Wydział Elektroniki
i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Programowanie obiektowe
Klasy autonomiczne (cd)

Krzysztof Gracki
kgr@ii.pw.edu.pl

pok. 312

Politechnika
Warszawska

Akcesoria specjalne klasy

Kompilator jest zobowiązany (pod pewnymi warunkami) do wygenerowania funkcji specjalnych odpowiedzialnych za cykl życia obiektów:

- tworzenie (domyślne)
- klonowanie / migrację
- usuwanie

Dla klasy `K` są to

`K();`	**Konstruktor domyślny**
`K(const K& ck);`	**Konstruktor kopiujący**
`K& operator=(const K& ck);`	**Kopiujący op. przypisania**
`K(K&& rk);`	**Konstruktor przesuwający**
`K& operator=(K&& rk);`	**Przesuwający op. przypisania**
`~K();`	**Destruktor**

Ogólna zasada (nie) generacji: jeżeli programista definiuje sam funkcję odpowiedzialną za pewien aspekt cyklu życia obiektu, to kompilator jest zwolniony z obowiązku generacji dla tego aspektu.

Zalecana reguła: jeśli trzeba zdefiniować jedno z tych akcesoriów, to zapewne trzeba zdefiniować albo jawnie wykluczyć pozostałe (Rule of Three/Five)

Rule of Three/Five


```
struct PersonCpyMove
{ // obiekt kopiowalny i przenoszony (copy and move semantics)
  int userId;
  std::string name;
};
struct PersonCpy
{ // objekt kopiowalny (copy semantic - move disabled)
int userId;
  std::string name;
  ~PersonCpy()  {
    //...
  };
};
```

Należy zdefiniować pozostałe funkcje

Klasy ze "zmienną" strukturą

Zawierają wskazanie(a) na inne obiekty albo zasoby
Zasadnicze pytanie: kto jest właścicielem tych obiektów/zasobów?
Takie klasy powinny zwykle mieć definicje:

- **Konstruktora(ów) zwykłych**
- **Konstruktora kopiującego**
- **Kopiującego operatora przypisania**
- **Konstruktora przesuwającego (opt) [C++11]**
- **Przesuwającego operatora przypisania (opt) [C++11]**
- **Destruktora**

```
class A {
  B *bptr;
  C *cptr;   // C może mieć także wskazania!
public:      // ...
          // Jaki jest cykl życia obiektów wskazanych
          // i ich semantyka widziana z poziomu klasy A?
};
```

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Mechanizmy pomocnicze

Operatory `new` i `delete`

Politechnika Warszawska

Operatory `new` i `delete`

Kategorie trwałości obiektów

- statyczne (czas życia = czas aktywności programu)
- dynamiczne kontrolowane (tworzone i usuwane z inicjatywy programu w wydzielonej pamięci)
- automatyczne (tworzone / usuwane w związku z aktywacją / deaktywacją bloku) Operatory new i delete
- tymczasowe (tworzone / usuwane z inicjatywy kompilatora)

Obiekty dynamiczne zawsze wymagają pośrednictwa wskazań lub referencji

Tylko obiekty statyczne i automatyczne mają bezpośrednią identyfikację

Politechnika Warszawska

Operatory new i delete (cd1)

C


```
void *malloc(size_t s);
void *calloc(size_t n, size_t s);
void *realloc(void *ptr, size_t s);
void free(void *ptr);
```

Czas życia obiektów dynamicznych: od alokacji, np. przez **malloc()** do zwolnienia pamięci przez **free()**:

```
Typ *p = malloc(sizeof(Typ));
if(p == NULL)
  { /* niepowodzenie */ }
else
  { /* Pamięć przydzielona poprawnie.
       Inicjuj i używaj *p ....... */
    free(p);
  }
```


Operatory new i delete (cd1)

C++


```
p = new Typ(argumenty_inicjalizacji);
// Niepowodzenie => wyjątek bad_alloc.

// Można używać obiektu *p, zainicjowanego
// .....
delete p;              // Zwolnienie pamięci
```

Typ obiektu: dowolny bez specyfikacji const, volatile;

