Wrocław 2020.09.25

Autor: Michał Przewoźniczek

Techniki Efektywnego Programowania – zadanie 1
Alokacja i dealokacja prostych typów, wskaźniki, wskaźniki wielokrotne
Alokacja statyczna i dynamiczna, konstruktory i destruktory

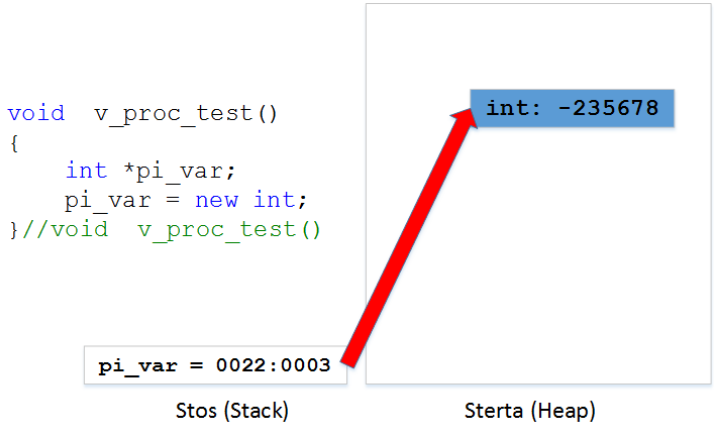
Dynamiczna alokacja i dealokacja.

Alokacja i dealokacja pamięci polega na jej zarezerwowaniu i zwalnianiu. Alokacja i dealokacja dzieli się na dwa rodzaje: statyczną i dynamiczną. Pamięć alokowana statycznie jest alokowana na stosie (ang. *stack*), a pamięć alkowana dynamicznie jest alokowana na tzw. stercie (ang. *heap*). To zadanie dotyczy przede wszystkim alokacji dynamicznej.

Żeby zaalokować pamięć dynamicznie należy zrobić to jawnie. Jedną z możliwości jest użycie operatora `new`. Na przykład:

```
int *pi_var;
pi_var = new int;
```

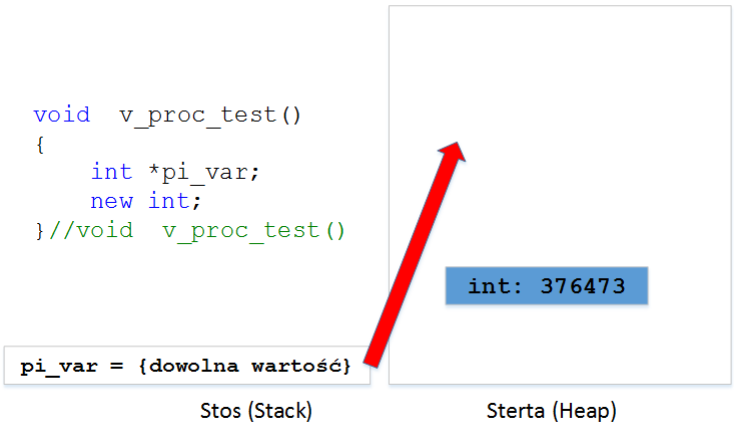
Zmienna `pi_var` jest wskaźnikiem na typ `int`. `pi_var` przechowuje adres w pamięci. Na początku (w momencie deklaracji), ten adres jest dowolny. Operator `new` alokuje dynamicznie pamięć dla zmiennej zadanego typu (w powyższym programie jest to `int`) po czym **zwraca adres** zaalokowanego obszaru pamięci. Ten adres jest przypisywany do zmiennej `pi_var`, która jest wskaźnikiem na typ `int`. Powyższą operację można zobrazować tak, jak na Rys. 1.



Rys. 1 Wizualizacja wykonania instrukcji „`pi_var = new int`”

Należy pamiętać, że w wyniku alokacji pamięć **nie jest** zerowana, to znaczy, że wartość zaalokowanego na stercie int’a może być dowolna.

Należy zauważyć, że gdyby zamiast instrukcji „`pi_var = new int;`” została wykonana instrukcja „`new int`” to wtedy alokacja doszła by do skutku i zmienna typu `int` zostałaby zaalokowana na stercie, ale w programie nie byłoby żadnej zmiennej znającej adres tak zaalokowanego int’a. W takiej sytuacji nie da się użyć zaalokowanej zmiennej typu `int`. Taki program jest zamieszczony na Rys. 2. Jeżeli program zaalokuje jakiś obszar w pamięci, ale nie posiada jego adresu, to nie może takiego obszaru zwolnić. Jest to jedna z sytuacji, w której występuje tzw. *wyciek pamięci* (ang. *memory leak*).



