Beata Jankowska
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

VI. Pola i metody statyczne. Szablony. Dynamiczny przydział pamięci.

1. Pola i metody statyczne

1.1. Do składowych (pól i metod) klasy odwołujemy się w kontekście obiektów, które są jej egzemplarzami. Jednakże, pewne charakterystyczne pola klasy (liczniki, bufory) mogą mieć charakter **zasobu wspólnego dla wszystkich obiektów tej klasy**. W takiej sytuacji, należy te pola związać z samą klasą, nie zaś z jej obiektami. Służy do tego słowo kluczowe `static`, którym należy poprzedzić **deklarację** odpowiedniego pola, np.

```
class K {
  public:
    static int zmienna;
    //...
}
```

Po zadeklarowaniu, zmienna statyczna powinna być **zdefiniowana** w zakresie jej widoczności, np.

```
int K::zmienna = 0;
```

1.2. Odwołania do pól statycznych wewnątrz klasy mają postać identyczną jak odwołania do jej „zwykłych" pól; odwołania do pól statycznych z zewnątrz klasy przyjmują natomiast charakterystyczną postać:

nazwa_klasy `::` *pole_klasy* ,
np.

```
int main
{ //...
  cout << K::zmienna;
  ///...
  return 0;
}
```

1.3. Statyczny charakter można też nadać metodom klasy. Takie metody mogą się odwoływać **tylko do pól statycznych klasy**,
np.

```
#include <iostream>
#include <math.h>
class K {
  public:
    static float zmienna;
    static bool statrownosc (int a) {
      if (a == int(zmienna)) return true;
      else return false;}
    //...
};
int K::zmienna = 3;
int main()
{ int b;
  //...
  if(K::statrownosc(b))
    cout << (K::zmienna);
    else cout << pow(K::zmienna,2.0);
  ///...
  return 0;
}
```

1.4. Do pól i metod statycznych klasy można się też odwoływać za pomocą obiektów tej klasy. Poprawne są więc następujące odwołania:

```
K *wskk = new K();
K k = K();

wskk->statrown (5);
k.statrown (5);
k.zmienna += 4;
wskk->zmienna += 4;
```

```
class Figura
{
  protected:
    int x, y;          //wspolrzedne srodka figury
    double obwod;      //obwod figury – obliczany
                       //w klasie potomnej
  public:
    Figura(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    ~Figura()
    {
        if(obwod > 10) LiczbaFigurObwod10--;
    }

    virtual void OpiszSie() = 0;

  protected:
    static int LiczbaFigurObwod10;    //statyczne pole
                    //zliczające figury o obwodzie > 10
  public:
    static void Wyswietl_LiczbeFigurObwod10 ()
    {
        printf("Obwod tej figury: %f\n", obwod);
        printf("Liczba wszystkich figur
                          o obwodzie > 10: %d\n",
                          LiczbaFigurObwod10);
    }
};
int Figura::LiczbaFigurObwod10 = 0;
```

```
class Kwadrat : public Figura
{
  private:
    int bok;

  public:
    Kwadrat(int x, int y, int a) : Figura(x, y)
    {
        bok = a;
        obwod = 4*a;
        if (obwod > 10) LiczbaFigurObwod10++;
    }
    void OpiszSie()
    {
        printf("Jestem kwadratem w [%d, %d]
                        o boku %d\n", x, y, bok);
    }

};

int main()
{
    Figura *f1 = new Kwadrat(3, 5, 8);
    f1->OpiszSie();
    Kwadrat k1 = Kwadrat(0, 0, 2);
    k1.Figura::OpiszSie(); // metoda czysto wirtualna
    k1.OpiszSie();
    K:: Wyswietl_LiczbeFigurObwod10();
    delete f1;
    k1.Wyswietl_LiczbeFigurObwod10();
    return 0;
}
```

2. Szablony

2.1. Szablony (wzorce) stanowią implementację paradygmatu **programowania uogólnionego**. Polega on na tworzeniu kodu niezależnego od typów danych. Zaletą takiego podejścia jest możliwość skoncentrowania się na algorytmie – jego poprawności i efektywności. Ponadto, stosowanie szablonów prowadzi zwykle do skrócenia kodu programu.

2.2. Abstrakcyjny kod wzorca jest zastępowany na etapie kompilacji programu kodem fizycznym. To zastąpienie, zwane **konkretyzacją wzorca**, dotyczy użycia wzorca, a nie jego definicji.

