Wstęp do programowania, część III

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

30 listopada 2011

Ograniczenia prostych typów danych

Proste typy danych z C++ mają ograniczone zastosowanie

1. Większość przedmiotów itp. ma wiele różnych własności (cech).
2. Nie można reprezentować wszystkich cech przedmiotu używając jednej liczby.
   - W zasadzie można użyć tablicy do opisu przedmiotu o wielu cechach, ale wtedy musimy je wszystkie reprezentować z pomocą tego samego typu danych.
   - Używając do tego celu tablicy musimy pamiętać jaka cecha odpowiada któremu indeksowi i na dłuższą metę nie jest to wygodne.
3. Potrzebna jest ogólna metoda reprezentowania złożonych obiektów.

Ograniczenia prostych typów danych

Proste typy danych z C++ mają ograniczone zastosowanie

1. Większość przedmiotów itp. ma wiele różnych własności (cech).
2. Nie można reprezentować wszystkich cech przedmiotu używając jednej liczby.
   - W zasadzie można użyć tablicy do opisu przedmiotu o wielu cechach, ale wtedy musimy je wszystkie reprezentować z pomocą tego samego typu danych.
   - Używając do tego celu tablicy musimy pamiętać jaka cecha odpowiada któremu indeksowi i na dłuższą metę nie jest to wygodne.
3. Potrzebna jest ogólna metoda reprezentowania złożonych obiektów.

Złożone typy danych pozwalają reprezentować złożone obiekty

Pozwalają tworzyć zmienne, którymi można posługiwać się tak, jak gdyby były zespołami zmiennych o różnych prostych typach danych.

Rodzaje złożonych typów danych w C++

W języku C++ mamy do dyspozycji trzy rodzaje złożonych typów danych:

- Struktury (*ang. structure*)
- Unie (*ang. union*)
- Klasy (*ang. class*)

Rodzaje złożonych typów danych w C++

W języku C++ mamy do dyspozycji trzy rodzaje złożonych typów danych:

- Struktury (*ang. structure*)
- Unie (*ang. union*)
- Klasy (*ang. class*)

Struktury

Obiekty o wielu cechach.

Unie

Obiekty, które w różnych sytuacjach mogą mieć różne własności.

Klasy

Obiekty o wielu cechach, które mogą oddziaływać z innymi obiektami.

Definiowanie struktur

Definicja struktury

1. Słowo kluczowe `struct`.
2. Nazwa struktury (identyfikator złożonego typu danych).
3. Otwierający nawias klamrowy.
4. Lista składników (*ang. member*) – dla struktur są to wyłącznie pola (*ang. field*).
5. Zamykający nawias klamrowy.
6. (Opcjonalnie) lista nazw zmiennych (o tym typie danych).
7. Średnik.

Definiowanie struktur

Definicja struktury

1. Słowo kluczowe `struct`.
2. Nazwa struktury (identyfikator złożonego typu danych).
3. Otwierający nawias klamrowy.
4. Lista składników (*ang. member*) – dla struktur są to wyłącznie pola (*ang. field*).
5. Zamykający nawias klamrowy.
6. (Opcjonalnie) lista nazw zmiennych (o tym typie danych).
7. Średnik.

Składniki struktury definiuje się tak, jak zmienne (bez wartości początkowych).

Przykład definicji struktury – z jądra Linuksa

```
struct wakeup_source {
    char                *name;
    struct list_head     entry;
    spinlock_t           lock;
    struct timer_list    timer;
    unsigned long        timer_expires;
    ktime_t              total_time;
    ktime_t              max_time;
    ktime_t              last_time;
    unsigned long        event_count;
    unsigned long        active_count;
    unsigned long        relax_count;
    unsigned long        hit_count;
    unsigned int         active:1;
};
```

Zmienne o typach danych będących strukturami

Deklaracja zmiennej zawiera słowo kluczowe `struct` i nazwę struktury, które łącznie pełnią rolę typu danych dla zmiennej, np.:

```
struct wakeup_source  zmienna;
struct wakeup_source *ws;  // Wskaźnik
```

Zmienne o typach danych będących strukturami

Deklaracja zmiennej zawiera słowo kluczowe `struct` i nazwę struktury, które łącznie pełnią rolę typu danych dla zmiennej, np.:

```
struct wakeup_source  zmienna;
struct wakeup_source *ws;  // Wskaźnik
```

Wykorzystanie pamięci

1. Zmienna o typie danych będącym strukturą zajmuje w pamięci co najmniej tyle miejsca, ile wynosi suma rozmiarów jej składników.
2. Składniki takiej zmiennej na ogół są rozmieszczane w pamięci zgodnie z kolejnością deklaracji i mogą być wyrównywane na granicy słowa o określonej długości (np. 32-bitowego).
3. `sizeof()` oblicza liczbę bajtów danych zajmowaną przez strukturę.

Bezpośrednie odwołania do pól struktury

Kropka oddziela nazwę zmiennej o złożonym typie danych od nazwy składnika, do którego odnosi się operacja, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w;

w.x = 1;
cout << w.x << endl;
```

Bezpośrednie odwołania do pól struktury

Kropka oddziela nazwę zmiennej o złożonym typie danych od nazwy składnika, do którego odnosi się operacja, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w;

w.x = 1;
cout << w.x << endl;
```

`w.x` pełni rolę zmiennej typu `double`.

