Wykład nr 2

Temat: Tablice statyczne, funkcje, klasy pamięci, reguły zasięgu, obszary nazw.

Cytaty:

Forma zawsze następuje po funkcji.

Louis Henri Sullivan

Nie możesz ufać kodowi, którego osobiście nie napisałeś w całości.

Ken Thompson

Gdy się nie wie, co się robi, to się dzieją takie rzeczy,
że się nie wie, co się dzieje.

znana prawda programistyczna

Wykład nr 2

Temat: Tablice statyczne, funkcje, klasy pamięci, reguły zasięgu, obszary nazw.

Zakres wykładu:

- kwalifikator **`const`**
- tablice statyczne jedno-, dwu- i wielowymiarowe
- inicjalizowanie tablic
- tablice znakowe
- deklarowanie, definiowanie i wywoływanie funkcji
- funkcje **`inline`** oraz funkcje z pustą listą parametrów
- przekazywanie danych do funkcji przez wartość
- argumenty domyślne funkcji
- przeciążanie nazw funkcji
- klasy pamięci
- reguły zasięgu
- zasłanianie nazw, operator rozróżniania zasięgu ::
- obszary nazw
- podsumowanie
- ćwiczenia powtórzeniowe i sprawdzające
- następny wykład

Kwalifikator `const`

Kwalifikator **const** (*ang. stały*) umożliwia programiście poinformowanie kompilatora, że wartość pewnej zmiennej nie powinna być modyfikowana.

Stałe symboliczne w trakcie definiowania muszą być inicjalizowane wyrażeniem stałym i nie mogą być później modyfikowane.

Przykład:

```
main()
{
   //const int x;  //błąd: zmienna x musi być zainicjalizowana w momencie definicji
   const int x=5;  //zainicjalizowanie stałej symbolicznej

   cout<<x;
   //x=6;          //błąd: nie można zmieniać wartości stałej symbolicznej

   //------------------------------------------------------------------------------

   const int rozmiar=10;

   int tablica[rozmiar];    //definicja tablicy 10-elementowej za pomocą stałej
}
```

Typowy błąd programisty

Przypisanie wartości stałej symbolicznej za pomocą instrukcji wykonywalnej jest **błędem składni**.

Typowy błąd programisty

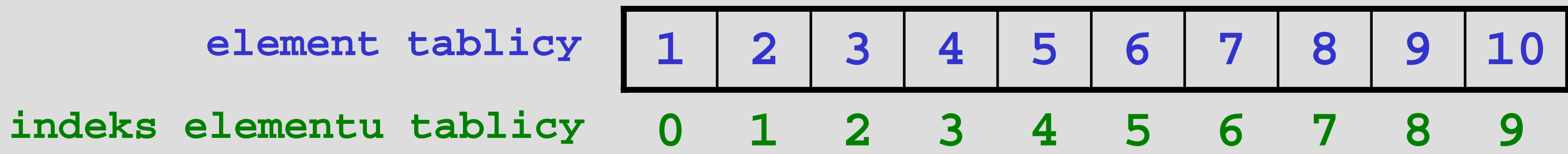
Tylko stałe oraz stałe symboliczne mogą być używane do deklarowania/definiowania automatycznych i statycznych tablic. Brak stałej w takim przypadku jest **błędem składni**.

Wskazówka praktyczna

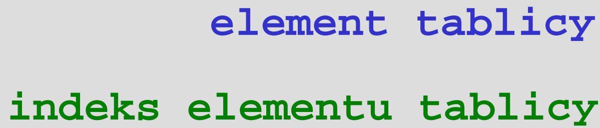
Definiowanie wielkości tablicy jako stałej symbolicznej zamiast stałej nadaje programowi większą skalowalność i czytelność.

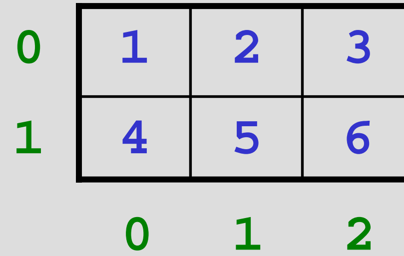
Tablice statyczne jedno-, dwu- i wielowymiarowe

Tablica – jest to ciąg obiektów tego samego typu, które zajmują ciągły obszar w pamięci.

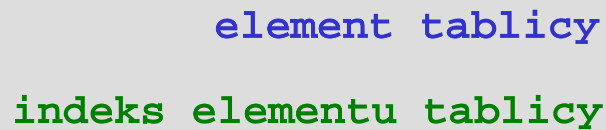


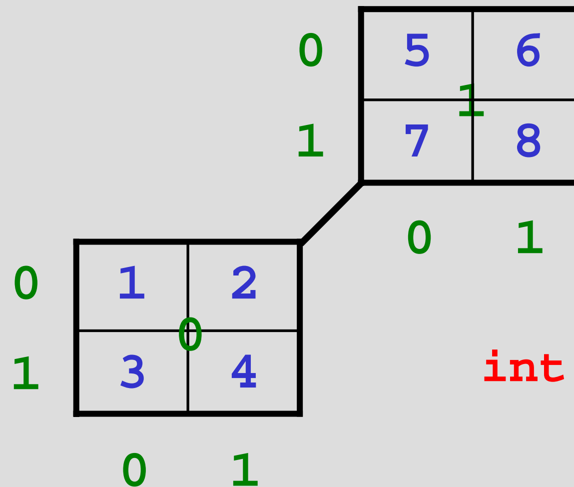

```
int tablica[10];
```






```
int tablica[2][3];
```






```
int tablica[2][2][2];
```

