Podstawy programowania obiektowego

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 27.03.2020

5. Konstruktory i destrultory

Konstruktory i destruktory

- **Konstruktor** jest specjalną funkcją odpowiedzialną za prawidłową inicjalizację obiektu – nadanie polom danych poprawnych wartości początkowych.
- Zadaniem **destruktora** jest przeprowadzenie wymaganych operacji przy usuwaniu obiektu – np.: zwolnienie przydzielonej pamięci na stercie, zamknięcie plików, zwolnienie innych przydzielonych zasobów systemowych.

Konstruktor

- Konstruktor jest specjalną funkcją wołaną w momencie tworzenia obiektu
- Konstruktor nie może zwracać wartości.
- Możliwe jest zdefiniowanie kilku różnych konstruktorów klasy różniących się parametrami.
- Konstruktor ma taką samą nazwę, jak nazwa klasy.

Przydział pamięci dla obiektu i wywołanie konstruktora należy traktować rozłącznie. Często pamięć jest przydzielana wcześniej, natomiast moment wywołania konstruktora wynika z logiki przetwarzania

Konstruktor

Konstruktory są wołane przy tworzeniu obiektów

Globalnych	Przed rozpoczęciem wykonania programu, np.: przed wejściem do funkcji `main`
Lokalnych	Deklarowanych wewnątrz funkcji lub instrukcji blokowej. Konstruktor jest wołany w momencie osiągnięcia odpowiedniej instrukcji.
Dynamicznie tworzonych	Obiekty tworzone są dynamicznie w wyniku wywołania operatora new. Operator new przydziela pamięć obiektu na stercie. Jeżeli alokacja pamięci przebiegła pomyślnie, woła natychmiast konstruktor.
Pól składowych klas	Jeżeli składowymi obiektu są podobiekty innych klas, wówczas podczas tworzenia obiektu nadrzędnego wołane będą konstruktory komponentów.
Podobiektów klas bazowych	W podczas konstrukcji obiektu klasy potomnej wołane są konstruktory inicjujące komponenty klas bazowych.
Obiektów tymczasowych	Obiekty tymczasowe mogą być tworzone jako rezultaty wywołania funkcji zwracających obiekty lub w niektórych przypadkach rzutowania.

Obiekty globalne

```
#include <stdio.h>

class A{
public:
    A(int i){
        printf("A(%d)\n",i);
    }
};

A globalny(0);

int main(){
    printf("main\n");
}
```

Konstruktor obiektu globalnego jest wołany przed wejściem do funkcji `main()`.

Pamięć dla obiektów globalnych przydzielana jest w bloku danych.

Problem: globalne obiekty w różnych modułach -- ich inicjalizacja zależy od kolejności konsolidacji (linkowania).

```
A()
main
```

Obiekty lokalne i tworzone dynamicznie

```
class A{
public:
    A(int i){
        printf("A(%d)\n",i);
    }
};


int main(){
    {
        A a1(1);
    }
    printf("statement#2\n");
    A* ptr = new A(2);
    A a3(3);
    delete ptr;
}
```

```
A(1)
statement#2
A(2)
A(3)
```

Czas życia obiektu a1 ogranicza się do instrukcji blokowej. Pamięć dla obiektu przydzielana jest na stosie.

Kolejny obiekt tworzony jest dynamicznie. Pamięć przydzielana jest na stercie. Musi być zwolniona za pomocą operatora `delete`.

Obiekt a3 tworzony jest na stosie. Zniknie przy wyjściu z funkcji `main`.

Podobiekty klas bazowych i atrybuty

```
class A{
public:
    A(int i){
        printf("A(%d)\n",i);
    }

};


class B : public A{
public:
    B():A(224){printf("B()\n");}
};

class C{
    A a;
    B b;
public:
    C():a(111),b(){
      printf("C()\n");}
};
```

```
int main()
{
printf("Podobiekt klasy bazowej:\n");
B b;
printf("Atrybut (pole klasy):\n");
C c;
}
```

Podobiekt klasy bazowej:
A(224)
B()
Atrybut (pole klasy):
A(111)
A(224)
B()
C()

Konstruktor

Operacje wykonywane przez konstruktor:

1. Woła konstruktor klasy bazowej
2. Woła konstruktory komponentów danych (atrybutów) w kolejności deklaracji.
3. Jeżeli klasa definiuje lub dziedziczy funkcje wirtualne, inicjalizuje wskaźnik `vptr` obiektu wskazujący `VTABLE` klasy.
4. Wykonuje kod zdefiniowany w ciele konstruktora.