Operator delete może zmienić zawartość obiektu wskazywanego przez argument

```
int* n = new int;      // *n==???
int& m = *new int(1); // m==1
std::cout << *n <<" "<< m;// -842150451 (==CDCDCDCD) 1
delete n;              // *n jest nieokreślone
delete& m;             // m  jest nieokreślone
std::cout << *n << ' ' << m;
                // -572662307 -572662307 (==DDDDDDDD)
```

Allokacja z użyciem szablonów

C++11

unique_ptr<>
shared_ptr<>
make_unique<>
make_shared<>

```
void zf_iptr() { //zła!
 int* intPtr = new int(1);
 int x = std::rand() % 10;
 // ...
 if (x == 5) return;
                    // wycieki pamięci !,
 // intPtr należy usunąć
 delete intPtr;
}}
```

```
void f_muptr() {
// "brak" wywołania operatora new()
  std::unique_ptr<int> intPtr = std::make_unique<int>(15);
  int  x = std::rand() % 10;
  // ...
  if (x == 5) return;
  // ...
}
```

Politechnika Warszawska

Przenoszenie własności std::move()


```
class Osoba {
  std::string nazwisko;
  std::string adres;
public:
  explicit Osoba(std::string n, std::string a)
    : nazwisko(n), adres(a) {}
};
```

patrz. dalej

kolega

znajomy

Osoba
Matysiak, Warszawa

```
void f_osoba()
{
auto kolega = std::make_unique<Osoba>("Matysiak",
"Warszawa");
auto znajomy = kolega; // nie można kopiować
                       // obiektu unique_ptr !
auto znajomy = std::move(kolega);
}
```

to jest rzutowania na rvalue reference

Politechnika Warszawska

Operatory new i delete (cd..)

Przy pomocy new można tworzyć tablice dynamiczne; zwracane jest wskazanie na zerowy element.

```
Typ *tab = new Typ[rozmiar_tablicy];
```

- Dla tablicy wielowymiarowej wszystkie wymiary z wyjątkiem pierwszego muszą być wyrażeniami stałymi
- Pierwszy wymiar może być określony dowolnym wyrażeniem

```
char (*strona)[80] = new char[n][80];
// new zwraca wskazanie char(*)[80]

// Przetwarzanie tablicy: ....
strona[i][j] ....
delete [] strona;
```

new[]
delete[]

Operatory new i delete (cd..)

- Tablic dynamicznych nie można inicjować z wyjątkiem przypadku, gdy elementy tablicy są obiektami klasy z konstruktorem domyślnym; wówczas wszystkie elementy zostaną zainicjowane tym konstruktorem w kolejności rosnących indeksów (==>).

- Przy zwalnianiu tablicy dynamicznej: '`delete [] p;`' wewnątrz nawiasów [] nie podaje się rozmiaru tablicy; destruktor, jeśli jest działa dla malejących indeksów (<==)

Operatory new i delete (cd..)

- Dla typów wbudowanych działa inicjowanie zerujące:

```
int* A = new int[3](); // wszystkie elementy zerowane
int* B = new int[7];   // elementy nieokreślone
cout<<A[0]<<' '<<B[0]<<endl; // 0 -842150451
```

- Argumentem operatora delete musi być wskazanie zwrócone przez operator **new** albo 0;
  argument równy 0 (NULL) nie powoduje żadnego skutku.

C++11 pozwala jednorodnie inicjować obiekty (uniform initialization), np.

```
int* p = new int[6] {3,1,4};
vector<int> vi {1,0,1,0,2};
```


Operatory new i delete (cd...)

- Operatory **new, new[], delete i delete[]** można przeciążać dostosowując ich funkcje do potrzeb konkretnej klasy.
- Wyrażenie **new Klasa** albo **new Klasa [n]** odwołuje się do operatora przeciążonego w klasie Klasa, jeśli takie przeciążenie zostało zdefiniowane;
  w przeciwnym przypadku oznacza wywołanie globalnego operatora `::new`.

Nie można mieszać alokacji! `new` – `delete[]`; `malloc()` – `delete`; `new` - `free()`.

Niepowodzenia alokacji

Niepowodzenie alokacji => wyjątek `bad_alloc`.

