Język C++ zajęcia nr 3

Operatory `new` i `delete`

Obiekty ze względu na **czas życia** dzielą się na **statyczne** (trwające przez cały czas wykonania programu) i **dynamiczne** (tworzone i usuwane z pamięci w różnych fazach wykonania programu).

- Statyczny obiekt to **obiekt globalny** lub **obiekt lokalny w klasie pamięci `static`**.
- Obiekt dynamiczny może być obiektem **automatycznym** (lokalny obiekt zdefiniowany bez podania klasy pamięci static) lub **kontrolowanym**.

Dynamiczny obiekt kontrolowany znajduje się pod kontrolą użytkownika i w odpowiednich momentach w sposób jawny jest tworzony w wolnej pamięci (np. na stercie) i usuwany z niej, gdy nie jest już potrzebny. Operator **`new`** służy do przydzielania pamięci, a operator **`delete`** do jej późniejszego zwalniania.

Najprostsze użycie operatora **`new`** ma postać wyrażenia:

`new` ***typ***

Efektem wykonania operacji **`new`** jest utworzenie w wolnej pamięci obiektu typu ***typ***. Wartością wyrażenia z operatorem **`new`** jest **wskaźnik na utworzony obiekt**. W przypadku, gdy nie jest możliwe utworzenie obiektu z powodu wyczerpania zapasu wolnej pamięci, wartością wyrażenia jest NULL.

> **Obiekt utworzony przy pomocy operatora `new` nie ma nazwy, poprzez którą można się do niego odwoływać. Jedyny dostęp do takiego obiektu uzyskiwany jest za pośrednictwem wskaźnika.**

Pamięć **alokowana** (przydzielona) dla obiektu jest z nim związana do zakończenia wykonania programu lub do jej zwolnienia operatorem **`delete`**, który można użyć w wyrażeniu:

`delete` ***wsk***

gdzie ***wsk*** ma wartość wskaźnika na obiekt utworzony wcześniej operatorem **`new`**.

Uwaga: Wyrażenie z operatorem **`delete`** nie zwraca żadnej wartości (jest typu **`void`**).

Obiekt dynamiczny bez jawnej inicjalizacji zawiera przypadkowe dane. Równocześnie z utworzeniem nowego obiektu możliwa jest jego inicjalizacja poprzez umieszczenie inicjalizatora w nawiasach po nazwie typu w wyrażeniu z operatorem **new**:

```
int *p;                 // Definicja wskaźnika na int
p=new int(18);          // Utworzenie nowego obiektu o wartości 18
cout << *p;             // Wyświetlenie wartości obiektu
delete p;               // Usunięcie obiektu
```

Operator **new** służy również do alokowania tablic dynamicznych o rozmiarach ustalanych w trakcie wykonania programu. Wyrażenie z operatorem **new** ma wówczas postać:

new ***typ*** **[** ***rozmiar*** **]**

gdzie ***rozmiar*** jest dowolnym wyrażeniem typu całkowitego. Wartością tego wyrażenia jest wskaźnik na początek tablicy.

Tablica dynamiczna utworzona operatorem **new** jest usuwana z pamięci w wyniku użycia operatora **delete** w wyrażeniu:

delete [ ] ***wsk***

przy czym ***wsk*** ma wartość wskaźnika na początek tablicy.

Uwaga: Tablice alokowane operatorem **new** nie mogą być inicjalizowane.

Sortowanie tablicy dynamicznej:

Wprowadź, skompiluj i uruchom program sortowania tablicy. Zinterpretuj poszczególne elementy tekstu źródłowego!!