Rys. 2 Wizualizacja wycieku pamięci

Pamięć zaalokowaną dynamicznie zwalania się w sposób jawny. Jednym ze sposobów jest użycie operatora `delete`, jak w poniższym programie. Zwalnianie pamięci, która nie będzie już używana jest ważne, ponieważ w przeciwnym wypadku program będzie powodował wycieki pamięci.

```
int *pi_var;
pi_var = new int;
//uzywaj zmiennej
delete pi_var;
```

Zmienna pi_var jest wskaźnikiem na typ int. Żeby odwołać się do jej wartości, używamy znaku „*”, tak jak w poniższym programie.

```
int *pi_var;
pi_var = new int;
*pi_var = 5;
delete pi_var;
```

Linijka `*pi_var = 5;` **nie jest** pojedynczym poleceniem. Jej wykonanie składa się z następujących operacji:

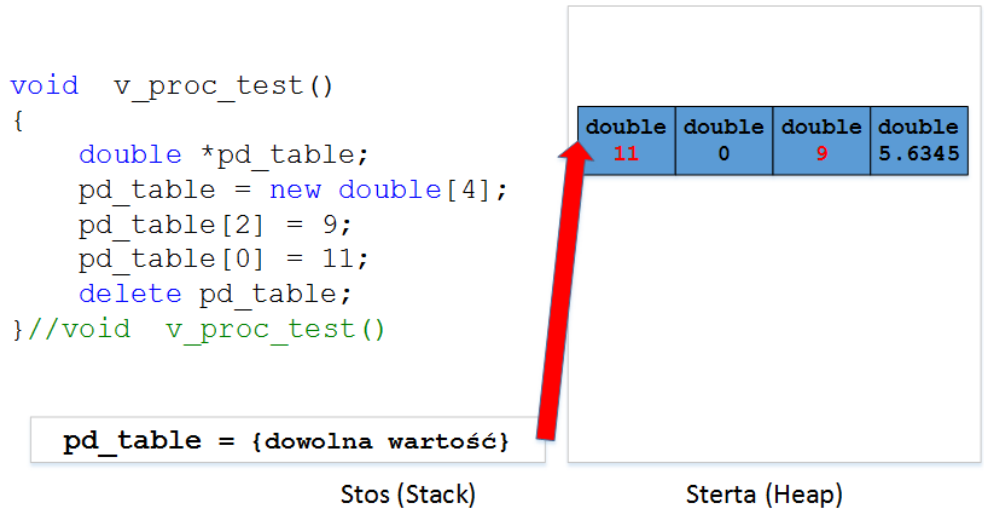
1. Weź adres przechowywany w zmiennej pi_var.
2. Znajdź obszar pamięci, który wskazuje pi_var.
3. Zinterpretuj ten obszar pamięci jako typ `int`.
4. Wpisz do tego obszaru pamięci, zinterpretowanego jako `int`, wartość 5.

Alokacja i dealokacja tablic.

Powyższe przykłady dotyczą alokacji i dealokacji pojedynczej zmiennej. Dynamicznie (statycznie też) można alokować również tablice. Na przykład:

```
double *pd_table;
pd_table = new double[4];
pd_table[2] = 9;
pd_table[0] = 11;
delete pd_table;
```

Powyższy program alokuje tablicę zmiennych typu `double`. Tablica **zawsze jest ciągłym obszarem pamięci**. Do elementów tablicy zazwyczaj odwołujemy się przy pomocy operatora tablicowego „[]”. Wartość w nawiasie to tzw. *offset*, czyli przesunięcie o zadaną liczbę elementów od początku tablicy. Zapis `pd_table[0] = 11;` oznacza przypisanie wartości 11 do pierwszej komórki w tablicy, ponieważ `pd_table` wskazuje na początek tablicy, a mamy przesunąć się o zero elementów. Efekt działania programu przedstawiono na



Rys. 3 Wizualizacja działania programu alokującego tablicę i wykonującego na niej operacje (na czerwono zaznaczono te wartości w tablicy, które zostały zmodyfikowane przez program)

Należy pamiętać, że tablica posiada określony rozmiar, jeśli w powyższym programie odwołamy się do elementu tablicy o offsecie 4 lub większym, to wykroczymy poza zakres tablicy. **W C/C++ programista ma dostęp do faktycznie zaalokowanej tablicy (obszaru pamięci)**, a nie do obiektu, który tablicę jedynie opakowuje (jak w Javie). Oznacza to, że **samo wykroczenie poza zakres tablicy nie spowoduje błędu**. **Błąd mogą spowodować jedynie operacje wykonywane na pamięci, która nie została zaalokowana przez program!**

Rozważmy program tak, jak na kolejnej stronie.