Odwołania do pól struktury przez wskaźnik

Symbol -> oznacza, że operacja ma odnosić się do pola o danej nazwie, będącego składnikiem zmiennej (o złożonym typie danych) wskazywanej przez wskaźnik, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w, *wsk;

wsk = &w;
wsk->x = 1;
cout << wsk->x << endl;
```

Odwołania do pól struktury przez wskaźnik

Symbol -> oznacza, że operacja ma odnosić się do pola o danej nazwie, będącego składnikiem zmiennej (o złożonym typie danych) wskazywanej przez wskaźnik, np.:

```
struct wektor {
    double x;
    double y;
    double z;
} w, *wsk;

wsk = &w;
wsk->x = 1;
cout << wsk->x << endl;
```

`wsk->x` pełni rolę zmiennej typu `double`.

Początkowe wartości pól struktury

```
struct wektor {
    double x;
    double y;
    double z;
};

struct wektor w1 = { 1, 2, 3 };
struct wektor w2 = { .y = 0, .x = 1, .z = 2 };
struct wektor w3 = { .x = 1, .y = 2, };
```

Początkowe wartości pól struktury

```
struct wektor {
    double x;
    double y;
    double z;
};

struct wektor w1 = { 1, 2, 3 };
struct wektor w2 = { .y = 0, .x = 1, .z = 2 };
struct wektor w3 = { .x = 1, .y = 2, };
```

Pole z zmiennej w3 nie ma określonej wartości początkowej.

Listy

Lista (*ang. link list*)

Struktura danych (*ang. data structure*), w której w każdym elemencie przechowywana jest informacja o położeniu w pamięci następnego elementu (często także poprzedniego elementu) np. w postaci wskaźnika.

Listy

Lista (*ang. link list*)

Struktura danych (*ang. data structure*), w której w każdym elemencie przechowywana jest informacja o położeniu w pamięci następnego elementu (często także poprzedniego elementu) np. w postaci wskaźnika.

Przykład

```
struct element_listy {
    double r;                       // Dane "użyteczne".
    struct element_listy *nast;     // Następny element (adres).
};
```

Potrzebny jest dodatkowy wskaźnik zawierający adres pierwszego elementu listy.

Kolejki i stosy

Dwa najczęściej używane rodzaje list

Kolejka (*ang. queue*)

- Lista typu FIFO (*ang. first in, first out*).
- Nowe elementy dołącza się na końcu.

Stos (*ang. stack*)

- Lista typu LIFO (*ang. last in, first out*).
- Nowe elementy dołącza się na początku.

Kolejki i stosy

Dwa najczęściej używane rodzaje list

Kolejka (*ang. queue*)

- Lista typu FIFO (*ang. first in, first out*).
- Nowe elementy dołącza się na końcu.

Stos (*ang. stack*)

- Lista typu LIFO (*ang. last in, first out*).
- Nowe elementy dołącza się na początku.

Przykład zastosowania stosu

Użytkownik wprowadza dowolnie dużo liczb (nie wiemy z góry ile ich będzie), a program powinien wydrukować je w kolejności odwrotnej w stosunku do kolejności wprowadzania. Można umieszczać je kolejno na stosie, a później tylko odczytać jego zawartość.

Dodawanie elementu do listy

```
struct element_listy {
    double r;
    struct element_listy *nast;
};

struct element_listy *dodaj(struct element_listy *stos, double x)
{
    struct element_listy *el = new struct element_listy;
    if (!el)
        return stos;
    el->r = x;
    el->nast = stos;
    return el;
}
```

Usuwanie elementu listy

```
struct element_listy *zdejmij(struct element_listy *stos)
{
    struct element_listy *el;

    if (!stos)
        return NULL;
    el = stos->nast;
    delete stos;
    return el;
}
```

Wprowadzanie i drukowanie liczb

```
struct element_listy *stos = NULL;
for (;;) {
    double x;

    cout << "Podaj x (Ctrl-d oznacza koniec): ";
    cin >> x;
    if (!cin)
        break;
    stos = dodaj(stos, x);
}

while (stos) {
    cout << stos->r << endl;
    stos = zdejmij(stos);
}
```

Definiowanie i używanie unii

1. Definicja unii wygląda tak samo, jak definicja struktury o identycznych polach, tylko zamiast słowa kluczowego `struct` używa się słowa kluczowego `union`.
2. Różnica polega na tym, że wszystkie pola unii znajdują się pod tym samym adresem w pamięci.
3. Stąd wynika, że pól unii używa się zamiennie.
4. Odwołania do pól unii zapisuje się tak samo, jak odwołania do pól struktury.

Definiowanie i używanie unii

1. Definicja unii wygląda tak samo, jak definicja struktury o identycznych polach, tylko zamiast słowa kluczowego `struct` używa się słowa kluczowego `union`.
2. Różnica polega na tym, że wszystkie pola unii znajdują się pod tym samym adresem w pamięci.
3. Stąd wynika, że pól unii używa się zamiennie.
4. Odwołania do pól unii zapisuje się tak samo, jak odwołania do pól struktury.

Przykład

```
union bajty {
    int n;
    unsigned char b[sizeof(int)];
};
```

Przykład użycia unii

Drukowanie wartości bajtów składowych dla wartości typu int

```
union bajty {
    int n;
    unsigned char b[sizeof(int)];
};

union bajty x;

cout << "Podaj n: ";
cin >> x.n;

