Programowanie, część I

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

22 lutego 2011

Kontakt

- `http://www.fuw.edu.pl/~rwys/prog`
- rwys@fuw.edu.pl
- tel. 22 55 32 263
- ul. Hoża 69, pok. 142

Materiał na ćwiczenia

- Klasy i dziedziczenie w C++
- Programowanie grafiki z użyciem biblioteki *Qt*
- Algorytmy numeryczne w C++

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
- Tworzenie prostej grafiki
- Konstruowanie graficznego interfejsu użytkownika (GUI)

Plan wykładu – Algorytmy numeryczne

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

Czym są klasy

Klasa (*ang. class*)

Złożony typ danych 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 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 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 typie danych, który jest klasą.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o typie danych, który jest klasą.

Realizacja (*ang. exemplar*) pewnej klasy.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o typie danych, który jest klasą.

Realizacja (*ang. exemplar*) pewnej klasy.

Instacja (*ang. instance*)

Obiekt pewnej klasy utworzony w czasie wykonywania programu.

Czym są obiekty

Obiekt (*ang. object*)

Zmienna o typie danych, który jest klasą.

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 dostępne 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 argumentów)
  - 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 argumentów)
  - 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.

Kopiowanie argumentów konstruktora

Kopiowanie argumentów konstruktora do pól obiektu można zapisywać w skrótowej formie `pole1(argument1), pole2(argument2), ...` po nagłówku konstruktora i znaku `:` (przed nawiasem klamrowym rozpoczynającym właściwą treść konstruktora).

Kopiowanie argumentów konstruktora

Kopiowanie argumentów konstruktora do pól obiektu można zapisywać w skrótowej formie `pole1(argument1), pole2(argument2), ...` po nagłówku konstruktora i znaku `:` (przed nawiasem klamrowym rozpoczynającym właściwą treść konstruktora).

```
class Wektor {
  public:
        double x, y;
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
        ...
};
```

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 preinkrementacji.

Do czego przydaje się this?

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

```
class Wektor {
        double x, y;
  public:
        Wektor(void): x(0), y(0) {}
        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 argumentem typu `int`.

Przeciążanie postinkrementacji

W celu przeciążenia postinkrementacji trzeba zdefiniować metodę z jednym argumentem typu `int`.

```
class Wektor {
        double x, y;
  public:
        Wektor(void): x(0), y(0) {}
        Wektor(double a, double b): x(a), y(b) {}
        double norma(void) const
        {
                return x*x + y*y;
        }
        Wektor operator ++(int zero)
        {
                Wektor w(x, y);

                x++;
                y++;
                return w;
        }
};
```

```
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;

// Dlaczego wykonanie tego nie spowoduje błędu?
cout << a[0] << " " << a[1] << " " << a[2] << 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;  // a.operator=(b)
```

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;  // a.operator=(r)
```

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

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): x(0), y(0) {}
        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): x(0), y(0) {}
        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);
};
```

Klasy zaprzyjaźnione z innymi klasami

Jeżeli pola danej klasy (np. A) są prywatne, ale metody z innej klasy (np. B) mają mieć do nich dostęp, to można zadeklarować klasę B jako „przyjaciela" (*ang. friend*) klasy A.

Klasy zaprzyjaźnione z innymi klasami

Jeżeli pola danej klasy (np. A) są prywatne, ale metody z innej klasy (np. B) mają mieć do nich dostęp, to można zadeklarować klasę B jako „przyjaciela" (*ang. friend*) klasy A.

```
class A {
        double a;
  public:
        A(void): x(0) {}
        A(double x): a(x) {}
        ...
        friend class B;  // Metody z B mają dostęp do pola a z A
};
```

Przeciążanie operacji konwersji typów danych

W C++ konwersja typów danych, jak np. `(double)zmienna`, może być przeciążona podobnie do operatorów.

Przeciążanie operacji konwersji typów danych

W C++ konwersja typów danych, jak np. `(double)zmienna`, może być przeciążona podobnie do operatorów.

W celu przeciążenia operacji konwersji typów danych w klasie, dla której chcemy zmienić działanie tej operacji, definiuje się metodę o nazwie `operator typ()`, gdzie `typ` jest typem danych, na który będą „konwertowane" obiekty danej klasy, np. `operator double()`.

Przeciążanie operacji konwersji typów danych

W C++ konwersja typów danych, jak np. `(double)zmienna`, może być przeciążona podobnie do operatorów.

W celu przeciążenia operacji konwersji typów danych w klasie, dla której chcemy zmienić działanie tej operacji, definiuje się metodę o nazwie `operator typ()`, gdzie `typ` jest typem danych, na który będą „konwertowane" obiekty danej klasy, np. `operator double()`.

W definicji metody `operator double()` oraz analogicznych metod dla innych typów danych nie deklaruje się typu zwracanego wyniku i argumentów (są one określone domyślnie).

Przeciążanie konwersji typów danych – przykład

Metoda `operator double()` określa znaczenie zapisu `(double)w` oraz jest wykorzystywana przy niejawnych konwersjach (np. nadaje sens wyrażeniu `w == 25`).