2.3. W języku C++, szablonów używa się – między innymi – do definiowania klas. Obowiązuje następująca składnia:

```
template<typename C> class nazwa_klasy
{//...}
```

gdzie *C* oznacza nazwę typu parametrycznego, zaś *nazwa_klasy* – nazwę definiowanego typu klasowego. Typu *C* używa się tak samo, jak innych typów. Zakresem tego typu jest deklaracja/definicja opatrzona przedrostkiem `template<typename C>`. Klasy zdefiniowanej przy użyciu wzorca używa się podobnie jak zwykłych klas, dodając do nazwy klasy nazwę właściwego typu, umieszczoną w nawiasach kątowych <>, np.

```
template<typename C> class Napis
{    struct Nrep;
     Nrep* rep;
  public:
     Napis();
     Napis(const C*);
     C czytaj(int i);
  //...
}
```

```
class Chznak {
  //znak chinski
};
// ...
Napis<char> nc();
Napis<unsigned char> nuc();
Napis<Chznak> nch();
```

Składowe wzorca klasy deklaruje się i definiuje tak samo, jak w wypadku zwykłych klas:

```
template<typename C>
struct Napis<C>::Nrep {
   C* s;     //wskaznik do tablicy elementow
   int roz;  //liczba elementow
   int n;    //licznik odwolan
   //...
};

template<typename C>
Napis<C>::Napis() {
   rep = new Nrep();
}

template<typename C>
Napis<C>::Napis(const C* c) {
   rep = new Nrep();
   rep->s = c;
}

template<typename C>
C Napis<C>::czytaj(int i) {
   return rep->s[i];
}
```

2.4. We wzorcu można użyć większej liczby parametrów. Parametry te mogą być typami, lecz także wartościami typów porządkowych i wzorcami, np.

```
template<typename T, int i>
class Bufor {
   T w[i];
   int indeks;
   //...
};
Bufor<char, 127> buf1();
Bufor<Rek, 20> buf2();
```

2.5. Oprócz wzorców klas, w języku C++ można używać także wzorców funkcji, np.

```
#include <vector>
//...
template<typename C>
void sortuj(vector<C>& v) {
    C el1, el2;
    for (int i=0; i<v.size()-1; i++)
        for (j=i; j<v.size()-1; j++) {
            el1 = v[j];
            el2 = v[j+1];
            /*...*/
        }
};
void f(vector<int>& vi, vector<string>& vs) {
   sortuj(vi);
   sortuj(vs);
}
```

Przy wywołaniu funkcji zdefiniowanej za pomocą wzorca, wersja obowiązującego wzorca wynika z postaci argumentów funkcji użytych w tym wywołaniu.

2.6. Szablony należy definiować tylko tam, gdzie są istotnie potrzebne!

2.7. Słowo kluczowe `typename` we wzorcu klasy:

```
template<typename C>
class nazwa_klasy //...
```

lub funkcji:

```
template<typename C>
typ_funkcji  nazwa_funkcji  sygnatura_ funkcji {
   //zwykle z wykorzystaniem typu C
}
```

może być – w zasadzie – używane zamiennie ze słowem kluczowym `class`:

```
template<class C>
class nazwa_klasy //...
```

2.8. Typy generyczne dobrze nadają się do definiowania **kolekcji**, czyli obiektów, które przechowują inne obiekty. Dla wszystkich kolekcji definiuje się operacje dodawania i usuwania elementów. Częstym zastosowaniem kolekcji jest przeglądanie jej elementów, jednego po drugim. W tym celu definiuje się klasę iteratora, charakterystyczną dla typu elementów kolekcji.

2.9. W języku C++, programista ma do dyspozycji **standardową bibliotekę STL** (ang. Standard Template Library) kolekcji i iteratorów. Są wśród nich: `array`, `vector`, `list`, `queue`, `stack`, `map`.

3. Dynamiczny przydział pamięci

3.1. W wielu językach programowania do implementacji procesów wykorzystuje się tzw. sterty. Sterta jest obszarem pamięci, przeznaczonym do przechowywania danych o charakterze globalnym lub o charakterze dynamicznym pewnego procesu. W językach obiektowych stertę wykorzystuje się, między innymi, do przechowywania rekordów aktywacji obiektów (egzemplarzy klas) tworzonych dynamicznie w trakcie działania procesu.

3.2. W językach obiektowych możliwość dynamicznego przydziału pamięci rozszerza się zwykle na wybrane wartości typów prostych lub złożonych.

3.3. W języku C++ można dynamicznie przydzielić pamięć zmiennej typu liczbowego, zmiennej typu tablicowego lub typu strukturalnego. Do realizacji tego przydziału służy operator `new`, używany również do dynamicznego tworzenia samych obiektów – egzemplarzy klas. Wynikiem działania tego operatora jest:

- `0`, jeśli na stercie brakuje pamięci potrzebnej do utworzenia danej lub struktury danych,
- wskaźnik do danej lub struktury danych, która została utworzona w obrębie sterty, w przeciwnym wypadku.

