1 Wskaźniki i listy jednokierunkowe

1.1 Model pamięci komputera

Pamięć komputera możemy wyobrażać sobie tak, jak na rysunku:

Zawartość: ...	01001011	01101010	11100101	00111001	00100010	01110011	...
adresy:	1753	1754	1755	1756	1757	1758	

Składa się ona z dużej liczby elementów, z których każdy jest w jednym z dwóch stanów, oznaczanych 0 i 1. Stan całej pamięci (jej zawartość) możemy więc wyobrażać sobie jako ciąg 0 i 1. Po włączeniu zasilania zawartość pamięci jest przypadkowa, później zależy od tego, jakie programy działały i działają na komputerze.

Elementy pamięci są grupowane po 8, każda taka grupa tworzy komórkę pamięci. Każdą komórkę można zidentyfikować za pomocą ciągu zer i jedynek ustalonej długości (a raczej za pomocą sygnału elektronicznego opisanego przez taki ciąg). O takim ciągu wolimy myśleć, jak o liczbie naturalnej przedstawionej przez ten ciąg. Tę liczbę nazywamy numerem komórki pamięci lub – częściej – adresem komórki.

1.2 Zmienne (w języku programowania)

Dość trudno zdefiniować pojęcie zmiennej, ale w językach programowania takich, jak Pascal lub C, można podać kilka elementów składających się na to pojęcie. Są to między innymi:

1) nazwa zmiennej (na przykład x lub *costam*, mogą być też nienazwane zmienne, nazwę zmiennej można uogólnić do syntaktycznego pojęcia zmiennej (znanego np. z Pascala), obejmującego odniesienia do elementów tablic, pól struktur (rekordów), zmiennych wskazywanych itp., czyli wszelkich napisów oznaczających w języku coś w rodzaju zmiennej),

2) typ zmiennej,

3) rozmiar zmiennej,

4) adres zmiennej,

5) wartość zmiennej

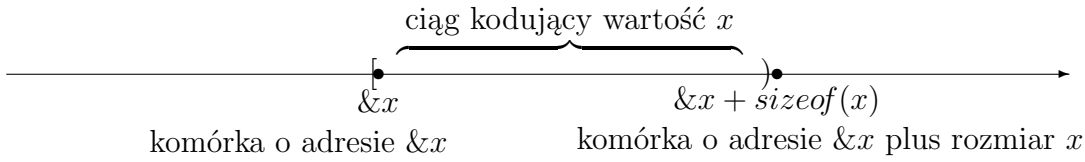
6) i inne.

Zwykle programista wymyśla zmiennej nazwę (może istnieć wiele zmiennych o tej samej nazwie, bywają też zmienne bez nazwy). Zmienna – mówiąc intuicyjnie – przechowuje (lub pamięta) pewną

informację zwaną wartością zmiennej. Jeszcze inaczej: ze zmienną jest związana informacja zwana wartością zmiennej. Programista jest zobowiązany zadeklarować zmienną i wtedy określa jej typ. Typ zmiennej wyznacza sposób reprezentowania jej wartości w pamięci komputera. Zwykle również wyznacza jej rozmiar. Rozmiar zmiennej to liczba komórek pamici potrzebna do zapisania jej wartości. W pascalowym programie możemy posłużyć się rozmiarem zmiennej x typu t pisząc $SizeOf(x)$ lub $SizeOf(t)$. Podobnie jest w C. Ze zmienną jest związany fragment pamięci komputera, w którym jest zapisana (zakodowana) wartość zmiennej Składa się z tylu kolejnych komórek, ile wynosi rozmiar zmiennej. Adresem zmiennej jest adres pierwszej komórki tego fragmentu. Jest on określany przez kompilator i jest ostatecznie ustalany podczas wykonania programu. W programie pascalowym możemy użyć adresu zmiennej x pisząc $@x$. Analogiczna konstrukcja w języku C ma postać $\&x$.