```
#include <iostream>
using namespace std;

class Wektor {
  public:
    double x, y;
    Wektor(void): x(0), y(0) {}
    Wektor(double a, double b): x(a), y(b) {}
    operator double() const { return x*x + y*y; }
};

int main()
{
    Wektor w(3, 4);

    cout << (double)w << endl;  // w.operator double()
    if (w == 25)  // w.operator double() == 25
        cout << "OK" << endl;

    return 0;
}
```

Klasy nadrzędne i podrzędne

Jedna klasa jest podrzędna (*ang. secondary*) w stosunku do drugiej lub jest jej podklasą (*ang. subclass*), lub jest klasą pochodną (*ang. derived class*) w stosunku do niej, jeżeli dziedziczy (*ang. inherit*) po niej część składników (tzn. część jej składników pochodzi z tamtej klasy).

Klasy nadrzędne i podrzędne

Jedna klasa jest podrzędna (*ang. secondary*) w stosunku do drugiej lub jest jej podklasą (*ang. subclass*), lub jest klasą pochodną (*ang. derived class*) w stosunku do niej, jeżeli dziedziczy (*ang. inherit*) po niej część składników (tzn. część jej składników pochodzi z tamtej klasy).

Klasa, po której składniki są dziedziczone, nazywana jest klasą nadrzędną (*ang. primary*) w stosunku do klasy dziedziczącej lub jej nadklasą (*ang. superclass*), lub klasą macierzystą (*ang. parent class*) albo klasą bazową (*ang. base class*) w stosunku do niej.

Klasy nadrzędne i podrzędne

Jedna klasa jest podrzędna (*ang. secondary*) w stosunku do drugiej lub jest jej podklasą (*ang. subclass*), lub jest klasą pochodną (*ang. derived class*) w stosunku do niej, jeżeli dziedziczy (*ang. inherit*) po niej część składników (tzn. część jej składników pochodzi z tamtej klasy).

Klasa, po której składniki są dziedziczone, nazywana jest klasą nadrzędną (*ang. primary*) w stosunku do klasy dziedziczącej lub jej nadklasą (*ang. superclass*), lub klasą macierzystą (*ang. parent class*) albo klasą bazową (*ang. base class*) w stosunku do niej.

Podklasy można traktować jako bardziej precyzyjne specyfikacje rzeczy mających wspólne własności (w takim sensie „jamnik" jest bardziej precyzyjnym okerśleniem własności zwierzęcia, niż „pies").

Definiowanie klas pochodnych

W C++ w definicji klasy można umieścić listę klas nadrzędnych w stosunku do niej (z modyfikatorami dostępu).

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};
```

```
class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

W zasięgu tej definicji klasa `Wektor_3D` będzie zawierała pola `x`, `y` i `z`. Metody zdefiniowane w klasie `Wektor_3D` zastępują analogiczne metody zdefiniowane w klasie macierzystej.

Definiowanie klas pochodnych

W C++ w definicji klasy można umieścić listę klas nadrzędnych w stosunku do niej (z modyfikatorami dostępu).

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};
```

```
class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

W zasięgu tej definicji klasa `Wektor_3D` będzie zawierała pola `x`, `y` i `z`. Metody zdefiniowane w klasie `Wektor_3D` zastępują analogiczne metody zdefiniowane w klasie macierzystej.

Modyfikator dostępu `public` oznacza, że dostęp do pól `x`, `y` w klasie pochodnej jest taki, jak w klasie macierzystej.

Dziedziczenie i operacje przypisania

Zasada przypisania dla obiektów

Obiektowi klasy bazowej zawsze można przypisać obiekt klasy pochodnej. Wówczas *domyślnie* (tzn. jeśli przypisanie nie jest przeciążone *w klasie bazowej*) wartości pól będących składnikami obydwu klas są kopiowane z obiektu klasy pochodnej do obiektu klasy bazowej (zgodnie z kierunkiem przypisania).

Dziedziczenie i operacje przypisania

Zasada przypisania dla obiektów

Obiektowi klasy bazowej zawsze można przypisać obiekt klasy pochodnej. Wówczas *domyślnie* (tzn. jeśli przypisanie nie jest przeciążone *w klasie bazowej*) wartości pól będących składnikami obydwu klas są kopiowane z obiektu klasy pochodnej do obiektu klasy bazowej (zgodnie z kierunkiem przypisania).

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

```
Wektor_3D w_3d;
Wektor_2D w_2d;

...
w_2d = w_3d;  // OK

// Wartości pól x, y z obiektu w_3d są kopiowane
// do pól x, y w obiekcie w_2d (odpowiednio).

// Przypisanie w odwrotnym kierunku
// byłoby błędne!
```

Dziedziczenie, referencje i operacje przypisania

Zasada przypisania dla referencji

Obiekt klasy pochodnej zawsze może być przedmiotem (celem) referencji do obiektu klasy bazowej.

Dziedziczenie, referencje i operacje przypisania

Zasada przypisania dla referencji

Obiekt klasy pochodnej zawsze może być przedmiotem (celem) referencji do obiektu klasy bazowej.

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

```
void drukuj(Wektor_2D& wektor)
{
        cout << '(' << wektor.x << ','
                << wektor.y << ')' << endl;
}

...
Wektor_3D w_3d;

