Lekcja 12
Wskaźniki. Dynamiczne struktury danych

wskaźniki – deklarowanie - przydzielanie i zwalnianie pamięci – lista - drzewo

Wskaźnik, to zmienna, która przechowuje adres (innej zmiennej).

Przykład. W zmiennej X przechowujemy liczbę 10. A my chcemy dodatkowo wiedzieć, w którym miejscu pamięci operacyjnej komputera, to 10 w zmiennej X się znajduje. Po co to wszystko? Gdy deklarujemy jakieś zmienne, to przydzielamy im konkretną ilość i miejsce w pamięci operacyjnej. Czasem jednak nie wiemy ile tej pamięci będzie nam potrzebne – na przykład: jaką wielką tablicę trzeba zadeklarować. Możemy więc deklarować nie tablicę statyczną (o konkretnej liczbie komórek) lecz dynamiczną, której wielkość ustalimy w trakcie działania programu, np. gdy użytkownik ją wpisze z klawiatury. W ten sposób algorytmy są być może bardziej skomplikowane, ale wykonują się szybciej i czasem jest to jedyna metoda by poradzić sobie z dużą ilością danych.

Deklarowanie i przypisywanie

Zmienna *liczba* jest typu *int*
Zmienna *adres* jest wskaźnikiem typu *int* – 4 bajty w pamięci
W zmiennej *liczba* znajduje się wartość *97*
W zmiennej *adres* znajduje się adres pamięci komputera, w którym przechowywana jest wartość 97 w zmiennej *liczba*
Na ekranie wypisujemy w kolejności:

liczba	zawartość zmiennej *liczba*
&liczba	adres zmiennej *liczba*
adres	zawartość wskaźnika *adres*
**adres*	zawartość komórki pamięci o adresie *adres*
*char(*adres)*	rzutowanie na inny typ – wypisany kod ASCII liczby 97

```
int liczba;
int *adres;
liczba=97;
adres=&liczba;
cout << liczba << endl;
cout << &liczba << endl;
cout << adres << endl;
cout << *adres << endl;
cout<<char(*adres)<<endl;
```

```
97
0x6bfef8
0x6bfef8
97
a
```

Tworzenie zmiennej dynamicznej

Typowa zmienna w momencie zadeklarowania zajmuje miejsce w pamięci operacyjnej i istnieje do momentu zakończenia działania bloku programu. Pamięć dla zmiennej dynamicznej jest przydzielana dopiero po wykonaniu instrukcji NEW i zwalniana jest po wykonaniu instrukcji DELETE. To programista decyduje kiedy zajmiemy i zwolnimy miejsce w pamięci.

Deklaracja zmiennej dynamicznej (wskaźnika) **ZmDyn*
Przydzielamy obszar w pamięci komputera
Do wskaźnika przypisujemy wartość 7
Na ekranie wypisujemy zawartość wskaźnika
Na ekranie wypisujemy adres zmiennej dynamicznej
Zwalniamy pamięć !!! bardzo ważne

```
int *ZmDyn;
ZmDyn=new int;
*ZmDyn=7;
cout << *ZmDyn << endl;
cout << ZmDyn << endl;
delete ZmDyn;
```

```
7
0x836218
```

Tablice i zmienne dynamiczne

Jeśli nie znamy wielkości tablicy albo tablica jest bardzo duża konieczne jest dynamiczne przydzielenie dla niej pamięci. W dawnych czasach, gdy pamięć komputerów liczona była w kilobajtach było to niezbędne. Z tego też powodu twórcy języka C zdecydowali, że tablice przekazywane jako parametry funkcji będą wskaźnikami! – nie można tablic przekazać inaczej.