Można ustanowić funkcję reagowania na niepowodzenia:

1. Dołączyć plik nagłówkowy <new>. W pliku tym znajduje się deklaracja typu wskaźnikowego i funkcji usługowej:

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler);
```

2. Zdefiniować własną funkcję reagowania na niepowodzenia alokacji:

```
void MojaFO()
{ /* ...(objaśnienia dalej)*/ }
```

3. "Zainstalować" tę funkcję przy pomocy wywołania:

```
new_handler old = set_new_handler(MojaFO);
```


Niepowodzenia alokacji (cd.)

4. Od tego momentu obowiązuje nowy sposób reagowania; poprzednio obowiązujące konwencje można przywrócić wywołując w odpowiednim miejscu:

```
set_new_handler(old);
```

5. Funkcja **`MojaFO()`** może:

- Uwolnić obszar pamięci i powrócić – spowoduje to ponowienie poprzednio nieskutecznej alokacji.
- Jeżeli zwolnienie pamięci nie jest możliwe - wygenerować wyjątek **`bad_alloc`** uruchamiając obsługę sytuacji wyjątkowych.
- Jeżeli żaden z powyższych scenariuszy nie stosuje się, to wywołać **`abort()`** albo **`exit()`**.
- Można wymusić zachowanie operatora **`new`** zgodne ze stylem **`malloc()`** wywołując **`set_new_handler(0)`**;

Przykład: klasa `String`


```
class String
{ char *p;        // Treść łańcucha w buforze p[]
  unsigned size;  // Liczba znaków w łańcuchu
  void allocate(uint siz) { p=new char[size=siz]; }
public:
  String(const char *s="")  // Dwie wersje konstruktora
  { allocate(::strlen(s) + 1);
    ::strcpy(p,s);
  }
  String(const String &Str) // Konstruktor kopiujący
  { allocate(Str.size);
    ::strcpy(p,Str.p);
  }
  String & operator=(const String &s); // Op. przypisania
 ~String(){ delete [] p; }
  String operator+ (const String &s) const;
  int    operator==(const String &s) const;
  // ... inne operatory i funkcje
};
```


Przykład: klasa `String` (cd1)

Operator przypisania

referencje

operator pobrania adresu

```
String& String::operator=(const String &s)
{ if(this != &s) // s=s?
  { if(size != s.size)
    { delete [] p;
      allocate(s.size);
    }
    strcpy(p, s.p);
  }
  return *this;
}
```

Przykład: klasa `String` (cd2)

Operator konkatenacji

```
// Wersja 1 (błędna)
String String::operator+ (const String &s) const
// Argumenty bez zmian
{
  char *temp = new[size + s.size-1];
  ::strcpy(temp, p);
  ::strcat(temp, s.p);
  return String(temp);
}
// Bufor temp[] nie jest zwalniany!
```


Przykład: klasa `String` (cd3)

Operator konkatenacji

```
// Wersja 2 (bardzo nieefektywna)
String String::operator+ (const String &s) const
{
  char *temp = new[size + s.size-1];
  ::strcpy(temp, p);
  ::strcat(temp, s.p);
  String t(temp);
  delete [] temp;
  return t;
}
```

Przykład: klasa `String` (cd4)

Operator konkatenacji

```
// Wersja 3 (z prywatnym konstruktorem).
// Postulujemy istnienie konstruktora String(uint n)
// tworzącego obiekt z buforem o rozmiarze n,
// ale bez wypełnienia tekstem.
String String::operator+(const String &s)const
{
  String temp(size + s.size-1);//String(uint);
  strcpy(temp.p, p);
  strcat(temp.p, s.p);
  return temp;
}
```


Przykład: klasa `String` (cd5)

Koszt operacji na łańcuchach: semantyka kopiowania

