Jeszcze o konstruktorach

Konstruktor:

- ma nazwę taką jak klasa,
- nie ma typu wyniku,
- nie może przekazać wyniku instrukcją `return`,
- może wywoływać metody.

Konstruktor domyślny — przykład

```
struct Error {char *what, *where, *why;};

class Zespolona {
double re,im ;
public:
  Zespolona(r,i) {re = r; im = i;}
};
Error e ;      // Ok, konstruktor domyślny
Zespolona z1; // Błąd, nie ma konstr. domyślnego
Zespolona z2(2.0,3.0); // Ok
```

Nie można teraz utworzyć niezainicjalizowanego obiektu klasy `Zespolona`.

Konstruktor bezargumentowy

- można go wywołać bez argumentów
- jest konieczny, jeśli chcemy mieć tablice elementów tej klasy
- jeśli nie zdefiniujemy żadnego konstruktora, kompilator wygeneruje domyślny konstruktor bezargumentowy
- konstruktor domyślny nie wykonuje żadnej inicjalizacji
- **Uwaga:** jeśli zdefiniujemy jakiś konstruktor, to konstruktor domyślny nie zostanie wygenerowany.

Konstruktor kopiujący

- ma jeden argument typu referencji do tej samej klasy, np.

  `Zespolona(Zespolona&) ;`
- jeśli go nie zdefiniujemy, kompilator go wygeneruje;
- wygenerowany konstruktor składowa po składowej, zatem zwykle nie nadaje się dla obiektów zawierających wskaźniki;
- jest wywoływay niejawnie przy przekazywaniu argumentów i wyników funkcji.

Obiekty tymczasowe

Każde użycie konstruktora powoduje powstanie nowego obiektu. Można w ten sposób tworzyć obiekty tymczasowe. Np. jeśli mamy funkcję

```
double policz(Zespolona) ;
```

możemy wywołać tak:

```
Zespolona z(3,4);
x = policz(z) ;
```

...ale jeśli samo `z` nie jest nam potrzebne, to można i tak:

```
x = policz(Zespolona(3,4)) ;
```

Utworzony obiekt tymczasowy będzie istniał tylko podczas wykonywania tej instrukcji.

Ułatwianie sobie życia

Jeśli chcemy zdefiniować liczby zespolone, które są w istocie rzeczywiste, możemy to uczynić tak:

```
Zespolona jeden(1,0);
```

Jeśli chcemy to robić często, warto dopisać konstruktor:

```
class Zespolona { ...
  Zespolona(double r) {re = r;im = 0;};
};
```

Prościej jednak użyć argumentu domyślnego:

```
Zespolona(double r, double i=0) ...
```

Konstruktor i Konwersja

Istnienie konstruktora, który można wywołać z jednym argumentem ma dalsze konsekwencje. Poniższe wywołanie jest teraz poprawne:

```
double policz(Zespolona) ;
x =policz(3) ;
```

Innymi słowy zdefiniowanie w klasie K konstruktora, który można wywołać z jednym argumentem typu T, oznacza zdefiniowanie konwersji z T do K.

Dziedziczenie

- Dziedziczenie jest jednym z najważniejszych elementów obiektowości
- Pozwala na pogodzenie dwóch sprzecznych dążeń:
  - raz napisany i przetestowany program powinien pozostać w niezmienionej postaci
  - programy wymagają stałego dostosowywania do zmieniających się wymagań użytkownika, sprzętowych itp.

Hierarchie klas

- Dziedziczenie umożliwia tworzenie hierarchii klas.
- Klasy odpowiadają pojęciom w świecie modelowanym przez program. Hierarchie klas pozwalają tworzyć hierarchie pojęć, wyrażając w ten sposón zależności między pojęciami.
- Klasa pochodna dziedziczy po klasie bazowej; tworzymy ją by opisać bardziej wyspecjalizowane obiekty klasy bazowej.
- Każdy obiekt klasy pochodnej jest obiektem klasy bazowej.

Zalety dziedziczenia

- Jawne wyrażanie zależności między klasami (pojęciami).
- Możemy np. jawnie zapisać, że każdy kwadrat jest prostokątem.
- Unikanie wielokrotnego pisania tych samych (lub podobnych) fragmentów kodu.

Przykładowa klasa bazowa

```
class A {
 private:
  int m1;
 protected:
  int m2;
public:
  int m3;
};
```

Składowe zadeklarowane w sekcji **protected** są widoczne w podklasach (bezpośrednich i dalszych), nie są natomiast widoczne z zewnątrz.

Przykładowa podklasa

```
class B : public A {
 private:
  int m4;
 protected:
  int m5 ;
 public:
  int m6;
  void f();
};
```

Dziedziczenie jest oznaczane przy pomocy dwukropka. Występujące po nim **public** oznacza, że fakt dziedziczenia jest jawny (zwykle tego chcemy).

Przykład użycia

```
void B::f() {
 m1 = 1; // Błąd, składowa prywatna A
 m2 = 2; // OK, składowa chroniona A
 m3 = 3; // OK, składowa publiczna A
 m4=m5=m6 = 4; // OK, skladowe B
}
```

```
main() {
  A a; B b; int i;
  i = a.m1; // Błąd, składowa prywatna
  i = a.m2; // Błąd, składowa chroniona
  i = a.m3; // OK
  i = a.m6; // Nie ma takiej składowej
  i = b.m2; // Błąd, składowa chroniona
  i = b.m3; // OK (odziedziczone po A)
  i = b.m4; // Błąd, składowa prywatna
  i = b.m6; // OK
}
```

Przyjaciele

```
struct X; // ``zapowiedź'' deklaracji
struct Y {
  void f(X*) ;
}
class X {
  int i ; // składowa prywatna
 public:
  friend void g(X*, int) // przyjaciel globalny
  friend void Y::f(X*) // zaprzyjaźniona metoda Y
  friend struct Z // Cała struktura jako przyjaciel
}
```

Zaprzyjaźnione funkcje uzyskują dostęp do prywatnych skladowych danej klasy.

Podsumowanie reguł dostępu

- Składowe prywatne są widoczne jedynie w swojej klasie i wśród przyjaciół
- Składowe chronione (protected) są ponadto widoczne w klasach pochodnych
- Składowe publiczne są widoczne wszędzie tam, gdzie jest widoczna sama klasa.

Zastępowanie (overriding)

W podklasach można deklarować składowe o takiej samej nazwie jak w nadklasach. W takim wypadku składowa podklasy zastępuje składową nadklasy.

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

Operator zasięgu

Co zrobić, aby w podklasie odwołać się do zasłoniętej składowej nadklasy? Należy użyć operatora zasięgu (::), np.

```
void D::f() { a = C::a; }
```

W podobny sposób można odwoływać się do zasłoniętych zmiennych globalnych:

```
int i;
void f(int i) {
  i = 3; // parametr
  ::i = 5; // zmienna globalna
}
```

```
void C::f{ a=1; // Składowa C }
void D::f{ a=2; // Składowa D }
main {
  C c;
  D d;
  c.a = 3; // Składowa C
  d.a = 4  // Składowa D
  a.f() ;  // Składowa C
  b.f() ;  // Składowa D
  A *e = &d ;
  e->f() ; // Składowa C!
}
```

```
class A {
 private:
  int m1;
 protected:
  int m2;
public:
  int m3;
};
class B : public A {
 private:
  int m4;
 protected:
  int m5 ;
 public:
  int m6;
};
```

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.