...
drukuj(w_3d);  // OK

// Wewnątrz funkcji drukuj() obiekt w_3d
// występuje pod nazwą wektor i
// jest traktowany jako obiekty klasy
// Wektor_2D.
```

Dziedziczenie, wskaźniki i operacje przypisania

Zasada przypisania dla wskaźników

Wskaźnikowi do obiektów klasy bazowej zawsze można przypisać adres obiektu klasy pochodnej. Wówczas wskaźnik ten może być używany przy odwołaniach do pól i wywoływaniu metod z tego obiektu, traktowanego jako obiekt klasy bazowej.

Dziedziczenie, wskaźniki i operacje przypisania

Zasada przypisania dla wskaźników

Wskaźnikowi do obiektów klasy bazowej zawsze można przypisać adres obiektu klasy pochodnej. Wówczas wskaźnik ten może być używany przy odwołaniach do pól i wywoływaniu metod z tego obiektu, traktowanego jako obiekt klasy bazowej.

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void);
        double abs(void);
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void);
        double abs(void);
};
```

```
Wektor_3D w_3d;
Wektor_2D *wsk;

...
wsk = &w_3d;  // OK

cout << wsk->abs() << endl;  // Wektor_2D::abs()

// Metoda abs() z klasy Wektor_2D zostanie
// wywołana w kontekście obiektu w_3d i będzie
// go traktować jako obiekt klasy Wektor_2D.
```

Polimorfizm

Obiekty klas pochodnych mogą zastępować obiekty klas bazowych we wszystkich sytuacjach. W szczególności dowolna funkcja operująca obiektami klasy bazowej może także operować obiektami dowolnej klasy pochodnej w stosunku do niej (wtedy te obiekty są traktowane jako obiekty klasy bazowej).

Polimorfizm

Obiekty klas pochodnych mogą zastępować obiekty klas bazowych we wszystkich sytuacjach. W szczególności dowolna funkcja operująca obiektami klasy bazowej może także operować obiektami dowolnej klasy pochodnej w stosunku do niej (wtedy te obiekty są traktowane jako obiekty klasy bazowej).

Polimorfizm (*ang. polymorphism*)

Własność języka programowania pozwalająca posługiwać się zmiennymi o różnych typach danych w jednakowy sposób.

Polimorfizm

Obiekty klas pochodnych mogą zastępować obiekty klas bazowych we wszystkich sytuacjach. W szczególności dowolna funkcja operująca obiektami klasy bazowej może także operować obiektami dowolnej klasy pochodnej w stosunku do niej (wtedy te obiekty są traktowane jako obiekty klasy bazowej).

Polimorfizm (*ang. polymorphism*)

Własność języka programowania pozwalająca posługiwać się zmiennymi o różnych typach danych w jednakowy sposób.

Polimorficzne funkcje

Mogą operować argumentami o różnych typach danych (np. funkcje w C++, których argumenty są referencjami lub wskaźnikami do zmiennych obiektowych).

Dziedziczenie i metody o jednakowych nagłówkach

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
        double abs(void);
        {
                return sqrt(norma());
        }
};
```

Dziedziczenie i metody o jednakowych nagłówkach

```
class Wektor_2D {
  public:
        double x, y;
        double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
        double abs(void);
        {
                return sqrt(norma());
        }
};
```

```
void drukuj_abs(Wektor_2D& wektor)
{
        cout << wektor.abs() << endl;
}

int main()
{
        Wektor_3D w;

        w.x = 0;
        w.y = 3;
        w.z = 4;

        drukuj_abs(w);  // Wydrukuje 3

        // Zmienna w jest traktowana przez
        // funkcję drukuj_abs() tak, jakby była
        // klasy Wektor_2D, więc zostaną użyte
        // metody abs() i norma() zdefiniowane
        // dla klasy Wektor_2D.

        return 0;
}
```

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda zdefiniowana w pewnej klasie w taki sposób, że w klasach pochodnych w stosunku do tej klasy można wstawić na jej miejsce (*ang. override*) nowe metody z takim samym nagłówkiem.

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda zdefiniowana w pewnej klasie w taki sposób, że w klasach pochodnych w stosunku do tej klasy można wstawić na jej miejsce (*ang. override*) nowe metody z takim samym nagłówkiem.

Jeżeli metoda zostanie zadeklarowana jako wirtualna (w klasie bazowej) i w klasie pochodnej w stosunku do niej jest zdefiniowana metoda o takim samym nagłówku, to dla obiektów klasy pochodnej w każdej sytuacji będzie wywoływana metoda zdefiniowana w klasie pochodnej.

Wirtualne metody

Wirtualna metoda (*ang. virtual method*)

Metoda zdefiniowana w pewnej klasie w taki sposób, że w klasach pochodnych w stosunku do tej klasy można wstawić na jej miejsce (*ang. override*) nowe metody z takim samym nagłówkiem.

Jeżeli metoda zostanie zadeklarowana jako wirtualna (w klasie bazowej) i w klasie pochodnej w stosunku do niej jest zdefiniowana metoda o takim samym nagłówku, to dla obiektów klasy pochodnej w każdej sytuacji będzie wywoływana metoda zdefiniowana w klasie pochodnej.

W szczególności będzie tak w przypadku, gdy obiekt klasy pochodnej zastępuje obiekt klasy bazowej zawierającej wirtualną metodę.

Deklarowanie wirtualnych metod

```
class Wektor_2D {
  public:
        double x, y;
        virtual double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
};
```

Deklarowanie wirtualnych metod

```
class Wektor_2D {
  public:
        double x, y;
        virtual double norma(void)
        {
                return x*x + y*y;
        }
        double abs(void)
        {
                return sqrt(norma());
        }
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        double norma(void)
        {
                return x*x + y*y + z*z;
        }
};
```

```
void drukuj_abs(Wektor_2D& wektor)
{
        cout << wektor.abs() << endl;
}

int main()
{
        Wektor_3D w_3d;

        w_3d.x = 0;
        w_3d.y = 3;
        w_3d.z = 4;

        drukuj_abs(w_3d);  // Wydrukuje 5

        // Zmienna w jest traktowana przez
        // funkcję drukuj_abs() tak, jakby była
        // klasy Wektor_2D, ale metoda norma()
        // jest wirtualna, więc dla zmiennej w_3d
        // będzie wywołana metoda norma()
        // zdefiniowana dla Wektor_3D.

