Zgodność typów

Rozważmy klasy

```
class A {
// ...
};
class B : public A {
// ...
};
```

Zgodność typów

Obiekt klasy pochodnej jest obiektem klasy bazowej

```
A a, *aw;
B b, *bw;
a = b;   // OK
b = a;   // Błąd
aw = bw; // OK
bw = aw; // Błąd

void fA(A) ;
fA(a);   // OK, konstr kopiujący A::A(&A)
fA(b);   // OK, A::A(&A) zadziała

void fAref(A&);
fAref(a); // OK, bez kopiowania
fBref(b); // OK

void fAwsk(A*);
fAwsk(&a); // OK
fAwsk(&b); // OK

void fB(B);
fB(a) // Błąd
fB(b) // OK, kopiowanie B::B(B&)

class C {
 public:
  int a ;
  void f() ;
};
```

```
class D: public C {
  public:
   int a ;
   void f() ;
};
```

Na co wskazują wskaźniki?

Zwróćmy uwagę na interesującą konsekwencję zgodności typów:

```
D d;
C *cw = &d;
cw->f();
```

Wprawdzie `cw` wskazuje na obiekt klasy D, ale sam jest wskaźnikiem do obiektu klasy C, zatem wywołana zostanie metoda z tej klasy.

Czasem jednak chcemy innego zachowania...

Rozważmy poniższy przykład:

```
class Figura { // ...
protected: int x, y ; // położenie na ekranie
void ustaw(int ax, int ay) {x=ax; y=ay;};
void pokaż(){}; // Nie umiemy narysować
void schowaj(){};
void przesuń(int,int);
};

Figura::przesuń(int ax, int ay){
  schowaj();
  ustaw(ax,ay);
  pokaż();
}
```

Teraz zdefiniujmy figurę okrąg:

```
class Okrąg : public Figura {
 protected:
  int promień ;
 public:
  Okrąg(int,int,int);
  pokaż();  // umiemy zdefiniować
  schowaj();
```

```
}
Okrąg o(30,40,10); // zadany środek i promień
o.pokaż();         // Rysuje okrąg
o.przesuń(100,200);// Nie przesuwa!
```

metoda `Figura::przesuń` wywołuje metodę `Figura::pokaż`, a nie `Okrąg::pokaż`.

Rozwiązaniem naszego problemu są metody wirtualne

Metody wirtualne

Metoda wirtualna różni się od zwykłej tym, że wybór metody zostaje dokonany dopiero w czasie wykonania, na podstawie faktycznego (a nie zadeklarowanego) typu obiektu.

```
struct A {
  void virtual f(int);
}
struct B : A {};
struct C : B {
  void f(int); // Ta metoda jest wirtualna!
}

main() {
  A *p = new A ;
  p->f(3); // A::f
  p = new B;
  p->f(3); // A::f
  p = new C
  p->f(3)  // C::f
  // Ale:
  A a;
  C c;
  a = c;
  a.f(3); // A::f, bo a jest obiektem klasy A
```

Poprawiona figura

```
class Figura { // ...
protected: int x, y ; // położenie na ekranie
void ustaw(int ax, int ay) {x=ax; y=ay;};
void virtual pokaż(){};
```

```
void virtual schowaj(){};
void przesuń(int,int);
};

Figura::przesuń(int ax, int ay){
  schowaj();
  ustaw(ax,ay);
  pokaż();
}

class Okrąg : public Figura {
 protected:
  int promień ;
 public:
  Okrąg(int,int,int);
  pokaż();  // umiemy zdefiniować
  schowaj();
}
Okrąg o(30,40,10); // zadany środek i promień
o.pokaż();         // Rysuje okrąg
o.przesuń(100,200);// Przesuwa!
```

Teraz metoda `Figura::przesuń` wywołana z obiektu klasy `Okrąg` wywołuje `Okrąg::pokaż`, gdyż `pokaż` jest wirtualna.