for (int i = 0; i < sizeof(int); i++)
    cout << "b[" << i << "] = 0x" << hex << (int)x.b[i] << endl;
```

Czym są klasy

Klasa (*ang. class*)

Złożony typ danych (analogiczny do struktury) z możliwością definiowania funkcji, zwanych metodami (*ang. method*), w których poszczególne pola wchodzące w skład zmiennej tego typu są traktowane tak, jak gdyby były zmiennymi lokalnymi.

Czym są klasy

Klasa (*ang. class*)

Złożony typ danych (analogiczny do struktury) z możliwością definiowania funkcji, zwanych metodami (*ang. method*), w których poszczególne pola wchodzące w skład zmiennej tego typu są traktowane tak, jak gdyby były zmiennymi lokalnymi.

Definiuje abstrakcyjną charakterystykę pewnej rzeczy (obiektu), określając jego cechy, reprezentowane przez pola, a także zachowanie się lub reakcje na bodźce, reprezentowane przez metody.

Czym są klasy

Klasa (*ang. class*)

Złożony typ danych (analogiczny do struktury) z możliwością definiowania funkcji, zwanych metodami (*ang. method*), w których poszczególne pola wchodzące w skład zmiennej tego typu są traktowane tak, jak gdyby były zmiennymi lokalnymi.

Definiuje abstrakcyjną charakterystykę pewnej rzeczy (obiektu), określając jego cechy, reprezentowane przez pola, a także zachowanie się lub reakcje na bodźce, reprezentowane przez metody.

Może być traktowana jako *model* (*ang. blueprint*) odzwierciedlający naturę czegoś.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o (złożonym) typie danych, który jest klasą (lub strukturą).

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o (złożonym) typie danych, który jest klasą (lub strukturą).

Realizacja (*ang. exemplar*) pewnej klasy (w takim sensie, w jakim element klasy abstrakcji relacji równoważności reprezentuje ją całą).

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o (złożonym) typie danych, który jest klasą (lub strukturą).

Realizacja (*ang. exemplar*) pewnej klasy (w takim sensie, w jakim element klasy abstrakcji relacji równoważności reprezentuje ją całą).

Instacja (*ang. instance*)

Obiekt pewnej klasy (tzn. zmienna) utworzony w czasie wykonywania programu.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o (złożonym) typie danych, który jest klasą (lub strukturą).

Realizacja (*ang. exemplar*) pewnej klasy (w takim sensie, w jakim element klasy abstrakcji relacji równoważności reprezentuje ją całą).

Instacja (*ang. instance*)

Obiekt pewnej klasy (tzn. zmienna) utworzony w czasie wykonywania programu.

Stanowi reprezentację stanu czegoś w danej chwili czasu, wyrażoną poprzez wartości pól wchodzących w jego skład, zaś dostępne metody określają jego możliwości działania.

Definicja klasy

1. Słowo kluczowe `class`.
2. Nazwa klasy (litery, znak _ i cyfry – oprócz pierwszego znaku).
3. Specyfikacja klas nadrzędnych (od których pochodzi ta klasa).
4. Lista składników (w nawiasie klamrowym).
   - Pola – typ danych i nazwa (jak dla struktur).
   - Metody – nagłówek (typ wyniku, nazwa, lista argumentów).
   - Dla każdego składnika można określić zasady dostępu.

Definicja klasy

1. Słowo kluczowe `class`.
2. Nazwa klasy (litery, znak _ i cyfry – oprócz pierwszego znaku).
3. Specyfikacja klas nadrzędnych (od których pochodzi ta klasa).
4. Lista składników (w nawiasie klamrowym).
   - Pola – typ danych i nazwa (jak dla struktur).
   - Metody – nagłówek (typ wyniku, nazwa, lista argumentów).
   - Dla każdego składnika można określić zasady dostępu.

```
class Vector {
  public:  // specyfikacja zasad dostępu
        double x, y;          // pola
        double norm(void);    // metoda
        double length(void); // metoda
};
```

Modyfikatory dostępu

Modyfikator dostępu (*ang. access modifier*)

Określa, z jakich miejsc w programie można odwoływać się do składników klasy („oddziałuje" na składniki zadeklarowane „pod" nim).

`public` – do tych składników klasy można odwoływać się z dowolnego miejsca w programie.

`private` – do tych składników klasy można odwoływać się tylko z metod będących składnikami tej klasy.

`protected` – do tych składników klasy można odwoływać się z metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

Modyfikatory dostępu

Modyfikator dostępu (*ang. access modifier*)

Określa, z jakich miejsc w programie można odwoływać się do składników klasy („oddziałuje" na składniki zadeklarowane „pod" nim).

`public` – do tych składników klasy można odwoływać się z dowolnego miejsca w programie.

`private` – do tych składników klasy można odwoływać się tylko z metod będących składnikami tej klasy.

`protected` – do tych składników klasy można odwoływać się z metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

Domyślnym modyfikatorem dostępu jest `private`.

Metody

Metody definiuje się podobnie jak zwykłe funkcje

Przed nazwą metody trzeba umieścić nazwę klasy, której składnikiem jest ta metoda oraz symbol `::`, np.:

```
double Wektor::length(void)
{
    // Tu znajduje się treść metody.
}
```

Metody

Metody definiuje się podobnie jak zwykłe funkcje

Przed nazwą metody trzeba umieścić nazwę klasy, której składnikiem jest ta metoda oraz symbol `::`, np.:

```
double Wektor::length(void)
{
    // Tu znajduje się treść metody.
}
```

Metodę zawsze wywołuje się dla konkretnego obiektu i jego składniki (pola oraz inne metody) mogą być używane w treści metody jak zmienne lokalne (w przypadku pól) lub „zwykłe” funkcje (w przypadku metod).

Składniki klas

Składniki (*ang. member*) klasy

Pola wchodzące w skład obiektów tej klasy oraz metody zdefiniowane w celu przeprowadzania operacji na nich.

Składniki klas

Składniki (*ang. member*) klasy

Pola wchodzące w skład obiektów tej klasy oraz metody zdefiniowane w celu przeprowadzania operacji na nich.

Do składników klas można odwoływać się tylko poprzez obiekty tych klas, z użyciem symboli `.` i `->`.

```
klasa obiekt;

obiekt.pole = wart;
obiekt.metoda(wart);
```

```
klasa *wsk;

wsk = new klasa;
wsk->pole = wart;
wsk->metoda(wart);
```

Odwołania do składników klas

Zmienna po lewej stronie symbolu . lub wskaźnik po lewej stronie symbolu -> określa kontekst odwołania do pola obiektu lub wywołania metody.

Odwołania do składników klas

Zmienna po lewej stronie symbolu . lub wskaźnik po lewej stronie symbolu -> określa kontekst odwołania do pola obiektu lub wywołania metody.