Błędy dostępu do pamięci i ich konsekwencje.

```
int i_table_size = 3;
int *pi_repeats;
pi_repeats = new int[i_table_size];


for  (int ii = 0; ii < i_table_size; ii++)
{
      pi_repeats[ii] = ii + 5;
}//for  (int ii = 0; ii < i_table_size; ii++)

int  i_loop_repeat;
i_loop_repeat = pi_repeats[5];

unsigned int i_loop_counter = 0;
while  ((double) i_loop_counter != (double) i_loop_repeat)
{
      //zrob cos
      i_loop_counter++;
}//while  ((double) i_loop_counter != (double) i_loop_repeat)

char  *pc_string;
pc_string = new char[i_loop_repeat];
//zrob cos z pc_string

pi_repeats[5] = 8;

delete pi_repeats;
delete pc_string;
```

W powyższym programie najpierw alokujemy tablicę `pi_repeats` o długości 3. Logika programu wskazuje, że tablica `pi_repeats` służy do tego, żeby przechowywać informację ile razy ma być powtórzona pętla `while`. W zależności od tego, który element z tablicy `pi_repeats` zostanie wybrany pętla `while` wykona się mniej lub więcej razy.
Niestety, programista popełnił błąd i do zmiennej `i_loop_repeat` pobiera wartość spoza zakresu tablicy `pi_repeats`. W związku z tym wartość zmiennej `i_loop_repeat` może być dowolna. Możliwe są następujące scenariusze:

- Wartość `i_loop_repeat` będzie ujemna i pętla `while` będzie wykonywać się w nieskończoność
- Wartość `i_loop_repeat` będzie wynosić 0 i pętla `while` nie wykona się ani razu
- Wartość `i_loop_repeat` będzie dodatnia i pętla `while` wykona się pewną liczbę razy, ale najprawdopodobniej nie tyle, ile chciałby programista

Jeżeli pętla `while` nie spowoduje zawieszenia programu, to wykona się linia `pc_string = new char[i_loop_repeat];`. W tym miejscu może też wystąpić wyjątek i program zostanie przerwany (jeśli wartość `i_loop_repeat` będzie ujemna lub zero, oraz jeśli będzie zbyt duża).

Wreszcie jeżeli program dotrze do linii `pi_repeats[5] = 8;`, to możliwe są dwa scenariusze:

- Operacja spowoduje wyjątek, ponieważ program podejmie próbę zapisu do pamięci, która nie jest jego (której nie zaalokował). Mogą być tam np. dane innego procesu. Można oczekiwać, że taki błąd wystąpi najczęściej.
- Przypadkiem może się zdarzyć, że pod adresem `pi_repeats[5]`, będzie się znajdować jedna ze zmiennych programu na przykład `i_table_size`. W takim przypadku wyjątek nie wystąpi, ale wartość jednej ze zmiennych zostanie zmodyfikowana w niekontrolowany sposób.

Powyższy przykład pokazuje, że pojedynczy błąd polegający na błędnym dostępie do pamięci nie powoduje bezpośredniego rzucenia wyjątku i przerwania działania programu. Może jednak spowodować całą serię różnych błędów. To które wystąpią nie zależy od napisanego programu, ale od stanu systemu operacyjnego w momencie wywołania programu. Programista nie ma więc nad tym kontroli.

Do komórek tablicy można uzyskiwać dostęp w różny sposób. W powyższym przykładzie, dostęp do drugiej komórki pamięci tablicy był uzyskiwany poprzez wykonanie `pi_repeats[2]`. Można to jednak zrobić inaczej używając następującego zapisu: `*(pi_repeats+2)`. Oznacza on:

1. Weź adres `pi_repeats` i przesuń go o dwa rozmiary wskazywanego typu. `pi_repeats` wskazuje na typ int. Zakładając, że `int` jest 4-bajtowy, nastąpi przesunięcie o 8 bajtów, do początku trzeciego `int`‘a w tablicy.
2. Za pomocą „*” zostanie uzyskany dostęp do zmiennej typu int znajdującej się na trzeciej pozycji w tablicy.

