Wskaźniki

- Wskaźnik to zmienna, zawierająca adres komórki w pamięci. Wskaźniki w języku C++ mają swoje typy, typem wskaźnika jest typ obiektu, który mieści się w pamięci począwszy od wskazywanej przez wskaźnik komórki.
- Jeżeli więc wskaźnik ma typ `int` to program wie, w jaki sposób odczytać wskazywaną przez wskaźnik zawartość. Należy odczytać te powiedzmy 4 kolejne bajty, i odkodować je jako zmienną `int`.
- Wskaźnik jest zmienną, i niezależnie od swojego typu ma określoną wielkość. To jest jasne, wskaźnik zawiera adres komórki pamięci, i format takiego adresu zależy tylko od ilości pamięci przydzielonej programowi w systemie.
- Do definicji wskaźników używamy operatora wskazywania `*`. Nazwa wskaźnika może być dowolna, tak jak nazwa każdej innej zmiennej, ale często do nazw wskaźników dodaje się człon `_ptr`, żeby podkreślić, że zmienna jest wskaźnikiem.

```
int *x_ptr;
```

Wskaźniki

- Zadeklarowaliśmy w ten sposób wskaźnik typu `int`, o nazwie `x_ptr`. Tak zdefiniowany wskaźnik może pokazywać zupełnie przypadkową komórkę w pamięci, ponieważ nie został zainicjalizowany (chyba, że jest zmienną globalną, wtedy jest automatycznie zainicjalizowany na 0).
- Podkreślmy, nazwa wskaźnika to `x_ptr`. Gwiazdka nie jest częścią nazwy, tylko informacją, że ma to być wskaźnik. Niektórzy programiści definiują wskaźniki następująco:

```
int* x_ptr;
```

  czyli gwiazdka stoi przy typie zmiennej, a nie przy jej nazwie. To podkreśla, że zmienna ma nazwę `x_ptr` i jest typu wskaźnik na `int`. Położenie gwiazdki w powyższej definicji nie ma znaczenia dla kompilatora.

Wskaźniki

- Wskaźniki powinno się inicjalizować od razu przy definicji:

```
int jakas_zmienna;
int *x_ptr = &jakas_zmienna;
```

- Najpierw utworzyliśmy w pamięci zmienną `jakas_zmienna`, a następnie jej adres (otrzymany przy pomocy znanego nam już operatora adresu &) podstawiliśmy do zmiennej `x_ptr`. Od tego momentu możemy używać zmiennej `jakas_zmienna` przy użyciu wskaźnika i operatora wskazywania:

```
inna_zmienna = (*x_ptr) * 2;
*x_ptr = 15;
```

- Operator wskazywania (czyli gwiazdka) ma wysoki priorytet, wyższy niż operacja arytmetyczne, więc w przypisaniu

```
inna_zmienna = (*x_ptr) * 2;
```

nawias nie jest formalnie potrzebny. Ale często się go stosuje dla pewności i czytelności.

Wskaźniki

- Zauważmy, że ten sam symbol * w powyższym podstawieniu występuje w dwóch zupełnie różnych rolach. Raz jest to mnożenie dwóch zmiennych, a w drugim przypadku jest operatorem wskazywania.
- Taka sytuacja ilustruje dostępny w C++ mechanizm *przeładowania* funkcji lub operatora. Tak samo nazywający się operator lub funkcja ma zupełnie inną definicję, w zależności od kontekstu w którym są użyte. Wspominaliśmy już o tym w przeszłości, i wkrótce wrócimy do tego tematu.
- Podkreślmy jeszcze raz: `x_ptr` to zmienna zawierająca adres komórki w pamięci.
- Natomiast `*x_ptr` to zmienna typu `int`, zapisana pod adresem `x_ptr`. W powyższym przypadku zmienna ta ma też swoją nazwę: `jakas_zmienna`.

