Konstruktor kopiujący

- Konstruktor kopiujący jest wywoływany wtedy, kiedy nowo tworzony obiekt jest inicjowany już istniejącym obiektem tej samej klasy:
  - inicjujemy nowy obiekt istniejącym obiektem

```
Punkt a(1,1), b(a); // definicja obiektu b za pomocą a
Punkt c=a(1,1);
Punkt d=Punkt(a);
Punkt *wPunkt = new Punkt(a);
```

  - przekazujemy obiekt do funkcji przez wartość – nowy obiekt jest kopią

```
funkcja(a);         // przekazanie obiektu a do funkcji
```

  - zwracamy obiekt z funkcji przez wartość

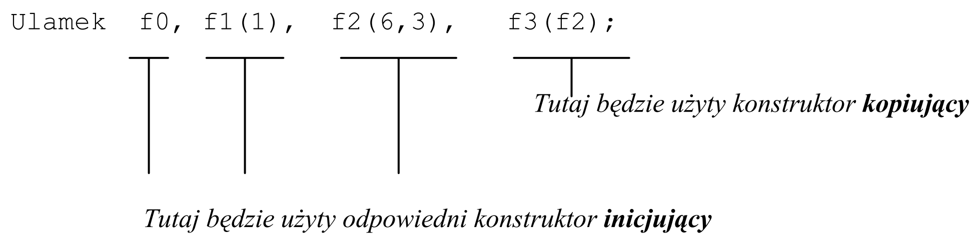
```
c=funkcja();        // zwrócenie obiektu z funkcji
```

- Jeśli użytkownik takiego konstruktora nie dostarczy, to stosowany jest domyślny konstruktor kopiujący, wykonujący dokładną, bit po bicie kopię obiektu.
- Przykład:

```
class Ulamek
{ int l;                  // licznik
  int m;                  // mianownik
  ...
public:
  Ulamek(int a=0, int b=1)     // Konstruktor inicjujący
  {
    ...
  }


};

Ulamek  f0, f1(1),  f2(6,3),   f3(f2);
```



W klasie `Ulamek` *brak* jest konstruktora kopiującego. Kompilator utworzy *własny* konstruktor kopiujący.

- Pytanie: Czy domyślny konstruktor kopiujący zawsze zrobi to co trzeba?

Kiedy potrzebny jest własny konstruktor kopiowania?

- Przykład: tablica dynamiczna, zdefiniowany jest tylko konstruktor inicjujący.

```
#include <iostream>
using namespace std;

class Tablica {
  int lElem;        // liczba elementów tablicy
  double *T ;       // wskaźnik do obszaru przydzielonego dla tablicy
public :
  Tablica (int n)
  { T = new double [lElem = n] ;
    cout << "++ Konstruktor - adres obiektu: " << this
         << " - adres tablicy: " << T << "\n" ;
  }
  ~Tablica ()
  { cout << "-- Destruktor - adres obiektu: "
         << this << " - adres tablicy: " << T << "\n" ;
    delete [] T ;
  }
  friend void func(Tablica b);
} ;
void func (Tablica b)
{
  cout << "*** func: Działam na kopii - adres obiektu: "
  << &b <<" - adres tablicy: " << b.T <<endl;
}

int main()
{ Tablica a(4) ;
  cout << "** Zaczynam main **" << endl;
  cout << "** Wywołuję func **\n" ;
  func (a) ;
  cout << "** Kończę main **" << endl;
  return 0;
}
```

- Jak wygląda przebieg programu?

```
++ Konstruktor - adres obiektu: 0xbffffc70 - adres tablicy: 0x8049f50
** Zaczynam main **
** Wywołuję func **
*** func: Działam na kopii - adres obiektu: 0xbffffc60
                           - adres tablicy: 0x8049f50
-- Destruktor - adres obiektu: 0xbffffc60 - adres tablicy: 0x8049f50
** Kończę main **
-- Destruktor - adres obiektu: 0xbffffc70 - adres tablicy: 0x8049f50
```

- Gdzie jest pułapka?

Jak działa konstruktor kopiujący?