Tablice można tworzyć z:

- typów fundamentalnych (z wyjątkiem `void`)
- typów wyliczeniowych (`enum`)
- wskaźników
- innych tablic (tablice dwu- i wielowymiarowe)
- obiektów typów zdefiniowanych przez użytkownika (obiektów klas)
- wskaźników do pokazywania na składniki klasy

Przykłady definicji tablic statycznych:

```
int tablica[20];                //tablica 20 liczb typu int

int tablica[3][4];              //tablica 12 liczb typu int

int tablica[2][3][4];           //tablica 24 liczb typu int

char zdanie[80];                //tablica 80 znaków typu char

float numery[20];               //tablica 20 liczb typu float

double liczby[5];               //tablica 5 liczb typu double

int *tabl_wsk[12];              //tablica 12 wskaźników do int

Klasa tabl_obiektow[10];        //tablica 10 obiektów klasy Klasa
```

Tablica statyczna – tablica, której rozmiar jest znany już na etapie kompilacji.
Tablica dynamicza – tablica, której rozmiar nie jest znany na etapie kompilacji.

Inicjalizowanie tablic

Inicjalizowanie tablic jednowymiarowych – wartości początkowe elementom tablicy można nadać w różny sposób:

- za pomocą przypisania z wykorzystaniem pętli

```
int tablica[10];                    //tablica 10 liczb typu int
for(int i=0;i<10;i++)
   tablica[i]=0;                    //inicjalizacja w pętli

for(int i=0;i<10;i++)
   cin>>tablica[i];                 //wczytanie elementów z klawiatury
```

- inicjalizacja w momencie definicji tablicy

```
int tab1[5]={0};                    //wszystkie elementy zerowe
int tab2[5]={1};                    //pierwszy element równy 1, pozostałe zerowe
int tab3[5]={2,5,6};                //elementy 4 i 5 zerowe (o indeksach 3 i 4)
int tab4[5]={1,2,3,4,5};            //jawna inicjalizacja elementów
int tab5[ ]={1,2,3,4};              //liczba elementów tablicy równa 4
int tab6[5]={0,1,2,3,4,5};          //błąd składni, za dużo elementów
const int rozmiar=5;                //rozmiar tablicy - stała symboliczna
int tab7[rozmiar]={0};              //rozmiar wcześniej zdefiniowany
```

- przypisanie każdego elementu tablicy z osobna

```
int tab[3];                         //tablica 3 liczb typu int
tab[0]=1; tab[1]=4; tab[2]=-3       //jawna inicjalizacja (w jednej linii)
```

Inicjalizowanie tablic dwuwymiarowych – wartości początkowe elementom tablicy można nadać w różny sposób:

- **za pomocą przypisania z wykorzystaniem pętli**

```
int tablica[5][10];              //macierz 5x10 (5 wierszy, 10 kolumn)
for(int i=0;i<5;i++)
   for(int j=0;j<10;j++)
      tablica[i][j]=0;           //inicjalizacja w pętli

for(int i=0;i<5;i++)
   for(int j=0;j<10;j++)
      cin>>tablica[i][j];        //wczytanie elementów z klawiatury
```

- **inicjalizacja w momencie definicji tablicy**

```
int tab1[2][2]={0};                 //wszystkie elementy zerowe
int tab2[2][2]={ {1}, {3,4} };      //element tab2[0][1] zerowy
int tab3[2][2]={ {1,2}, {3,4} };    //wszystkie elementy jawnie zainicjalizowane
int tab4[2][2]={ 1,2,3,4 };         //wszystkie elementy jawnie zainicjalizowane
```

- **przypisanie każdego elementu tablicy z osobna**

```
int tab[2][2];                   //tablica 4 liczb typu int
tab[0][0]=1; tab[0][1]=2;        //jawna inicjalizacja
tab[1][0]=3; tab[1][1]=4;        //jawna inicjalizacja
```

