:

```
auto r2 = power<double>(2, 4);
auto r3 = power<int>(2.1, 4);
```

szablony funkcji

- Automatyczna dedukcja parametrów szablonu jest możliwa jedynie wtedy, gdy parametry wywołania funkcji zależą od typów parametrów szablonu.
- Jeżeli parametr szablonu określa na przykład typ zwracany przez funkcję musi być on podany jawnie wraz z wywołaniem funkcji.
- Liczba parametrów szablonu nie jest w żaden sposób ograniczona.
- Próba konkretyzacji poprawnego szablonu w nieprawidłowymi typami argumentów, może generować błąd kompilacji

```
//auto r4 = power("napis", 4); // blad kompilacji
```

Dlaczego powyższa instrukcja jest nieprawidłowa a poniższa już tak:

```
auto r4 = power('n', 4);
```

szablony funkcji

W celu wywołania odpowiedniej funkcji kompilator postępuje według następujących kroków:

1. Poszukiwana jest funkcja o argumentach dokładnie pasujących do tych w wywołaniu
2. Poszukiwany jest wzorzec funkcji pozwalający na generację funkcji spełniającej warunek pierwszy
3. Poszukiwana jest funkcja przeciążona o pasujących argumentach
4. Zgłaszany jest błąd.

szablony klas

Podobnie jak w przypadku funkcji istnieje możliwość definiowania wzorców klas.

```
template<typename T>
class Polynomial
{
public:
    Polynomial();

    void setCoeff(size_t n, const T& coeff);

    T value(const T& arg) const;
    T coeff(size_t n) const;
    size_t degree() const;
private:
    void _resize(size_t degree);

    size_t currentPolynomialDegree_;
    ConstSizePolynomial<T> polynomial_;
};
```

szablony klas

W przypadku szablonu klasy nie istnieje możliwość automatycznej dedukcji argumentów szablonu. Należy podać je w sposób jawny.

```
int main(void){
    Polynomial<double> p1;
    Polynomial<float> p2;
};
```

Istnieje możliwość podania argumentów domyślnych

```
template<typename T = float>
class Polynomial {/*...*/};

int main(void){
    Polynomial<double> p1;
    Polynomial<> p2;
};
```

szablony klas

Parametrami szablonów mogą być również stałe niebędące typami (np. liczby całkowite, typ wyliczeniowy)

```
template<typename T, size_t N>
class Polynomial
{
public:
    Polynomial();
    void setCoeff(size_t n, float coeff);

    float value(float arg) const;
    float coeff(size_t n) const;
    size_t degree() const;
private:
    float coeffs_[N+1];
};
```

Uwaga. Każda nowa specjalizacja szablonu powoduje powstanie nowego typu danych. Typy te, o ile nie są zaprzyjaźnione, nic nie wiedzą o pozostałych specjalizacjach z danej rodziny.
W szczególności nie istnieje możliwość kopiowania danych pomiędzy obiektami różnych typów.

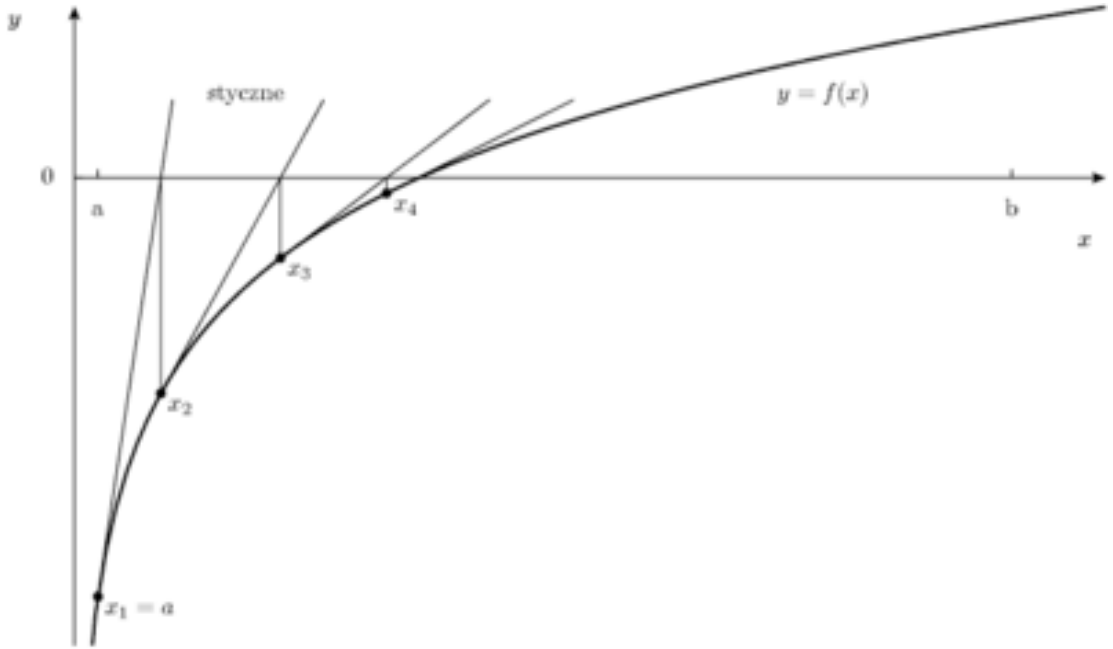
```
Polynomial<double, 10> p1;
Polynomial<double, 8> p2;
Polynomial<float, 10> p3;

Polynomial<double, 10> p4 = p1; // ok
// Polynomial<double, 10> p5 = p2; // blad
// Polynomial<double, 10> p6 = p3; // blad
```