        return 0;
}
```

Wirtualne destruktory

W C++ destruktor może być zadeklarowany jako wirtualna metoda.

Wirtualne destruktory

W C++ destruktor może być zadeklarowany jako wirtualna metoda.

W takim przypadku destruktor zdefiniowany w klasie pochodnej jest zawsze wykonywany dla obiektów tej klasy przed destruktorem zdefiniowanym dla klasy bazowej.

Wirtualne destruktory

W C++ destruktor może być zadeklarowany jako wirtualna metoda.

W takim przypadku destruktor zdefiniowany w klasie pochodnej jest zawsze wykonywany dla obiektów tej klasy przed destruktorem zdefiniowanym dla klasy bazowej.

Jeżeli destruktor nie jest wirtualny, to destruktor zdefiniowany dla klasy pochodnej może nie być wykonany, w zależności od sposobu operowania obiektem.

Wywoływanie konstruktora klasy bazowej

Konstruktor klasy pochodnej może (i na ogół powinien) jawnie określić który konstruktor klasy bazowej należy wykonać.

```
class Wektor_2D {
  public:
        double x, y;
        Wektor_2D(double a, double b): x(a), y(b) {}
        ...
};

class Wektor_3D : public Wektor_2D {
  public:
        double z;
        Wektor_3D(double a, double b, double c): Wektor_2D(a, b), z(c) {}
        ...
};
```

Modyfikator dostępu `protected`

Składniki klasy zadeklarowane z modyfikatorem dostępu `protected` są dostępne dla metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

Modyfikator dostępu protected

Składniki klasy zadeklarowane z modyfikatorem dostępu `protected` są dostępne dla metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

```
class Wektor_2D {
  protected:
        double x, y;
  public:
        virtual double norma(void);
        ...
};

class Wektor_3D : public Wektor_2D {
        double z;
  public:
        double norma(void)
        {
                return x*x + y*y + z*z;   // OK
        }
        ...
};
```

Modyfikator dostępu protected

Składniki klasy zadeklarowane z modyfikatorem dostępu `protected` są dostępne dla metod będących składnikami tej klasy oraz klas pochodnych w stosunku do niej.

```
class Wektor_2D {
  protected:
        double x, y;
  public:
        virtual double norma(void);
        ...
};

class Wektor_3D : public Wektor_2D {
        double z;
  public:
        double norma(void)
        {
                return x*x + y*y + z*z;   // OK
        }
        ...
};
```

```
...

int main()
{
        ...

        cout << w.x << endl; // Błąd!
        ...
}
```

Dostęp do składników klas bazowych

Domyślnie składniki klasy bazowej są traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu `private`.

Dostęp do składników klas bazowych

Domyślnie składniki klasy bazowej są traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu `private`.

Dla składników klasy bazowej można jawnie określić dostęp do nich w obiektach klas pochodnych, z pomocą modyfikatora dostępu przed nazwą klasy bazowej w definicji klasy pochodnej.

```
class Wektor_2D {
  public:
        double x, y;
        ...
};

class Wektor_3D : public Wektor_2D {
        ...
};
```

Dostęp do składników klas bazowych

Domyślnie składniki klasy bazowej są traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu `private`.

Dla składników klasy bazowej można jawnie określić dostęp do nich w obiektach klas pochodnych, z pomocą modyfikatora dostępu przed nazwą klasy bazowej w definicji klasy pochodnej.

```
class Wektor_2D {
  public:
        double x, y;
        ...
};

class Wektor_3D : public Wektor_2D {
        ...
};
```

`public` oznacza, że dostęp do składników klasy bazowej w obiekcie klasy pochodnej ma być taki, jaki byłby w obiekcie klasy bazowej.

Dostęp do składników klas bazowych i protected

protected oznacza, że składniki klasy bazowej zadeklarowane (w definicji klasy bazowej) z modyfikatorem dostępu public mają być traktowane tak, jakby były zadeklarowane w klasie pochodnej z modyfikatorem dostępu protected.

```
class Wektor_2D {
  public:
        double x, y;
        ...
};

class Wektor_3D : protected Wektor_2D {
        ...
};

int main()
{
        Wektor_3D w3d;

        w3d.x = 0;  // Błąd!
        ...
}
```

Klasy abstrakcyjne

Klasa abstrakcyjna (*ang. abstract class*)

Klasa, dla której nie można zdefiniować obiektu. Można tylko tworzyć klasy pochodne w stosunku do niej i tworzyć obiekty tych klas.

Klasy abstrakcyjne

Klasa abstrakcyjna (*ang. abstract class*)

Klasa, dla której nie można zdefiniować obiektu. Można tylko tworzyć klasy pochodne w stosunku do niej i tworzyć obiekty tych klas.

W C++ klasę abstrakcyją definiuje się deklarując przynajmniej jedną wirtualną metodę bez implementacji.

Klasy abstrakcyjne

Klasa abstrakcyjna (*ang. abstract class*)

Klasa, dla której nie można zdefiniować obiektu. Można tylko tworzyć klasy pochodne w stosunku do niej i tworzyć obiekty tych klas.

W C++ klasę abstrakcyją definiuje się deklarując przynajmniej jedną wirtualną metodę bez implementacji.