Na przykład zmienna x typu $Char$, o rozmiarze 1, umieszczona w pamięci pod adresem 1756 ma wartość opisaną przez ciąg 00111001. Wartość takiej zmiennej jest zapisywana przy użyciu kodu ASCII Zawartość wspomnianej komórki jest przedstawieniem dwójkowym liczby 57. Z kolei liczba 57 jest kodem ASCII cyfry (znaku) 9. W tym przypadku wartością zmiennej x jest znak 9. Gdyby zmienna x miała pascalowy typ $Word$, to jej wartością byłaby liczba 57.

Niżej mamy schematyczny rysunek przedstawiający zmienną o nazwie x. Pozioma linia to wyobrażenie pamięci komputera (lub jej zawartości). Kółko może przedstawiać pojedyńczą komórkę pamięci, pod nim mamy zapisany adres komórki.



Pojęcie zmiennej najczęściej obejmuje też m. in. elementy tablic i pola struktur, czyli rekordów. Jeżeli na przykład **a** jest nazwą tablicy, to w językach C i Pascal, w każdym miejscu, w którym może wystąpić zmienna nazwana, może też wystąpić (pod warunkiem zgodności typów i sensowności) napis **a[3]**, oznaczający element tablicy **a** o indeksie 3. Taki napis zwykle pełni też funkcję nazwy (określenia, odwołania do) zmiennej i, podobnie jak zwykła nazwa, zależnie od kontekstu, oznacza albo pewien obszar pamięci, pod który np. coś podstawiamy, albo zapisanę tam informację, którą np. gdzieś podstawiamy.

1.3 typy wskaźnikowe i zmienne tych typów

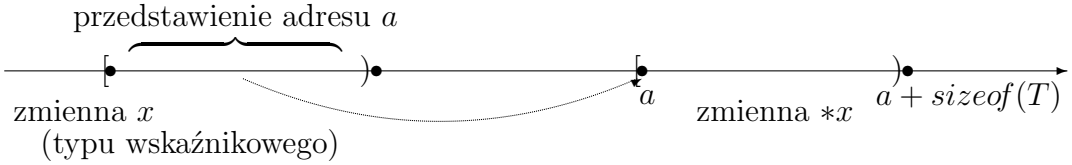
Wśród typów są też typy wskaźnikowe. Wartości zmiennych tych typów są adresami komórek pamięci komputera. Właściwie są to liczby, ale nie ma sensu wykonywać na nich operacji mnożenia, ani dodawania. Ewentualnie można dodać do adresu liczbę naturalną, i to się czasem robi. Dzięki typom wskaźnikowym możemy w Pascalu i w C stosować adresowanie pośrednie znane z definicji maszyny RAM.

Programując wyjątkowo posługujemy się pojedyńczymi komórkami pamięci. Zwykle posługujemy się zmiennymi, które zajmują fragment pamięci złożony z wielu komórek. Wobec tego, określając typ wskaźnikowy podaje się dodatkowo, iż oczekujemy, że pod wskazanym adresem znajduje się informacja określonego typu. Mając typ T możemy zdefiniować typ wskaźnikowy, wskazujący wartości typu T. Intuicyjnie, wartościami takiego typu są adresy, pod którymi znajdują się informacje typu T.

Jeżeli mamy typy wskaźnikowe, to możemy posługiwać się też zmiennymi tych typów. Posługujemy się nimi, jak każdymi innymi, w szczególności mogą one występować w instrukcji podstawiania po obu stronach operatora podstawiania. Dodatkowo, z takimi zmiennymi jest związane pojęcie zmiennej wskazywanej. Jeżeli **x** jest zmienną typu wskaźnikowego, wskazującą wartości typu **T**, to napis ***x** bywa nazywany zmienną wskazywaną przez **x**, pełni rolę nazwy zmiennej, jest używany jako odwołanie do zmiennej i może występować po obu stronach operatora podstawiania. Napis ***x** jest typu **T**, jest rozmiaru takiego, jak każda inna zmienna typu **T**, oznacza wartość zapisywaną tak, jak inne wartości typu **T**. W końcu, adresem zmiennej ***x** jest wartość zmiennej **x**.