Wskaźniki

- Mówiliśmy już, że wskaźniki nie powinny pozostawać niezainicjalizowane, czyli wskazujące na przypadkowy adres. Jeżeli musimy zdefiniować wskaźnik, ale nie mamy jeszcze dla niego gotowego adresu, powinniśmy zainicjalizować go przy pomocy specjalnego wskaźnika `nullptr`:

```
int *x_ptr = nullptr;
```

- Jeżeli przez pomyłkę będziemy chcieli coś zapisać do komórki wskazywanej przez taki wskaźnik, lub coś z niej odczytać, kompilator zgłosi błąd.
- Pozostawianie niezainicjalizowanych wskaźników jest uważane za błąd programisty, chociaż nie zakazuje tego standard C++. Trzeba pamiętać, że przypadkowe zapisanie czegoś pod adres wskazywany przez niezainicjalizowany wskaźnik z reguły *wysypie* program, a błąd taki będzie trudny do znalezienia. Przy każdym wywołaniu program może zachowywać się inaczej.

Wskaźniki

- `nullptr` to zmienna typu wskaźnikowego, o wartości 0. Można też stosować po prostu wartość 0, albo zmienną o nazwie `NULL`. Stosowanie `nullptr` jest zgodne ze standardem C++, i ma pewne subtelne korzyści. Ale zupełnie poprawnie możemy też napisać

```
int *x_ptr = NULL;
int *x_ptr = 0;
int *x_ptr = 0x0;
```

  Jest to pozostałość z języka C.
- Wskaźniki używane są w różnych sytuacjach, o których jeszcze wspomnimy, ale najważniejszym zastosowaniem są tak zwane zmienne dynamiczne.
- Mówiąc do tej pory o zmiennych powiedzieliśmy, że zmienne mogą być statyczne lub automatyczne.

Wskaźniki

- Statyczne to zmienne globalne i lokalne zadeklarowane jako `static`. Te zmienne mają ustalone miejsce w pamięci, zaraz po wszystkich funkcjach. Lokalizacja tych zmiennych nie zmienia się w trakcje działania programu, i ich ilość musi być ustalona w momencie kompilacji. Są automatycznie inicjalizowane jako 0, lub wartością ustaloną w programie. W pamięci komputera mieszczą się, wraz z funkcjami, na samym początku. Mają swoje nazwy, po których można się do nich odnosić.
- Zmienne automatyczne to zmienne lokalne, zadeklarowane na przykład wewnątrz funkcji, które tworzone są na stosie, po wywołaniu funkcji, i są usuwane po wyjściu z funkcji. Ich lokalizacja może mieć przypadkowy adres, nie podlegają domyślnej inicjalizacji przez 0, i podobnie jak zmienne statyczne mają swoje nazwy.

Wskaźniki

- Pomiędzy końcem pamięci zajętej przez funkcje i zmienne statyczne a „spodem” stosu rozciąga się obszar pamięci zwany stertą. Program może tworzyć tam zmienne, tak zwane zmienne dynamiczne. Ilość tak utworzonych zmiennych jest ograniczona tylko wielkością sterty. Tak utworzone zmienne dynamiczne nie mają swoich nazw, i można się odnosić do nich wyłącznie przy użyciu wskaźników.
- Do tworzenia zmiennych dynamicznych służy operator `new`. Jako swój argument przyjmuje typ zmiennej, a zwraca wskaźnik do nowo utworzonej zmiennej:

```
int *x_ptr;
x_ptr = new int;
```

- W powyższym przykładzie utworzony wskaźnik `x_ptr` jest pozostawiony niezainicjalizowany, ale tylko na moment, inicjalizacja następuje już w następnej linijce. W takiej sytuacji jest to dopuszczalne. Możemy jednak zrobić to od razu:

```
int *x_ptr = new int;
```

Wskaźniki

- Obiekty (zmienne) dynamiczne utworzone przez operator `new` istnieją aż do ich usunięcia specjalnym operatorem `delete`. Operator ten przyjmuje jako argument wskaźnik i zwalnia pamięć zajmowaną przez wskazywaną zmienna. Typ wskaźnika znowu jest ważny - operator dzięki niemu wie, ile bajtów pamięci zwolnić. Sam Operator `delete` nic nie zwraca. Na przykład:

```
int *x_ptr;
x_ptr = new int;
*x_ptr = 15;
cout << *x_ptr << endl;
delete x_ptr;
x_ptr = new int;
*x_ptr = 5;
cout << *x_ptr << endl;
delete x_ptr;
```

- Utworzyliśmy zmienną `x_ptr`, która jest wskaźnikiem typu `int`. Jest to zmienna statyczna lub lokalna, i jej zasięg jest regulowany zwykłymi zasadami.