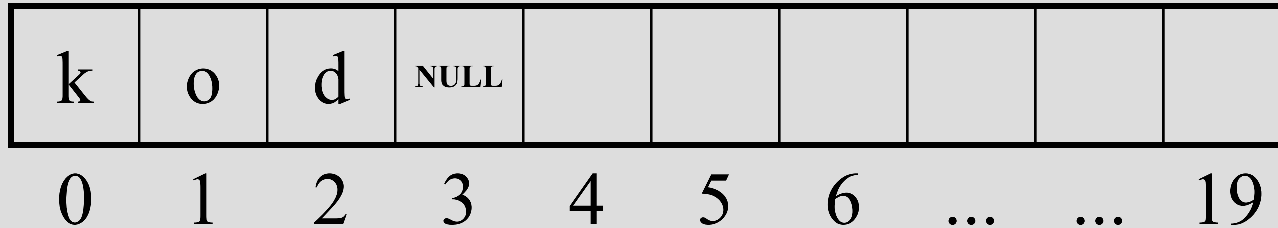
Tablice znakowe

Tablica znakowa – tablica do przechowywania znaków (np. liter).
String – ciąg znaków ASCII zakończony znakiem NULL (znak o kodzie 0).

```
char zdanie1[20];                //tablica znakowa char

char zdanie2[20]={”kod”};        //znak NULL automatycznie dopisany,
char zdanie3[  ]={”kod”};        //string poprawnie zakończony,
                                 //jest znak NULL

char zdanie4[20]={’k’,’o’,’d’}; //reszta inicjalizowana zerami,
                                 //a więc jest znak NULL –
                                 //string jest poprawnie zakończony
```




```
char zdanie5[  ]={’k’,’o’,’d’}; //nie ma znaku NULL, nie jest to
                                 //błąd, gdy litery nie mają być
                                 //używane jako ciąg znaków – czyli
                                 //string – ale jako luźne litery do
                                 //innych celów
```

UWAGA: do tablicy znakowej `zdanie1` można wpisywać dowolne ciągi znaków ale bez tzw. „białych znaków” (np. spacji). Wpisanie białego znaku pomiędzy kolejnymi wyrazami spowoduje pominięcie wyrazów od pierwszego białego znaku do końca tablicy.

DO ZAPAMIĘTANIA:

- Rozmiar tablicy statycznej musi być stałą, znaną już na etapie kompilacji
- Aby odwołać się do konkretnego elementu tablicy, należy podać nazwę tablicy i indeks odpowiedniego elementu
- Indeks elementu tablicy musi być liczbą całkowitą lub wyrażeniem całkowitym
- Pierwszy element w każdej tablicy ma indeks równy 0
- Elementy tablic wielowymiarowych umieszczane są kolejno w pamięci komputera tak, że najszybciej zmienia się najbardziej skrajny prawy indeks
- Nazwy tablic podlegają tym samym konwencjom co inne nazwy zmiennych
- **Indeks elementów tablicy w C++ zaczyna się od zera!**
- **NAZWA TABLICY jest równocześnie ADRESEM JEJ ZEROWEGO ELEMENTU! (STAŁYM WSKAŹNIKIEM do jej zerowego elementu)!**

Typowy błąd programisty

Zauważ różnicę między elementem tablicy, a indeksem elementu tablicy. Ponieważ indeks elementów tablicy zaczyna się od 0, siódmy element tablicy ma indeks 6, podczas gdy indeks 7 dotyczy ósmego elementu tablicy. Jest to źródło często popełnianego błędu, tzw. błędu „przesunięcia o jeden”.

Typowy błąd programisty

Brak zainicjalizowania elementów tablicy, która tego wymaga (np. zdefiniowanej w zakresie lokalnym) może powodować **błąd logiczny** (**błąd wykonania**).

Typowy błąd programisty

Umieszczenie na liście inicjalizującej tablicę większej ilości wartości niż elementów tablicy jest **błędem składni**.

Typowy błąd programisty

Do definiowania rozmiaru tablic statycznych mogą być używane tylko stałe, brak stałej jest **błędem składni**.

Typowy błąd programisty

Odwołanie się do elementu poza granicami tablicy jest **błędem logicznym** (**błądem wykonania**). Odwołanie takie jest zależne od systemu.

Typowy błąd programisty

Brak zapewnienia wystarczająco dużego rozmiaru tablicy znakowej do przechowania napisu może powodować utratę danych i poważne **błędy wykonania**.

Wskazówka dotycząca wydajności

Zamiast inicjalizowania tablicy za pomocą instrukcji przypisania w trakcie wykonywania programu, zainicjalizuj ją w czasie kompilacji za pomocą listy inicjalizującej, dzięki czemu program wykona się szybciej.

Dobry styl programisty

Definiowanie wielkości tablicy jako stałej symbolicznej (np. `rozmiar`) zamiast stałej (np. `10`) czyni program bardziej czytelnym. Ta technika jest używana do pozbycia się tzw. *magicznych liczb*, czyli np. kilkakrotne umieszczanie wielkości 10 w kodzie programu przetwarzającym 10-elementową tablicę nadaje jej sztuczne znaczenie i może być mylące, gdy program zawiera inne wartości 10 nie mające nic wspólnego z rozmiarem tablicy.

Wskazówka praktyczna

Kiedy przechodzisz tablicę z użyciem pętli, indeks elementu tablicy nigdy nie powinien stać się mniejszy od 0 i zawsze powinien być mniejszy o jeden od liczby elementów tablicy. Upewnij się, że warunek zakończenia pętli uniemożliwia dostęp do elementów poza tym zakresem.

Deklarowanie, definiowanie i wywoływanie funkcji

Funkcja – podprogram realizujący jakieś zadanie, który najczęściej jako rezultat zwraca jakąś wartość.