Oznacza to, że w trakcie wykonania kodu konstruktora dysponujemy zestawem funkcji wirtualnych odpowiadających danemu poziomowi hierarchii.

Przykład

```
class A
{
public:
    A(){f();}
    virtual void f(){
        printf("A::f ");
    }
};

class B : public A
{

public:
    B():A(){f();}
    void f(){
        printf("B::f ");
    }
};
```

```
class C : public B
{
public:
    C():B(){f();}
    void f(){
        printf("C::f ");
    }
};

void main()
{
    C c;
}
// wypisze A::f B::f C::f
```

Konstruktor

Konstruktor zdefiniowany jest jako

```
class-name([arg-list])
        [:ctor-initializer]
        {body}
```

`arg-list` jest opcjonalną listą argumentów konstruktora

`ctor-initializer` jest listą inicjalizacyjną.

Może ona zawierać:

- nazwy bezpośrednich klas bazowych
- nazwy wirtualnych klas bazowych (również niebezpośrednich)
- nazwy (identyfikatory) atrybutów klasy

Po odpowiednich nazwach w nawiasach podawane są argumenty inicjalizujące.

Lista inicjalizacyjna nie jest kodem wykonywanym, lecz elementem języka umożliwiającym przesłanie parametrów do konstruktorów klas bazowych i atrybutów klasy. Odpowiednie konstruktory wołane są w kolejności wynikającej z deklaracji.

Destruktor

- Destruktor jest funkcją wywoływaną przy usuwaniu obiektu.
  Zadeklarowany jest jako:

```
~class-name()
```

- W klasie może być zdefiniowany dokładnie jeden destruktor.
- Destruktor nie może mieć argumentów
- Destruktor nie może zwracać wartości.

Destruktor

Destruktory są wołane dla następujących typów obiektów

Globalnych	Po zakończeniu programu, np.: po wyjściu z funkcji main
Lokalnych	Destruktor jest wołany w momencie osiągnięcia końca bloku instrukcji.
Dynamicznie tworzonych	Dynamicznie tworzone obiekty zwalniane są za pomocą operatora delete. Operator ten wywołuje destruktor oraz zwalnia pamięć obiektu.
Obiektów tymczasowych	
Jawnie	W niektórych przypadkach destruktory mogą być wywołane jawnie. Dotyczy to głównie obiektów, dla których pamięć przydzielana jest w niestandardowy sposób oraz szczególnych przypadków dziedziczenia wielobazowego.

Globalne, lokalne, dynamiczne

```
class A{
    int v;
public:
    A(int _v):v(_v){}
    ~A(){
        printf("~A(%d)\n",v);
    }
};

A globalny(1);

int main(){
    A a2(2);
    A*ptr = new A(3);
    {
        A a3(4);
    }
    delete ptr;

}
```

Obiekt globalny istnieje przez cały czas działania programu. Zostanie usunięty po wyjściu z funkcji `main` (jako ostatni).
W momencie wyjścia z instrukcji blokowej zwalniana jest pamięć stosu. Pierwszy zostanie usunięty obiekt `a3`. Wcześniej wywołany zostanie jego destruktor.
Destruktor `A(3)` zostanie wywołany w momencie zwolnienia pamięci za pomocą `delete`.
Destruktor obiektu `a2` – przy wyjściu z instrukcji blokowej funkcji `main`.

```
~A(4)
~A(3)
~A(2)
~A(1)
```

Kolejność wykonania destruktorów

Destruktory wołane są w kolejności odwrotnej do konstruktorów:

1. Wpierw wykonywane jest ciało destruktora klasy
2. Następnie wywoływane są destruktory obiektów składowych klasy (komponentów) w kolejności odwrotnej do kolejności deklaracji.
   Dotyczy to komponentów, które nie są zadeklarowane jako `static`. Statyczne komponenty są usuwane jak obiekty globalne.
3. Dalej wołane są destruktory klas bazowych

Wywołanie funkcji wirtualnych w destruktorze

Zasady wywołania funkcji wirtualnych w destruktorach są analogiczne, jak w przypadku konstruktorów.