Wskaźniki

- Następnie tworzymy zmienną dynamiczną typu `int` (nie ma ona nazwy), i wpisujemy do niej wartość 15. Ta wartość rzeczywiście tam jest, o czym przekonuje nas `cout`. Następnie usuwamy tą zmienną dynamiczną.
- Oczywiście, sam wskaźnik pozostaje, i w dalszym ciągu wskazuje na tą samą komórkę pamięci. Ale pod tym adresem nie ma już zmiennej. Ten obszar pamięci może zaraz zostać przydzielony innej zmiennej zupełnie innego typu. Może zostać nadpisany czymkolwiek w każdej chwili.
- W kolejnej linijce tworzymy nową zmienną dynamiczną typu `int`, i jej adres przypisujemy wskaźnikowi `x_ptr`. Oczywiście, to może być już zupełnie inny adres. Do tej nowej zmiennej wpisujemy wartość 5, i sprawdzamy. W końcu usuwamy też tą następną zmienną dynamiczną.

Wskaźniki

- Wskaźnik `x_ptr` nadal istnieje, ale nie wskazuje już na nic konkretnego. Powinniśmy wpisać do niego `nullptr`.
- Jeszcze taka uwaga. Możemy mieć wiele wskaźników wskazujących na tą samą zmienną. Nie ma w tym niczego dziwnego, i wielokrotnie będziemy się z tym spotykać. Wiele struktur danych i wiele operacji na nich będzie tego wymagać. W tej sytuacji usunięcie zmiennej operatorem `delete` pozostawi wiele „pustych” wskaźników. Powinniśmy pilnować wszystkich.
- Podkreślmy: operator `delete` nie nadpisuje ani nie kasuje niczego w pamięci. Jedynie informuje system operacyjny, że pod danym adresem niczego już nie ma, i można go wykorzystać ponownie.

Wskaźniki

- W języku C zarządzanie zmiennymi dynamicznymi służyły funkcje `malloc()` i `free()`, i takie konstrukcje wciąż są dopuszczalne w C++. Ale nie powinniśmy ich stosować w nowych programach.
- Jeżeli utworzymy zmienną dynamiczną i nie usuniemy jej przy pomocy operatora `delete` zmienna zostanie usunięta przy zakończeniu działania programu.
- Dostęp do zmiennej dynamicznej jest możliwy wyłącznie przez wskaźnik. Jeżeli utracimy wskaźnik, zmienna stanie się trwale niedostępna. W dalszym ciągu zajmuje pamięć, ale nie może być już przez program ani zapisana ani odczytana.
- Wskaźnik do zmiennej dynamicznej jest najczęściej zmienną lokalną, na przykład zdefiniowaną w funkcji. Po wyjściu z funkcji przestaje istnieć. Jeżeli jest jedynym wskaźnikiem wskazującym na daną zmienną, zmienna zostanie utracona.

Wskaźniki

- Taka utrata zmiennej nazywa się czasem „wyciekiem pamięci”. Istniejąca bezużyteczna zmienna zajmuje miejsce. Może zabraknąć miejsca dla następnych zmiennych dynamicznych. Trzeba na to uważać.
- Oczywiście wszelki wyciek pamięci zakończy się wraz z zakończeniem działania programu, ale nie powinno się do niego dopuszczać.
- Jeżeli na stercie nie ma wystarczająco dużo pamięci, żeby stworzyć nową zmienną dynamiczną, operator `new` zwróci pusty wskaźnik `nullptr`. Program zgłosi też tak zwany wyjątek (błąd), dzięki któremu można spróbować rozwiązać problem. Jeżeli nie obsłużymy takiego wyjątku, program się „wysypie”.