```
class Wektor {
  public:
        double x, y;
        virtual double norma(void) = 0;
        ...
};
```

```
class Wektor_2D : public Wektor {
  public:
        double norma(void)
        {
                return x*x + y*y;
        }
        ...
};
```

Wejście-wyjście w C++

`ios_base`

Klasa bazowa dla wszystkich klas wejścia-wyjścia reprezentujących strumienie danych (*ang. data stream*).

Wejście-wyjście w C++

`ios_base`

Klasa bazowa dla wszystkich klas wejścia-wyjścia reprezentujących strumienie danych (*ang. data stream*).

`ios`

Klasa pochodna od `ios_base`. Zawiera składniki wspólne dla strumieni wejściowych (*ang. input*) i wyjściowych (*ang. output*).

Klasy wejścia-wyjścia w C++

`istream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wejściowych, niezależnie od źródła (*ang. source*) danych, m. in. rodzinę metod `operator >>()`.

`ostream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wyjściowych, niezależnie od miejsca przeznaczenia (*ang. destination*) danych, m. in. rodzinę metod `operator <<()`.

Klasy wejścia-wyjścia w C++

`istream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wejściowych, niezależnie od źródła (*ang. source*) danych, m. in. rodzinę metod `operator >>()`.

`ostream`

Klasa pochodna od `ios`. Zawiera składniki potrzebne do obsługi strumieni wyjściowych, niezależnie od miejsca przeznaczenia (*ang. destination*) danych, m. in. rodzinę metod `operator <<()`.

`iostream`

Klasa pochodna od `istream` i `ostream`. Zawiera składniki potrzebne do obsługi strumieni wejściowych i wyjściowych, niezależnie od źródła lub miejsca przeznaczenia danych.

Klasy wejścia-wyjścia w C++ – c. d.

`ifstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z pliku (tekstowego).

`ofstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych w pliku (tekstowym).

Klasy wejścia-wyjścia w C++ – c. d.

`ifstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z pliku (tekstowego).

`ofstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych w pliku (tekstowym).

`fstream`

Klasa pochodna od `iostream`. Zawiera składniki potrzebne do zapisywania danych w pliku (tekstowym) i odczytywania danych z pliku (tekstowego).

Klasy wejścia-wyjścia w C++ – c. d.

`istringstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z ciągu znaków (reprezentowanego przez obiekt klasy `string`).

`ostringstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych do ciągu znaków (reprezentowanego przez obiekt klasy `string`).

Klasy wejścia-wyjścia w C++ – c. d.

`istringstream`

Klasa pochodna od `istream`. Zawiera składniki potrzebne do odczytywania danych z ciągu znaków (reprezentowanego przez obiekt klasy `string`).

`ostringstream`

Klasa pochodna od `ostream`. Zawiera składniki potrzebne do zapisywania danych do ciągu znaków (reprezentowanego przez obiekt klasy `string`).

`stringstream`

Klasa pochodna od `iostream`. Zawiera składniki potrzebne do odczytywania danych z ciągu znaków i zapisywania danych do ciągu znaków (reprezentowanego przez obiekt klasy `string`).

Błędy wejścia-wyjścia i konwersja typów danych

W klasach `istream` i `ostream` przeciążenie konwersji typów danych pozwala na używanie obiektów tych klas w wyrażeniach typów całkowitych oraz typu `bool`.

Błędy wejścia-wyjścia i konwersja typów danych

W klasach `istream` i `ostream` przeciążenie konwersji typów danych pozwala na używanie obiektów tych klas w wyrażeniach typów całkowitych oraz typu `bool`.

Obowiązuje przy tym zasada, że jeśli obiekt reprezentuje wartość liczbową 0 lub wartość `false` typu `bool`, to nie jest skojarzony z żadnym źródłem lub miejscem przeznaczenia danych albo reprezentuje strumień, dla którego ostatnia operacja wejścia-wyjścia zakończyła się niepowodzeniem.

Błędy wejścia-wyjścia i konwersja typów danych

W klasach `istream` i `ostream` przeciążenie konwersji typów danych pozwala na używanie obiektów tych klas w wyrażeniach typów całkowitych oraz typu `bool`.

Obowiązuje przy tym zasada, że jeśli obiekt reprezentuje wartość liczbową 0 lub wartość `false` typu `bool`, to nie jest skojarzony z żadnym źródłem lub miejscem przeznaczenia danych albo reprezentuje strumień, dla którego ostatnia operacja wejścia-wyjścia zakończyła się niepowodzeniem.

Operacje wejścia-wyjścia reprezentowane przez symbole >> oraz << zwracają wyniki będące referencjami do obiektów klasy `istream` i `ostream`, odpowiednio.

Wyniki operacji wejścia-wyjścia

Operator wyjścia << zwraca wynik typu `ostream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem, tzn. czy „konwersja" tego obiektu na wartość liczbową (lub typu `bool`) daje liczbę różną od zera (lub `true`).

Wyniki operacji wejścia-wyjścia

Operator wyjścia << zwraca wynik typu `ostream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem, tzn. czy „konwersja" tego obiektu na wartość liczbową (lub typu `bool`) daje liczbę różną od zera (lub `true`).

Operator wejścia >> zwraca wynik typu `istream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem – jak dla operatora wyjścia.

Wyniki operacji wejścia-wyjścia

Operator wyjścia << zwraca wynik typu `ostream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem, tzn. czy „konwersja" tego obiektu na wartość liczbową (lub typu `bool`) daje liczbę różną od zera (lub `true`).

Operator wejścia >> zwraca wynik typu `istream&`, który można wykorzystać do sprawdzenia, czy ostatnia operacja zakończyła się sukcesem – jak dla operatora wyjścia.

Przykład