- Domyślny konstruktor kopiujący przepisuje kolejno wartość każdej składowej do jej odpowiednika w nowo tworzonym obiekcie. Jest to tzw. kopiowanie płytkie (ang. *memberwise copying*, *shallow copying*).
- W przypadku obiektów o składowych dynamicznych potrzebne jest kopiowanie głębokie zapewniane przez własny konstruktor kopiujący.
- Konstruktor taki ma prototyp:
  `MojaKlasa(const MojaKlasa &);`
- Przykład:

```
#include <iostream>

class Tablica
{
   int lElem ;
   double * T ;
 public :
   Tablica (int n)
   { T = new double [lElem = n] ;
   }
   ~Tablica ()
   { delete [] T ;
   }
   ///////////////////////////////////////////////////////////////////
   Tablica (const Tablica & v)            // konstruktor kopiujący
   { T = new double [lElem = v.lElem] ; // przydzielenie pamięci
     int i ;
     for (i=0 ; i<lElem ; i++)            // skopiowanie do nowej pamięci
       T[i]=v.T[i] ;
   }
   ///////////////////////////////////////////////////////////////////
} ;
```

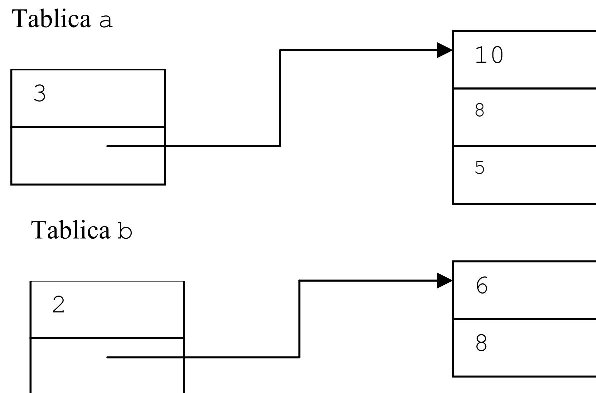
Operator przypisania =

- *Inicjalizacja:* tworzenie nowego obiektu.
- *Przypisanie:* zmiana wartości istniejącego obiektu.

```
Punkt a(1,1), b;  // inicjalizacja
b=a;              // przypisanie
```

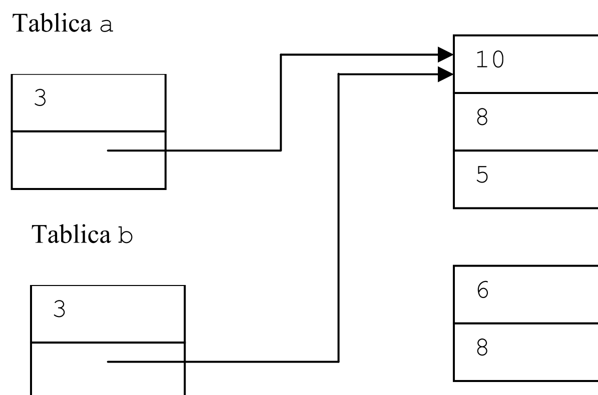
- Występuje *podobny* problem jak w przypadku kopiowania. W przypadku użycia domyślnego operatora przypisania wykonywane jest kopiowanie płytkie.

```
class Tablica
{
   int lElem ;
   int *T ;
 public :
   Tablica (int n)
   ...
};
int main() {
  Tablica a(3), b(2);
```



- Wykonujemy w programie instrukcję przypisania:

```
b=a
```



Wskazuje na ten sam obszar pamięci

Pozostaje zajęta pamięć ze sterty, nie ma do niej dostępu

- Rozwiązanie: przeciążenie operatora przypisania =

- Prototyp operatora przypisania =:

```
MojaKlasa& operator=(const MojaKlasa &);
```