Wskaźniki

- Możemy zapobiec zgłaszaniu takiego wyjątku używając parametru `nothrow` operatora `new`. Wtedy wyczerpanie sterty możemy zauważyć sprawdzając, czy zwrócony wskaźnik jest różny od `nullptr`:

```
int main()
{
    long ilosc = 0;
    do
    {
        zmienna_ptr = new (nothrow) double;
        ilosc++;
    }
    while( zmienna_ptr )
    cout << "Sterta␣wyczerpana␣przy␣" << ilosc;
    cout << "zmiennych␣double" << endl;
    return 0;
}
```

- Zwrócona ilość da nam pojęcie o rozmiarze sterty.

Wskaźniki

- Wskaźniki mogą być stałe, tak jak każda inna zmienna. Jeżeli wiemy, że wskaźnika nie będziemy zmieniać, zadeklarujmy go jako `const`. Wtedy ewentualny błąd polegający na próbie nadpisania takiego wskaźnika zgłosi kompilator:

```
int licznik;
int * const licznik_ptr = &licznik;
```

- Wskaźnik `const` można zainicjalizować tylko podczas definicji. W rozdziale o tablicach mówiliśmy, że nazwa tablicy jest właśnie wskaźnikiem typu `const`. To jest zupełnie logiczne.
- Wskaźnik może też pokazywać na zmienną typu `const`:

```
const int *licznik_ptr;
```

Wskaźniki

- Taki wskaźnik możemy zmieniać, ale nie możemy zmieniać zmiennej wskazywanej przez niego. Na przykład, w poniższym przykładzie kompilator zgłosi błąd:

```
int licznik = 0;
const int *licznik_ptr;
*licznik_ptr = 5;
```

- Zauważmy jeszcze, że sama zmienna `licznik` nie jest zadeklarowana jako `const`, więc możemy ją zmieniać w trakcie programu. Nie możemy jednak tego robić przy pomocy wskaźnika `licznik_ptr`.
- Oba kwalifikatory mogą występować jednocześnie, czyli jak najbardziej możemy zdefiniować stały wskaźnik do stałej zmiennej (stała zmienna - to zabawnie brzmi, ale jest chyba poprawne?).

Wskaźniki

- Przy pomocy operatora `new` możemy tworzyć tablice:

```
int *licznik_ptr = nullptr;
int rozmiar;
cin >> rozmiar;
licznik_ptr = new int[rozmiar];
```

- Tak powstała tablica, w odróżnieniu od poprzednio omawianych tablic, nazywa się *dynamiczna*. Dla takiej tablicy rozmiar musi być znany dopiero w momencie wykonywania programu, a nie w momencie kompilacji, jak to jest w przypadku tablic statycznych.
- Tak utworzoną tablicę dynamiczną usuwamy w nastepujący sposób:

```
delete [] licznik_ptr;
```

Wskaźniki

- Pracując ze wskaźnikami w zasadzie nigdy nie ustawiamy ich wartości bezpośrednio, to znaczy wpisując jakiś konkretny adres. Często wykorzystujemy adres istniejącej zmiennej:

```
int licznik, *licznik_ptr;
licznik_ptr = &licznik;
```

- Możemy też wykorzystać istniejący wskaźnik:

```
int licznik, *licznik_ptr, *temp_ptr;
licznik_ptr = &licznik;
temp_ptr = licznik_ptr;
```

- Inną typową metodą nadawania wartości wskaźnikowi jest przy pomocy operatora `new`:

```
int *licznik_ptr;
licznik_ptr = new int;
```

Wskaźniki

- Może się pojawić następująca wątpliwość. Mówiliśmy, że zmienne dynamiczne, do których dostęp jest poprzez wskaźniki rozwiązują problem, jeżeli w momencie kompilacji nie wiemy ilu zmiennych będziemy potrzebować. Ale przecież do każdej zmiennej potrzebujemy wskazujący na nią wskaźnik. Skoro wskaźniki deklarowaliśmy jako zmienne globalne lub lokalne, to *ich* ilość musi być znana w momencie kompilacji.
- Jeżeli więc ilość wskaźników musi być znana w momencie kompilacji, to ilość utworzonych zmiennych dynamicznych chyba też? Możemy tworzyć tyle zmiennych dynamicznych ile nam się podoba w czasie wykonywania programu, ale przecież żeby takie zmienne były użyteczne, do każdej potrzebujemy osobny wskaźnik.