```
if (cin >> r)  // Odczytaj liczbę z cin i sprawdź, czy udało się.
    cout << r*r << endl;
```

Łączenie operacji wejścia-wyjścia

Wyniki zwracane przez >> i << pozwalają na łączenie operacji wejścia-wyjścia w ciągi.

Przykład

1. Zapisz `a` do `cout` i zwróć referencję do `cout` jako wynik.
2. Zapisz `b` w strumieniu, do którego referencja została zwrócona jako wynik poprzedniej operacji i zwróć referencję do tego strumienia jako wynik.
3. Zapisz `endl` w strumieniu, do którego referencja została zwrócona jako wynik poprzedniej operacji i zwróć referencję do tego strumienia jako wynik.

```
cout << a << b << endl;  // lub ((cout << a) << b) << endl;
```

Przeciążanie operacji wejścia-wyjścia

Operatory wejścia-wyjścia >> i << mogą być zdefiniowane dla każdej klasy z wykorzystaniem standardowego mechanizmu przeciążania operatorów z użyciem funkcji o dwóch argumentach.

Przeciążanie operacji wejścia-wyjścia

Operatory wejścia-wyjścia >> i << mogą być zdefiniowane dla każdej klasy z wykorzystaniem standardowego mechanizmu przeciążania operatorów z użyciem funkcji o dwóch argumentach.

```
#include <iostream>
using namespace std;

class Wektor {
  public:
    double x, y;
    Wektor(void): x(0), y(0) {}
    Wektor(double a, double b): x(a), y(b) {}
    ...  // Inne metody i pola.
};

ostream& operator <<(ostream& out, Wektor& w)
{
    return out << '(' << w.x << ", " << w.y << ')';
}
```

```
int main()
{
    Wektor w(3, 4);

    ...  // Inne instrukcje.
    // Wykonaj (operator <<(cout, w)) << endl
    cout << w << endl;

    return 0;
}
```

Błędny argument metody-operatora

W następującym kodzie jest problem polegający na tym, że przekazanie błędnego argumentu metodzie `operator =()` zostanie zignorowane, ale wynik wykonania programu prawdopodobnie będzie błędny.

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

...
```

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

Błędny argument metody-operatora

W następującym kodzie jest problem polegający na tym, że przekazanie błędnego argumentu metodzie `operator =()` zostanie zignorowane, ale wynik wykonania programu prawdopodobnie będzie błędny.

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

...
```

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

Rozwiązaniem może być zapisanie metody `operator =()` tak, aby zgłaszała wyjątek, gdy warunek `n == t.n` nie jest spełniony.

Co to jest wyjątek?

Wyjątek (*ang. exception*)

Błąd ujawniający się podczas wykonywania programu, który powoduje, że wykonywanie go w normalny sposób nie może być kontynuowane.

Co to jest wyjątek?

Wyjątek (*ang. exception*)

Błąd ujawniający się podczas wykonywania programu, który powoduje, że wykonywanie go w normalny sposób nie może być kontynuowane.

Zgłoszenie wyjątku (*ang. throwing an exception*)

Czynność polegająca na przerwaniu aktualnie wykonywanego kodu (najczęściej funkcji) i utworzeniu obiektu (dowolnego typu) zawierającego informacje o błędzie.

Co to jest wyjątek?

Wyjątek (*ang. exception*)

Błąd ujawniający się podczas wykonywania programu, który powoduje, że wykonywanie go w normalny sposób nie może być kontynuowane.

Zgłoszenie wyjątku (*ang. throwing an exception*)

Czynność polegająca na przerwaniu aktualnie wykonywanego kodu (najczęściej funkcji) i utworzeniu obiektu (dowolnego typu) zawierającego informacje o błędzie.

Instrukcja `throw`

W C++ wyjątki zgłasza się z pomocą specjalnej instrukcji, złożonej ze słowa kluczowego `throw` i wyrażenia (dowolnego typu), które stanowi jej argument.

Zgłoszenie wyjątku z użyciem `throw`

Argument instrukcji `throw` służy do utworzenia obiektu zawierającego informacje o błędzie, który spowodował zgłoszenie wyjątku (reprezentującego wyjątek).

Zgłoszenie wyjątku z użyciem `throw`

Argument instrukcji `throw` służy do utworzenia obiektu zawierającego informacje o błędzie, który spowodował zgłoszenie wyjątku (reprezentującego wyjątek).

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

...
```

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n != t.n)
                throw -1;

        for (int i = 0; i < n; i++)
                elem[i] = t.elem[i];
        return *this;
}
```

Zgłoszenie wyjątku z użyciem `throw`

Argument instrukcji `throw` służy do utworzenia obiektu zawierającego informacje o błędzie, który spowodował zgłoszenie wyjątku (reprezentującego wyjątek).

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

...
```

