Lekcja 4: Listy

Wstęp

Ta lekcja zaczyna się powtórką rozdziału o wskaźnikach - tego z podręcznika Programowania. Tam pełnił on rolę ostatniego rozdziału kończącego wiedzę o języku. Tutaj został rozszerzony o tablice dynamiczne i jest wstępem do zrozumienia nowej struktury danych - listy dynamicznej, czyli takiej, która tworzy się i znika w trakcie działania programu. I zarazem dopasowuje się długością do potrzeb użytkownika.

To jedna z najważniejszych lekcji w tym podręczniku. Być może nie zrozumiecie jej od razu. Nie przejmujcie się. To normalne. Trzeba czasem parę razy powrócić do początku, by "zaskoczyć" i zrozumieć, jak poruszać po listach. Trzeba wykonać testy końcowe i rozwiązać sporo zadań. I wtedy powiecie - przecież to takie proste... I wtedy dopiero będziecie mogli zacząć chodzić po drzewach :) - już zaraz, w następnej lekcji.

Wskaźniki i zmienne dynamiczne

Wiecie już, jak definiować zmienne różnych typów, wiecie jak są przechowywane w pamięci komputera, czyli macie pełne podstawy ku temu, aby trochę bliżej poznać bardziej zaawansowane mechanizmy pozwalające na świadome zarządzanie zawartością pamięci komputera przez programistę.

Lecz najpierw kilka słów wyjaśnienia, po co jest to Wam potrzebne. Dostępem programów do pamięci steruje system operacyjny. Tutaj zastosujemy pewne uproszczenie: w momencie uruchomienia programu system operacyjny przydziela pamięć (proces przydzielania pamięci nazywać będziemy **alokacją**) :

- na kod programu (czyli listę rozkazów dla procesora)
- na zmienne statyczne, czyli te definiowane w sposób już Wam znany:

```
int i;   // potrzebne 4 bajty - tyle zajmuje liczba całkowita
char zn; // potrzebny 1 bajt - znak jest dla komputera liczbą przypisaną mu w kodzie ASCII

int t[10][10]; //potrzebne 10x10x4 = 400 bajtów, itd
```

Jeżeli pamięci na te zmienne nam zabraknie - program po prostu się nie uruchomi. Pamięć przydzielona na zmienne statyczne na początku działania programu pozostaje przez nie zajęta aż do końca jego pracy - i nic nie możemy z tym zrobić. Nakłada to na programistę poważne ograniczenia - już w momencie pisania programu musimy przewidzieć maksymalne zapotrzebowanie na pamięć.

Załóżmy więc na przykład, że w tablicy umieszczamy rekordy zawierające dane osobowe naszych znajomych. Ponieważ nie mamy żadnej możliwości zmiany rozmiaru tablicy statycznej podczas pracy programu, już w trakcie jego tworzenia musimy wiedzieć, ilu maksymalnie znajomych mieć możemy. Oczywicie to ograniczenie można obejść, podając za każdym razem absurdalnie wysokie wartości, tak aby na przykład tablica zawierała milion elementów. System operacyjny musi nam wtedy przydzielić pamięć na milion rekordów, co przy założeniu, że jeden z nich zajmuje (jak chcielibyśmy pamiętać zdjęcie - choćby małe) 10 kB, daje nam 10 000 000 kB, czyli prawie 10 GB. Jeśli macie komputer o takiej pamięci, to możecie próbować :-).

To może nie był najmądrzejszy przykład, ale pamiętajmy, że nie tylko nasz program pracuje na komputerze. Tworzenie zmiennych statycznych o maksymalnym możliwym rozmiarze nie jest więc żadnym rozwiązaniem - jest nieeleganckie, nieefektywne i samolubne - odbieramy w ten sposób "przestrzeń życiową" dla innych, równolegle pracujących programów.

Wartość zmiennej i wskaźnik do niej

Aby uniknąć wszelkich ograniczeń wymienionych powyżej, wymyślono **zmienne dynamiczne**. Zamiast na początku, podczas uruchamiania programu, alokować pamięć "z zapasem", można poprosić system o przydzielenie nam pamięci na taką zmienną dokładnie w tym momencie, w którym nam to będzie potrzebne. Kiedy skończymy z niej korzystać, zawiadomimy o tym system i pamięć do tej pory przez nas zajęta zostanie mu zwrócona do ponownego wykorzystania, bądź to przez nas, bądź przez inny program. Jeżeli zażądamy dostępu do większej ilości pamięci, niż jest w dyspozycji systemu - zawiadomi on nas o tym i pamięci nie przydzieli. Programista musi więc po każdym żądaniu alokacji sprawdzić, czy dostał to, co chciał - i zareagować odpowiednio w przypadku niepowodzenia.