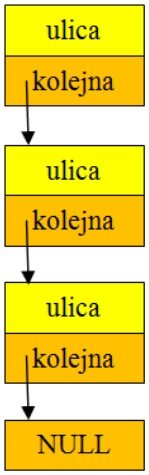
```
ile komórek: 10
  6  0x792270
  3  0x792274
  3  0x792278
  5  0x79227c
  4  0x792280
  6  0x792284
  3  0x792288
  1  0x79228c
  2  0x792290
  4  0x792294
```

```
cout << "ile komórek: ";
int kom;
cin >> kom;
int *Tliczby=new int [kom];
for (int i=0;i<kom;i++) {
  Tliczby[i]=rand() % 6 + 1;
  cout.width(3);
  cout<<Tliczby[i]<<"  "
      <<&Tliczby[i]<<endl;
}
delete [] Tliczby;
```

Do tablicy dynamicznej *Tliczby*, której wielkość ustalamy w trakcie działania programu wpisujemy losowe liczby. Każda z liczb zajmuje 4 bajty. Na końcu zwolnienie pamięci tablicy.

Lista jednokierunkowa

Prócz tablic, w których poszczególne elementy przechowywane są w ponumerowanych komórkach, w algorytmice mamy do czynienia z innymi strukturami do przechowywania dużej liczby elementów. Najpopularniejsze z nich to: **listy, stosy, kolejki, drzewa, grafy i sieci**. Ze względu na swoją specyfikę struktury te operują na wskaźnikach. Strukturę i podstawowe operacje na listach możemy definiować samodzielnie, ale istnieją też gotowe biblioteki.



Lista jest strukturą podobną do tablicy, elementy nie są jednak ponumerowane, ale każdy elementy listy składa się z dwóch części: pierwszy przechowuje dane, drugi wskazuje na kolejny element listy. Do takiej listy można dołączać nowe elementy lub usuwać. Przeglądać listę można jedynie od początku do końca – podobnie jak w pliku na dysku. Pewną modyfikacją takie listy jednokierunkowej jest lista dwukierunkowa – każdy element listy składa się z trzech części: dane, wskaźnik na następny i wskaźnik na poprzedni element.

Definiowanie elementu listy

Deklaracja listy powinna być umieszczona przed fragmentami programu, które z listy korzystają. W tym przypadku może być wewnątrz głównego programu. Deklaracja identyczna, jak w typowej strukturze, ale jednym z elementów jest wskaźnik na kolejny element tej listy.

Inicjujemy listę tworząc wskaźnik i przypisując mu wartość NULL.

W pętli wczytujemy z klawiatury napisy i tworzymy kolejne elementy listy (inicjując wskaźnik pomocniczy). Ponieważ lista jest wskaźnikiem, dlatego przypisanie wygląda odmiennie – za pomocą operatora „->”.

Kolejnemu wskaźnikowi przypisujemy adres elementu listy, który istnieje, tzw. „głowa listy”.

Przesuwanie po kolejnych elementach listy za pomocą pętli – ostatni element listy ma ustawiony adres NULL, dlatego pętla zakończy się na tym elemencie listy.

Po wypisaniu elementów listy na ekranie, głową listy staje się następny element, po czym zwalniamy pamięć wypisanego na ekranie elementu.

Strzałkami na rysunku zaznaczono adresy kolejnych elementów listy i adresy elementów kolejnych.

W profesjonalnych zastosowaniach tworzone są funkcje, które wykonują powyższe operacje związane z listą. Słowem PUSH określa się dodawanie elementu do listy, słowem POP – zdejmowanie z listy i usuwanie z pamięci.

```
//deklaracja listy
struct LISTAulica{
  LISTAulica *kolejna;
  string ulica;
};
//inicjowanie listy
LISTAulica *LU = NULL;
string ul;
//wpisujeju ulice do listy
do{
  getline(cin,ul);
  if (ul!=""){
    LISTAulica *U = new LISTAulica;
    U->ulica=ul;
    U->kolejna=LU;
    LU=U;
    cout<<LU<<"  "<<LU->kolejna<<endl;
  }
} while (ul!="");
//ulice na ekran
//i usuwamy elementy listy z pamięci
while (LU){
  cout<<LU<<"  "<<LU->kolejna
      <<"  "<<LU->ulica<<endl;
  LU=LU->kolejna;
  delete LU;
}
```

```
Dlugoszowskich 1
0xc86228  0
Rynek 12/3
0xc86238  0xc86228
Grunwaldzka 72
0xc86248  0xc86238

0xc86248  0xc86238  Grunwaldzka 72
0xc86238  0xc86228  Rynek 12/3
0xc86228  0  Dlugoszowskich 1
```

Dołączanie elementu na początek listy

Parametrami funkcji PUSH są: „głowa” listy (element na wierzchołku listy) oraz napis do umieszczenia w nowym elemencie listy.