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n != t.n)
                throw -1;

        for (int i = 0; i < n; i++)
                elem[i] = t.elem[i];
        return *this;
}
```

Metoda `operator =()` zgłasza wyjątek, jeżeli obiekty po obu stronach symbolu przypisania nie są ze sobą zgodne.

Skutki użycia `throw`

Funkcja zgłaszająca wyjątek

W wyniku wykonania `throw` funkcja (lub metoda) zgłaszająca wyjątek jest przerywana (podobnie, jak w przypadku wykonania `return`), a obiekt reprezentujący wyjątek jest przekazywany do funkcji (lub metody), która ją wywołała.

Skutki użycia `throw`

Funkcja zgłaszająca wyjątek

W wyniku wykonania `throw` funkcja (lub metoda) zgłaszająca wyjątek jest przerywana (podobnie, jak w przypadku wykonania `return`), a obiekt reprezentujący wyjątek jest przekazywany do funkcji (lub metody), która ją wywołała.

Funkcja, która wywołała funkcję zgłaszającą wyjątek

Odebranie wyjątku (obiektu reprezentującego wyjątek) od wywołanej przez nią funkcji (metody) ma taki skutek, jakby ona sama zgłosiła wyjątek (tzn. jakby w trakcie jej wykonywania nastąpiło wykonanie `throw` z takim argumentem, jak obiekt reprezentujący wyjątek).

Przekazywanie obiektów reprezentujących wyjątki

Jeżeli `a()` wywołuje `b()`, która wywołuje `c()`, to zgłoszenie wyjątku przez `c()` spowoduje przekazanie obiektu reprezentującego wyjątek do `b()`, a następnie do `a()` (i tak dalej).

Przekazywanie obiektów reprezentujących wyjątki

Jeżeli `a()` wywołuje `b()`, która wywołuje `c()`, to zgłoszenie wyjątku przez `c()` spowoduje przekazanie obiektu reprezentującego wyjątek do `b()`, a następnie do `a()` (i tak dalej).

Obiekty reprezentujące wyjątki są przekazywane „w dół” łańcucha wywołań funkcji (*ang. function call chain*) i ostatecznie docierają do funkcji `main()`, chyba że „po drodze” zostaną przechwycone (*ang. catch*).

Przekazywanie obiektów reprezentujących wyjątki

Jeżeli `a()` wywołuje `b()`, która wywołuje `c()`, to zgłoszenie wyjątku przez `c()` spowoduje przekazanie obiektu reprezentującego wyjątek do `b()`, a następnie do `a()` (i tak dalej).

Obiekty reprezentujące wyjątki są przekazywane „w dół" łańcucha wywołań funkcji (*ang. function call chain*) i ostatecznie docierają do funkcji `main()`, chyba że „po drodze" zostaną przechwycone (*ang. catch*).

Dotarcie obiektu reprezentującego wyjątek do funkcji `main()` powoduje natychmiastowe przerwanie wykonywania programu (z odpowiednim komunikatem o błędzie), chyba że (obiekt reprezentujący) wyjątek zostanie przechwycony wewnątrz funkcji `main()`.

Przechwytywanie wyjątków

Bloki `try` i `catch`

Wyjątek może być przechwycony, jeżeli:

1. Wywołanie zgłaszającej go funkcji miało miejsce w bloku poprzedzanym przez słowo kluczowe `try` oraz
2. z tym blokiem związany jest blok poprzedzany słowem kluczowym `catch` i definicją zmiennej (w nawiasie okrągłym), której typ danych odpowiada typowi danych obiektu reprezentującego wyjątek.

Przechwytywanie wyjątków

Bloki `try` i `catch`

Wyjątek może być przechwycony, jeżeli:

1. Wywołanie zgłaszającej go funkcji miało miejsce w bloku poprzedzanym przez słowo kluczowe `try` oraz
2. z tym blokiem związany jest blok poprzedzany słowem kluczowym `catch` i definicją zmiennej (w nawiasie okrągłym), której typ danych odpowiada typowi danych obiektu reprezentującego wyjątek.

```
Tablica& Tablica::operator =(Tablica& t)
{
        if (n != t.n)
                throw -1;
        for (int i = 0; i < n; i++)
                elem[i] = t.elem[i];
        return *this;
}
```