Zauważmy, że po wykonaniu (zapisanego w C) podstawienia **y = &x** wartości wyrażeń ***y** oraz **x** powinny być równe.

Związek między zmienną **x** (typu wskazującego wartości typu **T**) i zmienną wskazywaną ***x** (typu **T**) został przedstawiony na poniższym rysunku:



Kolejna tabelka pozwala na porównanie tych elementów składni języków C i Pascal, które dotyczą zmiennych wskaźnikowych.

	C	Pascal
typ wskaźnikowy wskazujący typ **T**	**T***	**^T**
deklaracja zmiennej **x** typu wskaźnikowego	**T *x**	**var x:^T**
zmienna lub wartość wskazywana przez **x**	***x**	**x^**
wskaźnik „pusty”	**null**	**nil**

1.4 Stała null (ewentualnie nil)

W C i w Pascalu mamy stałą, której wartość można przypisać dowolnej zmiennej typu wskaźnikowego. W C stała ta nazywa się **null**, a w Pascalu – **nil**. Jej wartością jest „wskaźnik, który niczego nie wskazuje”, albo myśląc o adresach – jest to adres poprawny składniowo, ale oznaczający lokalizację, w której nic nie ma.

Z punktu widzenia programisty istotne jest to, że jeżeli podczas wykonania programu zdarzy się, że zmienna wskaźnikowa **x** ma wartość taką, jak **null** (a więc jeżeli warunek **x == null** jest prawdziwy), to próba posłużenia się w tym momencie zmienną wskazywaną ***x** jest błędem i powoduje przerwanie działania programu.

1.5 Nadawanie wartości zmiennym typów wskaźnikowych

Jeżeli mamy zmienne, to powinniśmy przypisywać im jakieś wartości, chociażby wartości początkowe. Oczywiście, jeżeli pewna zmienna coś pamięta, to możemy jej wartość przypisać każdej innej zmiennej tego samego typu i typy wskaźnikowe nie są tu wyjątkiem. W ten sposób jednak kopiujemy tylko wcześniej istniejące wartości. Niewiele też daje posługiwanie się jedyną stałą typu wskaźnikowego. Istotnie nowe wartości typów wskaźnikowych uzyskuje się w specjalny sposób.

Jeżeli korzystamy ze zmiennej, to musi zostać ustalony fragment pamięci, w którym będzie zapisywana jej wartość. Sposób wykorzystania pamięci komputera zależy od kompilatora. W przypadku zmiennych nazwanych, kompilator po zapoznaniu się z deklaracją zmiennej jakoś decyduje o umiejscowieniu jej w pamięci komputera.

Podobnie jest w przypadku zmiennych wskazywanych. O umiejscowieniu zmiennej wskazywanej w pamięci komputera decyduje kompilator, ale tym razem robi to życzenie programisty wyrażone w programie, a przy okazji przypisuje wskaźnik do tego miejsca odpowiedniej zmiennej typu wskaźnikowego. Aby wyrazić to życzenie, trzeba w programie umieścić wywołanie odpowiedniej funkcji, którą w dalszej części tego tekstu będziemy nazywać **nowy_wskanik** (wywołanie będzie miało uproszczoną postać **x = nowy_wskaźnik**).

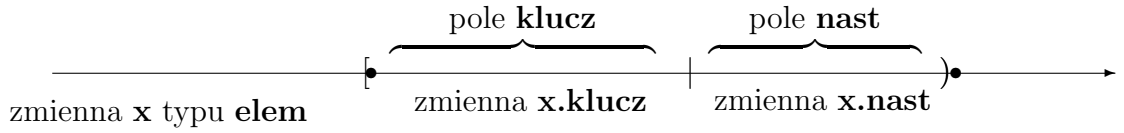
W języku C w takim przypadku używa się funkcji o nazwie **malloc** wymagającej parametru będącego rozmiarem tworzonej zmiennej (patrz definicja funkcji **utwórz** wykorzystywanej na wykadzie, **x = malloc(sizeof(...))**). W Pascalu do przypisania zmiennej wskaźnikowej **x** jest używana procedura **New** z parametrem **x** wywoływanym przez nazwę (wywołanie ma postać **New(x)**).

