Tablice, wskaźniki, referencje

- **Tablice obiektów są tworzone dokładnie tak samo, jak i tablice, składające się z elementów innego typu**

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

class cl
{
   int i;
public:
   void  set_i(int k) { i = k; }
   int     get_i() { return i; }
};

int main()
{
   cl ob[10];
   for(int i=0; i<9; i++)
               ob[i].set_i(i+1);
   ……..
   return 0;
}
```

➢ **Jeśli klasa ma konstruktor sparametryzowany, należy wykonywać inicjowanie tablicy obiektów:**

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

class cl
{
   int i;
public:
   cl(int k) { i = k; }
   int     get_i() { return i; }
};

int main()
{
   cl ob[3] = { 1, 2, 3 };    //wartości początkowe

   for(int i=0; i<3; i++)
              cout  <<  ob[i].get_i() << endl;

   return 0;
}
```

➢ **składnia cl ob[3] = { 1, 2, 3 }; jest skrótem bardziej ogólnej postaci inicjowania: cl ob[3] = { cl(1), cl(2), cl(3) };**

➢ **Skrócona postać może być używana wyłącznie dla konstruktorów, pobierających tylko jeden argument. Jeśli ilość argumentów konstruktora większa od jedynki, trzeba używać dłuższą postać inicjowania:**

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

class cl
{
   int i, j;
public:
   cl(int k, int m) { i = k;  j = m; }
   int     get_i() { return i; }
   int     get_j() { return j; }
};

int main()
{
   cl ob[3] = {
                     cl(1,2),
                     cl(2,4),
                     cl(5,2)
                  };    //inicjowanie

   for(int i=0; i<3; i++)
              cout  <<  ob[i].get_i() << ”  ” << ob[i].get_j()  << endl;

   return 0;
}
```

➢ **A jeśli program potrzebuje w jednych przypadkach używać inicjowanie obiektów, a w innych – nie (dana są wprowadzone operatorami we/wy, dynamiczna alokacje pamięci, …) ? Trzeba przeciążyć konstruktor:**

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

class cl
{
   int i;
public:
    cl()  { i = 0; }          //wywołujemy dla tablic nieinicjowanych
   cl(int k) { i = k; }       //wywołujemy dla tablic inicjowanych
   int     get_i() { return i; }
};

int main()
{
   cl ob[3] = { 1, 2, 3 };    //inicjowanie
   cl rt[256];                //OK

   cl *ptr_cl = (cl *)malloc(10000*sizeof(cl));
   if(!ptr_cl)
   {   //opracowanie błędu alokacji pamięci
   }

   return 0;
}
```

Wskaźniki do obiektów

- **Wskaźnikami do obiektów można posługiwać się podobnie jak wskaźnikami do zmiennych innych typów**

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

class cl
{
   int i;
public:
    cl()  { i = 0; }          //wywołujemy dla tablic nieinicjowanych
   cl(int k) { i = k; }       //wywołujemy dla tablic inicjowanych
   void set_i(int k) { i = k; }
   int     get_i() { return i; }
};
```

```
int main()
{
   int i;
   cl *ptr_cl = (cl *)malloc(10000*sizeof(cl));
   if(!ptr_cl)
   {   //opracowanie błędu alokacji pamięci
   }

   cl *p1 = ptr_cl;  //ustawiamy wskanik p1 na początek tablicy

   for(i=0; i<10000; i++, p1++)
            p1 -> set_i(i+1);     //to samo:  ptr_cl[i].set_i(i+1);

   p1 = ptr_cl; //ustawiamy wskanik p1 na początek tablicy

   for(i=0; i<10000; i++, p1++)
   {
            if(ptr_cl[i].get_i() != p1->get_i())
                     cout << "błąd\t";
   }

   //free(p1)     //błąd !!!
   free(ptr_cl);
   ptr_cl = NULL;

   return 0;
}
```

➢ **Podczas wywoływania funkcji – składowej klasy jest do niej automatycznie przekazywany niejawny argument, który jest wskaźnikiem do wywołującego obiektu (do obiektu, z którego wywoływana jest funkcja). Taki wskaźnik to *this*.**

```
class myclass
{
    double base;
    int exp;
public:
    myclass(double a, int p);
     double  comp();
};

myclass::myclass (double a, int p)
{
     base = a;     //this->base = a;
     exp   = p;     //this->exp   = p;
}

double myclass::comp()
{
     double ret = base;        //ret = this->base;
     int ti = exp;                  //ti   = this->exp;
     for( ; ti > 0; ti - - )
               ret  * = base;      //ret *= this->base;
     return ret;
}
```