3.4. Rozważmy przykład użycia operatora `new` do konstrukcji drzewiastej reprezentacji wyrażeń wyodrębnianych w procesie analizy składniowej [B. Stroustrup, Język C++]:

```
///...
struct wezelw {
    wartosc_symbolu oper;
    wezelw *lewy;
    wezelw *prawy;
    // ..
};
```

```
wezelw *wyrazenie(bool daj) {
    wezelw *lewy = skladnik(daj);
    for (;;)
        switch (biezacy_symbol=gener_leks()) {
        case PLUS:
        case MINUS:
           { wezelw *n = new wezelw; //utworz wezelw
                                     //na stercie
             n->oper = biezacy_symbol;
             n->lewy = lewy;
             n->prawy = skladnik(true);
             lewy = n;
             break;
           }
        default:
           return lewy;              //przekaz wezel
        }
}
```

3.5. Przy użyciu operatora `new` można tworzyć także tablice dynamiczne, o elementach dowolnych typów. W poniższym przykładzie zastosowano `new` do zbudowania tablicy znaków [B. Stroustrup, Język C++]:

```
///...
char *zachowaj_napis(const char *p) {
   char *s = new char[strlen(p)+1];
   s = p;
   return s;
}

int main(int argc, char *argv[]) {
   //...
   if (argc<2) exit(1);
   char *p = zachowaj_napis(argv[1]);
   //...
}
```

3.6. Utworzone przy użyciu operatora `new` obiekty oraz dane i struktury danych są przechowywane na stercie tak długo, dopóki nie nastąpi ich programowa likwidacja. Pamięć dynamiczna umożliwia więc wydłużenie czasu życia obiektu (danej, struktury danych) poza zasięg wyznaczony aktywnością funkcji, w obrębie której powstał. Takie działanie, choć w pewnych okolicznościach bardzo pożądane, pociąga za sobą niebezpieczeństwo przepełnienia sterty. Przy braku istnienia standardowego odśmiecacza (ang. *garbage collector*), można do niego doprowadzić całkiem łatwo.

3.7. Klasyczny **odśmiecacz** usuwa z pamięci dynamicznej, w regularnych odstępach czasowych wszystkie obiekty (dane, struktury danych), do których nie ma w bieżącym momencie żadnych dowiązań. Zajmowany przez te obiekty (dane, struktury danych) obszar zostaje zwolniony i dołączony do puli obszarów wolnych. Takie postępowanie prowadzi często do tzw. **fragmentacji pamięci**. W celu zwiększenia obszaru ciągłego pamięci wolnej, można dodatkowo przeprowadzić **skupianie** (ang. *compactification*) jej wolnych podobszarów.
Standardowy mechanizm zarządzania pamięcią dynamiczną w języku C++ nie obejmuje odśmiecania.

3.8. Programowa likwidacja obiektów (danych, struktur danych) aktywowanych w sposób dynamiczny przy użyciu operatora `new` następuje przy użyciu operatora `delete`. Operatora tego używa się zawsze w odniesieniu do wskaźnika obiektu, danej, lub struktury danych.

3.9. Węzły drzewiastej reprezentacji wyrażeń tworzone przez program z punktu 3.4 będą używane przez generator kodu pośredniego. Po wykorzystaniu węzła, generator kodu powinien zwolnić zajmowaną przez niego pamięć i przywrócić ją do puli obszarów wolnych [B. Stroustrup, Język C++]:

```
void generuj (wezelw *n) {
    switch (n->oper) {
      case PLUS:
      case MINUS:
           {
             // generacja kodu pośredniego dla wezla n
             // reprezentujacego korzen drzewa rozbioru
             // wyrazenia
             delete n;        //usun wezel z pamieci
           }
    }
}
```

3.10. Do likwidacji struktury tablicowej trzeba się posłużyć operatorem `delete` w postaci `delete[]`. I tak, funkcję `main` z punktu 3.5, używającą tablicy dynamicznej `p`, należałoby zredefiniować do postaci [B. Stroustrup, Język C++]:

```
int main(int argc, char *argv[]) {
    if (argc<2) exit(1);
    char *p = zachowaj_napis(argv[1]);
    //...
    delete[] p; //zwolnienie obszaru zajmowanego
                //w pamieci przez tablice dynamiczna p
}
```

3.11. Warto zwrócić uwagę, że poprawna realizacja operacji `delete p` (`delete[] p`) wymaga znajomości wielkości obszaru pamięci zajmowanego przez obiekt (daną, strukturę danych) wskazywany przez `p`. Z tego powodu, w obszarze zajmowanym przez obiekt umieszcza się dodatkową informację na temat wielkości tego obszaru. Używanie pamięci dynamicznej wiąże się więc z pewnymi narzutami pamięciowymi w stosunku do posługiwania się pamięcią statyczną.