- Przykład:

```
#include <iostream>
using namespace std;
class Tablica {
  int lElem ;
  int *T ;
public :
  Tablica (int n)
  {
    T = new int [lElem = n] ;
    for (int i=0 ; i<lElem ; i++)
      T[i] = 0 ;
  }
  ~Tablica ()
  {
    delete T ;
  }
// przeciążony operator przypisania
    Tablica& operator= (const Tablica &) ;
};
///////////////////////////////////////////////////
// przeciążony operator przypisania
Tablica & Tablica::operator = (const Tablica & v)
{
  if (this != &v)     // czy przypisujemy obiekt do siebie samego
  {
// 1. zwolnienie pamięci obiektu po lewej stronie =
    delete [ ] T ;
// 2. przydzielenie na nowo pamięci
    T = new int [lElem = v.lElem] ;
// 3. przepisanie do nowej pamięci obiektu po prawej stronie =
    for (int i=0 ; i<lElem ; i++)
      T[i] = v.T[i] ;
  }
  else cout << "  nic nie robimy \n" ;
  return * this ;
}
////////////////////////////////////////////////////
main() {
  Tablica a(5), b(3), c(4) ;
  cout << "** przypisanie a=b \n" ;
  a = b ; // czyli a.operator=(b)
  cout << "** przypisanie c=c \n" ;
  c = c ; // czyli c.operator=(c)
  cout << "** przypisanie a=b=c \n" ;
  a = b = c ; // czyli a.operator=(b.operator=(c))
}
```

Przykład

- Autor- S.Prata, Szkoła programowania, język C++, rozdział 14

```
class Student
{
  String name;      // obiekt klasy String
  ArrayDb scores;   // obiekt klasy ArrayDb
  ...
};
```

Klasa `String`

- Cechy klasy:
  - powinna umożliwiać przechowywanie napisów zmiennej długości.
  - powinna umożliwiać dostęp do pojedynczych znaków napisu
  - powinna umożliwiać przypisywanie jednego napisu drugiemu
  - powinna umożliwiać porównywanie napisów

```
// Plik string1.h -- definicja klasy
#include <iostream>
using namespace std;
#ifndef STRING1_H_
#define STRING1_H_
class String {
private:
  char * str;             // wskaźnik do napisu - alokacja pamięci
                          // zostanie wykonana w konstruktorze
  int len;                // długość napisu
  static int num_strings; // ilość obiektów - wprowadzona, aby
                          // pokazac korzystanie ze składowej statycznej
  static const int CINLIM = 80; // ograniczenie długości danych wejściowych
public:
// konstruktory
  String(const char * s); // constructor
  String();               // default constructor
  String(const String &); // copy constructor
  ~String();              // destructor
// przeciążone operatory - metody
  String & operator=(const String &);
  String & operator=(const char *);
  char & operator[](int i);
  const char & operator[](int i) const;
// przeciążone operatory - funkcje zaprzyjaźnione
  friend bool operator<(const String &st, const String &st2);
  friend bool operator>(const String &st1, const String &st2);
  friend bool operator==(const String &st, const String &st2);
  friend ostream & operator<<(ostream & os, const String & st);
  friend istream & operator>>(istream & is, String & st);
// funkcje pomocnicze
  static int HowMany();
  int length () const { return len; }
};
#endif
```

```
// Plik string1.cpp -- metody klasy String
#include <iostream>
#include <cstring>
#include "string1.h"
using namespace std;

// inicjalizacja składowych statycznych

int String::num_strings = 0; // musi być inicjalizowana na zewnatrz

// metody statyczne

int String::HowMany()
{
  return num_strings;
}

// konstruktory i destruktor

// tworzenie obiektu String na podstawie napisu w stylu języka C
String::String(const char * s)
{
  len = strlen(s);
  str = new char[len + 1];
  strcpy(str, s);
  num_strings++;             // śledzenie tworzonych obiektów
}

// konstruktor domyślny
String::String()
{
  len = 0;
  str = new char[1];
  str[0] = '\0';
  num_strings++;            // śledzenie tworzonych obiektów
}

// konstruktor kopiujący
String::String(const String & st)
{
    num_strings++;
    len = st.len;
    str = new char [len + 1];
    strcpy(str, st.str);    // śledzenie tworzonych obiektów
}

// destruktor
String::~String()
{
  --num_strings;           // śledzenie tworzonych obiektów
  delete [] str;
}
```

```
// przeciążone operatory -- metody

// operator przypisania -- obiekt String do obiektu String
String & String::operator=(const String & st)
{
  if (this == &st)
    return *this;
  delete [] str;
  len = st.len;
  str = new char[len + 1];
  strcpy(str, st.str);
  return *this;
}

// operator przypisania -- napis w stylu języka C do obiektu String
String & String::operator=(const char * s)
{
  delete [] str;
  len = strlen(s);
  str = new char[len + 1];
  strcpy(str, s);
  return *this;
}

// operator [] dla obiektu nie-stałego String
char & String::operator[](int i)
{
  return str[i];
}

// operator [] dla obiektu stałego String
const char & String::operator[](int i) const
{
  return str[i];
}
```

```
// przeciążone operatory -- funkcje zaprzyjaźnione

bool operator<(const String &st1, const String &st2)
{
  return (strcmp(st1.str, st2.str) < 0);
}

bool operator>(const String &st1, const String &st2)
{
  return st2.str < st1.str;
}

bool operator==(const String &st1, const String &st2)
{
  return (strcmp(st1.str, st2.str) == 0);
}

ostream & operator<<(ostream & os, const String & st)
{
  os << st.str;
  return os;
}

istream & operator>>(istream & is, String & st)
{
  char temp[String::CINLIM];
  is.get(temp, String::CINLIM);
  if (is)
    st = temp;
  while (is && is.get() != '\n')
    continue;
  return is;
}
```