Deklaracja funkcji – informuje kompilator jaką wartość funkcja będzie zwracała oraz liczbę i typ argumentów (w kolejności ich oczekiwanego pojawienia się)

```
typ_zwracany nazwa_funkcji(lista_typów_argumentów);
```

np.

```
int dodaj(int a,int b);                //deklaracja (prototyp) funkcji
```

Definicja funkcji – definicja jest deklaracją, w której przedstawiona jest treść funkcji

```
typ_zwracany nazwa_funkcji(lista_typów_argumentów)
{
   //ciało (treść) funkcji
}
```

np.

```
int dodaj(int a,int b)                 //definicja funkcji
{
   return a+b;
}
```

Wywołanie funkcji – to napisanie nazwy funkcji z podaniem argumentów funkcji w nawiasie okrągłym () (jeśli funkcja takie argumenty posiada)

np.

```
int liczb1=5,liczba2=10,wynik;

wynik=dodaj(liczba1,liczba2);                //wywołanie funkcji

...
...
...

int dodaj(int a,int b)                       //definicja funkcji
{
   return a+b;
}
```

Argumenty (parametry) formalne – to argumenty zdefiniowane w prototypie funkcji,
Argumenty (parametry) aktualne – to argumenty użyte w wywołaniu funkcji.

Funkcje `inline` oraz funkcje z pustą listą parametrów

Funkcje **inline**

Wywołania funkcji wymagają pewnego czasu wykonywania. Czas ten można zredukować poprzez zdefiniowanie funkcji jako **inline**, zwłaszcza dla małych, niedługich funkcji.

Kwalifikator ten „doradza” kompilatorowi wygenerowanie kopii kodu funkcji w miejscach jej wywołań, aby uniknąć jej wywołania. Kompilator może ignorować kwalifikator **inline**, zwłaszcza dla większych funkcji, dlatego powinien być on używany tylko z małymi, często używanymi funkcjami.

Przykład:

```
int liczb1=5,liczba2=10,wynik;

wynik=dodaj(liczba1,liczba2); //wywołanie funkcji

...
...
...

inline int dodaj(int a,int b) //definicja funkcji inline
{
   return a+b;
}
```

Funkcje z pustą listą parametrów

W języku C++ można definiować tzw. **funkcje z pustą listą parametrów**, nie pobierającą żadnych argumentów. Funkcja taka jest określona przez napisanie **`void`** w nawiasie okrągłym w nagłówku definicji, albo przez pozostawienie pustego nawiasu.

Jeżeli funkcja ma dodatkowo nie zwracać żadnej wartości, należy umieścić słowo **`void`** także przed nazwą tej funkcji.

Przykład:

```
int funkcja1(void)       //definicja funkcji z pustą listą parametrów
{                        //zwracająca liczbę całkowitą
   ...
   return wartosc_calkowita;
}
//**************************************************************************
int funkcja2()           //definicja funkcji z pustą listą parametrów
{                        //zwracająca liczbę całkowitą
   ...
   return wartosc_calkowita;
}
//**************************************************************************
void funkcja3()          //definicja funkcji z pustą listą parametrów
{                        //nie zwracająca żadnej wartości
   ...
}
```

Przekazywanie danych do funkcji przez wartość

Przekazywanie argumentów do funkcji (wywoływanie funkcji) przez wartość powoduje, że funkcja pracuje na kopii przekazywanej zmiennej, więc NIE MA możliwości jej modyfikowania. Kopia ta oraz inne zmienne definiowane w obrębie funkcji przechowywane są najczęściej na stosie.

```
int funkcja(int aa)                   //definicja funkcji
{
   aa=aa+100;
   return aa;
}

main()                                //definicja funkcji main
{
   int a=5, b;                        //definicja zmiennych

   cout<<a<<" "<<b;

   b=funkcja(a);                      //wywołanie funkcji

   cout<<a<<" "<<b;
}
```

Przed wywołaniem funkcji: a=5, b=?
Po wywołaniu funkcji: a=5, b=105

Argumenty domyślne funkcji

Argumenty domyślne funkcji

Programista może określić, że dany argument funkcji jest **argumentem domyślnym** i dostarczyć dla niego wartość domyślną.

Argumenty domyślne muszą być położone najdalej z prawej strony na liście parametrów funkcji.

Argumenty domyślne powinny być określone wraz z pierwszym wystąpieniem nazwy funkcji – zazwyczaj w jej prototypie (w definicji funkcji już nie!!!).

Przykład:

```
int Objetosc_Pudelka(int dlug=1,int szer=1,int wys=1);

main()
{
   //przykładowe wywołania funkcji Objetosc_Pudelka
   Objetosc_Pudelka();         //dlug=1,szer=1,wys=1       objetosc=1
   Objetosc_Pudelka(10);       //dlug=10,szer=1,wys=1      objetosc=10
   Objetosc_Pudelka(10,5);     //dlug=10,szer=5,wys=1      objetosc=50
   Objetosc_Pudelka(10,5,2);   //dlug=10,szer=5,wys=2      objetosc=100
}

int Objetosc_Pudelka(int dlug,int szer,int wys)
{
   return dlug*szer*wys;
}
```

Przeciążanie nazw funkcji

Przeciążanie nazw funkcji

Przeciążenie nazw funkcji – sytuacja, gdy zdefiniowanych jest kilka funkcji o takiej samej nazwie, różniących się zestawem parametrów (argumentów).

Przeciążanie funkcji jest często stosowane do tworzenia różnych funkcji o takiej samej nazwie, które przeprowadzają podobne zadania, ale na różnych typach danych.

Przykład:

```
int kwadrat(int x) {return x*x;}
double kwadrat(double x) {return x*x;}
```

UWAGA: W języku C++ istnieje także możliwość **przeciążania operatorów**, tak, aby operowały na **obiektach** typów danych zdefiniowanych przez użytkownika.