1.6 Struktury

Deklaracja

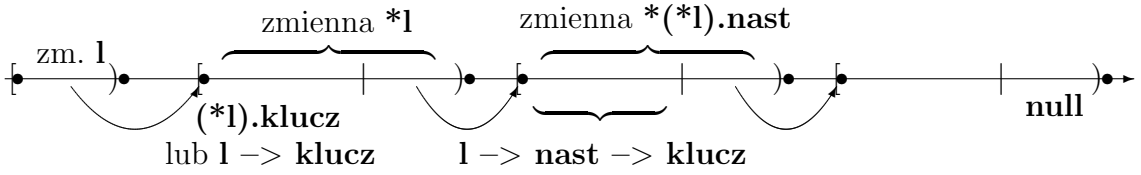
```
1) struct elem {
2)     int klucz;
3)     elem *nast;
4)     }
```

wprowadza nowy typ strukturowy (?, dawniej: rekordowy) o nazwie **elem**. Zmienne tego typu pamiętają dwie informacje: liczbę całkowitą w polu o nazwie **klucz** i wskaźnik do zmiennej typu **elem** w polu **nast**. Są reprezentowane w pamięci komputera w sposób przedstawiony na rysunku:



1.7 Listy jednokierunkowe

Listy można przedstawiać jak niżej



Typ odpowiadający pojęciu listy można definiować na dwa sposoby. Standardowo przyjmuje się definicję **typedef elem *lista**. Wtedy listę utożsamia się ze wskaźnikiem do pierwszego elementu listy. Przy takim rozumieniu pojęcia listy istnieje też lista pusta, która odpowiada wskaźnikowi **null**.

Można też niestandardowo przyjąć, że lista jest synonimem pojęcia elementu listy. Wtedy deklarujemy **typedef elem lista**. Tak rozumiane listy utożsamiamy z pierwszym elementem listy, który w tym przypadku musi istnieć. W ten sposób możemy mówić wyłącznie o listach niepustych.

1.8 Rozwiązanie zadania 2 z listy 11 przy niestandardowej definicji

1) Dane: zmienna **l** typu **elem**, czyli pierwszy element pewnej listy, i liczba całkowita **k**.

2) **elem *x; x = &l;**

3) **while (x −> nast !== null) {x = x −> nast; }**

4) Komentarz: Zauważmy, że stwierdzenie "x wskazuje element listy l" jest niezmiennikiem powyższej instrukcji. Podobnie, niezmiennikiem jest stwierdzenie "x jest różne od null". Po wykonaniu pętli while prawdziwe są następujące stwierdzenia: "x wskazuje niepusty element listy l, którego pole nast jest puste (ma wartość **null**)".

5) **przydziel(x −> nast);**

6) **x −> nast −> nast = null;**

7) **x −> nast −> klucz = k;**

1.9 Rozwiązanie zadania 2 z listy 11 przy standardowej definicji

1) Dane: zmienna **l** typu **lista**, czyli wskaźnik do pierwszego elementu pewnej listy, i liczba całkowita **k**.

2) **elem *p;**

3) **przydziel(p);**

4) **p –> nast = null;**

5) **p –> klucz = k;**

6) **if (l == null) { l = p; }**

7) **else {**

8) **elem *x = l;**

9) **while (x –> nast !== null) {x = x –> nast; };**

10) **x = p;**

11) **}**

1.10 Odwracanie listy jednokierunkowej (zadania 7 lub 12 z listy 11)

Odwracanie listy jest zadaniem, które wymaga doprecyzowania. Będziemy zakładać, że dana lista jest nam niepotrzebna, a z jakichś powodów interesuje nas lista, która ma elementy takie jak dana, ale wymienione w odwrotnej kolejności, i w związku z tym elementy dana lista może zostać zniszczona, a jej elementy – wykorzystane w konstrukcji listy odwróconej.