```
int main()
{ String a("ABRA"), b("KAD"); b = a + b + a; }
```

W trybie **DEBUG** dodatkowe allokacje

Akcja	Total	Treść	Gdzie
Alokuj	5 B: 5 B	"ABRA“	main(), konstruktor a
Alokuj	4 B: 9 B	"KAD“	main(), konstruktor b
Alokuj	8 B: 17 B	""	operator+(), zmienna temp
Alokuj	8 B: 25 B	"ABRAKAD“	operator+(), return temp;
Zwolnij	8 B: 17 B	"ABRAKAD"	operator+(), usuń temp
Alokuj	12 B: 29 B	""	operator+(), zmienna temp
Alokuj	12 B: 41 B	"ABRAKADABRA"	operator+(), return temp;
Zwolnij	12 B: 29 B	"ABRAKADABRA"	operator+(), usuń temp
Zwolnij	4 B: 25 B	"KAD"	operator=(), zwolnij bufor b
Alokuj	12 B: 37 B	"ABRAKADABRA"	operator=(), nowy bufor b
Zwolnij	12 B: 25 B	"ABRAKADABRA"	zwolnij bufor wyniku a+b+a
Zwolnij	8 B: 17 B	"ABRAKAD"	zwolnij bufor wyniku a+b
Zwolnij	12 B: 5 B	"ABRAKADABRA"	destrukcja b
Zwolnij	5 B: 0 B	"ABRA"	destrukcja a

Przykład: klasa String (cd6)

Operacje na łańcuchach: semantyka przesunięcia [C++11]

```
typedef unsigned uint;
class String
{ void allocate(uint siz, const char *s);   // Wspolny alokator
  void deallocate();
  String(uint n) { allocate(n,""); }
public:
  String(String&& s)            // Konstruktor przesuwający
  { size = s.size; p = s.p;
    s.size = 0; s.p = nullptr;
  }
  String& operator=(String &&s) // Przypisanie przesuwające
  { if(this != &s)
    { deallocate();
      size = s.size; p = s.p;
      s.size = 0; s.p = nullptr;
    }
    return *this;
  }  // Pozostałe konstruktory, przypisanie kopiujące, destruktor, ...
};
```


Przykład: klasa String (cd5)

Koszt operacji na łańcuchach: semantyka przesunięcia

```
int main()
{ String a("ABRA"), b("KAD"); b = a + b + a; }
```

```
Akcja    Total        Treść          Gdzie
Alokuj    5 B: 5 B,   "ABRA"         main(), utworzenie a
Alokuj    4 B: 9 B,   "KAD"          main(), utworzenie b
Alokuj    8 B: 17 B,   ""            operator+(), zmienna temp;
String(String&& s), s == ABRAKAD
Zwolnij   0 B: 17 B,                 pusta destrukcja po move
Alokuj   12 B: 29 B, ""              operator+(), zmienna temp;
String(String&& s), s == ABRAKADABRA
Zwolnij   0 B: 29 B,                 pusta destrukcja po move
operator=(String &&s), s == ABRAKADABRA
Zwolnij   4 B: 25 B, "KAD"           zwolnienie bufora b
Zwolnij   0 B: 25 B,                 pusta destrukcja po move
Zwolnij   8 B: 17 B, "ABRAKAD"       bufor w obiekcie tymczasowym s1+s
Zwolnij  12 B: 5 B,  "ABRAKADABRA"   main(), destrukcja b
Zwolnij   5 B: 0 B,  "ABRA"          main(), destrukcja a
```

Klasy zagnieżdżone - struktura


```
class Ext
{ struct Pri
  {  int x;
     void prvf();
  };
public:
  struct Pbi
  { int y;
    void pubf();
  };
  Pri  efun(Pbi);
  // ...
};
```

```
// Definicje funkcji
void Ext::Pri::prvf()
{ /*...*/ }
void Ext::Pbi::pubf()
{ /*...*/ }
Ext::Pri
Ext::efun(Ext::Pbi p)
{ Ext::Pri z;
  z.x = p.y;
  return z;
}
```

Klasa otaczająca nie ma żadnych specjalnych uprawnień względem klas zagnieżdżonych.

Listy inicjacyjne


```
class Osoba
{ String nazwisko;
  String adres;
  // Inne składowe prywatne
public:
  Osoba(const String &n, const String &a);
  // Inne składowe publiczne
};
```

Konstruktor z przypisaniami