Konstruktory i destruktory w hierarchiach

- Obiekt podklasy składa się z wielu warstw;
- każda warstwa odpowiada jednej z nadklas.
- Musimy zadbać o inicjalizację wszystkich warstw.
- Klasy mogą też mieć składowe będące obiektami.
- W podklasie musimy zadbać o inicjalizację:
  - bezpośredniej nadklasy,
  - własnych składowych.
- Konstruktor naszej nadklasy zainicjalizuje swoje nadklasy.

Inicjalizacja nadklas

```
class MyFrame : public Frame ...
// Przykład inicjalizacji nadklasy:
MyFrame::MyFrame()
        : Frame(0,"Basic Test", 0,0,640,480)
// Przykład inicjalizacji składowych:
MWindow::MWindow(int x, int y, int w, int h)
  : _x(x),_y(y),_w(w),_h(h)
```

- Jak widać możemy w ten sposób inicjalizować również składowe typów prostych.
- Nie musimy inicjalizować nadklasy, jeśli ta ma konstruktor domyślny (i wystarczy nam taka inicjalizacja).
- Podobnie, nie musimy inicjalizować składowych, które mają konstruktor domyślny.

Kolejność inicjalizacji

- Najpierw inicjalizuje się klasę bazową,
- następnie składowe w kolejności deklaracji, **niezależnie** od kolejności inicjalizatorów.
- Służy to zagwarantowaniu, że podobiekty i składowe będą niszczone w odwrotnej kolejności niż były inicjalizowane.

```
struct Bar { Bar(){ cout << "Bar" << endl; }; };
struct Foo { Foo(){ cout << "Foo" << endl; }; };
struct Baz { Foo f; Bar b;
             Baz(): b(),f(){};
           };
main() { Baz baz; }
```

Zobaczymy najpierw `Foo` potem `Bar`

Znaczenie inicjalizacji składowych

```
struct A { A(int); A(); } ;
struct B { A a; B(A&); }
```

Rozważmy dwie wersje konstruktora dla B:

```
B::B(A& a2) { a = a2; }
B::B(A& a2):a(a2) {}
```

W pierwszej wykonują się dwie operacje:

- tworzenie i inicjalizacja konstruktorem domyślnym,
- przypisanie

W drugiej tylko jedna:

- tworzenie i inicjalizacja konstruktorem kopiującym.

Destrukcja obiektu

- Treść destruktora wykonuje się przed destruktorami dla obiektów składowych
- Destruktory dla obiektów składowych wykonuje się przed destruktorami klas bazowych.
- W konstruktorach i destruktorach można wywoływać metody, także wirtualne.
- Destruktor może być wirtualny; powinien być wirtualny w klasach zawierających metody wirtualne.

Operatory

- Nowodefiniowane klasy muszą być tak samo dobrymi typami jak typy wbudowane:
  - muszą się dać efektywnie zaimplementować,
  - muszą dać się wygodnie używać.
- to wymaga, by twórca klasy mógł definiować operatory.
- Większość operatorów można przeciążać, tj. definiować ich znaczenie dla własnych klas
- Przeciążenie operatora to zdefiniowanie metody o nazwie składającej się ze słowa operator i nazwy operatora (np. `operator+`).

Przeciążanie operatorów

Poniższe operatory można przeciążać:

- `+ - * / % ^ & | ~ ! && || << >>`
- `= += -= *= /= %= ^= &= |= <<= >>=`
- `< > >= <= == !=`
- `++ --`
- `, ->  ->*`
- `[] ()`
- `new delete` (ale na innych zasadach)

Nie można przeciążać: `. .* :: ?: sizeof`

Wywoływanie operatorów

Operatory można wywoływać w postaci operatorowej:

```
a = b + c;
```

jak i funkcyjnej

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

Są one równoważne; postaci funkcyjnej praktycznie się nie stosuje.