Przykład:

```
cout<<"tekst"<<endl;        //przeciążony jest operator << który
                            //pracuje na obiekcie cout
```

DO ZAPAMIĘTANIA:

- Wywołanie funkcji zwracającej rezultat samo w sobie ma jakąś wartość (ściślej taką, jaką ma rezultat zwracany przez tę funkcję), można je zatem użyć w dowolnym wyrażeniu, np. `2+3.2+a+dodaj(2,3)+dodaj(a,5);`

- Argumenty formalne funkcji to te, które występują w pierwszej linijce definicji funkcji (w nawiasie), natomiast argumenty aktualne to te, które występują w wywołaniu funkcji

- W języku C++ pojedyncze zmienne (w tym elementy tablic) przekazywane są domyślnie przez wartość

- **NAZWA FUNKCJI jest równocześnie JEJ ADRESEM w pamięci!**

Typowy błąd programisty

Brak typu zwracanej wartości w definicji przed nazwą funkcji, podczas gdy funkcja zwraca wartość, jest **błędem składni**. Podobnie błędem jest zapomnienie zwrócenia wartości, gdy funkcja ma taką zwracać. Zwracanie wartości z funkcji, której zwracany typ został określony jako `void`, jest **błędem składni**.

Typowy błąd programisty

Ponowne definiowanie parametru (argumentu) funkcji jako zmiennej lokalnej w ciele funkcji jest **błędem składni**.

Typowy błąd programisty

Definiowanie funkcji wewnątrz innej funkcji jest **błędem składni**.

Typowy błąd programisty

Jeżeli prototyp funkcji, jej nagłówek w definicji oraz wywołania nie zgadzają się pod względem liczby, typów oraz porządku argumentów aktualnych i formalnych, a także typu zwracanego przez funkcję, jest to **błąd składni**.

Typowy błąd programisty

Brak średnika na końcu prototypu (deklaracji) funkcji jest **błędem składni**.

Typowy błąd programisty

Brak prototypu (deklaracji) funkcji, podczas gdy funkcja nie jest zdefiniowana przed jej pierwszym wywołaniem, jest **błędem składni**.

Wskazówka praktyczna
Programy powinny być tworzone jako kolekcje małych funkcji.

Wskazówka praktyczna
Prototyp funkcji nie jest wymagany, jeżeli definicja funkcji pojawia się przed pierwszym użyciem tej funkcji w programie. W tym przypadku definicja funkcji jest jednocześnie jej deklaracją.

Wskazówka praktyczna
Zmienne używane tylko w określonej funkcji powinny być raczej definiowane jako zmienne lokalne w tej funkcji, niż zmienne globalne.

Dobry styl programisty
Nie używaj takich samych nazw dla argumentów przekazywanych do funkcji i odpowiednich argumentów w jej definicji (chociaż nie jest to błędem). Dzięki temu unikniesz dwuznaczności.

Dobry styl programisty
Używaj nazw argumentów funkcji także w deklaracji funkcji, chociaż są one konieczne jedynie w jej definicji. Łatwiejsze jest zrozumienie programu przez jego samo czytanie.

Klasy pamięci

Atrybuty zmiennych obejmują *nazwę*, *typ*, *rozmiar* i *wartość*. Za pomocą tzw. specyfikatorów klas pamięci można określić ***klasę pamięci***, ***zasięg*** oraz ***połączenia*** zmiennych (obiektów).

Klasa pamięci (czas życia obiektu) obiektu (zmiennej) określa okres, podczas którego obiekt znajduje się w pamięci, tzn. od momentu, gdy zostaje on zdefiniowany (definicja przydziela mu miejsce w pamięci), do momentu, gdy przestaje on istnieć (miejsce w pamięci zajmowane przez obiekt zostaje zwolnione). Niektóre obiekty istnieją krótko, niektóre są wielokrotnie tworzone i niszczone, a inne trwają przez cały czas wykonywania programu.

Zasięg (zakres ważności nazwy obiektu) określa tę część programu, w której nazwa obiektu jest znana kompilatorowi i gdzie można się do obiektu odwoływać w programie. Do niektórych nazw można się odwoływać w całym programie, a do innych tylko z jego ograniczonych części.

Połączenia identyfikatorów (nazw obiektów) dla programów składających się z wielu plików źródłowych określają, czy nazwa zmiennej jest znana tylko w bieżącym pliku źródłowym, czy w dowolnym pliku źródłowym z prawidłowymi deklaracjami.

Jaka jest różnica między ***klasą pamięci*** obiektu a ***zasięgiem*** nazwy obiektu?
Różnica między powyższymi pojęciami jest taka, że w jakimś momencie obiekt może istnieć, ale nie być dostępny. To dlatego, że np. znajdujemy się chwilowo poza zakresem ważności jego nazwy.

Specyfikatory klas pamięci mogą być podzielone na dwie klasy pamięci:

- **automatyczną** (**auto**, **register**)
- **statyczną** (**extern**, **static**)

C++ zawiera cztery specyfikatory klas pamięci: **auto**, **register**, **extern**, **static**

auto – zmienne tego typu są tworzone w zakresie bloku, w którym są definiowane, istnieją, gdy blok jest aktywny i są niszczone, gdy następuje wyjście z bloku. Specyfikator klasy pamięci **auto** wyraźnie deklaruje zmienne **automatycznej klasy pamięci**. Zmienne lokalne są domyślnie elementami automatycznej klasy pamięci, więc słowo kluczowe **auto** jest zwykle pomijane.