Rzutowanie wskaźników

Typy wskazywane przez wskaźniki mogą być rzutowane. Na przykład:

```
int  i_value = 58;
int *pi_val;
char *pc_val;
void *pv_val;

pi_val = &i_value;
pc_val = (char *) &i_value;
pv_val = (void *) &i_value;
```

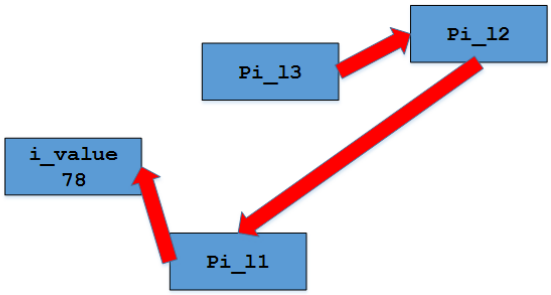
W powyższym programie deklarujemy (jest to alokacja statyczna) zmienną typu int. Za pomocą „&” pobieramy jej adres i wpisujemy do wskaźnika `pi_val`. W ten sposób za pomocą wskaźnika `pi_val` możemy modyfikować wartość zmiennej `i_value`. Wskaźnik `pc_val` jest typu `char`. Jeśli chcemy przypisać mu wartość adresu `i_value`, to musimy dokonać jawnego rzutowania `pc_val = (char *) &i_value;`.Po tej operacji za pomocą wskaźnika ca, będziemy mogli interpretować pierwszy bajt (typ char jest 1-bajtowy) `i_value` jako zmienną typu `char` i modyfikować jego wartość np. wykonując: `*pc_val = 'a';`
Typ `void *` jest typem ogólnym, może wskazywać na cokolwiek. Jeśli będziemy chcieli odwoływać się do adresu wskazywanego przez zmienną `pv_val`, to zawsze będziemy musieli podać jak interpretujemy pamięć wskazywaną przez `pv_val`. Na przykład:
`*((int*) pv_val) = 5;`
zintepretuje `pv_val` jako wskaźnik na int i za pomocą „*” ustawi jego wartość na 5.

Wskaźniki wielokrotne, tablice wielowymiarowe.

Wskaźniki mogą być wielokrotne, co zostało pokazane na Rys. 4.

```
void  v_proc_test()
{
    int  i_value = 78;
    int  *pi_l1, **pi_l2, ***pi_l3;

    pi_l1 = &i_value;
    pi_l2 = &pi_l1;
    pi_l3 = &pi_l2;
}//void  v_proc_test()
```



Rys. 4 Wskaźniki wielokrotne

`pi_l3` jest potrójnym wskaźnikiem na `int` i wskazuje na `pi_l2` (podwójny wskaźnik na `int`), ten z kolei wskazuje na `pi_l1`, który jest wskaźnikiem na `int` (pojedynczym wskaźnikiem na `int`) i ten wreszcie wskazuje na zmienną `i_value` typu `int`. Taka konstrukcja wskaźników przypomina nieco angielską bajkę i wydaje się być niepotrzebna. W rzeczywistości jest inaczej i możliwość operowania na wielokrotnych wskaźnikach jest bardzo przydatna. Jeśli za pomocą `pi_l3` chcemy odwołać się do wartości `i_value` to wystarczy zapis `***pi_l3 = 5;`

Wskaźniki wielokrotne są szczególnie przydatne, przy przekazywaniu informacji do/z funkcji, oraz do alokacji wielowymiarowych tablic. Na przykład:

```
void  vSet5(int  iVal) {iVal = 5;}
void  vSet5(int  *piVal) {*piVal = 5;}

int  i_want_to_be_5;
int  i_will_be_5;

vSet5(i_want_to_be_5);
vSet5(&i_will_be_5);
```