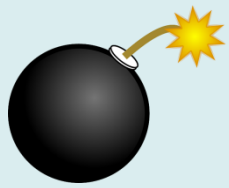
```
class A
{
public:
    ~A(){f();}
    virtual void f(){
        printf("~A::f ");
    }
};
class B : public A
{
public:
    ~B(){
        printf("~B ");
        f();
    }
};
```

```
class C : public B
{
public:
    ~C(){f();}
    void f(){
        printf("~C::f ");
    }
};
void main()
{
    C c;
}
// ~C::f ~B~A::f ~A::f
```

Wirtualne destruktory

```
class Geometry{
public:
    vector<GeoObject*> elements;
    ~Geometry();
    string asWKT();
};

Geometry::~Geometry(){
    for(int i=0;i<elements.size();i++){
        delete elements[i];
    }
}
```



Kiedy usuwamy obiekt za pomocą `delete elements[i]` – wołamy destruktor klasy `GeoObject`, a nie rzeczywistego obiektu wskazywanego przez wskaźnik.

```
class Linestring:public GeoObject{
public:
    vector<Coord> line;
    Linestring(const vector<double>&v);
    // . . .
};
```

To nie zostanie zwolniione

Wirtualne destruktory

Rozwiązanie – w klasie bazowej należy zadeklarować pusty wirtualny destruktor

```
class GeoObject{
public:
    virtual ~GeoObject(){}
    virtual string asWKT(bool printHeader = true)const{
        return "";
    }
};
```

Dzięki temu wskaźnik do kodu destruktora stanie się składnikiem tablicy VTABLE i wywołanie destruktora za pośrednictwem operatora `delete` wywoła zawsze właściwy destruktor klasy potomnej (który na końcu wywoła destruktor klasy bazowej)..

Przykład

```
class A
{
public:
    virtual ~A(){
       printf("~A ");
    }
};

class B : public A
{
public:
    ~B(){
        printf("~B ");
    }
};
```

```
int main()
{
    A*a=new B;
    delete a;
// dla
// virtual ~A(){printf("~A ");}
//  wypisze ~B ~A
// dla ~A(){printf("~A ");}
// wypisze ~A
}
```

Standardowy konstruktor

- Konstruktor standardowy nie ma argumentów. Jest on używany w kilku sytuacjach:
  - Tworzenie tablic obiektów:
  - Tworzenie pustych obiektów i odczyt ich zawartości (np. ze strumienia)

```
class A{
    int v;
public:
    A(int _v):v(_v){}
    void dump(){printf("%d ",v);}
};

int main(){
    A*tab = new A[10];
    for (int i=0;i<      10;i++){
        tab[i].dump()
    }
    delete []tab;
}
```

```
error: no matching function for call to 'A::A()'
A*tab = new A[10];
                ^
```

Dwa możliwe rozwiązania

```
class A{
    int v;
public:
    A(int _v):v(_v){}
    // drugi konstruktor
    A():v(0){}

    void dump(){
        printf("%d ",v);
    }
};
```

```
class A{
    int v;
public:
    // standardowy argument
    A(int _v=0):v(_v){}

    void dump(){
       printf("%d ",v);
    }
};
```

- Klasa może mieć kilka konstruktorów – w tym standardowy
- Możliwe jest użycie standardowych argumentów `A(int _v=0)`

Konstruktor placement new

```
class A{
    int v;
public:
    A(int _v):v(_v){}
    ~A(){printf("~A()[v=%d] ",v);}
    void dump(){
        printf("%d ",v);
    }
};

void test_1(){
    unsigned char buf[100];
    A *ptr = new (buf) A(2);
    ptr->dump();
    ptr->~A();
}
```

```
2 ~A()[v=2]
```

Za pomocą mechanizmu *placemnt new* możemy wywołać konstruktor dla przydzielonego w niestandardowy sposób miejsca w pamięci.

`A *ptr = new (buf) A(2);` -- wskazujemy tablicę znaków `buf`, jako pamięć obiektu -- zajmie tylko jej początkową część.

Na zakończenie **jawnie** wywoływany jest destruktor (w tym przypadku nie jest to konieczne).

Nie zawalniamy pamięci za pomocą `delete ptr`.