Przykład: zwykle zmienne lokalne i parametry funkcji są elementami automatycznych klas pamięci.

```
auto float x;
```

register – specyfikator ten może być umieszczony przed deklaracją zmiennych automatycznych, sugerując, by kompilator utworzył zmienną raczej w jednym z rejestrów sprzętowych komputera o wysokim stopniu prędkości, niż w pamięci. W ten sposób koszty związane z wielokrotnym ładowaniem często używanych zmiennych z pamięci do rejestrów i składowanie wyników z powrotem w pamięci mogą być wyeliminowane. Kompilator może ignorować deklaracje zmiennych **register** (np. może brakować wystarczającej liczby rejestrów dostępnych do użycia przez kompilator). Słowo kluczowe **register** może być wykorzystane tylko ze zmiennymi lokalnymi i parametrami funkcji. Nie można uzyskać adresu zmiennej **register**, rejestry adresowane są inaczej niż pamięć.

Przykład: intensywnie używane zmienne, takiej liczniki lub sumy.

```
register int licznik=1;
```

Specyfikatory klas pamięci mogą być podzielone na dwie klasy pamięci:

- **automatyczną** (**auto**, **register**)
- **statyczną** (**extern**, **static**)

C++ zawiera cztery specyfikatory klas pamięci: **auto**, **register**, **extern**, **static**

extern – używane do deklarowania identyfikatorów dla zmiennych i funkcji **statycznej klasy pamięci**. Takie zmienne istnieją od momentu, w którym program rozpoczyna wykonywanie. Jest im przydzielana pamięć i inicjalizowana jednorazowo w trakcie uruchamiania programu. Jednak mimo, iż nazwy zmiennych czy funkcji tej klasy istnieją od początku wykonywania programu, nie oznacza to, że nazwy te są dostępne przez cały czas i w każdym miejscu programu, gdyż obowiązują je zwykłe zasady zasięgu. Globalne zmienne i funkcje oznaczają domyślnie specyfikator klasy pamięci **extern**. Zmienne globalne są tworzone przez umieszczenie deklaracji zmiennych poza definicją jakiejkolwiek funkcji. Do zmiennych tych może się odwoływać każda funkcja z pliku, która może odczytania ich deklarację lub definicję.

Przykład: globalne zmienne i globalne nazwy funkcji (czyli ogólnie identyfikatory zewnętrzne)

```
extern int a;
```

static – zmienne lokalne zadeklarowane za pomocą słowa kluczowego static są znane tylko w funkcji, w której zostały zdefiniowane (podobnie jak inne zmienne), ale w przeciwieństwie do zmiennych automatycznych, zachowują swoje wartości po wyjściu z funkcji. Przy ponownym wejściu do tej samej funkcji, zmienne te mają takie wartości, jakie miały przy ostatnim wyjściu z funkcji.

Przykład: globalne i lokalne zmienne i funkcje

```
static int ilosc_wywolan;
```

Reguły zasięgu

Reguły zasięgu

Zasięg pliku – identyfikator jest dostępny we wszystkich funkcjach od miejsca, w którym został zadeklarowany, aż do końca pliku.
Przykład: wszystkie zmienne globalne, definicje funkcji i jej prototypy umieszczone poza funkcją.

Zasięg funkcji – jedynymi identyfikatorami mającymi zasięg funkcji są etykiety. Etykiety mogą być używane gdziekolwiek w funkcji, w której się pojawiają, ale nie są dostępne spoza ciała tej funkcji
Przykład: etykiety w strukturach **`switch`** (jako etykiety **`case`**) i etykiety w wyrażeniach **`goto`**.

Zasięg prototypu funkcji – jedynymi identyfikatorami mającymi zasięg prototypu funkcji są identyfikatory użyte na jej liście parametrów. Prototypy funkcji nie wymagają nazw na liście parametrów – wymagane są tylko typy.

Zasięg bloku – identyfikatory zadeklarowane wewnątrz bloku. Zasięg bloku rozpoczyna się w miejscu deklaracji identyfikatora i kończy w momencie napotkania prawego nawiasu klamrowego }.
Przykład: zmienne lokalne w funkcji, parametry funkcji, dowolne zmienne w bloku {}, itp.

Zasięg klasy – omówiony zostanie podczas omawiania klas.

ZALECENIE: Staraj się unikać korzystania z takich samych nazw identyfikatorów (zmiennych) w programie!!! (problemy z „zasłanianiem” nazw)

Zasłanianie nazw, operator rozróżniania zasięgu ::

Przykład:

```
#include <iostream>

using namespace std;

int zmienna;                        //definicja zmiennej globalnej

main()
{
   int zmienna=1;                   //definicja zmiennej lokalnej
   cout<<zmienna<<endl;             //zmienna lokalna zasłania globalną
   {                                //otwarcie bloku lokalnego
      int zmienna=2;                //definicja zmiennej lokalnej

      cout<<zmienna<<endl;          //zmienna lokalna zasłania globalną
      cout<<::zmienna<<endl;        //tylko dla zasłoniętego obiektu globalnego
   }                                //zamknięcie bloku

   cout<<”Poza blokiem, ”<<zmienna<<endl;
}
```

Widok ekranu:

```
1
2
0
Poza blokiem, 1
```

Obszary nazw

Obszary nazw (relatywnie nowa cecha C++) są przeznaczone dla programistów do pomocy w rozwijaniu nowych elementów programu bez wywoływania konfliktów nazw z istniejącymi elementami oprogramowania.