```
class wektor {
 public:
        double x, y;
        double norma(void);
        double abs(void);
};

double wektor::norma(void)
{
        return x*x + y*y;
}

double wektor::abs(void)
{
        return sqrt(norma());
}
```

```
wektor wk, *ptr;

wk.x = 0;
wk.y = 1;
// Metoda norma() będzie wywołana w
// metodzie abs() dla pól z obiektu wk
cout << wk.abs() << endl;

ptr = new wektor;
ptr->x = 1;
ptr->y = 1;
// Metoda norma() będzie wywołana w
// metodzie abs() dla pól z obiektu
// pod adresem ptr
cout << ptr->abs() << endl;
```

Programowanie obiektowe i proceduralne

Programowanie zorientowane obiektowo (*ang. object-oriented programming*) jest sposobem zapisu kodu źródłowego programów komputerowych tak, aby przepływ kontroli w programie można było interpretować jako interakcje między obiektami różnych klas.

Programowanie obiektowe i proceduralne

Programowanie zorientowane obiektowo (*ang. object-oriented programming*) jest sposobem zapisu kodu źródłowego programów komputerowych tak, aby przepływ kontroli w programie można było interpretować jako interakcje między obiektami różnych klas.

Pozwala to na szybkie tworzenie aplikacji z wykorzystaniem gotowych komponentów (np. obiektów reprezentujących składniki graficznego interfejsu użytkownika) i z pomocą środowisk typu RAD (*ang. Rapid Application Development*).

Programowanie obiektowe i proceduralne

Programowanie zorientowane obiektowo (*ang. object-oriented programming*) jest sposobem zapisu kodu źródłowego programów komputerowych tak, aby przepływ kontroli w programie można było interpretować jako interakcje między obiektami różnych klas.

Pozwala to na szybkie tworzenie aplikacji z wykorzystaniem gotowych komponentów (np. obiektów reprezentujących składniki graficznego interfejsu użytkownika) i z pomocą środowisk typu RAD (*ang. Rapid Application Development*).

Programowanie proceduralne polega na zapisywaniu kodu źródłowego programu w formie funkcji (procedur) realizujących różne części algorytmu.

Implementowanie metod

Definicja klasy określa tylko nagłówki metod. Aby można było z nich korzystać, trzeba podać ich implementację (sposób działania).

Implementacja metody jest bardzo podobna do definicji funkcji.

Implementowanie metod

Definicja klasy określa tylko nagłówki metod. Aby można było z nich korzystać, trzeba podać ich implementację (sposób działania).

Implementacja metody jest bardzo podobna do definicji funkcji.

```
double Vector::norm(void)
{
        return x*x + y*y;
}

double Vector::length(void)
{
        return sqrt(norm());
}
```

Metody „inline”

Jeżeli implementacja metody jest szczególnie prosta, można umieścić ją wewnątrz definicji klasy.

Metody „inline”

Jeżeli implementacja metody jest szczególnie prosta, można umieścić ją wewnątrz definicji klasy.

```
class Vector {
  public:
        double x, y;
        double norm(void)
        {
                return x*x + y*y;
        }
        double length(void)
        {
                return sqrt(norm());
        }
};
```

Metody „const”

Jeżeli pola, z których korzysta metoda, nie są przez nią zmieniane, można to zadeklarować umieszczając `const` w nagłówku.

Metody „const”

Jeżeli pola, z których korzysta metoda, nie są przez nią zmieniane, można to zadeklarować umieszczając `const` w nagłówku.

```
class Vector {
  public:
        double x, y;
        double norm(void) const
        {
                return x*x + y*y;
        }
        double length(void) const
        {
                return sqrt(norm());
        }
};
```

Statyczne pola

Pola będące składnikami klas można zadeklarować z atrybutem `static`.

Wówczas są one wspólne dla wszystkich obiektów danej klasy i są inicjowane przed rozpoczęciem wykonywania funkcji `main()`.

Statyczne pola

Pola będące składnikami klas można zadeklarować z atrybutem `static`.

Wówczas są one wspólne dla wszystkich obiektów danej klasy i są inicjowane przed rozpoczęciem wykonywania funkcji `main()`.

```
class Klasa {
  public:
        static int statyczne_pole;  // pole wspólne dla wszystkich obiektów tej klasy
        ...
};

...
Klasa a, b;

...
a.statyczne_pole = 123;
cout << b.statyczne_pole << endl; // wydrukuje 123
cout << Klasa::statyczne_pole << endl; // wydrukuje 123
```

Stałe skojarzone z klasami

W definicji klasy można umieścić definicję stałej.

```
class Klasa {
  public:
        static const double jeden = 1;  // stała
        ...
};
```

Stałe skojarzone z klasami

W definicji klasy można umieścić definicję stałej.

```
class Klasa {
  public:
        static const double jeden = 1;  // stała
        ...
};
```

Dalej można odwoływać się do takiej stałej łącząc nazwę klasy z jej nazwą.