Konstruktor placement new

```
void test_2() {
    char *mem = (char *) malloc(10 * sizeof(A));
    for (int i = 0; i < 10; i++) {
        new (mem + i * sizeof(A)) A(i);
    }
    A*tab = (A*)mem;
    for (int i=0;i<10;i++){
        tab[i].dump();
    }
    for (int i=0;i<10;i++)
        tab[i].~A();
    free(mem);
}
```

```
0 1 2 3 4 5 6 7 8 9 ~A()[v=0]
~A()[v=1] ~A()[v=2] ~A()[v=3]
~A()[v=4] ~A()[v=5] ~A()[v=6]
~A()[v=7] ~A()[v=8] ~A()[v=9]
```

- Przydzielana jest pamięć o rozmiarze wystarczającym dla 10 obiektów za pomocą funkcji `malloc()`.
- Dziesięciokrotnie wołany jest operator placement new. Obiekty w tablicy mają różne wartości.
- 10 razy wołany jest jawnie destruktor `~A()`.
- Pamięć jest zwalniana za pomocą funkcji `free()`

Operator przypisania i konstruktor kopiujący

Operator przypisania

- Podczas **przypisania** danemu obiektowi nadaje się wartość drugiego obiektu.
- Standardowa implementacja przypisania polega na skopiowaniu *kolejnych pól* obiektu. To działanie może zostać przedefiniowane poprzez dostarczenie własnego operatora przypisania postaci:

```
X&X::operator=(const X&);
```

- Podczas przypisania poprzednia zawartość obiektu zostaje zastąpiona nową. Dla kontenerów (listy, tablice) wiąże się to ze zwolnieniem pamięci i przydzieleniem nowej.

Konstruktor kopiujący

- Podczas **inicjalizacji** z użyciem konstruktora kopiującego obiekt jest inicjowany wartością innego obiektu.
- Podobnie, jak w przypadku operatora przypisania, standardowa implementacja polega na skopiowaniu kolejnych pól drugiego obiekt.
- W przypadku klas alokujących pamięć konieczna jest odrębna implementacja konstruktor kopiującego:

```
X::X(const X&);
```

Kiedy wołany jest konstruktor kopiujący?

Jawna inicjalizacja obiektu

```
class A {...};
A a1;
A a2=a1; /* wołany jest konstruktor
kopiujący, a nie operator przypisania! */
A a3(a1);
```

Kiedy wołany jest konstruktor kopiujący?

- Inicjalizacja związana z przesyłaniem argumentów do funkcji.
- Argumenty do funkcji mogą być przesyłane przez wartość lub referencję (adres). W pierwszym przypadku na stosie przydziela się miejsce na nowy obiekt, a następnie automatycznie wołany jest konstruktor kopiujący, który inicjuje formalny parametr funkcji wartością argumentu wywołania

```
class A {...};
void foo(A a)
{ // tu zostanie stworzona kopia argumentu
}
A a1;
foo(a1);
```

Kiedy wołany jest konstruktor kopiujący?

- Inicjalizacja związana z przesyłaniem rezultatów wywołania funkcji.
- Funkcje mogą zwracać obiekty przez wartość lub referencje. W przypadku zwracanej referencji obiekt nie może być obiektem automatycznym. W przypadku zwracanej wartości pamięć obiektu jest przydzielana na stosie.

Przykład

```
class Ret{
public:
    Ret(const Ret&o){
        printf("Ret::copy constuctor\n");
    }
    Ret(){
        printf("Ret::constuctor\n");
    }
    Ret&operator=(const Ret&r){
        printf("Ret::assignment\n");
    }
};

Ret foo() {
    Ret r;
    return r;  // tu konstr. kopiujący
}

int main(){
    Ret r2;
    r2=foo();
}
```

Rezultat

```
Ret::constuctor
Ret::constuctor
Ret::copy constuctor
Ret::assignment
```

Dla standardu C++14 takie zachowanie jest obserwowane po wyłączeniu opcji „elizji konstruktorów” czyli pominięcia konstruktorów w procesie przetwarzania.
Odpowiada to w g++ fladze `-fno-elide-constructors`
W przypadku elizji wynikiem będzie:

```
Ret::constuctor
Ret::constuctor
Ret::assignment
```