Wskaźniki

- W rzeczywistości obiekty tworzone dynamicznie przy pomocy operatora `new` z reguły są bardziej rozbudowane niż typy fundamentalne. Take obiekty, które zawierają w sobie więcej niż tylko jeden element typu fundamentalnego nazywamy strukturami. Typowo taka struktura zawiera jakąś ilość danych, oraz wskaźnik do takiej samej struktury. Wtedy tworząc taki obiekt dynamiczny tworzymy jednocześnie „miejsce" na kolejny.
- Które z obiektów zawierają wskaźniki na które, to struktura zmiennych dynamicznych. Zmienne mogą tworzyć listę, gdzie każdy element zawiera wskaźnik na jeden kolejny. Żeby zachować dostęp do takiej listy, potrzebujemy tylko jednego wskaźnika: tego, który pokazuje na początek listy. Zmienne dynamiczne mogą w pamięci tworzyć strukturę drzewa, gdzie jedynym wskaźnikiem staycznym/automatycznym jest wskaźnik na korzeń drzewa. Następnie każdy węzeł zawiera wskaźniki na potomków, oraz wskaźnik na przodka.

Wskaźniki

- Na przykład, jeżeli chcemy utworzyć listę osób, z reguły w momencie pisania programu nie wiemy, ile tych osób będzie. Tworzymy więc taką listę ze zmiennych dynamicznych. Najpierw deklarujemy odowiednią stukturę, na przykład:

```
struct osoba
{
  char nazwisko[40];
  char imie[20];
  osoba *nastepny;
}
```

- Zauważmy, że wewnątrz struktury znajduje się wskaźnik na właśnie deklarowaną strukturę. C++ pozwala na taką składnię, nie stanowi to zresztą problemu, bo wskaźniki mają zawsze ten sam rozmiar, niezależnie od typu.

Wskaźniki

- Mając tak zadeklarowaną strukturę możemy tworzyć zmienne dynamiczne, powiązane wskaźnikami w listę

```
osoba *lista_ptr = nullptr; //to jest wskaźnik na początek listy
lista_ptr = new osoba;
(*lista_ptr).nastepny = new osoba;
```

- W ten sposób mamy na stercie dwie zmienne. Wskaźnik `lista_ptr` wskazuje na jedną z nich, a wskaźnik `(*lista_ptr).nastepny` pokazuje na następną.
- Zauważmy, że do poszczególnych *pól* struktury odnosimy się używając kropki.

Wskaźniki

- Tak utworzone zmienne dynamiczne powinniśmy usunąć przed zakończeniem programu. Możemy to zrobić przy pomocy funkcji zdefiniowanej rekurencyjnie:

```
void usun( osoba *osoba_ptr )
{
 if( (*osoba_ptr).nastepny ) usun( (*osoba_ptr).nastepny);
 delete osoba_ptr;
 return;
}
```

- Jeżeli `lista_ptr` jest wskaźnikiem na początek listy, to wystarczy wywołać funkcję `usun()` z tym wskaźnikiem.

```
usun( lista_ptr );
```

- Funkcja będzie się wywoływała kolejno ze wskaźnikami pokazującymi na kolejne elementy, aż dojdzie do końcowego. Potem będzie kolejno w odwrotnej koljności kasować elementy listy aż do pierwszego.

Wskaźniki

- Oprócz struktur wkrótce poznamy tak zwane *obiekty*, które oprócz zmiennych zawierają także różnego rodzaju funkcje. Obiekty i struktury danych to będzie nasz kolejny obszar zainteresowania.
- Wspomnijmy jeszcze o roli wskaźników w przekazywaniu argumentów i odbieraniu wartości od funkcji. Wiemy, że argumenty można przekazywać do funkcji przez wartość (funkcja otrzymuje kopię istniejącej zmiennej, a do oryginalnej zmiennej nie ma dostępu), albo przez referencję (funkcja otrzymuje adres istniejącej zmiennej, i może ją zmieniać).
- Pamiętamy składnię obu rodzajów przekazywania:

```
void jakasFunkcja1( int ilosc, int &cena );
```