Zmienne dynamiczne są to zmienne tworzone odpowiednim poleceniem podczas wykonywania programu i istniejące aż do ich jawnego skasowania.

To jawne kasowanie zmiennych dynamicznych jest bardzo istotne - jeśli tego nie zrobimy, pamięć na te zmienne pozostanie zarezerwowana nawet po zamknięciu programu. W ten sposób będzie następował tzw. **wyciek pamięci**; przy każdym kolejnym uruchomieniu programu będą zajmowane następne obszary pamięci i w końcu szybko może jej zabraknąć. Nie zapominajcie więc o usuwaniu zmiennych dynamicznych w programie, jak tylko przestaną byc potrzebne, i nie liczcie na to, że kompilator za Was to zrobi (choć bywają takie języki i sytuacje, że to robi).

Do zmiennych dynamicznych najczęściej mamy dostęp poprzez ich adres, nazywany powszechnie **wskaźnikiem**. Wskaźnik jest specjalnym typem zmiennej, w której przechowywany jest ... adres innej zmiennej, czyli adres początku obszaru pamięci przydzielonego na zapamiętanie naszych danych.

> Odnosząc to do przykładu z lekcji 0: odpowiednikiem wskaźnika będzie szufladka, do której możecie włożyć numer innej szufladki. Czyli niezbyt rozgarnięty człowieczek może poprosić swojego nadzorcę (system operacyjny), aby przydzielił mu dodatkowe szufladki, a numer pierwszej z nich umieścił w wyznaczonej szufladce - wskaźniku.

A więc do danych w pamięci możemy odwoływać się nie tylko przez zwykłe nazwy (wtedy są to zmienne statyczne, powoływane w programie lub podprogramie w momencie ich deklaracji), ale również przez adresy (wskaźniki) do nich. Odnosi się to w równym stopniu do zmiennych dynamicznych jak i statycznych. My w tym podręczniku będziemy wykorzystywali wskaźniki głównie w celu pracy ze zmiennymi dynamicznymi, lecz pamiętajcie - sposób odwoływania się do zmiennej nie jest równoważny sposobowi tworzenia zmiennej.

Wskaźnik wskazujący jakąś daną w pamięci ilustruje się zwykle następująco:

Zmienna `wsk_i` jest w tym przykładzie wskaźnikiem - wskazuje na jakąś daną w pamięci.

Nazwy wskaźników mogą być zupełnie dowolne, tak jak dowolne mogą być nazwy wszystkich innych zmiennych. Załóżmy więc, że `wsk_i` jest wskaźnikiem do danych całkowitych (wskazuje na adres przechowujący zmienną typu int). Jeśli chcemy wpisać liczbę 20 pod zmienną wskazywaną przez wsk_i, zapisujemy to z użyciem gwiazdki * stojącego po lewej stronie wskaźnika (**operator wyłuskania**).

```
*wsk_i = 20;
```

i czytamy tak: *pod zmienną wskazywaną przez `wsk_i` podstaw liczbę 20.*

Analogicznie zapisujemy i czytamy:

```
*adres = 3.8;  //pod zmienną wskazywaną przez adres podstaw wartoć 3.8
cout << *adres; //wydrukuj zmienną wskazywaną przez adres
```

W taki właśnie sposób musicie czytać wszystkie nazwy, które po lewej stronie mają symbol *.

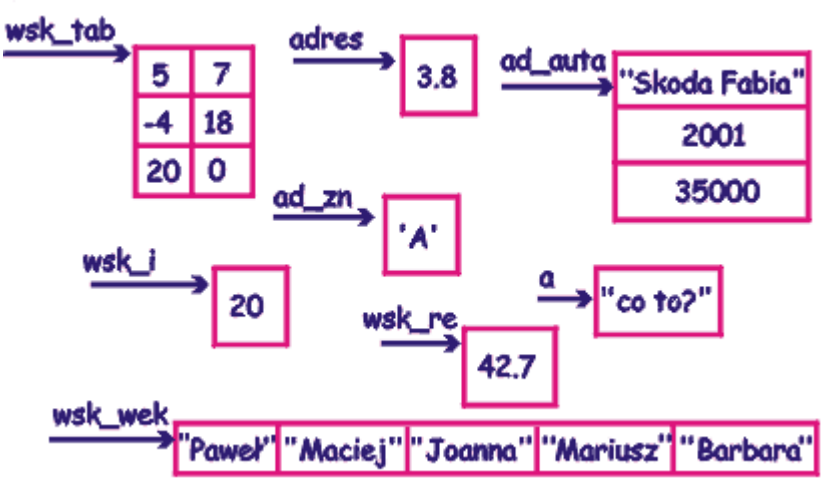
Na zmiennych dynamicznych można wykonywać te same operacje, co na zmiennych statycznych - tylko trzeba się do nich inaczej odwoływać. Porównajcie podstawienie pod zwykłą zmienną statyczną `i` (*pod zmienną `i` podstaw liczbę 12*) i pod zmienną dynamiczną `wsk_i` (*pod zmienną wskazywaną przez `wsk_i` podstaw liczbę 20*):