```
cout << Klasa::jeden << endl;
```

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Metody w klasach są specjalnym rodzajem funkcji.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Metody w klasach są specjalnym rodzajem funkcji.

Aby funkcje mogły spełniać swoje zadania, muszą operować na określonych danych, które trzeba przekazywać do kodu zapisanego w postaci funkcji.

Przykład – dodawanie macierzy 2x2

```
class Matrix22 {
  public:
    double w1k1, w1k2, w2k1, w2k2;
    Matrix22(void): w1k1(0), w1k2(0), w2k1(0), w2k2(0) {}
    Matrix22(double r): w1k1(r), w1k2(0), w2k1(0), w2k2(r) {}
    Matrix22(double a, double b, double c, double d):
             w1k1(a), w1k2(b), w2k1(c), w2k2(d) {}
    double det(void) const { return w1k1*w2k2 - w1k2*w2k1; }
    Matrix22 operator +(Mattix22 m)
    {
        return Matrix22(w1k1 + m.w1k1, w1k2 + m.w1k2,
                        w2k1 + m.w2k1, w2k2 + m.w2k2);
    }
};

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    cout << C.w1k1 << " " << C.w1k2 << " " << C.w2k1 << " " << C.w2k2 << endl;
    A = C + B;
    cout << A.w1k1 << " " << A.w1k2 << " " << A.w2k1 << " " << A.w2k2 << endl;
    B = A + C;
    cout << B.w1k1 << " " << B.w1k2 << " " << B.w2k1 << " " << B.w2k2 << endl;
    return 0;
}
```

Przekazywanie argumentu przez wartość

Aby uniknąć wielokrotnego powtarzania tego samego kodu wprowadzamy funkcję `drukuj()`:

```
void drukuj(Matrix22 M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    drukuj(C);
    A = C + B;
    drukuj(A);
    B = A + C;
    drukuj(B);
    return 0;
}
```

Przekazywanie argumentu przez wartość

Aby uniknąć wielokrotnego powtarzania tego samego kodu wprowadzamy funkcję `drukuj()`:

```
void drukuj(Matrix22 M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    drukuj(C);
    A = C + B;
    drukuj(A);
    B = A + C;
    drukuj(B);
    return 0;
}
```

Przy każdym wywołaniu `drukuj()` tworzona jest kopia obiektu przekazywanego jako argument (używana w funkcji pod nazwą `M`).

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Koszty te nie byłyby ponoszone, gdyby można było „powiedzieć" funkcji, że ma posługiwać się tym samym obiektem, który jest przekazywany jako argument (bez kopiowania go).

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Koszty te nie byłyby ponoszone, gdyby można było „powiedzieć" funkcji, że ma posługiwać się tym samym obiektem, który jest przekazywany jako argument (bez kopiowania go).

Do tego celu służą referencje (*ang. reference*).

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy funkcja będzie używać tego samego obiektu, który został przekazany jako argument (bez kopiowania go), ale pod inną nazwą.

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy funkcja będzie używać tego samego obiektu, który został przekazany jako argument (bez kopiowania go), ale pod inną nazwą.

Wówczas modyfikacje tego obiektu przez funkcję zostaną zachowane i będą „widoczne" po zakończeniu wykonywania jej.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Ma to sens jedynie w przypadku obiektów, o których wiadomo, że będą istniały po zakończeniu wykonywania funkcji zwracających referencje do nich (tzn. nie wolno zwracać referencji do obiektów będących zmiennymi lokalnymi w funkcji zwracającej referencję do obiektu).

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Ma to sens jedynie w przypadku obiektów, o których wiadomo, że będą istniały po zakończeniu wykonywania funkcji zwracających referencje do nich (tzn. nie wolno zwracać referencji do obiektów będących zmiennymi lokalnymi w funkcji zwracającej referencję do obiektu).

```
Matrix22& drukuj(ostream& out, Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
    return M;  // Przekazanie kontroli nad (obiektem reprezentowanym przez) M z powrotem.
}
```

Na czym polega przeciążanie operatorów

Zamiast podawać nazwę metody, można ją oznaczyć z pomocą symbolu operatora. Wtedy liczba argumentów metody zależy od liczby argumentów operatora.

Na czym polega przeciążanie operatorów

Zamiast podawać nazwę metody, można ją oznaczyć z pomocą symbolu operatora. Wtedy liczba argumentów metody zależy od liczby argumentów operatora.

```
class Wektor {
  public:
        double x, y;
        void add(Wektor w);
};

void Wektor::add(Wektor w)
{
        x += w.x;
        y += w.y;
}
...

Wektor w, v;

...
w.add(v);
```

Na czym polega przeciążanie operatorów

Zamiast podawać nazwę metody, można ją oznaczyć z pomocą symbolu operatora. Wtedy liczba argumentów metody zależy od liczby argumentów operatora.

```
class Wektor {
  public:
        double x, y;
        void add(Wektor w);
};

void Wektor::add(Wektor w)
{
        x += w.x;
        y += w.y;
}
...

Wektor w, v;

...
w.add(v);
```

```
class Wektor {
  public:
        double x, y;
        void operator +=(Wektor w);
};

void Wektor::operator +=(Wektor w)
{
        x += w.x;
        y += w.y;
}
...

Wektor w, v;

...
w += v;  // wywołanie metody operator +=
```

Argumenty

Argumenty funkcji wywoływanej jako operator nie muszą być tego samego typu jak obiekt, w kontekście którego wywoływana jest metoda.

Argumenty

Argumenty funkcji wywoływanej jako operator nie muszą być tego samego typu jak obiekt, w kontekście którego wywoływana jest metoda.

```
class Wektor {
  public:
        double x, y;
        void operator *=(double r);
};

void Wektor::operator *=(double r)
{
        x *= r;
        y *= r;
}
...

Wektor w;

...
w *= 2;  // wywołanie metody operator *=
```

Argumenty

Argumenty funkcji wywoływanej jako operator nie muszą być tego samego typu jak obiekt, w kontekście którego wywoływana jest metoda.

```
class Wektor {
  public:
        double x, y;
        void operator *=(double r);
};

void Wektor::operator *=(double r)
{
        x *= r;
        y *= r;
}
...