Komentarze

- Konstruktory i destruktory

```
// tworzenie obiektu String na podstawie napisu w stylu języka C
String::String(const char * s)
{
  len = strlen(s);
  str = new char[len + 1];
  strcpy(str, s);
  num_strings++;              // śledzenie tworzonych obiektów
}

// konstruktor domyślny
String::String()
{
  len = 0;
  str = new char[1];
  str[0] = '\0';
  num_strings++;             // śledzenie tworzonych obiektów
}
```

lub

```
String::String()
{
  len = 0;
  str=0;
  num_strings++;             // śledzenie tworzonych obiektów
}

// destruktor
String::~String()
{
  --num_strings;            // śledzenie tworzonych obiektów
  delete [] str;
}
```

Wszystkie konstruktory muszą być zgodne z destruktorem. Ponieważ w destruktorze odwołujemy się do tablicy: `delete [] str`, musimy w obydwu konstruktorach używać `new []`. Użycie operatora `delete []` w stosunku do obiektu, który nie został utworzony jako tablica obiektów jest nie zdefiniowane. Dopuszczalne jest jednak użycie wskaźnika pustego.

- Składowe statyczne
  - Inicjalizacja składowej statycznej musi być dokonana na zewnątrz klasy.
  - Instrukcję inicjalizacji należy umieścić w pliku z kodem metod.
  - Pole statyczne może być inicjalizowane wewnątrz klasy, jeśli jest stałą (`const`) lub typu wyliczeniowego.

- Metody statyczne
  - Metoda statyczna nie jest związana z żadnym obiektem
  - Jedyne pola, z których korzysta to pola statyczne
  - Przykład użycia funkcji statycznej:
    `cout << String::HowMany()<< endl;`

- Przeciążony operator przypisania
  - W klasie mamy dwa przeciążone operatory przypisania:
    ```
    String & operator=(const String &);
    String & operator=(const char *);
    ```

Pierwszy z operatorów jest wymagany, ponieważ klasa zawiera składową dynamiczną.
Drugi z operatorów został wprowadzony po to, aby można było działać bezpośrednio na napisach w stylu języka C, bez konieczności tworzenia obiektów tymczasowych.

Przykład 1: Brak przeciążonego operatora = dla przypisywania napisu.

```
String nazwisko;
nazwisko="Nowak";
```

Napis w stylu języka C musi być zamieniony na obiekt `String`. Wykorzystany zostanie konstruktor jednoargumentowy. Powstanie obiekt tymczasowy. Po przypisaniu zostanie usunięty.

Przykład 2: Zdefiniowano przeciążony operatora `String & operator=(const char *)`:

```
// operator przypisania -- napis w stylu języka C do obiektu String
String & String::operator=(const char * s)
{
  delete [] str;
  len = strlen(s);
  str = new char[len + 1];
  strcpy(str, s);
  return *this;
}

String nazwisko;
nazwisko="Nowak";
```

Użyty zostanie bezpośrednio operator przypisania.

Zasada: Konwersje zdefiniowane przez użytkownika (za pomocą konstruktorów lub odpowiednich funkcji konwersji) są brane pod uwagę tylko wtedy, kiedy są niezbędne.