```
Osoba::Osoba(const String &n, const String &a)
{ nazwisko = n;
  adres = a;
}
```

Listy inicjacyjne (cd1)

Deklaracja:

```
Osoba znajoma("Matysiak", "Warszawa");
```

- powoduje aktywację:

`String(char*)`	konwersja tekstu "Warszawa"
`String(char*)`	konwersja tekstu "Matysiak"
`String()`	konstruktor dla składowej nazwisko
`String()`	konstruktor dla składowej adres
`operator=`	przypisanie dla składowej nazwisko
`operator=`	przypisanie dla składowej adres

Listy inicjacyjne (cd2)

Konstruktor z listą inicjacyjną

```
Osoba::Osoba(const String &n, const String &a)
: nazwisko(n), adres(a) // Lista inicjacyjna
{ /* pusta treść konstruktora */ }
```

Sekwencja akcji dla tej wersji konstruktora:

`String(char*)`	konwersja "Warszawa"
`String(char*)`	konwersja "Matysiak"
`String(String &)`	kopiowanie nazwiska
`String(String &)`	kopiowanie adresu.

Uwagi:

- Kolejność inicjacji: wg porządku składowych w klasie !!
- Składowe referencyjne i składowe `const` muszą być tak inicjowane!

Listy inicjacyjne (cd3) – przykład pułapka


```
#include <iostream>
struct Box
{ int x, y, z;
  Box(int xx=1, int yy=1, int zz=1): x(xx), y(yy), z(zz){}
  friend std::ostream& operator<<(std::ostream& os, const Box& b)
  { return os<<'('<<b.x<<','<<b.y<<','<<b.z<<")\n"; }
};
struct S
{ Box &br;
  int n;
  S(): n(10), br(*new Box(n, n, n)){}
  ~S() { delete &br; }
};
int main(){ std::cout<<S().br; } // (-858993460,-858993460,-858993460)
```

Uwaga: Lista inicjacyjna jest rozpatrywana w zasięgu swojego konstruktora; ewentualne konflikty można usuwać stosując operator rezolucji '::' albo wskazanie `this`.

Delegowanie konstruktorów

- Konstruktor może wywoływać inny konstruktor klasy

```
class DC //C++11 {
   int x, y;
   char *name;
 public:
   DC(int v) : x(v), y(0), name("DC") {
     cout << " DC("<< v <<")" << endl;
   }
   DC() : DC(0) { cout << " DC() " << endl; }
 };

int main() {
 DC d;
}
```

```
DC(0)
DC()
```

Wskazania na składowe


```
int fun(int a, float b) { return 10; };

class C {
public:
   int mfun(int a, float b)        { return 20; };
   static int sfun(int a, float b) { return 30; };
};

typedef int (*fptr)(int, float);


void test() {
   fptr fp1 = &fun;
   fptr fp2 = &C::mfun;  // Błąd
   fptr fp3 = &C::sfun;
}
```


Wskazania na składowe - przykład


```
#include <iostream>
class Menu {
private:
  void Open(void)  { std::cout << "Open()"  << std::endl; };
  void Print(void) { std::cout << "Print()" << std::endl; };
public:
  typedef void (Menu::*member_fun_ptr)(void); // (2)
  //member_fun_ptr ftbl[2]; // //(2)
  void (Menu::* ftbl[2])(void); // (1)

  Menu() :
      ftbl{ &Menu::Open, &Menu::Print}  { };
  void test() {
    static int i = 0;
    (this->*ftbl[i])();
    i = (i == 0) ? 1 : 0;
  }
};
```

```
int main() {
    Menu menu;
    menu.test();
    menu.test();
    menu.test();
    std::cout << "ftbl[0]:";
    Menu::member_fun_ptr fp = menu.ftbl[0];
    (menu.*fp)();
}
```

```
Open()
Print()
Open()
ftbl[0]:Open()
```

Wskazania na składowe (cd1)

Terminologia jest (nieco) myląca:

- notacja przypomina konwencje dotyczące wskaźników
- wartości wskazań na składowe **nie są adresami** (można traktować jako przesuniecie względem początku obiektu)
- nie można wykonywać konwersji (np. do typu int)