Każdy **plik nagłówkowy** w projekcie standardu C++ używa **obszaru nazw** nazywanego `std`. Projektanci nie powinni używać obszaru `std` do definiowania nowych bibliotek klas.

Instrukcja **`using namespace`** `std` mówi nam, że używamy elementów oprogramowania z biblioteki standardowej C++.

Instrukcja **`using`** umożliwia nam używanie krótkich wersji każdej nazwy w bibliotece standardowej C++ lub jakimkolwiek, określonym przez programistę **obszarze nazw** **`namespace`**.

Jeżeli używamy dwóch lub więcej bibliotek klas, które mają opisy z identycznymi nazwami, może wystąpić konflikt nazw. Należy wtedy w pełni określić nazwę, jakiej chcemy użyć z jej obszarem nazw za pomocą **operatora rozróżniania zasięgu ::**.

Przykład:

```
#include <iostream>

namespace student1
{
   int zmienna=1;
   int zmienna1=1;
}
////////////////////////////////////////////////////////////////////////////////////
////////
namespace student2
{
   int zmienna=2;
   int zmienna2=2;
}

using namespace std;
using namespace student1;
using namespace student2;

main()
{

   cout<<student1::zmienna<<endl;
   cout<<student2::zmienna<<endl;

   cout<<zmienna1<<endl;
   cout<<zmienna2<<endl;
}
```

Typowy błąd programisty

Tylko jeden specyfikator klasy pamięci może być zastosowany do nazwy zmiennej. Na przykład jeśli dołączysz `register`, nie dołączaj już `auto` itp.

Typowy błąd programisty

Przypadkowe użycie takiej samej nazwy zmiennej w wewnętrznym i zewnętrznym bloku może powodować **błędy logiczne**.

Wskazówka praktyczna

Automatyczna pamięć jest przykładem zasady najmniejszego przywileju: dlaczego zmienne mają być przechowywane w pamięci jeśli nie są potrzebne?

Wskazówka dotycząca wydajności

Deklaracje `register` są zwykle niepotrzebne. Dzisiejsze optymalizujące kompilatory są zdolne rozpoznawać często używane zmienne i same mogą decydować o ich umieszczeniu w rejestrach.

Podsumowanie

PODSUMOWANIE 1:

- Tablica jest ciągłą grupą powiązanych komórek pamięci, które mają wspólną nazwę i są tego samego typu.
- Indeks elementu tablicy musi być liczbą całkowitą lub wyrażeniem całkowitym. Indeks w tablicy zaczyna się od 0.
- Chcąc zarezerwować 100 komórek pamięci dla tablicy liczb całkowitych, napisz `int t[100]`, a nie `int t[99]`.
- Jeśli jest mniej wartości inicjalizujących niż elementów tablicy, pozostałe elementy są inicjalizowane wartością zero.
- C++ nie zapobiega odwoływaniu się do elementów poza granicami tablic, co może być przyczyną poważnych błędów.
- Pojedyncze znaki napisu przechowywanego w tablicy są dostępne bezpośrednio przez użycie notacji indeksowej tablicy.
- Nazwa tablicy jest adresem jej pierwszego elementu.
- W tablicach dwuwymiarowych pierwszy indeks określa umownie wiersz, a drugi kolumnę.
- Najlepszym sposobem do rozwijania i utrzymywania dużego programu jest podzielenie go na mniejsze moduły, którymi w C++ są funkcje i klasy.
- Celem ukrycia informacji przez funkcję jest dostęp tylko do tych danych, których funkcja potrzebuje do wykonania swojego zadnia. Jest to tzw. zasada najmniejszego przywileju, charakteryzująca dobrą technikę programowania.
- Każdy argument funkcji może być stałą, zmienną lub wyrażeniem.
- Argumenty przekazywane do funkcji powinny mieć zgodną liczbę, typ i porządek z parametrami w jej definicji.
- Kompilator pomija nazwy zmiennych wymienionych w prototypie (deklaracji) funkcji.
- Pusta lista parametrów jest określona przez puste nawiasy lub umieszczony w nich typ `void`.

PODSUMOWANIE 2:

- Kiedy argument jest przekazywany do funkcji wywołaniem przez wartość, utworzona zostaje kopia wartości argumentu, a zmiany dokonywane na tej kopii w wywołanej funkcji nie zmieniają oryginalnej wartości zmiennej.

- Każdy identyfikator zmiennej ma atrybuty obejmujące klasę pamięci, zasięg i połączenia.

- C++ zawiera cztery specyfikatory klas pamięci: **`auto, register, extern, static`**.

- Klasa pamięci identyfikatora określa czas istnienia w pamięci identyfikatora.

- Zasięg identyfikatora dotyczy miejsc, z których można się odwoływać do identyfikatora w programie.

- Połączenia identyfikatorów, w przypadku programów składających się z wielu plików, określają, czy identyfikator jest znany tylko w bieżącym pliku źródłowym, czy w dowolnym pliku źródłowym z prawidłowymi deklaracjami.