Wektor w;

...
w *= 2;  // wywołanie metody operator *=
```

```
class Wektor {
  public:
        double x, y;
        void operator +=(double r);
};

void Wektor::operator +=(double r)
{
        x += r;
}
...

Wektor w;

...
w += 1;  // wywołanie metody operator +=
```

Przeciążanie symbolu []

Można także przeciążać operator wskazania elementu tablicy [].
Reprezentująca go funkcja najczęściej zwraca referencję, aby można było użyć jej po lewej stronie operator przypisania =.

Przeciążanie symbolu []

Można także przeciążać operator wskazania elementu tablicy []. Reprezentująca go funkcja najczęściej zwraca referencję, aby można było użyć jej po lewej stronie operator przypisania =.

```
class Wektor {
  public:
        double x, y;
        double& wsp(int n)
        {
                return n == 1 ? x : y;
        }
};

...

Wektor w;

...
w.wsp(1) = 2;  // wywołanie metody wsp()
```

Przeciążanie symbolu []

Można także przeciążać operator wskazania elementu tablicy []. Reprezentująca go funkcja najczęściej zwraca referencję, aby można było użyć jej po lewej stronie operator przypisania =.

```
class Wektor {
  public:
        double x, y;
        double& wsp(int n)
        {
                return n == 1 ? x : y;
        }
};

...

Wektor w;

...
w.wsp(1) = 2;  // wywołanie metody wsp()
```

```
class Wektor {
  public:
        double x, y;
        double& operator [](int n)
        {
                return n == 1 ? x : y;
        }
};

...

Wektor w;

...
w[1] = 2;  // wywołanie metody operator []
```

Konstruktor

Jeśli w czasie tworzenia obiektu danej klasy należy wykonać jakąś akcję (np. zarezerwować pamięć), to jednym z jej składników powinien być konstruktor (*ang. constructor*).

Konstruktor

Jeśli w czasie tworzenia obiektu danej klasy należy wykonać jakąś akcję (np. zarezerwować pamięć), to jednym z jej składników powinien być konstruktor (*ang. constructor*).

Konstruktor

Metoda wywoływana automatycznie (z odpowiednimi argumentami) podczas tworzenia obiektu. Zawsze ma taką nazwę, jak klasa, której jest składnikiem.

Konstruktor

Jeśli w czasie tworzenia obiektu danej klasy należy wykonać jakąś akcję (np. zarezerwować pamięć), to jednym z jej składników powinien być konstruktor (*ang. constructor*).

Konstruktor

Metoda wywoływana automatycznie (z odpowiednimi argumentami) podczas tworzenia obiektu. Zawsze ma taką nazwę, jak klasa, której jest składnikiem.

```
class Tablica {
  public:
        double *elem;
        int n;
        double& operator [](int n);
        Tablica(int n_el);
};
```

```
Tablica::Tablica(int n_el)
{
        elem = new double[n_el];
        if (elem)
                n = n_el;
}

Tablica tab(10);  // wywołanie konstruktora
```

Definiowanie wielu konstruktorów

W jednej klasie może być wiele konstruktorów, ale muszą one różnić się jednoznacznie liczbą lub typami danych argumentów.

Definiowanie wielu konstruktorów

W jednej klasie może być wiele konstruktorów, ale muszą one różnić się jednoznacznie liczbą lub typami danych argumentów.

```
class Tablica {
  public:
        double *elem;
        int n;
        double& operator [](int n);
        void init(int n_el);
        Tablica(int n_el);
        Tablica(int n_el, double r);
};

void Tablica::init(int n_el)
{
        elem = new double[n_el];
        if (elem)
                n = n_el;
}
```

```
Tablica::Tablica(int n_el)
{
        init(n_el);
}

Tablica::Tablica(int n_el, double r)
{
        init(n_el);
        for (int i = 0; i < n; i++)
                elem[i] = r;
}

...
Tablica tab(10);  // Tablica(int)

...
Tablica tmp(10, -1);  // Tablica(int, double)
```

Kiedy wywoływane są konstruktory

Dla zmiennych lokalnych

Bezpośrednio po utworzeniu zmiennej w czasie wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu definicji zmiennej w obrębie bloku).

Kiedy wywoływane są konstruktory

Dla zmiennych lokalnych

Bezpośrednio po utworzeniu zmiennej w czasie wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu definicji zmiennej w obrębie bloku).

Dla zmiennych dynamicznych

Bezpośrednio po utworzeniu zmiennej z użyciem `new`, np.:

```
wsk = new Tablica(10);
```

Kiedy wywoływane są konstruktory

Dla zmiennych lokalnych

Bezpośrednio po utworzeniu zmiennej w czasie wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu definicji zmiennej w obrębie bloku).

Dla zmiennych dynamicznych

Bezpośrednio po utworzeniu zmiennej z użyciem `new`, np.:

```
wsk = new Tablica(10);
```

Dla zmiennych globalnych lub statycznych

Przed rozpoczęciem wykonywania funkcji `main()`, bezpośrednio po zarezerwowaniu pamięci na te zmienne.

Destruktor

Jeżeli przed usunięciem obiektu z pamięci trzeba przeprowadzić jakąś dodatkową czynność (np. zwolnić pamięć zarezerwowaną przez konstruktor), to jednym ze składników klasy powinien być destruktor (*ang. destructor*).

Destruktor

Jeżeli przed usunięciem obiektu z pamięci trzeba przeprowadzić jakąś dodatkową czynność (np. zwolnić pamięć zarezerwowaną przez konstruktor), to jednym ze składników klasy powinien być destruktor (*ang. destructor*).

Destruktor

Metoda wykonywana automatycznie bezpośrednio przed usunięciem obiektu z pamięci. Jej nazwa musi składać się ze znaku ~ i nazwy klasy, której jest składnikiem.

Destruktor

Jeżeli przed usunięciem obiektu z pamięci trzeba przeprowadzić jakąś dodatkową czynność (np. zwolnić pamięć zarezerwowaną przez konstruktor), to jednym ze składników klasy powinien być destruktor (*ang. destructor*).

Destruktor

Metoda wykonywana automatycznie bezpośrednio przed usunięciem obiektu z pamięci. Jej nazwa musi składać się ze znaku ~ i nazwy klasy, której jest składnikiem.

Do destruktora nie można przekazywać argumentów, więc w każdej klasie może być co najwyżej jeden destruktor.

Definiowanie destruktora

