Programowanie, część I

Rafał J. Wysocki

11 marca 2010

Kontakt

- `http://www.fuw.edu.pl/~rwys/prog`
- rwys@fuw.edu.pl
- tel. 22 55 32 263

Materiał na ćwiczenia

- Wykorzystanie standardowych klas STL

Materiał na ćwiczenia

- Wykorzystanie standardowych klas STL
- Programowanie grafiki z użyciem biblioteki *Qt*

Materiał na ćwiczenia

- Wykorzystanie standardowych klas STL
- Programowanie grafiki z użyciem biblioteki *Qt*
- C++ i standardowe algorytmy numeryczne

Plan wykładu – Klasy i programowanie obiektowe

- Czym są klasy
- Definiowanie klas
- Typowe składniki klas i ich role
- Ograniczanie dostępu do składników klas
- Dziedziczenie
- Wirtualne metody i klasy abstrakcyjne
- Hierarchie klas
- Szablony

Plan wykładu – Biblioteka *Qt*

- Struktura programu z oknami, wątek obsługi zdarzeń
- Sygnały i sloty
- Klasy z biblioteki *Qt* i ich powiązania
- Projektowanie GUI
- Własne rysunki, wykresy i animacje
- Biblioteka *qwt*

Plan wykładu – Algorytmy numeryczne

- Eliminacja Gaussa i Gaussa-Jordana (przypomnienie)
- Równania różniczkowe zwyczajne i metoda Rungego-Kutty
- Rozwiązywanie równań nieliniowych metodą Newtona

Czym są klasy

Klasa (*ang. class*)

Złożony typ danych z możliwością definiowania funkcji, zwanych metodami, w których poszczególne pola wchodzące w skład zmiennej tego typu są traktowane jak zmienne statyczne.

Czym są klasy

Klasa (*ang. class*)

Złożony typ danych z możliwością definiowania funkcji, zwanych metodami, w których poszczególne pola wchodzące w skład zmiennej tego typu są traktowane jak zmienne statyczne.

Klasa

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 z możliwością definiowania funkcji, zwanych metodami, w których poszczególne pola wchodzące w skład zmiennej tego typu są traktowane jak zmienne statyczne.

Klasa

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

Klasa

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

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o typie danych, który jest klasą.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o typie danych, który jest klasą.

Obiekt

Realizacja (*ang. exemplar*) pewnej klasy.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o typie danych, który jest klasą.

Obiekt

Realizacja (*ang. exemplar*) pewnej klasy.

Instacja (*ang. instance*)

Obiekt pewnej klasy utworzony w czasie wykonywania programu. Stanowi reprezentację stanu czegoś w danej chwili czasu, wyrażoną poprzez wartości pól wchodzących w jego skład, zaś metody określają jego możliwości działania.

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

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

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

Programowanie obiektowe vs proceduralne

Obiektowo (C++)

```
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 *wsk;

wsk = new wektor;
wsk->x = 1;
wsk->y = 2;
cout << wsk->abs() << endl;
```

Proceduralnie (C)

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

double norma_wektora(struct wektor *w)
{
        return w->x * w->x + w->y * w->y;
}

double abs_wektora(struct wektor *w)
{
        return sqrt(norma_wektora(w));
}

...

struct wektor *wsk;

wsk = malloc(sizeof(*wsk));
wsk->x = 1;
wsk->y = 2;
cout << abs_wektora(wsk) << endl;
```

Programowanie obiektowe vs proceduralne

W kodzie obiektowym

- Zmienna, na której mają być przeprowadzone operacje, jest wyznaczana na podstawie *kontekstu wywołania* metody.
- Kontekst `wsk->` oznacza, że ma to być zmienna pod adresem `wsk`.
- Jeżeli podczas wykonywania metody nastąpi odwołanie do innego składnika tej samej klasy, jego kontekst nie zmienia się.

W kodzie proceduralnym

- Adres zmiennej, na której mają być przeprowadzone operacje, jest przekazywany do funkcji jako argument.
- Kontekst wywołania funkcji (procedury) jest określany przez przekazywane do niej argumenty.

Definicja klasy

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

Definicja klasy

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

```
class Wektor {
  public:  // specyfikacja zasad dostępu
        double x, y;          // pola
        double norma(void); // metoda
        double abs(void);   // metoda
};
```

Implementacja 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.

Implementacja 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 Wektor::norma(void)
{
        return x*x + y*y;
}

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

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 Wektor {
  public:
        double x, y;
        double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};
```

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 Wektor {
  public:
        double x, y;
        double norma(void) const
        {
                return x*x + y*y;
        }
        double abs(void) const
        {
                return sqrt(norma());
        }
};
```

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
```

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;
```

Przeciążanie operatorów

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

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);
```

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 +=
```

Przeciążanie operatorów – argumenty

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

Przeciążanie operatorów – 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 *=
```

Przeciążanie operatorów – 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!
```

Wskaźnik `this`

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

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`

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.

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);

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

Przeciążanie przypisania

Rozwiązaniem problemu jest przeciążenie operatora przypisania.

Przeciążanie przypisania

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
```

Przeciążanie przypisania (inny typ argumentu)

Argument metody wywoływanej jako operator = może być dowolny.

Przeciążanie przypisania (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. Podonie może być z przeciążonym przypisaniem.

Przeciążone przypisanie zwracające wynik

„Normalne” przypisanie zwraca wynik, który można wykorzystać w wyrażeniu. Podonie 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);
};
```

Literatura


B. Stroustrup, *Język C++* (Wydawnictwo Naukowo-Techniczne, Warszawa 2002).


B. Eckel, *Thinking in C++. Edycja polska* (Wydawnictwo Helion, Gliwice 2002).


J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4 (2nd Edition)* (Prentice Hall, Westford, 2008).


Pang Tao, *Metody obliczeniowe w fizyce* (Wydawnictwo Naukowe PWN, Warszawa 2001).


T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów* (Wydawnictwa Naukowo-Techniczne, Warszawa 2005).