Powtórzmy raz jeszcze:

Zmienne statyczne są tworzone w momencie uruchamiania programu i istnieją do końca jego pracy. Zmienne dynamiczne tworzone i usuwane są w trakcie pracy programu poprzez wywoływanie odpowiednich poleceń. Do zmiennych statycznych zwykle odwołujemy się przez ich nazwę, do zmiennych dynamicznych zwykle odwołujemy się przez wskaźnik.

Podkreślamy słowo zwykle - ponieważ w C++ można się odwoływać do dowolnego typu zmiennych w dowolny sposób. Ponieważ C++ wywodzi się z języka C, który przewidziany był także do programowania sprzętu (a więc na stosunkowo niskim poziomie), cały mechanizm wskaźników i operacji bezpośrednio na pamięci jest w C++ silnie rozwinięty. My jednak na ogół ograniczymy się do zastosowań podstawowych - możliwych do wykonania także w innych językach programowania.

Zmienne dynamiczne mogą być różnych typów, tak jak wszystkie inne zmienne. Na rysunku poniższym możecie zobaczyć dużo różnych zmiennych dynamicznych, najrozmaitszych typów, wraz z ich wskaźnikami. Zmienne dynamiczne mogą być dowolnie rozrzucone w pamięci komputera.



Powinniście w tym miejscu od razu zapytać: *a skąd będzie wiadomo, co wskazują te różne wskaźniki?*. Rzeczywiście - jeśli używamy zmiennej `i`, najpierw ją deklarujemy. Więc żeby skorzystać ze zmiennej dynamicznej, musimy najpierw zadeklarować wskaźnik do tej zmiennej (zauważcie, że wskaźnik tak naprawdę też jest zmienną, i to do tego statyczną - w niej zapamiętany jest adres zmiennej dynamicznej). Deklaracje wskaźników również oznacza się poprzez gwiazdkę - również po lewej stronie nazwy zmiennej. Piszemy więc tak:

```
int *wsk_i;
```

i czytamy: *zmienna `wsk_i` jest wskaźnikiem do zmiennej typu int*. Jeśli jednak wskaźnik wskazuje strukturę, to wcześniej trzeba ją zdefiniować. Ogólnie deklaracja wskaźnika ma postać:

```
typ_zmiennej_wskazywanej *nazwa_wskaźnika;
```

W przypadku języka C++, jak zdążyliście już zauważyć - tablice traktowane są w inny sposób niż pozostałe typy zmiennych. Zazwyczaj (w większości przypadków) zmienna typu tablicowego jest równoważna zmiennej zawierającej wskaźnik do pierwszego elementu.

```
int A[10];
```

to `A` może być traktowane jako wskaźnik do zmiennej typu **int**. Może być traktowane - co oznacza, że może być używane zamiennie z wskaźnikiem. To dlatego w niektórych przykładach przedstawionych we wcześniejszych lekcjach przekazywaliśmy tablicę do podprogramu jako wskaźnik.

Teraz już powinnicie umieć zadeklarować różne wskaźniki z naszego obrazka:

```
struct auta
{
  string marka;
  int rocznik;
  double cena;
};

int *wsk_i;
```

```
double *adres, *wsk_re;
string *a;
char *ad_zn;
int *wsk_tab;
auta *ad_auta;
```

Chcesz więcej wiadomości ?

Wskaźnik do zmiennej jest adresem pierwszego z bajtów, w których jest ta zmienna przechowywana. A więc wskaźnik na zmienną typu **int** to adres pierwszego z 4 bajtów, gdzie ta zmienna jest pamiętana; wskaźnik na łańcuch o 50 znakach to znowu adres pierwszego z nich, itd.