```
class Tablica {
  public:
        double *elem;
        int n;
        double& operator [](int n)
        {
                return elem[n];
        }
        void init(int n_el);
        Tablica(int n_el)
        {
                init(n_el);
        }
        Tablica(int n_el, double r);
        ~Tablica(void);
};

void Tablica::init(int n_el)
{
        elem = new double[n_el];
        if (elem)
                n = n_el;
}
```

```
Tablica::Tablica(int n_el, double r)
{
        init(n_el);
        for (int i = 0; i < n; i++)
                elem[i] = r;
}

Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

...
Tablica *wsk;

...
wsk = new Tablica(10, 0);
...
delete wsk;  // ~Tablica()
```

Kiedy wywoływany jest destruktor

Dla zmiennych lokalnych

Bezpośrednio przed usunięciem zmiennej z pamięci po zakończeniu wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu klamry } kończącej blok).

Kiedy wywoływany jest destruktor

Dla zmiennych lokalnych

Bezpośrednio przed usunięciem zmiennej z pamięci po zakończeniu wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu klamry } kończącej blok).

Dla zmiennych dynamicznych

Bezpośrednio przed usunięciem zmiennej z użyciem `delete`.

Kiedy wywoływany jest destruktor

Dla zmiennych lokalnych

Bezpośrednio przed usunięciem zmiennej z pamięci po zakończeniu wykonywania bloku zawierającego jej definicję (przeważnie w miejscu odpowiadającym położeniu klamry } kończącej blok).

Dla zmiennych dynamicznych

Bezpośrednio przed usunięciem zmiennej z użyciem `delete`.

Dla zmiennych globalnych lub statycznych

Po zakończeniu wykonywania funkcji `main()`, bezpośrednio przed zwolnieniem pamięci zajmowanej przez te zmienne.

Składniki klasy dostępne tylko dla metod

Często wygodnie jest zadeklarować składnik klasy jako dostępny tylko dla metod będących jej składnikami.

Składniki klasy dostępne tylko dla metod

Często wygodnie jest zadeklarować składnik klasy jako dostępny tylko dla metod będących jej składnikami.

W tym celu trzeba poprzedzić jego definicję modyfikatorem dostępu `private`.

Składniki klasy dostępne tylko dla metod

Często wygodnie jest zadeklarować składnik klasy jako dostępny tylko dla metod będących jej składnikami.

W tym celu trzeba poprzedzić jego definicję modyfikatorem dostępu `private`.

```
class Tablica {
  private:
        double *elem;
        int n;
  public:
        double& operator [](int n);
        void init(int n_el);
        Tablica(int n_el);
        Tablica(int n_el, double r);
        ~Tablica(void);
};
```

```
void Tablica::init(int n_el)
{
        elem = new double[n_el]; // OK
        if (elem)                // OK
                n = n_el;        // OK
}

...
Tablica tab(10);

...
cout << tab.n << endl;  // Błąd!
```

Czym jest wskaźnik this

Każdy obiekt dowolnej klasy zawiera pole o nazwie this, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Czym jest wskaźnik this

Każdy obiekt dowolnej klasy zawiera pole o nazwie this, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik this jest typu (Klasa *), gdzie Klasa jest typem danych (klasą) odpowiadającą obiektowi, w skład którego on wchodzi.

Czym jest wskaźnik this

Każdy obiekt dowolnej klasy zawiera pole o nazwie this, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik this jest typu (Klasa *), gdzie Klasa jest typem danych (klasą) odpowiadającą obiektowi, w skład którego on wchodzi.

Wskaźnik this jest polem prywatnym, więc może być wykorzystywany tylko przez metody będące składnikami klasy obiektu.

Czym jest wskaźnik this

Każdy obiekt dowolnej klasy zawiera pole o nazwie this, które jest wskaźnikiem, a jego wartością jest adres tego obiektu.

Wskaźnik this jest typu (Klasa *), gdzie Klasa jest typem danych (klasą) odpowiadającą obiektowi, w skład którego on wchodzi.

Wskaźnik this jest polem prywatnym, więc może być wykorzystywany tylko przez metody będące składnikami klasy obiektu.

Obiekt, w którego skład wchodzi pole this, jest wartością wyrażenia *this.

Do czego przydaje się this?

Wskaźnik `this` przydaje się (między innymi) przy przeciążaniu operatora inkrementacji.

Do czego przydaje się this?

Wskaźnik this przydaje się (między innymi) przy przeciążaniu operatora inkrementacji.

```
class Wektor {
  public:
        double x, y;
        Wektor(void) {}
        Wektor(double a, double b)
        {
                x = a;
                y = b;
        }
        double norma(void) const
        {
                return x*x + y*y;
        }
        Wektor operator ++(void)
        {
                x++;
                y++;
                return *this;
        }
};
```

```
Wektor w(1, 1), v;

v = ++w;

cout << w.norma() << " " << v.norma() << endl;
```

Przeciążanie postinkrementacji

W celu przeciążenia postinkrementacji trzeba zdefiniować metodę z jednym parametrem typu `int`, który otrzymuje wartość 0.

Przeciążanie postinkrementacji

W celu przeciążenia postinkrementacji trzeba zdefiniować metodę z jednym parametrem typu `int`, który otrzymuje wartość 0.