Tworzony jest nowy element listy, wpisywana jest nazwa ulicy, a element kolejny wskazuje na następny (pod nim) element listy.

```
void PUSH(LISTAulica *&head, string u){
  LISTAulica *U = new LISTAulica;
  U->ulica = u;
  U->kolejna = head;
  head = U;
}
```

Nową głową listy staje się wskaźnik na właśnie utworzony element – jest zwracany poprzez parametr funkcji – tzw. referencja.

Usuwanie początkowego elementu listy

Parametrem jest głowa listy – ten element zostanie usunięty z pamięci. Tworzymy tymczasowy wskaźnik na głowę listy.

Jeżeli istnieje (nie jest równy NULL), to nową głową staje się wskaźnik na kolejny element listy i zwalniamy poprzednią głowę listy.

```
void POP(LISTAulica *&head){
  LISTAulica *U;
  U = head;
  if(U){
    head = U->kolejna;
    delete U;
  }
}
```

Zliczanie elementów listy

Parametrem jest głowa listy i w pętli, dopóki głowa nie jest elementem pustym „przenosimy” wskaźnik głowy na kolejny element listy.

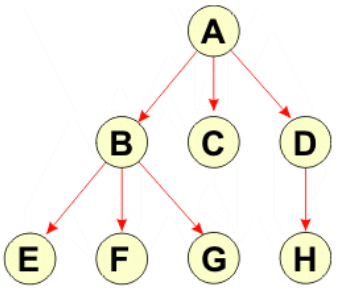
```
int COUNT(ULICAlista *head){
  int ile = 0;
  while(head){
    ile++;
    head = head->kolejna;
  }
  return ile;
}
```

Korzystanie z listy za pomocą gotowych funkcji staje się bardzo proste. Dodajemy nowe elementy zawsze na początku listy, tym samym pierwszy dodany element staje się ostatnim. Jeśli usuwamy, to zawsze od ostatnio dodanego. Taki sposób operowania na liście nazywamy też **kolejką LIFO** – ostatni dodany – pierwszy pobrany lub **stosem**.

```
//inicjowanie listy
lista *L = NULL;
//dodawanie nowych elementów
PUSH(L, "Długoszowskich 1");
PUSH(L, "Rynek 12/3");
PUSH(L, "Grunwaldzka 72");
//liczba elementów listy
cout << COUNT(L)<<endl;
//zwolnienie jednego elementu
POP(L);
cout << COUNT(L)<<endl;
//zwolnienie pozostałych
while(L){
  L = L->kolejna;
  delete L;
}
```

Jeśli zdefiniujemy możliwość usuwania elementów listy od końca, to otrzymamy tzw. **kolejką FIFO** – pierwszy dodany jest też pierwszym pobieranym.

Bardzo często stosowanymi w informatyce strukturami danych są **Drzewa**. Znacznie ułatwiają wyszukiwanie i sortowanie danych i z tego względu stosowane są w bazach danych, przetwarzaniu tekstu grafice komputerowej, telekomunikacji…



Dane przechowywane są w węzłach. Węzły powiązane są z innymi węzłami za pomocą **krawędzi**. Pierwszy węzeł drzewa nazywamy **korzeniem**, pozostałe (te które wyrastają z korzenia) **synami**. Każdy węzeł nadrzędny względem syna nazywamy **ojcem**, a synowie jednego ojca nazywamy **braćmi**. Jeśli węzeł nie posiada synów, to nazywamy go **liściem**. Taki układ przypomina drzewo, ale w odróżnieniu od tych prawdziwych, w informatyce korzeń zawsze jest na górze. Wyjątkową sytuację tworzy tzw. **drzewo binarne** – z jednego węzła wyrasta tylko dwie gałęzie.

Zadania

1) *Wygeneruj plik z losowymi liczbami, a następnie wczytaj je do listy*
2) *Sprawdź czy na liście nie ma duplikatów i je usuń*
3) *Podziel listę na dwie połowy – dwie listy*
4) *Posortuj elementy listy*