- Przeciążone operatory porównywania
  - Dzięki temu, że są zdefiniowane jako funkcje zaprzyjaźnione, możliwe jest porównywanie obiektów `String` z napisami w stylu języka C

- Przeciążony operator `[]`
  - Musi umożliwiać korzystanie zarówno z obiektów nie-stałych jak i stałych.
  - W przypadku obiektów nie-stałych: może być użyty zarówno do wyprowadzania jak i przypisywania wartości.

```
// operator [] dla obiektu nie-stałego String
// możliwy odczyt -- zapis
char & String::operator[](int i)
{
  return str[i];
}

// operator [] dla obiektu stałego String
// możliwy tylko odczyt
const char & String::operator[](int i) const
{
  return str[i];
}

String nazwisko("nowak");  // obiekt nie jest stały
const String odp("Tak");   // obiekt stały

// obiekt nie jest stały
cout << nazwisko[0];
nazwisko[0]='N';
cin >> nazwisko[0];

// obiekt stały
cout << odp[0];
```

Klasa `ArrayDb`

- Cechy klasy:
  - przechowywanie elementów typu `double`
  - dostęp do elementów za pomocą indeksu
  - możliwość przypisania jednej tablicy drugiej
  - sprawdzanie czy podany indeks jest z zakresu

```
// Plik arraydb.h -- definicja klasy ArrayDb
#ifndef ARRAYDB_H_
#define ARRAYDB_H_
#include <iostream>
using namespace std;

class ArrayDb
{
private:
  unsigned int size;       // ilość elementów tablicy
  double * arr;            // adres pierwszego elementu
public:
// konstruktory i destruktor
  ArrayDb();
  explicit ArrayDb(unsigned int n, double val = 0.0);
  ArrayDb(const double * pn, unsigned int n);
  ArrayDb(const ArrayDb & a);
  virtual ~ArrayDb();

// metody
  unsigned int ArSize() const {return size;} // rozmiar tablicy
    double Average() const;                  // średnia wartość

// przeciążone operatory -- metody
    double & operator[](int i);
    const double & operator[](int i) const;
    ArrayDb & operator=(const ArrayDb & a);

// przeciążone operatory -- funkcje zaprzyjaźnione
    friend ostream & operator<<(ostream & os, const ArrayDb & a);
};

#endif
```

```
// arraydb.cpp -- metody klasy ArrayDb
#include <iostream>
using namespace std;
#include <cstdlib>   // exit() prototype
#include "arraydb.h"

// konstruktory i destruktor

// konstruktor domyślny
ArrayDb::ArrayDb()
{
  arr = 0;
  size = 0;
}

// konstruktor tworzący tablicę n-elementową,
// o wartościach elementów równych val
ArrayDb::ArrayDb(unsigned int n, double val)
{
  arr = new double[n];
  size = n;
  for (int i = 0; i < size; i++)
    arr[i] = val;
}

// konstruktor tworzący tablicę n-elementową,
// na podstawie tablicy pn
ArrayDb::ArrayDb(const double *pn, unsigned int n)
{
  arr = new double[n];
  size = n;
  for (int i = 0; i < size; i++)
    arr[i] = pn[i];
}

// konstruktor kopiujący
ArrayDb::ArrayDb(const ArrayDb & a)
{
  size = a.size;
  arr = new double[size];
  for (int i = 0; i < size; i++)
    arr[i] = a.arr[i];
}

// destruktor
ArrayDb::~ArrayDb()
{ delete [] arr; }
```

```
// przeciążanie operatorów -- metody

// operator [] -- nie-stały obiekt
double & ArrayDb::operator[](int i)
{
  if (i < 0 || i >= size)
  {
     cerr << "Error in array limits: "  << i << " is a bad index\n";
     exit(1);
  }
  return arr[i];
}

// operator [] -- stały obiekt
const double & ArrayDb::operator[](int i) const
{
  if (i < 0 || i >= size)
  {
    cerr << "Error in array limits: "  << i << " is a bad index\n";
    exit(1);
  }
  return arr[i];
}

// operator przypisania
ArrayDb & ArrayDb::operator=(const ArrayDb & a)
{
  if (this == &a)      // if object assigned to self,
     return *this;   // don't change anything
  delete [] arr;
  size = a.size;
  arr = new double[size];
  for (int i = 0; i < size; i++)
    arr[i] = a.arr[i];
  return *this;
}

// przeciążanie operatorów -- funkcje zaprzyjaźnione

// operator wyjścia: drukuj po pięć w wierszu
ostream & operator<<(ostream & os, const ArrayDb & a)
{
  int i;
  for (i = 0; i < a.size; i++)
  {
    os << a.arr[i] << " ";
    if (i % 5 == 4)
      os << "\n";
  }
  if (i % 5 != 0)
     os << "\n";
  return os;
}
```

```
// metody

// średnia wartość elementów tablicy
double ArrayDb::Average() const
{
  double sum = 0;
  int i;
  int lim = ArSize();
  for (i = 0; i < lim; i++)
    sum += arr[i];
  if (i > 0)
    return sum / i;
  else
  {
    cerr << "No entries in score array\n";
    return 0;
  }
}
```

