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 ;
}
```

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
}
```

Operator wywołania funkcji

Wyrażenie

$$\textit{wyrażenie}(\textit{lista wyrażeń}]$$

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

$$\texttt{x}(\texttt{y})$$

interpretuje się jako

$$\texttt{x.operator}(\,)(\texttt{y})$$

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

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

Rozważmy klasę reprezentującą listy:

```
class Elem {
  int glowa ;
  Elem* ogon ;
public:
  Elem(int g,Elem* o):glowa(g),ogon(o){};
  friend class Iterator ;
};
```

Chcielibyśmy móc przechodzić po liście tak łatwo, jak po tablicy...

Iteratory

Wygodnym rozwiązaniem jest 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();) ...
```

Operatory konwersji

- Są metodami o nazwie

$$operator\ nazwa\_typu$$

- nie deklarujemy typu wyniku, bo musi być taki sam jak w nazwie operatora,
- musi instrukcją return przekazywać obiekt odpowiedniego typu

Dziedziczenie public, protected i private

- Sposób dziedziczenia określa gdzie dziedziczenie jest widoczne
- Wpływa na możliwość użycia obiektu podklasy jako nadklasy oraz
- Determinuje widoczność składowych nadklasy.
- `public` — widoczne wszędzie (zwykle tego chcemy)
- `protected` — tylko w podklasach
- `private` — nigdzie
- Dla klas domyślne jest `private`

Przykład

...ilustrujący również co się stanie jeśli zapomnimy `public`:

```
class X { public: int a; };
class Y : X   // Domyślne dziedziczenie: private
 int f() { return a;  } // BŁĄD!
;
X *px = new X;
Y *py = new Y;
cout << px->a; // OK
cout << py->a; // BŁĄD!
px = new Y;   // BŁĄD!
```

Przykład

```
class X { public: int a; };
class Z : protected X {
public:
 int f() { return a; } // OK
 X* asX { return (X*) this; }
};
X *px = new X;
Z *pz = new Z;
cout << px->a; // OK
cout << py->a; // BŁĄD!
px = pz;   // BŁĄD!
px = pz->asX(); // OK
```

Wielodziedziczenie

- W C++ klasa może mieć więcej niż jedną klasę podstawową.
- Konstrukcję taką nazywamy **wielodziedziczeniem**.
- Często jest to użyteczne, czasem prowadzi do problemów.
- Wiele innych języków (np. Java, Smalltalk) nie dopuszcza wielodziedziczenia (lub tylko jego okrojoną formę).

Klasa `Square` reprezentująca kwadrat:

- zna swoje położenie, wymiary i kolor

- umie się narysować w danym oknie

```
class Square {
public:
  Square(int x, int y, int a, Rgb col)
    : m_x(x),m_y(y),m_a(a),m_col(col) {};
  void draw(MWindow& win) ;
protected:
  int m_x, m_y, m_a;
  Rgb m_col ;
};
```

Klasa `Gadget`:

- reaguje na kliknięcie myszą
- umie (potencjalnie) się narysować
- umie zmienić swoje położenie
- może zawierać inne Gadgety

```
class Gadget {
  Gadget();
  virtual void onClick(int x, int y);
  virtual void paint(MWindow& win) ;
  virtual void move(int dx,int dy);
  Gadget& operator+=(Gadget* c);
  Gadget& operator+=(Gadget& c);
};
```

Kwadrat reagujący na kliknięcie:

```
class GSquare : public Square, public Gadget {
public:
  GSquare(int x, int y, int a, Rgb col)
    : Square(x,y,a,col),Gadget(){}
  void paint(MWindow& w) {draw(w);}
  void move(int dx,int dy){m_x+=dx;m_y+=dy;}
};
```

Klasa GSquare dziedziczy po klasach `Square` oraz `Gadget`.

```
Gadget allGadgets;
allGadgets += new GSquare(0,0,20,red);
```

Problem

```
class Scrollbar {
private:
 int x; int y;
public:
 void scroll(units n);
};
class HorizontalScrollbar : public Scrollbar {};
class VerticalScrollbar : public Scrollbar {};
class ScrollWindow : public VerticalScrollbar,
                  public HorizontalScrollbar {};

ScrollWindow w;
w.scroll(5); // BŁĄD!
```

Przykład

```
class Scrollbar {
private:
 int x; int y;
public:
 void scroll(units n);
};
class HorizontalScrollbar : public Scrollbar {};
class VerticalScrollbar : public Scrollbar {};
class ScrollWindow : public VerticalScrollbar,
                  public HorizontalScrollbar {};

ScrollWindow w;
w.HorizontalScrollbar::scroll(5); // w prawo
w.VerticalScrollbar::scroll(12); //...i w dół
```

Atrybuty i metody klasowe

Składowe klasowe (statyczne)

- Każdy obiekt klasy posiada własny zestaw atrybutów.
- Metody używają atrybutów odpowiedniego obiektu.

- Czasem potrzeba atrybutów wspólnych dla wszystkich obiektów danej klasy (i ew. odpowiednich metod).
- Do tego celu służą atrybuty i metody **statyczne** (klasowe).

Przykład

```
struct Licznik {
  static int ile; // Ile aktywnych obiektów klasy

  Licznik(){ ile++; }
  ~Licznik(){ ile--; }
};
int Licznik::ile=0;
main() {
  Licznik l ;
  Licznik* l2 = new Licznik();
  cout << Licznik::ile <<endl;
  cout << l.ile <<endl;
}
```

Zauważmy, że:

- Zmienna `ile` jest wspólna dla wszystkich obiektów klasy licznik.
- Jej deklaracja jest wewnątrz klasy, ale definicja jest globalna.
- Dostęp jest możliwy zarówno przez operator zasięgu, jak i dowolny obiekt klasy.

Metody klasowe

```
class Licznik {
  static int licz; // Ile aktywnych obiektów klasy
public:
  Licznik(){ licz++; }
  ~Licznik(){ licz--; }
  static int ile() { return licz; }
};
int Licznik::licz=0;
main(){
```

```
  Licznik l ;
  Licznik* l2 = new Licznik();

  cout << Licznik::ile() <<endl;
  cout << l.ile() <<endl;
}
```