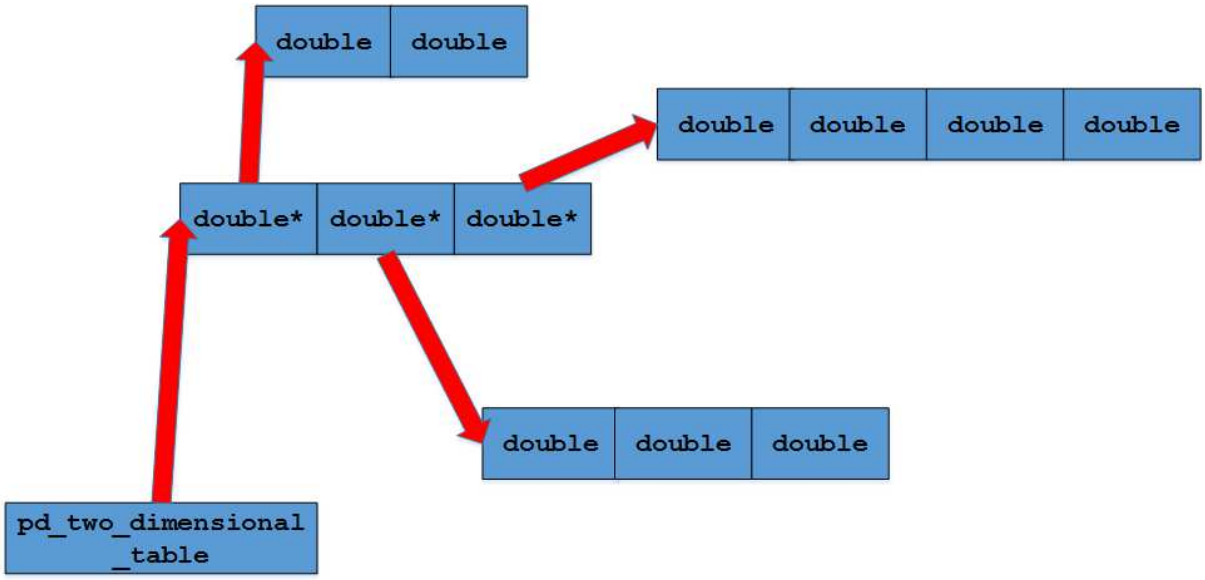
W powyższym programie wartość zmiennej `i_want_to_be_5`, pozostanie niezmieniona po wywołaniu `vSet5(int iVal)`, ponieważ wartość 5 zostanie ustawiona dla zmiennej lokalnej `iVal`. Jednak wartość zmiennej `i_will_be_5`, po wykonaniu `void vSet5(int *piVal)` będzie wynosić 5, ponieważ do funkcji trafia **adres zmiennej** `i_will_be_5`, a nie jej wartość. Funkcja `void vSet5(int *piVal)`, zmienia **wartość zmiennej pod adresem** `piVal`.

```
double  **pd_two_dimensional_table;
int  i_tab_size = 3;

pd_two_dimensional_table = new double*[i_tab_size];

for  (int  ii = 0; ii < i_tab_size; ii++)
      pd_two_dimensional_table[ii] = new double[ii+2];
```

Powyższy program alokuje dwuwymiarową tablicę typu `double`. Nie jest to jednak ciągły obszar ani pojedynczy byt w pamięci. Najpierw alokowana jest tablica (ciągły obszar) pojedynczych wskaźników na typ `double`. Potem alokowane są trzy tablice typu `double`. Każda z nich może być (i w przykładzie jest!) różnej długości! Stan pamięci po alokacji jest przestawiony na



Rys. 5 Tablica dwuwymiarowa – stan pamięci

Żeby skasować dwuwymiarową tablicę. Nie wystarczy wykonanie polecenia `delete  pd_two_dimensional_table;`. Skasuje ono wyłącznie tablicę wskaźników, a pozostałe tablice nie zostaną usunięte z pamięci. Dlatego niezbędne jest wykonanie:

```
for  (int  ii = 0; ii < i_tab_size; ii++)
      delete  pd_two_dimensional_table[ii];
delete  pd_two_dimensional_table;
```

Alokacja statyczna i dynamiczna.

Alokacja pamięci dzieli się na dwa rodzaje: statyczną i dynamiczną. Pamięć alokowana statycznie jest alokowana na stosie (ang. *stack*), a pamięć alokowana dynamicznie jest alokowana na tzw. stercie (ang. *heap*). Wykonanie alokacji dynamicznej zostało omówione w ramach zadania numer 1. Alokacja statyczna ma miejsce w momencie deklaracji zmiennej.

```
void  v_test_proc()
{
      int  i_test;
      double *pd_table;
      //do something
      if  (/*condition*/)
      {
            char c_sign;
            //do something
      }//if  (/*condition*/)
}//void  v_test_proc()
```

W programie powyżej najpierw zostanie utworzona zmienna `i_test`, a potem `pd_table`. Jeśli program wejdzie do bloku `if  (/*condition*/)`, to zostanie utworzona zmienna `c_sign`. Zmienne alokowane statycznie są tworzone na stosie. Dlatego kolejność ich usuwania jest odwrotna do kolejności tworzenia (kładzenia) na stosie. Zmienne zaalokowane statycznie są usuwane na końcu bloku, w którym zostały zaalokowane (zadeklarowane). Tak więc zmienna `c_sign` zostanie usunięta na końcu bloku if, w miejscu „`}//if  (/*condition*/)`” (oczywiście o ile program do tego bloku wejdzie). Na końcu procedury `v_test_proc()` zostaną usunięte zmienne najpierw `pd_table`, a potem `i_test`.