Idea jest prosta: zapoznajemy się z kolejnymi elementami danej listy i z obejrzanych budujemy stopniowo listę odwróconą. Będziemy starać się, aby podczas działania programu był zachowywany następujący niezmiennik:

dana lista L jest konkatenacją dwóch list L_1 i L_2 takich, że lista wskazywana przez **odwr** *jest odwróceniem L_1, a lista wskazywana przez* **dana** *jest równa L_2.*

1) Dane: zmienna **dana** typu **lista**, czyli wskaźnik do pierwszego elementu pewnej listy.

2) **lista odwr, pom;**

3) **odwr = null;**

4) **while (dana ! == null) {** (teraz lista **dana** jest niepusta)

5) **pom = dana –> nast;** (w **pom** zapamiętujemy wskaźnik do tzw. ogona listy, czyli reszta listy po pominięciu pierwszego jej elementu)

6) **dana –> nast = odwr;** (odcinamy pierwszy element listy od jej ogona, i dołączamy do niego całą listę wskazywaną przez **odwr**)

7) **odwr = dana;** (wyżej utworzona lista jest kolejnym przybliżeniem listy odwróconej i wskaźnik do niej jest zapamiętywany jako wartość **odwr**)

8) **dana = pom;** (zapamiętany ogon staje się nową daną listą)

9) **}**

10) **return odwr;** (Teraz cała odwrócona lista jest wskazywana przez zmienną **odwr**. Trzeba coś zrobić z wynikiem obliczeń, np. zwrócić).

1.11 Czy drzewo jest drzewem BST? (zadanie 5 z listy 12)

Uznałem, że do rozwiązania tego zadania jest potrzebna pomocnicza funkcja z dodatkowymi parametrami. Główna funkcja ma następującą postać:

1) **int min, max;**

2) **if (d = null) return(true);**

3) **else return(bst(d, &min, &max));**

Przyjmujemy, że drzewa składają się z elementów typu

1) **struct elem {**

2) **int k;**

3) **elem *l, *p;**

4) **}**

a same drzewa są typu **tree** definiowanego poprzez **typedef elem *tree**. Tak więc zmienna **d** powinna zostać zadeklarowana jako **elem *d** lub **tree d**. Poniżej została zdefiniowana pomocnicza funkcja **bst**. Napisy **true** i **false** mogą być rozumiane jako całkowitoliczbowe stałe równe odpowiednio 1 i 0.

Funkcja **bst** jest pisana tak, aby miała następujące własności

1) Funkcja **bst** przyjmuje jako wartości **true** i **false**, czyli 1 i 0.

2) Jeżeli **bst(d, min, max) = true**, to **d** wskazuje drzewo BST, a **min** i **max** wskazują zmienne, których wartości są równe najmniejszemu i największemu elementowi drzewa wskazywanego przez **d**.

3) Jeżeli **bst(d, min, max) = false**, to **d** wskazuje drzewo, które nie jest drzewem BST.

Własności te powinno dać się łatwo dowieść przez indukcję ze względu na budowę danego drzewa.

Definicja funkcji **bst**:

```
1)  int bst(d, min, max)
2)  tree d; int *min, *max;
3)  {
4)    int mn, mx;
5)    if (d -> l  =  null)
6)    {
7)        if (d -> p  =  null)
8)        { *min = d -> k;  *max = d -> k;  return ( true ); }
9)        else
10)       {
11)           if ( bst(d -> p, &mn, max)  &&  d -> k <= mn )
12)           { *min = d -> k;  return( true ); }
13)           else return( false );
14)       }
15)   }
16)   else
```

```
17)  {
18)      if (d -> p  =  null)
19)      {
20)          if ( bst(d -> l, min, &mx)  &&  d -> k  >=  mx)
21)          { *max = d -> k;  return( true ); }
22)          else return( false );
23)      }
24)      else
25)      {
26)          if ( bst(d -> l, min, &mx)  &&  bst(d -> p, &mn, max)
27)                  && mx <= d -> k && d -> k <= mn)
28)              return( true );
29)          else return( false );
30)      }
31)  }
32) }
```