- Zmienne automatycznej klasy pamięci są tworzone po wejściu do bloku, w którym są one zdefiniowane, istnieją dopóki blok jest aktywny i są niszczone po wyjściu z bloku.

- Zmienne statycznej klasy pamięci są przydzielane i inicjalizowane w trakcie uruchamiania programu.

- Dwa typy identyfikatorów mają statyczną klasę pamięci: identyfikatory zewnętrzne i zmienne lokalne deklarowane za pomocą specyfikatora klasy pamięci **`static`**.

- Zmienne globalne są tworzone przez umieszczenie ich deklaracji poza definicją jakiejkolwiek funkcji i zachowują swoje wartości przez cały czas wykonywania programu.

- Zmienne lokalne deklarowane za pomocą słowa kluczowego **`static`**, zachowują swoje wartości po wyjściu z funkcji, w której są zadeklarowane.

Ćwiczenia powtórzeniowe

1. Określ, które z poniższych zdań jest prawdziwe, a które fałszywe:
   a) tablica może przechowywać wiele różnych typów wartości (**FAŁSZ**)
   b) indeks tablic zwykle powinien być typu `float` (**FAŁSZ**)
   c) jeśli lista inicjalizująca zawiera więcej wartości niż tablica elementów, wystąpi błąd (**PRAWDA**)
   d) pojedynczy element przekazany do funkcji i zmodyfikowany w niej zachowa zmodyfikowaną wartość po zakończeniu wywołanej funkcji (**FAŁSZ**)

2. Podaj nagłówek definicji oraz prototyp (deklarację) dla każdej z poniższych funkcji:
   a) funkcja `przeciwprostokatna`, pobierająca dwa zmiennoprzecinkowe argumenty podwójnej precyzji, `bok1` i `bok2` oraz zwracająca zmiennoprzecinkowy wynik podwójnej precyzji

```
double przeciwprostokatna(double bok1, double bok2)
double przeciwprostokatna(double, double);
```

   b) funkcja `calkowitaNaZmiennoprzec` pobierająca argument całkowity `liczba` i zwracająca zmiennoprzecinkowy wynik

```
float calkowitaNaZmiennoprzec(int liczba)
float calkowitaNaZmiennoprzec(int);
```

   c) funkcja instrukcje nie pobierająca żadnych argumentów i nie zwracająca wartości

```
void instrukcje(void)      LUB    void instrukcje()
void instrukcje(void);     LUB    void instrukcje();
```

3. Napisz poniższe definicje:
   a) zmienna całkowita licznik zainicjalizowana wartością 0, która winna być umieszczona w rejestrze
   ODP: `register int licznik=0;`
   b) zmienna zmiennoprzecinkowa `ostatniaWart` zachowująca swoją wartość między wywołaniami funkcji, w której jest zdefiniowana
   ODP: `static float ostatniaWart;`

Ćwiczenia sprawdzające

1. Napisz pojedyncze instrukcje wykonujące następujące operacje na jednowymiarowej tablicy:
   a) zainicjalizuj 10 elementów tablicy liczb całkowitych `licznik` wartością zero
   b) dodaj 1 do każdego z 15 elementów tablicy liczb całkowitych `premia`
   c) wczytaj z klawiatury 12 wartości do tablicy liczb zmiennoprzecinkowych `miesieczneTemp`
   d) wyświetl 5 ostatnich wartości 10-elementowej tablicy liczb całkowitych `najlepszeWyniki`

2. Napisz program do rozwiązania następującego problemu z użyciem tablicy jednowymiarowej: Wczytaj 20 liczb z zakresu od 10 do 100 włącznie, a następnie wyświetl zdublowane i nie zdublowane liczby w tej tablicy.

3. Na parkingu pobierana jest minimalna opłata 5PLN za parkowanie do trzech godzin. Powyżej tych godzin opłata za godzinę lub jej część wynosi 1PLN. Maksymalna opłata za dowolny okres 24h wynosi 20PLN. Napisz program, który obliczy i wyświetli opłaty za parkowanie dla klientów parkingu, dla których wprowadzasz godziny postoju. Program powinien wykorzystać funkcję `obliczOplate` do określenia opłaty każdego z klientów.

4. Napisz funkcję `wielokrotnosc` określającą dla pary liczb całkowitych, czy druga liczba jest wielokrotnością pierwszej. Funkcja powinna pobierać dwa argumenty całkowite i zwracać **`true`**, jeśli drugi jest wielokrotnością pierwszego, lub **`false`** w przeciwnym wypadku.

5. Napisz segmenty programu realizujące następujące zadania:
   a) obliczanie całkowitej części ilorazu, kiedy liczba całkowita a jest dzielona przez liczbę całkowitą b
   b) obliczanie całkowitej reszty z dzielenia liczby całkowitej a przez liczbę całkowitą b
   c) użyj fragmentów programu opracowanych w a) i b) do napisania funkcji pobierającej czterocyfrową liczbę całkowitą i wyświetlającą ją jako serię cyfr, z których każda oddzielona jest dwiema spacjami. Na przykład liczbę całkowitą 7345 wyświetl jako 7 3 4 5.

Następny wykład

Wykład nr 3

Temat: Wskaźniki i referencje.