przykład - implementacja metody Newtona

Problem

Należy przygotować klasę, która będzie udostępniała mechanizm znajdowania najbliższego miejsca zerowego metodą Newtona-Raphsona. Należy umożliwić:

- podanie jako argumentu dowolnej implementacji wielomianu spełniającej zadany interfejs,
- wybranie metody wyznaczania nachylenia funkcji w danym punkcie.



przykład - implementacja metody Newtona

newton_method.hpp

```
template<typename DerivativeMethod, typename T>
class NewtonMethod
{
public:
    NewtonMethod(const T& eps = static_cast<T>(1e-6));

    template<typename Polynomial>
    float solve(const Polynomial&, T = static_cast<T>(0));

private:
    T eps_;
};
```

definicja metod szablonu klas poza ciałem szablonu

newton_method.hpp

```
template<typename DerivativeMethod, typename T>
NewtonMethod<DerivativeMethod, T>::NewtonMethod(const T& eps)
: eps_(eps)
{
}
```

Definiując daną metodę poza ciałem szablonu klasy należy ponownie powtórzyć parametry szablonu:

```
template<typename DerivativeMethod, typename T>
```

Konieczne jest aby przy nazwie typu, dla którego definiowana jest metoda pojawiły się nazwy parametrów szablonu:

```
NewtonMethod<DerivativeMethod, T>::
```

Następnie podawane są parametry wywołania metody:

```
NewtonMethod(const T& eps)
```

metoda szablonowa

W języku C++ istnieje również możliwość definiowania metod szablonowych zarówno w klasach jak i wzorcach klas.

```
template<typename DerivativeMethod, typename T>
class NewtonMethod
{
public:
    /* ... */
    template<typename Polynomial>
    float solve(const Polynomial&, T = static_cast<T>(0));
    /* ... */
};

template<typename T>
template<typename U>
MemoryT<T>::MemoryT<T>(const MemoryT<U>& rhs)
   : MemoryT<T>({rhs.size_.size})
{
    for (size_t i = 0; i < size_.size; ++i)
        data_[i] = static_cast<T>(rhs.data_[i]);
}
```

metoda szablonowa

```
template<typename DerivativeMethod, typename T>
template<typename Polynomial>
float NewtonMethod<DerivativeMethod, T>::solve(
    const Polynomial& p, T arg)
{
    auto derivative_method = DerivativeMethod();

    while (p.value(arg) * p.value(arg) > eps_)
    {
        auto step =
            p.value(arg) / derivative_method.calculate(p, arg);
        arg = arg - step;

        if (step * step <= eps_)
        {
            break;
        }
    }

    return arg;
}
```

newton_method.hpp

```
#ifndef NEWTON_METHOD_HPP
#define NEWTON_METHOD_HPP

template<typename DerivativeMethod, typename T>
class NewtonMethod
{
public:
    NewtonMethod(const T& eps = static_cast<T>(1e-6));

    template<typename Polynomial>
    float solve(const Polynomial&, T = static_cast<T>(0));

private:
    T eps_;
};

#include "newton_method.ipp"

#endif // NEWTON_METHOD_HPP
```

newton_method.ipp

```
#include "newton_method.hpp"

template<typename DerivativeMethod, typename T>
NewtonMethod<DerivativeMethod, T>::NewtonMethod(const T& eps)
: eps_(eps)
{
}

template<typename DerivativeMethod, typename T>
template<typename Polynomial>
float NewtonMethod<DerivativeMethod, T>::solve(const Polynomial& p, T arg)
{
    auto derivative_method = DerivativeMethod();

    while (p.value(arg) * p.value(arg) > eps_)
    {
        auto step = p.value(arg) / derivative_method.calculate(p, arg);
        arg = arg - step;

        if (step * step <= eps_)
        {
            break;
        }
    }

    return arg;
}
```