OOP w C++; zastosowania ref

Zastosowania referencji: konstruktor kopiujący.

Ref pozwala zdefiniować tzw konstruktor kopiujący:

```
class CCC {
public:
  CCC() {
    printf("CCC, konst. bezpar., %p\n", this); pole=123;
  }
  CCC(CCC const & c) { // konst. kopiujący MUSI mieć taki parametr (ew bez const)
    printf("CCC, konst. kopiujacy., %p\n", this); pole=c.pole;
  }
  ~CCC() { printf("CCC, destr., %p\n", this); }
  void setPole(int p) {pole=p;}
  int getPole() {return pole;}
private:
  int pole;
};
// ...
CCC c1; c1.setPole(321);
CCC c2=c1; // tutaj zostanie wywołany konst kopiujący !!!
printf("%d\n", c2.getPole());
```

```
/* wynik działania powyższego kodu:
CCC, konst. bezpar., 0064FE00
CCC, konst. kopiujacy., 0064FDFC
321
CCC, destr., 0064FDFC
CCC, destr., 0064FE00
*/
```

Konstruktor kopiujący jest wywoływany:

1. przy inicjalizowaniu obiektu obiektem
2. przy przekazywaniu przez parametr do funkcji (BEZ ref!!!)

OOP w C++; zastosowania ref

Przekazywanie obiektów przez parametr do funkcji/metody:

```
// tu wstaw kod z def klasy CCC ...
void fun(CCC c1, CCC& c2r) {
  /* jakis kod */
}
// ...
fun(c1,c2);
  // jest tworzona kopia c1 (i jest wywoływany konstr. kopiujący)
  // nie jest tworzona kopia c2 (z powodu "&");
  // parametr formalny c2r jest przekazywany "przez ref" (a nie "przez wartość")
```

Wywołanie konstruktora kopiującego świadczy o tym, że kompilator tworzy tzw *obiekt tymczasowy*, który jest inicjalizowany naszym obiektem.

Kompilator sam dba o właściwy "czas życia" obiektu tymczasowego. Tworzenie obiektów tymczasowych zajmujących dużo pamięci może być czaso/pamięcio chłonne!!!

Jak uniknąć tworzenia obiektów tymczasowych? Odp: przekazuj parametry przez ref.

OOP w C++; zastosowania ref

Zwracanie obiektu przez ref lub bez:

```
// tu wstaw kod z def klasy CCC ...
CCC fun1(CCC& c) { return c; }
CCC& fun2(CCC& c) { return c; }
// ...
{
  CCC c1,c2;
  printf("-----------------\n");
  fun1(c1);
  fun2(c2);
  printf("-----------------\n");
}
/* wynik działania powyższego kodu:
CCC, konst. bezpar., 0064FE00
CCC, konst. bezpar., 0064FDFC
-----------------
CCC, konst. kopiujacy., 0064FDF8 // tylko jeden obiekt tymczasowy
CCC, destr., 0064FDF8            // został utworzony!
-----------------
CCC, destr., 0064FDFC
CCC, destr., 0064FE00
*/
```

OOP w C++; przeciążanie operatorów

Mamy klasę Wektor2 reprezentującą wektory o 2 składowych; chcemy w naturalny sposób dodawać wektory przy pomocy operatora "+":

```
  Wektor2 a(1,2), b(3,4), c; c=a+b; c.pisz(); // powinno wyswietlic "Wektor2(4,6)"
```