Sortowanie tablicy dynamicznej:

```
#include <iostream>
using namespace std;
int main()
{
   int *t,x;
   long n,k,z;
   cout << "Podaj rozmiar tablicy: ";
   cin >> n;
   if(n<=0)
   {
      cout << "Ujemny lub zerowy rozmiar";
      return 1;
   }
   //------------------- Próba utworzenia tablicy
   if((t=new int[n])==NULL)
   {
      cout << "Brak wolnej pamieci";
      return 2;
   }
   //------------------- Wczytanie elementów tablicy
   cout << "Podaj kolejne elementy tablicy" << endl;
   for(k=0;k<n;k++)cin >> t[k];
   //------------------- Sortowanie przez zamianę
   do
      for(k=0,z=0;k<n-1;k++)
         if(t[k]>t[k+1])
         {
            x=t[k];
            t[k]=t[k+1];
            t[k+1]=x;
            z++;
         }
   while(z);
   //------------------- Wyświetlenie tablicy
   cout << "Tablica posortowana: " << endl;
   for(k=0;k<n;k++)cout << t[k] << " ";
   //------------------- Usunięcie tablicy
   delete[]t;
   return 0;
}
```

Przykładowe wyniki wykonania programu:

```
Podaj rozmiar tablicy: 12          KLAWIATURA 12↵

Podaj kolejne elementy tablicy:

62 45 12 30 73 6 12 94 11 0 52 49  KLAWIATURA 62 45 12 30 73 6 12 94 11 0 52 49↵

Tablica posortowana:

0 6 11 12 12 30 45 49 52 62 73 94
```

Zadanie domowe:

- **zinterpretuj poszczególne kroki algorytmu sortowania tablicy przez zamianę,**
- **zinterpretuj kod źródłowy całości programu (obowiązuje umiejętność jego interpretacji)!**

Uwaga: W przypadku tablic dwuwymiarowych można utworzyć tablicę dynamiczną, której elementami są tablice. Rozmiar tablic będących elementami musi być stały i znany w momencie kompilacji (wyłącznie takie tablice stanowią **typ** w rozumieniu języka C++). Przy tworzeniu tablicy dynamicznej jedynie pierwszy indeks może być ustalany w trakcie wykonania programu. Nie można stosować operatora **new** w wyrażeniu mającym postać:

```
new int[n][k];
```

Dynamiczna tablica tablic (zinterpretuj program):

```
#include <iostream>
using namespace std;
int main()
{
   int n;
   int (*t)[12];        // Wskaźnik na tablicę 12 elementów int
   cin >> n;
   t=new int[n][12];  // Utworzenie tablicy rozmiarów n*12
   delete[]t;
}
```

Funkcje rozwijane

Funkcje, biorąc pod uwagę sposób ich wywołania, mają postać funkcji zamkniętych lub funkcji otwartych (rozwijanych):

> **Funkcja zamknięta** stanowi fragment kodu, który jest umieszczony w jednym obszarze pamięci i może być wielokrotnie wykonywany po wywołaniu funkcji, co jest zwykle realizowane przez procesor za pomocą rozkazu skoku ze śladem. Funkcja, której kod ma duże rozmiary, powinna mieć ze względu na oszczędność pamięci formę funkcji zamkniętej.
>
> **Funkcja otwarta (rozwijana)** ma postać fragmentu kodu skopiowanego w wielu miejscach pamięci i zastępującego każde wywołanie tej funkcji. Rozwijanie funkcji w miejscu wywołania daje oszczędność czasu podczas wykonania programu, a w przypadku funkcji o niewielkich rozmiarach nie zwiększa wymaganej dla programu wielkości pamięci.

Funkcje w języku C++ mają standardowo postać funkcji zamkniętych. Użycie w deklaracji funkcji słowa **`inline`** jest wskazówką dla kompilatora, że dana funkcja powinna być funkcją rozwijaną.

```
inline double delta(double a, double b, double c)
{
   return b*b-4*a*c;
}
```

Uwaga: Słowo **`inline`** w deklaracji funkcji może zostać zignorowane przez kompilator i wówczas dana funkcja ma formę funkcji zamkniętej.