Definiowanie operatorów

- Można dowolnie ustalać typy argumentów i wyniku operatora.
- Nie można zmieniać priorytetu, łączności, ani liczby argumentów.
- Jeśli definiujemy operator jako funkcję, to musi mieć co najmniej jeden argument będący klasą lub referencją do klasy.
- W takim wypadku operator jest zwykle zaprzyjaźniony z klasą, np.

```
class Zespolona { double re, im;
public: // ...
  Zespolona operator+(Zespolona& z) ;
  friend Zespolona
    operator*(Zespolona z1, Zespolona z2);
};
```

Definiowanie operatorów — ograniczenia

- Operatory `= () [] ->` można deklarować tylko jako metody.
- Metody operatorów podlegają dziedziczeniu (za wyjątkiem operatora przypisania).
- Operator nie może mieć argumentów domyślnych.
- Dla operatorów `+ - * &` można przeciążąć zarówno ich postać jednojak i dwuargumentową. Jest to potencjalne źródło błędów i należy uważać.

```
class Zespolona { ... };
Zespolona operator+(Zespolona& z) //...
```

Jaki operator zdefiniowaliśmy?

Operatory jednoargumentowe

Operator jednoargumentowy (prefiksowy) `@` można zadeklarować jako:

- metodę bez argumentów:

  ```
  T operator@()
  ```

  i wówczas `@a` jest interpretowane jako

  ```
  a.operator@()
  ```

- funkcję jednoargumentową

  ```
  T1 operator@(T2)
  ```

  i wtedy `@a` jest interpretowane jako

  ```
  operator@(a)
  ```

Jak zdefiniować `x++`?

Operatorów `++` i `--` można używać w postaci tak prefiksowej jak i postfiksowej.

Dla zdefiniowania ich w wersji postfiksowej wprowadza się "ślepy" argument typu `int`, np

```
struct X {
  X operator++();    // prefiksowe ++x
  X operator++(int); // postfiksowe x++
};
main() {
X x;
++x; // to samo co x.operator++();
x++; // to samo co x.operator++(0);
}
```

Operatory dwuargumentowe

Operator dwuargumentowy (infiksowy) `@` można zadeklarować jako:

- metodę z jednym argumente,:

  ```
  T1 operator@(T2)
  ```

  i wówczas `a@b` jest interpretowane jako

  ```
  a.operator@(b)
  ```

- funkcję dwuargumentową

  ```
  T1 operator@(T2,T3)
  ```

  i wtedy `a@b` jest interpretowane jako

  ```
  operator@(a,b)
  ```

Definiować funkcję czy metodę?

- Operator zwykle definiujemy jako **metodę** — wtedy jest częścią definicji klasy.
- Są jednak sytuacje, kiedy lepsza jest **funkcja**:

- operator ma argumenty dwu różnych klas (operator mógłby być zdefiniowany w pierwszej)
- mamy możliwość modyfikowania tylko jednej z tych klas i może to być akurat druga z nich (np. przy operacjach na strumieniach `operator<<`)
- czasem zamiast definiować wszystkie kombinacje typów argumentów, definiujemy jedną jego postać i odpowiednie konwersje.

Przykład

```
class Zespolona {
  double re, im;
 public:
  Zespolona(double); // konstruktor, ale i konwersja
  Zespolona operator+(Zespolona&);
};
```

Teraz można napisać

```
Zespolona z1, z2;
z2 = z1+1;
```

ale nie można

```
z2 = 1+z1
```

Nie ma tego problemu gdy zdefiniujemy + jako funkcję.

Jeszcze jeden przykład

```
class Zespolona {
  double re, im;
 public:
  friend ostream& operator<<(ostream&, Zespolona&);
  // Teraz << ma dostęp do składowych prywatnych
  // ...
};
ostream& operator<<(ostream& os, Zespolona& z)
{
 os << '(' << z.re << ',' << z.im << ')' ;
 return os ;
}
```

Dokładniej o tym przykładzie przy omawianiu strumieni, zapamiętajmy jednak ten idiom.