```
void kopiuj(Tablica& a, Tablica& b)
{
        try {
                a = b;
        } catch (int kod) {
                cerr << "BŁĄD: " << kod << endl;
        }
}
```

Zasady przechwytywania wyjątków

Z jednym blokiem `try` można związać dowolną liczbę bloków `catch`. Wówczas muszą one następować jeden po drugim (bez żadnych instrukcji między nimi).

Zasady przechwytywania wyjątków

Z jednym blokiem `try` można związać dowolną liczbę bloków `catch`. Wówczas muszą one następować jeden po drugim (bez żadnych instrukcji między nimi).

Jeżeli wiele bloków `catch` „pasuje” do obiektu reprezentującego wyjątek, wybrany zostanie ten, który jest zdefiniowany jako pierwszy.

Zasady przechwytywania wyjątków

Z jednym blokiem `try` można związać dowolną liczbę bloków `catch`. Wówczas muszą one następować jeden po drugim (bez żadnych instrukcji między nimi).

Jeżeli wiele bloków `catch` „pasuje" do obiektu reprezentującego wyjątek, wybrany zostanie ten, który jest zdefiniowany jako pierwszy.

Jeżeli blok `catch` zostanie „dopasowany" do obiektu reprezentującego wyjątek (tzn. jego typ danych odpowiada typowi danych zmiennej zdefiniowanej w nawiasie okrągłym po słowie kluczowym `catch` dla tego bloku), to zostaną wykonane instrukcje znajdujące się wewnątrz tego bloku.

Zasady przechwytywania wyjątków c. d.

Jeżeli blok `catch` zostanie „dopasowany” do wyjątku (tzn. do obiektu reprezentującego wyjątek), to:

1. zmienna zdefiniowana w nawiasie okrągłym po słowie kluczowym `catch` dla tego bloku otrzyma wartość równą obiektowi reprezentującemu wyjątek oraz
2. zmienna ta może być wykorzystywana w instrukcjach wewnątrz tego bloku tak, jak gdyby była argumentem funkcji (można ją nazwać zmienną reprezentującą wyjątek).

Zasady przechwytywania wyjątków c. d.

Jeżeli blok `catch` zostanie „dopasowany” do wyjątku (tzn. do obiektu reprezentującego wyjątek), to:

1. zmienna zdefiniowana w nawiasie okrągłym po słowie kluczowym `catch` dla tego bloku otrzyma wartość równą obiektowi reprezentującemu wyjątek oraz
2. zmienna ta może być wykorzystywana w instrukcjach wewnątrz tego bloku tak, jak gdyby była argumentem funkcji (można ją nazwać zmienną reprezentującą wyjątek).

Domyślna (*ang. default*) obsługa wyjątku

Blok `catch`, dla którego w nawiasie okrągłym po słowie kluczowym `catch` (zamiast definicji zmiennej) znajduje się wielokropek (`...`), zostanie dopasowany do każdego wyjątku. W tym bloku nie można wykorzystywać zmiennej reprezentującej wyjątek.

Natychmiastowe przechwytywanie wyjątków

Jeżeli wykonanie `throw` nastąpi bezpośrednio w bloku `try`, z którym związany jest blok `catch` „pasujący” do obiektu reprezentującego wyjątek, to przerwane zostanie tylko wykonywanie bloku `try` zawierającego instrukcję `throw`.

Natychmiastowe przechwytywanie wyjątków

Jeżeli wykonanie `throw` nastąpi bezpośrednio w bloku `try`, z którym związany jest blok `catch` „pasujący" do obiektu reprezentującego wyjątek, to przerwane zostanie tylko wykonywanie bloku `try` zawierającego instrukcję `throw`.

W takim przypadku obiekt reprezentujący wyjątek jest przekazywany bezpośrednio do odpowiedniego bloku `catch` bez przerywania wykonywania funkcji, w której zgłaszany jest wyjątek.

Natychmiastowe przechwytywanie wyjątków

Jeżeli wykonanie `throw` nastąpi bezpośrednio w bloku `try`, z którym związany jest blok `catch` „pasujący" do obiektu reprezentującego wyjątek, to przerwane zostanie tylko wykonywanie bloku `try` zawierającego instrukcję `throw`.

W takim przypadku obiekt reprezentujący wyjątek jest przekazywany bezpośrednio do odpowiedniego bloku `catch` bez przerywania wykonywania funkcji, w której zgłaszany jest wyjątek.

```
try {
        ...
        throw MyException;
        ...
} catch (MyException e) {
        ...
}
```

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

```
int moja_funkcja(MojaKlasa arg) throw(int)
{
        // Ta funkcja może (ale nie musi) zgłaszać tylko wyjątki typu int
        ...
}
```

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

```
int moja_funkcja(MojaKlasa arg) throw(int)
{
        // Ta funkcja może (ale nie musi) zgłaszać tylko wyjątki typu int
        ...
}
```

Użycie pustej listy typów danych przy `throw()` oznacza, że funkcja w ogóle nie będzie zgłaszać wyjątków.

Deklarowanie typów zgłaszanych wyjątków dla funkcji

Można zadeklarować, że funkcja będzie zgłaszać tylko wyjątki określonego typu (np. `int`).

```
int moja_funkcja(MojaKlasa arg) throw(int)
{
        // Ta funkcja może (ale nie musi) zgłaszać tylko wyjątki typu int
        ...
}
```

Użycie pustej listy typów danych przy `throw()` oznacza, że funkcja w ogóle nie będzie zgłaszać wyjątków.

```
int moja_funkcja(MojaKlasa arg) throw()
{
        // Ta funkcja nie może zgłaszać wyjątków
        ...
}
```

Standardowe klasy reprezentujące wyjątki

`exception`

Klasa bazowa dla standardowych klas używanych do reprezentowania wyjątków.

Standardowe klasy reprezentujące wyjątki

`exception`

Klasa bazowa dla standardowych klas używanych do reprezentowania wyjątków.

Klasami pochodnymi w stosunku do `exception` są m. in. `bad_alloc`, `bad_cast`, `bad_exception`, `bad_typeid`, `logic_error`, `runtime_error`.

Standardowe klasy reprezentujące wyjątki

`exception`

Klasa bazowa dla standardowych klas używanych do reprezentowania wyjątków.

Klasami pochodnymi w stosunku do `exception` są m. in. `bad_alloc`, `bad_cast`, `bad_exception`, `bad_typeid`, `logic_error`, `runtime_error`.

Definiując nową klasę pochodną w stosunku do `exception` zwykle przesłania się metodę `what()`, której wynikiem jest wskaźnik do tablicy znakowej (o elementach typu `char`) zawierającej komunikat o błędzie.

Przykład – wyjątek bad_alloc

```
#include <iostream>
using namespace std;

int main () {
    try {
        double *myarray = new double[1000000000];
    } catch (exception& e) {
        cout << "Standard exception: " << e.what() << endl;
    }

    return 0;
}
```

Przykład – rozszerzanie klasy exception

```
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
 public:
    virtual const char* what() const throw()
    {
        return "My exception happened";
    }

    int x;
    myexception(void): x(0) {}
};
```

```
int main()
{
    myexception myex;

    try {
        throw myex;
    } catch (myexception& e) {
        cout << e.what() << endl;
        e.x = 1;
    }

    cout << myex.x << endl;

    return 0;
}
```

Literatura


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


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


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).