Komentarze

- Konstruktory i destruktor
  - Konstruktor `explicit ArrayDb(unsigned int n, double val = 0.0)` może być wywołany z jednym argumentem, pełni więc dodatkowo rolę niejawnej funkcji konwersji. Za pomocą słowa `explicit` rezygnujemy z tej funkcji.

Klasa `Student`

- Cechy klasy:
  - student jest opisywany za pomocą nazwiska i zestawu ocen

```
// Plik studentc.h -- definicja klasy Student
#ifndef STUDNTC_H_
#define STUDENTC_H_

#include <iostream>
using namespace std;
#include "arraydb.h"
#include "string1.h"

class Student
{
private:
  String name;       // nazwisko
  ArrayDb scores;    // oceny
public:
// konstruktory i destruktor
  Student() : name("Null Student"), scores() {}
  Student(const String & s) : name(s), scores() {}
  Student(int n) : name("Nully"), scores(n) {}
  Student(const String & s, int n) : name(s), scores(n) {}
  Student(const String & s, const ArrayDb & a) : name(s), scores(a) {}
  Student(const char * str, const double * pd, int n)
                                        : name(str), scores(pd, n) {}
  ~Student() {}

// metody
  double Average() const;

// przeciążone operatory -- metody
  double & operator[](int i);
  const double & operator[](int i) const;

// przeciążone operatory -- funkcje zaprzyjaźnione
  friend ostream & operator<<(ostream & os, const Student & stu);
  friend istream & operator>>(istream & is, Student & stu);
};

#endif
```

```
// studentc.cpp -- metody klasy Student
#include "studentc.h"

// metody
double Student::Average() const
{
  return scores.Average();  // ArrayDb::Average()
}

// przeciążone operatory -- metody
// operator [] -- nie stałe obiekty
double & Student::operator[](int i)
{
  return scores[i];         // ArrayDb::operator[]()
}

// operator [] -- stałe obiekty
const double & Student::operator[](int i) const
{
  return scores[i];
}

// przeciążone operatory -- funkcje zaprzyjaźnione
// operator wyjścia
ostream & operator<<(ostream & os, const Student & stu)
{
  os << "Scores for " << stu.name << ":\n";
  os << stu.scores;
  return os;
}

// operator wejścia
istream & operator>>(istream & is, Student & stu)
{
  is >> stu.name;
  return is;
}
```

Komentarze

- Konstruktory i destruktory
    - Przy utworzeniu obiektu zewnętrznego muszą być utworzone wszystkie obiekty wewnętrzne. W tym celu wykorzystywana jest lista inicjatorów konstruktora: podawane są nazwy obiektów i wartości początkowe. W przypadku typów wbudowanych, można je traktować tak, jakby posiadały pojedynczy konstruktor posiadający jeden argument.

```
class Student
{
private:
  String name;       // nazwisko
  ArrayDb scores;    // oceny
public:
// konstruktory i destruktor
  Student() : name("Null Student"), scores() {}
  Student(const String & s) : name(s), scores() {}
  Student(int n) : name("Nully"), scores(n) {}
  Student(const String & s, int n) : name(s), scores(n) {}
  Student(const String & s, const ArrayDb & a) : name(s), scores(a) {}
  Student(const char * str, const double * pd, int n)
                                          : name(str), scores(pd, n) {}
  ...
};
```