W C++, w odróżnieniu od języków takich jak C#, czy Java można alokować statycznie również obiekty.

```
void  v_test_proc()
{
      CAllocTest c_obj;

      //do something
      if  (/*condition*/)
      {
            CAllocTest c_obj_2;
            //do something
      }//if  (/*condition*/)
}//void  v_test_proc()
```

W powyższym programie, najpierw zostanie utworzony obiekt `c_obj`, a potem obiekt `c_obj_2` (o ile program wejdzie do bloku if). Obiekt `c_obj` zostanie usunięty na końcu bloku if, w miejscu `}//if  (/*condition*/)`, a obiekt `c_obj` na końcu procedury `v_test_proc`.

Poza miejscem wywołania i automatycznym wywołaniem destruktora, alokacja statyczna niczym nie różni się od dynamicznej. Tak więc w momencie konstrukcji obiektu wywoływany jest konstruktor, a przed destrukcją destruktor.

```
CAllocTest *pc_obj_tab;
pc_obj_tab = new CAllocTest[4];
//do something...
delete [] pc_obj_tab;
```

W przypadku alokacji tablicy obiektów, dla każdego zostanie wywołany konstruktor. W przykład w powyższym programie konstruktor zostanie wywołany 4 razy, oddzielnie dla każdego obiektu. W ramach instrukcji `delete [] pc_obj_tab;`, dla 4 obiektów zostanie wywołany destruktor, a potem zostanie zwolniona pamięć.

Uwaga. W przypadku usuwania tablic obiektów istotne jest użycie operatora `delete[]`, zamiast `delete`. Pierwszy z nich wywołuje destruktor dla każdego elementu tablicy, drugi jedynie dla pierwszego obiektu z tablicy. Pamięć oba operatory zwalniają w ten sam sposób.

Tablice można też alkować statycznie. W tym przypadku robi się to w poniższy sposób.

```
CAllocTest pc_obj_tab[4];
```

Konstruktor bezparameterowy

Konstruktor bezparameterowy jest wywoływany natychmiast po zaalokowaniu pamięci dla obiektu. Konstruktor bezparametrowy, podobnie jak każdy inny konstruktor służy do przygotowania nowego obiektu do pracy **i tylko do tego**. W związku z tym, dodanie tam jakiejkolwiek innej funkcjonalności poza inicjacją zmiennych będzie karane jako błąd. Na przykład jeżeli oprogramowaujemy klasę do wyświetlania interfejsu, to czym innym jest utworzenie obiektu, a czym innym uruchomienie jego działania, jak poniżej:

```
CInterface c_int;
c_int.vShowInterface();
```

Jeżeli do uruchomienia działania interfejsu wystarczy:

```
CInterface c_int;
```

…to będzie to traktowane jak błąd.

Nazwy zmiennych w deklaracjach klas

Deklaracje klas w plikach nagłówkowych służą do zaprezentowania programiście, co jest w danej klasie i jak jej używać. W szczególności, w dobrze napisanym programie, deklaracje klas **muszą** zawierać nazwy zmiennych. Program taki jak poniżej, jest błędny ponieważ wiadomo, że konstruktor (z parametrami) bierze 4 wartości typu `int`. **Nie wiadomo** jednak, co oznaczają te wartości (współrzędne prostokąta, długości boków, jeszcze co innego).

```
class  CRectangle
{
public:
      CRectangle(int, int, int, int);

private:
      int i_x;
      int i_y;
      int i_length;
      int i_height;
}//class  CRectangle
```

Konstruktor kopiujący i przekazywanie przez wartość

Kontruktor kopiujący pozwala na stworzenie kopii istniejącego obiektu. Jego składnia jest następująca:

```
{nazwa klasy} (const {nazwa klasy} &pcOther);
```

Na przykład:

```
CAllocTest(const CAllocTest &pcOther)
```

Znak `&` oznacza referencję do obiektu/zmiennej. Tak więc zmienna `&pcOther` jest referencją do typu `CAllocTest`. Referencja to wskaźnik, ale w treści programu odwołujemy się tak jakby nie była wskaźnikiem.