- Funkcja przyjmuje 2 argumenty, `ilosc` jest przekazywany przez wartość, a `cena` jest przekazywany przez referencję

Wskaźniki

- Mając gdzieś zadeklarowane zmienne `m,n` możemy wywołać funkcję:

```
jakasFunkcja1( m, n );
```

- W przypadku zmiennej `m` funkcja operuje na jej kopii, która przestanie istnieć po wyjściu z funkcji, a oryginalna zmienna pozostanie niezmieniona. Natomiast w przypadku zmiennej `n` funkcja otrzymuje adres tej zmiennej, i cokolwiek z nią zrobi to zrobi.
- Nietrudno się domyśleć, że przekazywanie zmiennej przez referencję to tylko sposób zapisu przekazywania do funkcji wskaźnika. Tą samą funkcję moglibyśmy zadeklarować następująco:

```
void jakasFunkcja2( int ilosc, int *cena );
```

Wskaźniki

- Wywołanie takiej funkcji wyglądałoby następująco:

```
int m, n;
int *n_ptr = &n;
...
jakasFunkcja2( m, n_ptr );
```

albo

```
int m, n;
...
jakasFunkcja2( m, &n );
```

- Efekt jest dokładnie taki sam. Najczęściej posługujemy się jawnym wskaźnikiem, bo w większości przypadków używając wskaźników jawnie łatwiej jest uniknąć błędów. Zauważmy, że przy wywołaniu funkcji `jakasFunkcja1()` fakt, że przesyłanie jest przez referencję jest ukryty. Czasem chcemy to ukryć, bo jest nieistotne, czasem nie.

Wskaźniki

- Pamiętamy, że jeżeli z jakiegoś powodu przekazywaliśmy argument przez referencję (na przykład, kiedy mieliśmy do czynienia z dużą tablicą) ale nie chcieliśmy, żeby funkcja cokolwiek w oryginalnej zmiennej zmieniała, dodawaliśmy kwalifikator `const`:

```
void jakasFunkcja( int ilosc, const int &cena );
```

- Używając wskaźnika, ten sam efekt otrzymujemy odpowiednio typ wskaźnika będącego argumentem formalnym:

```
void jakasFunkcja( int ilosc, const int *cena );
```

- Oczywiście funkcja może zwracać wskaźnik:

```
int *jakasFunkcja( int ilosc, const int *cena );
```

Wskaźniki

- Wspomnijmy jeszcze o jednej rzeczy. Funkcja `main()` też może przyjmować argumenty. Bardzo często program wywołuje się wraz z parametrami, które wypisuje się kolejno po nazwie programu. To są tak zwane argumenty wiersza poleceń. System operacyjny odczytuje te argumenty i przekazuje je do programu. W przypadku programu napisanego w języku C++ przekazuje je konkretnie do funkcji `main()`
- Żeby można było wykorzystać te argumenty, w definicji funkcji `main()` musimy dodać parametry formalne:

```
int main( int argc, char *argv[] )
```

- Wewnątrz `main()` `argc` to jest ilość przekazanych argumentów wiersza poleceń. Natomiast `argv[]` to tablica kolejnych argumentów. Każdy z nich jest C-stringiem, czyli jest tablicą znaków, innymi słowy ma typ `char *`. Rozmiar tej tablicy to oczywiście `argc`.

Wskaźniki

- Możemy odczytać kolejne parametry:

```
char *pierwszyParametr = argv[0];
...
```

  aż do `argv[argc-1]`.

- Dodawanie argumentów wiersza poleceń można łatwo zaaranżować w Codeblokach. W menu `Project` znajduje się pozycja `Set programs' arguments...` gdzie możemy wypisać kolejne argumenty. CodeBlocks doda te argumenty przy uruchomieniu kompilowanego programu

Zadanie domowe 5 (do oddania 24.05.)

Napisz program, który wczyta 3 liczby `double` (dodatnie) podane przez użytkownika, ale utworzy je jako dynamiczne. Następnie obliczy średnie: arytmetyczną oraz harmoniczną, i wypisze je użytkownikowi. Wszystkie zmienne (oprócz wskaźników) powinny być dynamiczne, utworzone na stercie.
Definicje średnich można znaleźć w googlu.