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.

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`

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.

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);
};
```

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)`

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 jedno- jak i dwuargumentową. Jest to potencjalne źródło błędów i należy uważać.

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

Jaki operator zdefiniowaliśmy?

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
  operator+(Zespolona&,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.

Operator przypisania

- Jest **czym innym** niż konstruktor kopiujący!
- Domyślnie jest zdefiniowany jako przypisanie składowa po składowej (zatem niekoniecznie bajt po bajcie).
- Dla klasy X definiuje się jako

  `X& operator=(const X&);`
- Uwaga na przypisania `x = x`. Jeśli chcemy dla danej klasy zablokować przypisania, najlepiej zdefiniować `operator=` w części prywatnej (jako pusty).

Przykład

```
struct napis {
  char *p;  int rozmiar; // rozmiar tablicy
  napis(int r){p = new char[rozmiar=r];};
  ~napis(){delete p;};
};
void f() {
 napis n1(10);
 napis n2(20);
 n1 = n2;
}
```

- Pamięć n2 nie zostanie zwolniona (wyciek pamięci).
- Pamięć n1 zostanie zwolniona dwukrotnie (groźne).
- Zmiana zawartości n1 automatycznie zmieni zawartość n2.

Rozwiązanie

Musimy zdefiniować operator przypisania

```
napis& operator=(const napis& n) {
  if(this != &n) { // uważaj na n=n
    delete p;
    p = new char[rozmiar=n.rozmiar];
    strcpy(p, n.p);
  }
  return *this;
}
```

Przypisanie vs inicjalizacja

Rozważmy drobną modyfikację funkcji `f`

```
void f() {
 napis n1(10);
 napis n2 = n1;
}
```

Wbrew pozorom “n2=n1” nie oznacza tu przypisania, lecz inicjalizację.

Tutaj wykona się konstruktor kopiujący.

Inaczej zwalnialibyśmy niezaalokowaną pamięć.

Operator indeksowania

Wyrażenie

wyrażenie1[*wyrażenie2*]

interpretuje się jako operator dwuargumentowy.

Zatem wyrażenie

`x[y]`

interpretuje się jako

`x.operator[ ](y)`

Na przykład

```
char napis::operator[](int i) {
  if(i<rozmiar) return p[i] ;
  else return 0 ;
}
```

Operator wywołania funkcji

Wyrażenie

wyrażenie(*lista wyrażeń*]

interpretuje się jako operator dwuargumentowy, gdzie *wyrażenie* jest pierwszym argumentem, a *lista wyrażeń* — drugim. Zatem wyrażenie

`x(y)`

interpretuje się jako

`x.operator( )(y)`

W praktyce używa się tylko `operator()()`, dla klas z jedną operacją.

Jeszcze o operatorze indeksowania

Argument operatora indeksowania nie musi być typu liczbowego, np.

```
int napis::operator[](const char *n) {
// Znajdź pierwsze wystąpienie n i daj jego indeks
// Jeśli n nie występuje, daj -1
}
```

Przykład

```
struct K {
  int operator()() {return 1 ;};
  int operator()(int a1) {return a1 ;};
  int operator()(int a1,int a2) {return a1+a2 ;};
} ;

main() {
  K k ;
  cout << k() << endl;
  cout << k(k()) << endl ;
  cout << k(k(),k(k())) << endl ;
}
```

Iteratory

```
class Elem {
  int glowa ;
  Elem* ogon ;
public:
  Elem(int g,Elem* o):glowa(g),ogon(o){};
  friend class Iterator ;
};
struct Iterator {
  Elem* lista;
  Iterator(Elem* el):lista(el){};
  int operator()(){if(!lista) return 0;
    int g=lista->glowa;lista=lista->ogon;return g;}
};
for(i=Iterator(el);k=i();) ...
```