Przykład wywołania konstruktora kopiującego w kodzie:

```
CAllocTest c_static_obj;
CAllocTest *pc_dynamic_obj_copy;

pc_dynamic_obj_copy = new CAllocTest(c_static_obj);
CAllocTest c_static_obj_copy(c_static_obj);
```

W powyższym programie obiekt `c_static_obj` jest alokowany statycznie w momencie deklaracji (natychmiast jest tworzony i wywoływany jest dla niego konstruktor bezparametrowy). Dla dynamicznie alokowanego obiektu, którego adres będzie przechowywany w `pc_dynamic_obj_copy`, wywoływany jest konstruktor kopiujący, który

powinien utworzyć obiekt będący kopią `c_static_obj`. Konstruktor kopiujący zostanie również wywołany dla tworzonego statycznie obiektu `c_static_obj_copy`.

W ramach listy nr 1 jedno z zadań pokazywało możliwość przekazania do danych do funkcji Poprzez wskaźnik, lub poprzez wartość. Podobnie można przekazywać obiekty.

```
void  v_test_rect()
{
      CRectangle  c_my_rect;

      v_mod_rect(&c_my_rect, 5,6);
      v_mod_rect(c_my_rect, 2,3);
      v_try_to_mod_rect(c_my_rect, 1,2);
}//void  v_test_rect()


void  v_mod_rect(CRectangle  *pcRect, int iNewLength, int iNewHeight)
{
      pcRect->vSetLen(iNewLength);
      pcRect->vSetHeight(iNewHeight);
}//void  v_mod_rect(CRectangle  *pcRect, int iNewLength, int iNewHeight)


void  v_mod_rect(CRectangle  &pcRect, int iNewLength, int iNewHeight)
{
      pcRect.vSetLen(iNewLength);
      pcRect.vSetHeight(iNewHeight);
}//void  v_mod_rect(CRectangle  &pcRect, int iNewLength, int iNewHeight)


void  v_try_to_mod_rect(CRectangle  cRect, int iNewLength, int iNewHeight)
{
      cRect.vSetLen(iNewLength);
      cRect.vSetHeight(iNewHeight);
}//void  v_try_to_mod_rect(CRectangle  cRect, int iNewLength, int
iNewHeight)
```

W procedurze `v_test_rect` wywołujemy 3 różne próby modyfikacji obiektu `c_my_rect`. Oba wywołania `v_mod_rect` odwołują się do `c_my_rect` poprzez wskaźnik. Dlatego zmiany wprowadzone w `v_mod_rect` zmienią obiekt `c_my_rect`. Jednak wywołanie `v_try_to_mod_rect` nie zmieni `c_my_rect`, ponieważ do procedury `v_try_to_mod_rect` **jest przekazywana kopia** obiektu `c_my_rect`. Ta kopia jest tworzona w momencie wywołania za pomocą konstruktora kopiującego. A zatem, `v_try_to_mod_rect` będzie modyfikować kopię `c_my_rect` o nazwie `cRect`, a nie `c_my_rect`.

Uwaga: dla każdej klasy kompilator domyślnie tworzy konstrktor kopiujący. Jego działanie polega na skopiowaniu wartości wszystkich pól. **Zastanów się, kiedy dla klasy trzeba zdefiniować konstruktor kopiujący, a kiedy nie jest to potrzebne.**

Zadania

UWAGI:

1. **Pisząc własny program można użyć innego nazewnictwa niż to przedstawione w treści zadania i w przykładach. Należy jednak użyć jakiejś spójnej konwencji kodowania, zgodnie z wymaganiami kursu.**
2. **Nie wolno używać wyjątków (jest to jedynie przypomnienie, wynika to wprost z zasad kursu).**
3. **Wolno używać wyłącznie komend ze standardu C++98**

1. Napisz funkcję `void v_alloc_table_fill_34(int iSize)`, która dynamicznie alokuje jednowymiarową tablicę zmiennych typu `int`. Alokowana tablica ma mieć rozmiar podany w parametrze (`iSize`). Wartości zmiennych w tablicy mają mieć wartość 34.

   Po zaalokowaniu tablicy i wypełnieniu jej wartościami, wypisz stan tablicy na ekranie.

   Pamiętaj o konieczności skasowania (dealokacji) tablicy.

   Zabezpiecz funkcję przed nieprawidłową wartością paramteru `iSize`. Czy ta funkcja będzie użyteczna i wygodna jeśli będzie wypisywać wartości na ekran w przypadku błędu?