Funkcje przeciążone

Język C++ dopuszcza jednoczesne wykorzystanie różnych **funkcji mających jednakową nazwę** w tym samym obszarze zakresu danej nazwy. Zespół takich funkcji to **funkcje przeciążone**. Kompilator rozpoznaje poszczególne funkcje w grupie funkcji przeciążonych po typie ich argumentów. Każda funkcja przeciążona musi różnić się argumentami od innej funkcji o identycznej nazwie. Różnica dotyczy liczby, kolejności i typów poszczególnych argumentów.

Funkcje o przeciążonych nazwach upraszczają programowanie pozwalając tym samym operacjom wykonywanym na obiektach różnych typów nadawać jednakowe nazwy. Dotyczy to także operatorów (w języku C ten sam operator ***** jako mnożenie stosuje się między innymi do typów char, int, float, double).

Różna liczba argumentów:

```
float pole(float);                    // pole koła
float pole(float, float);             // pole prostokąta
float pole(float, float, float);      // pole trójkąta
```

Różne typy argumentów:

```
char root(char);                      // pierwiastek kwadratowy z char
int root(int);                        // pierwiastek kwadratowy z int
double root(double);                  // pierwiastek kwadratowy z double
```

Różna liczba i typy argumentów:

```
void ekran(char, int);                // wyświetlenie wyników
void ekran(int*, char);               // wyświetlenie wyników
void ekran(int, float*, int);         // wyświetlenie wyników
void ekran(char(*)[10], int, char, char);   // wyświetlenie wyników
```

Dopasowanie argumentów

Podczas wywołania funkcji przeciążonej następuje dopasowanie argumentów poprzez porównanie typów argumentów wywołania z typami parametrów formalnych deklaracji funkcji. Wykonywana jest ta spośród funkcji przeciążonych, która wykazuje zgodność typów przy dopasowaniu argumentów. W przypadku braku ścisłej zgodności typów wykonywane są standardowe konwersje lub konwersje zdefiniowane przez użytkownika. **Jeżeli pomimo konwersji nie można dopasować typów argumentów, sygnalizowany jest błąd**.

Dopasowanie argumentów (zinterpretuj program):

```
#include <iostream>
using namespace std;
void wypisz(int x)
{
   cout << "   Liczba calkowita = " << x << endl;
   return;
}
void wypisz(double x)
{
   cout << " Liczba rzeczywista = " << x << endl;
   return;
}
void wypisz(char x)
{
   cout << "Znak alfanumeryczny = " << x << endl;
   return;
}
void wypisz(char* x)
{
   cout << "    Lancuch znakowy = " << x << endl;
   return;
}
int main()
{
   wypisz(128);
   wypisz(3976.28);
   wypisz('R');
   wypisz("Tygrys");
}
```

```
   Liczba calkowita = 128

 Liczba rzeczywista = 3976.28

Znak alfanumeryczny = R

    Lancuch znakowy = Tygrys
```

Jednakowo nazwane różne funkcje przeciążone muszą różnić się między sobą typami swoich argumentów. Przy ustalaniu, czy typy argumentów dwóch funkcji przeciążonych różnią się wystarczająco, stosowana jest zasada **różnych inicjalizatorów**: Jeżeli dwa argumenty wymagają różnych inicjalizatorów, to mogą być argumentami dwóch jednakowo nazwanych funkcji przeciążonych.

Rozróżnialne typy argumentów:

- **Wskaźnik na zmienną** pewnego typu i **wskaźnik na stałą** tego samego typu jako argumenty różnią się wystarczająco do przeciążenia funkcji.
- **Referencja do zmiennej** pewnego typu i **stała referencja** do tego samego typu jako argumenty różnią się wystarczająco do przeciążenia funkcji.

```
int fun(char*);              // Przeciążenie możliwe, różne
int fun(const char*);              // typy argumentów funkcji


char fun(int&);              // Przeciążenie możliwe, różne
char fun(const int&);              // typy argumentów funkcji
```

Nierozróżnialne typy argumentów:

- Wskaźnik na pewien typ i tablica elementów tego typu jako argumenty nie różnią się wystarczająco do przeciążenia funkcji.
- Zmienna pewnego typu i stała tego samego typu jako argumenty nie różnią się wystarczająco do przeciążenia funkcji.
- Zmienna pewnego typu i referencja do tego samego typu jako argumenty nie różnią się wystarczająco do przeciążenia funkcji.

```
int fun(int*);          // Błąd, typy argumentów obydwu
int fun(int[]);         // funkcji nie są rozróżnialne

int fun(char);          // Błąd, typy argumentów obydwu
int fun(const char);    // funkcji nie są rozróżnialne

float fun(int);         // Błąd, typy argumentów obydwu
float fun(int&);        // funkcji nie są rozróżnialne
```

W przypadku funkcji przeciążonych o argumentach tablicowych różniących się **wyłącznie rozmiarem** tablic rozróżnialność argumentów nie występuje dla tablic jednowymiarowych i tablic wielowymiarowych o różnych jedynie pierwszych od lewej rozmiarach.

Funkcje z argumentami tablicowymi:

Nierozróżnialne:

```
void fun(int[20]);

void fun(int[12]);


void fun(char[16][14][5]);

void fun(char[30][14][5]);
```

Rozróżnialne:

```
void fun(char[16][14][5]);

void fun(char[30][14][8]);
```

Problem dopasowania typów argumentów występuje również przy wyznaczaniu adresu funkcji przeciążonej, a także związany jest ze zwracaniem takiego adresu przez inną funkcję.

Argumenty domyślne wywołania funkcji

Liczba argumentów podawanych przy wywołaniu funkcji może być mniejsza niż liczba parametrów w deklaracji funkcji. Pominięte **argumenty domyślne** (domniemane) przybierają wartości wyrażeń przypisanych w deklaracji funkcji odpowiednim parametrom formalnym. Na liście argumentów w deklaracji funkcji parametry bez wartości domyślnych muszą poprzedzać wszystkie parametry domyślne.

Funkcja z argumentami domyślnymi (program):

```
#include <iomanip>
#include <iostream>
using namespace std;
int g=0,m=0;
void czas(int godz=g+6, int min=m+30, int sek=0)
{
   cout<<setfill('0')<<setw(2)<<godz<<":"<<setw(2)<<min<<":"
       <<setw(2)<<sek<<endl;
   return;
}

int main()
{
   czas(10,25,12);
   czas(16,40);
   czas(19);
   czas();
   g=8;
   m=20;
   czas(5);
   czas();
}
```

```
10:25:12

16:40:00

19:30:00

06:30:00

05:50:00

14:50:00
```

Uwaga: Wyrażenie, które reprezentuje domyślną wartość argumentu, nie może zawierać obiektów lokalnych ani pozostałych argumentów deklarowanej funkcji.

Zadanie CPP03.

Zmodyfikuj program C06-1 stosownie do następujących założeń:

- nie ma ograniczenia *n*<=100, *n* jest z zakresu *int*, tablica składa się z elementów typu *int*, **zastosuj dynamiczną alokację tablicy z wykorzystaniem operatora new, po jej wykorzystaniu usuń tablicę za pomocą operatora delete;**

- do wprowadzania/wyprowadzania danych zastosuj mechanizmy strumieniowe (cin, cout), wykorzystaj odpowiednie manipulatory do wyrównania liczb całkowitych w kolumnie tak, aby pozycje dziesiętne zostały wyrównane w pionie.

Kod źródłowy programu CPP03 wraz z oknem zawierającymi efekty przykładowego uruchomienia programu, należy zapisać w **jednym** dokumencie (**format Word**) o nazwie **CPP03.doc** i przesłać za pośrednictwem platformy Moodle w wyznaczonym terminie. Na początku dokumentu podać nazwiska autorów. Obaj autorzy niezależnie muszą przesłać dokument.