```
int main()
{
     myclass ob(10.0, 3), rr(100.0, 2);
     double val = ob.comp();    //val = 10^3 -  wskaźnik this wskazuje na obiekt ob
     val = rr.comp();                //val = 100^2 – wskaźnik this wskazuje na obiekt rr

    return 0;
}
```

- **Funkcje zaprzyjaźnione nie są składowymi klasy, dla tego wskaźnik this nie jest do nich przekazywany.**

- **Statychne funkcje składowe nie mają wskaźnika this.**

Operatory dynamicznej alokacji w C++

- **C++ ma dwa operatory dla roboty z pamięcią dynamiczną: *new* i *delete***

- **Operator *new* alokuje pewien obszar pamięci i zwraca wskaźnik do jego początku:**

  ptr_var = new typ;

- **Operator *delete* zwalnia pamięć, zaalokowaną przez *new* :**

  delete ptr_var;

- **Tu *ptr_var* jest wskaźnikem do typu *typ*: *typ *ptr_var;***

- **Jeśli rozmiar pamięci dostępnej jest niewystarczający, są możliwe dwie sytuacji:**

- **Domyślnie będzie wygenerowany wyjątek *bad_alloc* . Wyjątek – to jest błąd dynamiczny (który powstaje w trakcie wykonania programu). Program powinien obsłużyć wyjątek i podjęć odpowiednie akcje. Dokładnie o wyjątkach i obsługiwaniu sytuacji wyjątkowych będziemy mówili później.**

```
……..
myclass *ptr_ob = NULL;

try
{
        //próbujemy alokować pamieć
        ptr_ob = new myclass;
}
catch(bad_alloc aa)
{
        //jeśli pamięć nie pozostanie zaalokowaną, powstaje wyjątek bad_alloc
        //obsługa  tej wyjątkowej sytuacji
        cout  <<  ”brak pamięci \n”;
        system(”pause”);
        exit(1);
}
……
delete ptr_ob;
```

Jeśli wyjątek nie pozostanie obsługiwany

```
ptr_ob = new myclass;
```

działanie programu będzie przerwane (PAGE FAULT).

- **Zwraca wskaźnik NULL. Standard C++ dopuszcza to:**

```
#include <new>
using namespace std;
……..
myclass *ptr_ob = NULL;
ptr_obj = new(nothrow)  myclass;
 if(!ptr_obj)
{
            //obsługa błedu
……..
}
……
delete  ptr_ob;
```

Będziemy używali pierwszego trybu użycia operatora new.

- **Jeśli wskaźnik nie jest poprawny, działanie operatora *delete* zakończy się rujnowaniem układu pamięci dynamicznej procesu – PAGE FAULT.**

- **Zalety użycia operatorów *new, delete* (w porównaniu do funkcji C *malloc*, *free*):**

  - **Nie potrzebują użycia *sizeof(typ)* – są bardziej bezpieczne**
  - **Nie trzeba wykonywać rzutowania jawnego – zwraca wskaźnik określonego typu**
  - **Można przeciążyć *new* i *delete* (przeciążenie operatorów) – to pozwala tworzyć swoją własną systemu alokacji pamięci.**

- **Jeśli blok danych był zaalokowany przez *malloc*, musi być zwolniony przez *free*. Jeśli blok danych był zaalokowany przez *new*, musi być zwolniony przez *delete*.**

- **W dużych systemach programowych zwykłe są napisany specjalny klas, który dokonuje alokacje i zwolnienie pamięci (menedżer pamięci). Decyzje o to, jakie sposoby alokacji pamięci pozostają użyte, podejmuje doświadczony programista systemowy i sam pisze ten klas. Dla innych programistów jest koniecznie wszystkie działania z pamięcią dynamiczną wykonywać tylko przez metody tej klasy. (SCAD, Iter)**

- **Inicjacja alokowanej pamięci:**

ptr_var = new typ (wartość inicjująca);

Tu *ptr_var* – wskaźnik na obiekt typu *typ*, a typ wartości inicjującej powinien być zgodnym z typem *typ*.

o

```
double  *tt;
try
{
        tt = new double (3.14);
}
catch(bad_alloc aa)
{
……..
}
……..
delete tt;
```

o

```
class samp
{
        int i, j;
public:
        samp(int ii, int jj) { i = ii; j = jj; }
……….……….……….……….……….……….……….……….……….……….……….……….
} ;

int  main()
{
        samp *ptr_ob;
        //inicjowanie powoduje wywołanie konstruktora klasy
        ptr_ob = new samp (10, 20);
……….……….……….……….……….……….……….……….……….……….……….……….
        delete ptr_ob;
        return 0;
}
```

- **Alokowanie tablic:**

  ptr_var = *new typ* [ilość_elementów_tablicy];
  ***delete* [] ptr_var;**

ptr_var – wskaźnik do typu *typ.*

- **Uwaga! Jeśli opuścić ”[]” w operatorze *delete*, będzie zwolniony tylko element tablicy o indeksie zero.**

- **Podczas alokowania tablic nie można im przypisywać wartości początkowe**

- **Przykłady: AllocNew, AllocNew1**

- **Dostęp do składowych elementów tablicy:**

```
wskaźnik_do_tablicy[numer_elementu_tablicy].zmienna  = …..