   Czy wartość 34 powinna występować bezpośrednio w kodzie `v_alloc_table_fill_34`?

2. Napisz funkcję

```
bool  b_alloc_table_2_dim(int ???piTable, int iSizeX, int iSizeY);
```

, która:

- Alokuje dwuwymiarową tablicę dla typu `int`, dla parameteru podanego w `piTable`. Alokacja ma być wykonana tak, żeby w przypadku wywołania.

```
int  **pi_table;
b_alloc_table_2_dim(???pi_table, 5, 3);
```

`pi_table` ma wskazywać na tablicę dla typu int o wymiarach 5*3.

- Jeżeli operacja się uda funkcja ma zwrócić wartość `true`, lub `false` w przeciwnym przypadku.
- Zastanów się jak dokładnie co wstawić zamiast `???`, jeżeli użycie referencji jest niemożliwe.

3. Napisz funkcję

```
bool b_dealloc_table_2_dim(int ???piTable, int iSizeX, int iSizeY);
```

, która

- Dealokuje dwuwymiarową tablicę typu `int`.

- Jeżeli operacja się uda funkcja ma zwrócić wartość `true`, lub `false` w przeciwnym przypadku.
- Zastanów się jak dokładnie co wstawić zamiast `???`, jeżeli użycie referencji jest niemożliwe. Czy będzie jakaś różnica w porównaniu z `b_alloc_table_2_dim`?
- Czy `b_dealloc_table_2_dim` może mieć mniej parametrów?

4. Zaimplementuj klasę CTable.

Klasa „CTable” musi posiadać następujące konstruktory chrakteryzujące się następującym działaniem:

- **bezparametrowy**: CTable()
  - przypisuje polu s_name domyślną wartość (proszę pamiętać o użyciu odpowiednich stałych)
  - Wypisuje na ekran tekst: „bezp: ‘<s_name>’”, gdzie <s_name> oznacza wartość pola s_name
  - Tworzy tablicę int o domyślnej długości
- **z parametrem**: CTable (string sName, int iTableLen)
  - przypisujący polu s_name, wartość sName
  - Wypisuje na ekran tekst: „parametr: ‘<s_name>’”
  - Przypisuje długość tablicy równą iTableLen
- **kopiujący:** CTable (CTable &pcOther)
  - przypisujący polu s_name, wartość pcOther.s_name i doklejający tekst „_copy”. Na przykład, jeśli pcOther.s_name = „test” to wartość pola s_name dla obiektu utworzonego konstruktorem kopiującym będzie: „test_copy”
  - Kopiuje tablicę int
  - Wypisuje na ekran tekst: „kopiuj: ‘<s_name>’”

Ponadto klasa ma posiadać:

- **Destruktor,** wypisujący na ekran następujący tekst: „usuwam: ‘<s_name>’”
- **Metodę,** void vSetName(string sName), przypisującą polu s_name, wartość sName
- **Metodę,** bool bSetNewSize(int iTableLen), zmieniającą długość tablicy i zwracającą informację, czy udało się to zrobić, czy nie.
- **Metodę,** CTable *pcClone(), która zwraca nowy obiekt klasy CTable, będący klonem obiektu dla którego pcClone() zostało wywołane. Na przykład:

```
CTable  c_tab;
CTable  *pc_new_tab;
pc_new_tab = c_tab.pcClone();
```

Zastanów się, czy metoda pcClone() jest podobna funkcjonalnie do innych elementów kodu? Jeśli tak, to jak napisać program tak, żeby wykorzystać ten fakt i nie kopiować kodu niepotrzebnie?

Napisz dwie procedury:

```
void  v_mod_tab(CTable  *pcTab, int iNewSize);
void  v_mod_tab(CTable  cTab, int iNewSize);
```

Sprawdź, która procedura faktycznie zmodyfikuje obiekt podany jako parametr. Czy zostaną utworzone jakieś kopie?

Przetestuj statyczną i dynamiczną alokację obiektów CTable. Kiedy wywoływany jest konstruktor, a kiedy destruktor? Sprawdź wywołanie konstruktora i destruktorów w przypadku alokacji tablicy obiektów CTable.

Zalecana literatura

Jerzy Grębosz „Symfonia C++”, Wydawnictwo Edition, 2000.
Lub inna zalecana dla przedmiotu.
Binky pointer fun (https://www.youtube.com/watch?v=5VnDaHBi8dM)