W tym celu należy przeciążyć operator "+" dla klasy Wektor2:

```
class Wektor2 {
public:
  Wektor2() { x1=0; x2=0; }
  Wektor2(int p1, int p2) { x1=p1; x2=p2; }
  Wektor2(Wektor2 const & w) { x1=w.x1; x2=w.x2; }
  ~Wektor2() {}
  void pisz() { printf("Wektor2(%d,%d)\n", x1, x2); }
  Wektor2 operator+(Wektor2& w) {
    return Wektor2(x1+w.x1, x2+w.x2);
      // to tzw "konstruktor typu"; tworzy obiekt tymczasowy,
      // który jest potrzebny w wyrażeniu z wektorami ...
      // NA PEWNO wszystki obiekty tymczasowe są niszczone gdy stają się niepotrzebne
  }
    // powinno się jeszcze zdef operator= !!!
private:
  int x1, x2;
};
```

```
// ...
{
  Wektor2 a(1,2), b(3,4), c;
  printf("-----------------\n");
  c=a+b;
  printf("-----------------\n");
  c.pisz();
}
/* wynik działania powyższego kodu:
Wektor2, konst. (int,int), 0064FDFC
Wektor2, konst. (int,int), 0064FDF4
Wektor2, konst. bezpar., 0064FDEC
-----------------
Wektor2, konst. (int,int), 0064FDE4
-----------------
Wektor2(4,6)
*/
```

OOP w C++; przeciążanie operatorów c.d.

Dodatkowo przeciążyliśmy operator przypisania "=" ...

```
class Wektor2 {
public:
  Wektor2() {
    printf("Wektor2, konst. bezpar., %p\n", this);
    x1=0; x2=0;
  }
  Wektor2(int p1, int p2) {
    printf("Wektor2, konst. (int,int), %p\n", this);
    x1=p1; x2=p2;
  }
  Wektor2(Wektor2 const& w) {
    printf("Wektor2, konst. kop., %p\n", this);
    x1=w.x1; x2=w.x2;
  }
  ~Wektor2() {
    printf("Wektor2, destructor, %p\n", this);
  }
```

```
  Wektor2 operator+(Wektor2& w) {
    printf("Wektor2, operator+, %p\n", this);
    return Wektor2(x1+w.x1, x2+w.x2);
  }
  Wektor2& operator=(Wektor2 w) { // "Wektor2& w" powoduje błąd kompilatora ???
    printf("Wektor2, operator=, %p\n", this);
    x1=w.x1; x2=w.x2;
    return *this;
  }
  void pisz() { printf("Wektor2(%d,%d)\n", x1, x2); }
private:
  int x1, x2;
};

int main(int argc, char* argv[])
{
  Wektor2 a(1,2), b(3,4), c;
  printf("-----------------\n");
  c=a+b;
  printf("-----------------\n");
  c.pisz();
}
```

```
/* wydruk:
Wektor2, konst. (int,int), 0064FDFC
Wektor2, konst. (int,int), 0064FDF4
Wektor2, konst. bezpar., 0064FDEC
-----------------
Wektor2, operator+, 0064FDFC
Wektor2, konst. (int,int), 0064FDC0
Wektor2, operator=, 0064FDEC
Wektor2, destructor, 0064FDC0
-----------------
Wektor2(4,6)
Wektor2, destructor, 0064FDEC
Wektor2, destructor, 0064FDF4
Wektor2, destructor, 0064FDFC
*/
```

Uwaga: jest zasadnicza różnica między inicjalizowaniem a przypisywaniem przy pomocy operatora przypisania "=":

```
Wektor w1(10,20), w2= w1, w3; w3= w1;
```

```
int main() {
  Wektor2 a(1,2), b(3,4), c, d;
  printf("-----------------\n");
  d=c=a+b;
  printf("-----------------\n");
  c.pisz();
}
/* wydruk:
Wektor2, konst. (int,int), 0064FDFC
Wektor2, konst. (int,int), 0064FDF4
Wektor2, konst. bezpar., 0064FDEC
Wektor2, konst. bezpar., 0064FDE4
-----------------
Wektor2, operator+, 0064FDFC
Wektor2, konst. (int,int), 0064FDB0
Wektor2, operator=, 0064FDEC
Wektor2, destructor, 0064FDB0
Wektor2, konst. kop., 0064FDB8
Wektor2, operator=, 0064FDE4
Wektor2, destructor, 0064FDB8
-----------------
Wektor2(4,6)
Wektor2, destructor, 0064FDE4
Wektor2, destructor, 0064FDEC
Wektor2, destructor, 0064FDF4
```

```
Wektor2, destructor, 0064FDFC
*/
```