wskaźnik_do_tablicy[numer_elementu_tablicy].nazwa_metody(argum_faktyczne) ;
```

Referencje

- **Referencja – to jest niejawny wskaźnik. Można ją wykorzystać:**
  - **jako parametr funkcji**
  - **jako wartość zwracana funkcją**
  - **jako zmienna referencyjna**

- **Referencje jako parametr funkcji. Parametry mogą być przekazywany do funkcji w dwa sposoby:**
  - **przez wartość – do funkcji przekazywana jest kopia argumentu.**
  - **przez referencje – przekazywany jest wskaźnik do argumentu.**

- **Wskaźnik do argumentu w C++ można przekazać w sposób jawny (dokładnie tak, jak w języku C), a można posłużyć się parametrem referencyjnym.**

Przekazanie argumentu faktycznego przez wskaźnik	Przekazanie argumentu faktycznego w sposób referencyjny
#include <iostream> using namespace std; void fun(int *n); //argument formalny – wskaźnik int main() { int i = 0; f(&i); //pobieranie adresu zmiennej i i //przekazanie go do funkcji cout << ”nowa wartość i: ” << i << endl; return 0; } void fun(int *n) { *n = 100; //przypisanie liczby 100 zmiennej, //do której wskazuje wskaźnik n } Wydruk: Nowa wartość i: 100	**#include <iostream> using namespace std; void fun(int &n); //argument formalny - referencje int main() { int i = 0; f(i); //przekazanie przez referencje cout << ”nowa wartość i: ” << i << endl; return 0; } void fun(int &n) { n = 100; //przypisanie liczby 100 argumentowi, //który zastępuje argument formalny n } Wydruk: Nowa wartość i: 100**

- **Dla deklarowania parametru referencyjnego przed nazwą argumentu formalnego stawimy &**

- **W obszarze funkcji przed nazwą argumentu formalnego nie piszemy * (operacje wskazania pośredniego). Instrukcje *n* = 100; oznacza, że liczba 100 pozostaje umieszczona w argument aktualny, który zastępuje argument formalny *n* przy wywołani funkcji**

- **Przy wywołani funkcji przed nazwą argumentu aktualnego nie stawimy &. Przy deklaracji funkcji pozostało zadeklarowane, że argument formalny jest parametrem referencyjnym – kompilator automatycznie przekazuje funkcji adres argumentu aktualnego.**

- **Parametr referencyjny nie jest wskaźnikiem – arytmetyka wskaźników tu nie działa:**

```
void fun(int &n)
{
      n++;          //zwiększa wartość n o jedynkę
      n = n+20;     //oznacza, że wartość n będzie
                    //zwiększona o 20
}
```

```
void fun(int *n)
{
      n++;          //  n wskazuje na adres n+sizeof(int)
      n = n+20;  //n wskazuje na adres n + 20*sizeof(int)
      *n = *n+20;  //zwiększenie zmiennej z adresem n
                    //o 20
}
```

- **Przykład W2_0_0**

Przekazywanie obiektów przez referencje

- **Przekazywanie obiektu jako argumentu (przez wartość) powoduje utworzenie kopii tego obiektu -> wywołanie destruktora przy zakończeniu roboty funkcji -> możliwe uboczne efekty, czas wywołania funkcji jest odnośnie długi w skutek kopiowania argumentów i zapisów-odczytów w/z stosu. Zmiany danych obiektu, dokonywane w funkcji, odbywają się w kopii, po zakończeniu funkcji nie występują w oryginale obiektu.**

- **Przekazywanie obiektu przez referencje: kopia obiektu nie jest tworzoną. Po zakończeniu działania funkcji obiekt przekazywany jako parametr nie jest niszczony -> destruktor nie jest wywoływany. Przekazywanie jest szybsze od przekazywania przez wartość (nie powoduje kopiowania, zapisów/odczytów stosu). Zmiany danych obiektu, dokonywane w funkcji, odbywają się w oryginale obiektu (kopii obiektu niema), po zakończeniu funkcji te zmiany występują w oryginale obiektu. Składnia odwołania do składowych obiektu, przekazywanego przez referencje, jest taka sama, jak i przy przekazaniu przez wartość:**

nazwa_obiektu.nazwa_zmiennej
nazwa_obiektu.nazwa_metody(lista_argumetów_aktualnych);

- **Przykład W2_1_0**

Zwracanie referencji

- **Funkcje może zwracać referencje. Jest to korzystne przy przeciążeniu operatorów, a również nadaje możliwość występowania tej funkcji po lewej stronie operatora przypisania.**

- **Przykład: W2_1_1**

Zmienne referencyjne

- **Referencje można zadeklarować jako samodzielną zmienną – powstają zmienne referencyjne. Podczas definiowania zmiennej referencyjnej obowiązkowa jest jej inicjacja. Faktycznie zmienna referencyjna występuje jako druga nazwa zmiennej. To powoduje bałagan w programie – doświadczone programisty unikają samodzielnych zmiennych referencyjnych**

- **Przykład W2_1_1**

Ograniczenia dotyczące referencji

- **Nie można pobrać adresu referencji**
- **Nie można tworzyć tablic referencji**
- **Nie można stosować wskaźników do referencji**
- **Nie można tworzyć referencji do wartości bitowych**
- **Zmienna referencyjna musi być zainicjowana w momencie deklaracji (nie do tyczy to składowych klasy, argumentów funkcji i wartości zwracanych)**
- **Nie można używać referencji o wartości NULL**