```
class Wektor {
  public:
        double x, y;
        Wektor(void) {}
        Wektor(double a, double b)
        {
                x = a;
                y = b;
        }
        double norma(void) const
        {
                return x*x + y*y;
        }
        Wektor operator ++(int zero)
        {
                x++;
                y++;
                return *this;
        }
};
```

```
Wektor w(1, 1), v;

v = w++; // v = w.operator++(0)

cout << w.norma() << " " << v.norma() << endl;
```

Ostrożnie z przypisaniem!

Domyślna operacja przypisania (kopiowanie wartości pól z jednego obiektu do drugiego) nie musi działać zgodnie z oczekiwaniami.

Ostrożnie z przypisaniem!

Domyślna operacja przypisania (kopiowanie wartości pól z jednego obiektu do drugiego) nie musi działać zgodnie z oczekiwaniami.

```
class Tablica {
        double *elem;  // prywatne!
        int n;         // prywatne!
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

...
Tablica a(2), b(3);

for (int i = 0; i < 3; i++)
        b[i] = 0;
a = b;
a[0] = 1;
a[1] = 2;
// Jaki będzie wynik?
cout << b[0] <<  " " << b[1] << endl;
```

Jak przeciążyć przypisanie

Rozwiązaniem problemu jest przeciążenie operatora przypisania.

Jak przeciążyć przypisanie

Rozwiązaniem problemu jest przeciążenie operatora przypisania.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
        void operator =(Tablica& t);
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

void Tablica::operator =(Tablica& t)
{
        if (n == t.n)
                for (int i = 0; i < n; i++)
                        elem[i] = t.elem[i];
}

...
Tablica a(2), b(2);

...
a = b;  // OK
```

Inny typ argumentu

Argument metody wywoływanej jako operator = może być dowolny.

Inny typ argumentu

Argument metody wywoływanej jako operator = może być dowolny.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
        void operator =(double r);
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

void Tablica::operator =(double r)
{
          for (int i = 0; i < n; i++)
                elem[i] = r;
}

...
Tablica a(2);

...
a = 0;  // OK
```

Przeciążone przypisanie zwracające wynik

„Normalne” przypisanie zwraca wynik, który można wykorzystać w wyrażeniu. Podobnie może być z przeciążonym przypisaniem.

Przeciążone przypisanie zwracające wynik

„Normalne” przypisanie zwraca wynik, który można wykorzystać w wyrażeniu. Podobnie może być z przeciążonym przypisaniem.

```
class Tablica {
        double *elem;
        int n;
  public:
        Tablica(int nr_el);
        ~Tablica(void);
        double& operator [](int i)
        {
                return elem[i];
        }
        Tablica& operator =(Tablica& t);
};

Tablica::Tablica(int nr_el)
{
        elem = new double[nr_el];
        if (elem)
                n = nr_el;
}
```

```
Tablica::~Tablica(void)
{
        n = 0;
        if (elem)
                delete [] elem;
}

Tablica& Tablica::operator =(Tablica& t)
{
        if (n == t.n)
                for (int i = 0; i < n; i++)
                        elem[i] = t.elem[i];
        return *this;
}

...
Tablica a(2), b(2), c(2);

...
a = b = c;  // OK
```

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Muszą one różnić się typem argumentów tak, aby kompilator mógł jednoznacznie stwierdzić która z nich ma być wywołana w danym miejscu programu.

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Muszą one różnić się typem argumentów tak, aby kompilator mógł jednoznacznie stwierdzić która z nich ma być wywołana w danym miejscu programu.

Operatory mogą być również przeciążane z pomocą funkcji, które nie są metodami.

Różne operatory o tym samym symbolu

W jednej klasie może być zdefiniowanych wiele metod wywoływanych jako operator o tym samym symbolu (np. +=).

Muszą one różnić się typem argumentów tak, aby kompilator mógł jednoznacznie stwierdzić która z nich ma być wywołana w danym miejscu programu.

Operatory mogą być również przeciążane z pomocą funkcji, które nie są metodami.

Funkcje te mogą reprezentować operatory o takich symbolach, jakie zostały już przeciążone z pomocą metod.

Przeciążanie dodawania

W celu przeciążenia dodawania wygodnie jest użyć funkcji nie będącej metodą.

Przeciążanie dodawania

W celu przeciążenia dodawania wygodnie jest użyć funkcji nie będącej metodą.

```
class Wektor {
  public:
        double x, y;
        Wektor(void) {}
        Wektor(double a, double b)
        {
                x = a;
                y = b;
        }
        double norma(void) const
        {
                return x*x + y*y;
        }
};
```

```
Wektor operator +(Wektor w, Wektor v)
{
        Wektor wynik;

        wynik.x = w.x + v.x;
        wynik.y = w.y + v.y;
        return wynik;
}

...
Wektor a(1, 1), b(1, 0), c;

...
c = a + b;  // operator +(a, b)
```

Funkcje zaprzyjaźnione z klasami

Jeżeli pola danej klasy są prywatne, ale funkcja (np. przeciążająca operator) ma mieć do nich dostęp, to można zadeklarować ją jako „przyjaciela” (*ang. friend*) tej klasy.

Funkcje zaprzyjaźnione z klasami

Jeżeli pola danej klasy są prywatne, ale funkcja (np. przeciążająca operator) ma mieć do nich dostęp, to można zadeklarować ją jako „przyjaciela" (*ang. friend*) tej klasy.

```
class Wektor {
        double x, y;
  public:
        Wektor(void) {}
        Wektor(double a, double b)
        {
                x = a;
                y = b;
        }
        double norma(void) const
        {
                return x*x + y*y;
        }
        friend Wektor operator +(Wektor w, Wektor v);
};
```