3.12. W wersji C++11 wprowadzono tzw. inteligentne wskaźniki, które – przy braku odśmiecacza – znacznie ułatwiają zarządzanie pamięcią dynamiczną. Dynamiczne wskaźniki stanowią implementację paradygmatu programowania zwanego RAII (ang. Resource Acquisition is Initialization), która opiera się na wykorzystaniu klas szablonowych `unique_ptr`, `shared_ptr` i `weak_ptr`.
Inteligentny wskaźnik powstaje na podstawie deklaracji obiektu odpowiedniej klasy szablonowej. Utworzeniu (**na stosie**) tego obiektu towarzyszy inicjalizacja tzw. **surowego wskaźnika**, któremu zostaje przypisany adres w **pamięci dynamicznej**, przydzielonej na reprezentację obiektu przedmiotowego typu (parametr szablonu).
Likwidacja inteligentnego wskaźnika następuje przy wyjściu z zakresu deklaracji obiektu klasy szablonowej. Uruchamiany przy tej okazji destruktor klasy szablonowej zawiera instrukcję zwolnienia obszaru pamięci dynamicznej wskazywanego przez surowy wskaźnik likwidowanego obiektu.

3.13. Programista ma dyspozycji trzy typy inteligentnych wskaźników:
– `unique_ptr`, przeznaczony do tworzenia obiektów, które nie udostępniają swoich wskaźników; wskaźnik tego obiektu nie może być ani kopiowany, ani przekazywany przez wartość do funkcji – można go co najwyżej przenieść; odbywa się to poprzez przeniesienie własności zasobu pamięci na inny obiekt typu `unique_ptr`, np.

```
#include "stdafx.h"
#include <memory>
#include <iostream>
using namespace std;
typedef struct lzesp
{
    double r;
    double i;
};

int main()
{
    std::unique_ptr<lzesp> l1(new lzesp{ 2.0, 3.5 });
    std::unique_ptr<lzesp> l2;
    cout << l1.get() << endl;
    // dzialania dotyczace wskaznika l1
    l2 = l1; // blad
    l2 = std::move(l1);
    cout << l2.get() << endl;
    // dzialania dotyczace wskaznika l2
}
```

– `shared_ptr`, przeznaczony do tworzenia obiektów zawierających wskaźniki współdzielone; jest to w istocie typ wskaźnika ze zliczaniem referencji do zasobu pamięci – każdy obiekt `shared_ptr` zawiera – oprócz surowego wskaźnika do zasobu pamięci – licznik odwołań; zwolnienie pamięci przez obiekt wskazywany następuje dopiero w momencie wyzerowania tego licznika, np.

```
#include "stdafx.h"
#include <memory>
#include <iostream>
using namespace std;
typedef struct lzesp
{
    double r;
    double i;
};
int main()
{
    std::shared_ptr<lzesp> l1(new lzesp{ 2.0, 3.5 });
    // zliczanie referencji
    cout << l1.use_count() << endl;
    std::shared_ptr<lzesp> l2;
    l2 = l1;
    { std::shared_ptr<lzesp> l3(l1);
      // zliczanie referencji
      cout << l1.use_count() << endl;
    }
    // zliczanie referencji
    cout << l1.use_count() << endl;
    cout << l1.get() << endl;
    cout << l2.get() << endl;
}
```

– `weak_ptr`, wskaźnik o charakterze niezarządzającym, może istnieć dłużej niż obiekt przedmiotowego typu, na który wskazywał. Metoda `expired()` umożliwia badanie ważności wskaźnika, zaś metoda `lock()` – przedłużenie jego żywotności (podtrzymanie przy życiu istniejącego obiektu). Wskaźnik `weak_ptr` posiada licznik referencji, podobnie jak `shared_ptr`. Nie można go jednak stosować wprost do wyłuskania przedmiotowego obiektu, np.

```
#include "stdafx.h"
#include <memory>
#include <iostream>
using namespace std;
typedef struct lzesp
{
  double r;
  double i;
};
int main() {
  weak_ptr<lzesp> l1;
  // wyswietlanie referencji
  std::cout << l1.use_count() << endl;
  {
    {  std::shared_ptr<lzesp>
              l2(new lzesp{ 2.0, 3.5 });
       l1 = l2;
       std::cout << l1.use_count() << endl;
    }
    std::cout << l1.use_count() << endl;
    // dalszy przebieg programu zalezy od ponizszych
    // instrukcji, np.
    // auto l3 = shared_ptr<lzesp>
    //              (new lzesp{ 1.0, 2.0 });
    // l1 = l3;
    if (!l1.expired())
      {
        auto l4 = l1.lock();
        std::cout << (*l4).r << endl << (*l4).i
                                     << endl;
        //wyswietlanie referencji
        std::cout << l1.use_count() << endl;
      }
  }
  // wyswietlanie referencji
  std::cout << l1.use_count() << endl;
}
```