Tworzenie i usuwanie zmiennych dynamicznych

Moglibyście w tym miejscu uznać, że skoro umiecie wskaźnik zadeklarować i odwołać się do zmiennej przez niego wskazywanej, to możecie już działać na zmiennych dynamicznych, czyli na przykład:

```
...
struct auta
{
  string marka;
  int rocznik;
  double cena;
};
...

int main(int argc, char *argv[])
{
  ...
  int *wsk_i;
  char *ad_zn;
  auta *ad_auta;

  ...
  *wsk_i = 20;
  *ad_zn = 'A';
  (*ad_auta).cena = 35000;

  ...
  return 0;
}
```

Niestety to jeszcze nie koniec. Wiecie już przecież, że zmienne dynamiczne to takie zmienne, które mogą w trakcie pracy programu pojawiać się i znikać. Musimy więc im to pojawianie się i znikanie umożliwić. Zatem potrzebne będą dwie instrukcje:

- instrukcja, która powołuje do życia zmienną dynamiczną - przydziela jej pamięć i zapamiętuje jej adres;
- instrukcja, która odbiera pamięć zarezerwowaną na zmienną dynamiczną i kasuje jej adres.

Takie dwie instrukcje dostępne są w każdym zaawansowanym języku programowania, który pozwala na używanie zmiennych dynamicznych. W C++ są to odpowiednio **new** i **delete**. Tak więc:

- polecenie **new** *nazwa_typu* przydziela pamięć na zmienną dynamiczną takiego typu, jaki wskazuje *nazwa_typu*. Adres przydzielonej pamięci jest zwracany jako wynik. Jeżeli alokacja (przydział pamięci) się nie powiedzie, np. pamięci już nie wystarczy, zostanie wstawiony adres zerowy, oznaczany w C++ poprzez NULL. Fachowo rzecz ujmując - **new** nie jest poleceniem, tylko operatorem o następującej składni wywołania:

  ```
  zmienna_wskaźnikowa = new nazwa_typu;
  ```

- polecenie **delete** *zmienna_wskaźnikowa* odbiera pamięć zarezerwowaną na zmienną dynamiczną umieszczoną pod adresem pamiętanym w *zmienna_wskaznikowa*. Składnia wywołania tego operatora ma postać:

  ```
  delete zmienna_wskaźnikowa;
  ```

Oba operatory mają specjalną postać w przypadku alokacji i zwalniania tablic. C++ pozwala alokować bezpośrednio jedynie tablice jednowymiarowe (wektory). Aby zaalokować tablicę jednowymiarową, wykorzystujemy operator **new** w następującej postaci:

```
zmienna_wskaźnikowa = new nazwa_typu_elementów[rozmiar];
```

Tablicę jednowymiarową usuwamy zaś z pamięci za pomocą operatora **delete** wywoływanego następująco:

```
delete[] zmienna_wskaźnikowa;
```

Jak poradzić sobie z tablicami o większej liczbie wymiarów - o tym trochę niżej.

I jescze uwaga odnośnie odwoływania się do pól zmiennych strukturalnych (rekordowych) dostępnych przez wskaźnik. Ponieważ operator wyłuskania * ma niższy priorytet niż operator . wykorzystywany do odwoływania się do pól rekordów, zapis:

```
*ad_auta.cena = 35000;
```

jest nieprawidłowy - bo w ten sposób próbujemy się dostać do pamięci, której adres jest umieszczony w polu `cena` struktury - a to nie jest wskaźnik, tylko zwykła liczba. Więc prawidłowo powinniśmy zapisać:

```
(*ad_auta).cena = 35000;
```

Jest także inna metoda. W przypadku wskaźników można zastąpić operator kropki jego specjalną postacią `->`. Tak więc jeśli `ad_auta` jest wskaźnikiem na strukturę, to możemy równoważnie do powyższego zapisu napisać:

```
ad_auta->cena = 35000;
```

I taki właśnie zapis będziemy wykorzystywali.

Powróćmy więc do naszych przykładów. Przed pierwszym wykorzystaniem zmiennej wskazywanej przez `wsk_i`, musimy zarezerwować dla niej miejsce w pamięci:

```
wsk_i = new int;
```

Jeśli pamięć zostanie przydzielona, w zmiennej `wsk_i` znajdzie się adres tej zmiennej.

Następnie możemy zacząć korzystanie ze zmiennej:

```
*wsk_i = 20;
cout << "Wartosc wskazywana powiekszona o 5 " << (*wsk_i)+5 << endl;
```

Na koniec pozostaje nam zwolnienie pamięci (posprzątanie po sobie).

```
delete wsk_i;
```

Cały program mógłby wyglądać więc tak:

```
...
int *wsk_i;
wsk_i = new int;
*wsk_i = 20;
cout << "Wartosc wskazywana powiekszona o 5 " << (*wsk_i)+5 << endl;
delete wsk_i;
...
```

W tym miejscu moglibyście powiedzieć, że nie bardzo widzicie sens pisania w ten sposób, skoro zastosowanie wskaźników nie daje praktycznie żadnej nowej funkcjonalności. W następnym segmencie tej lekcji zobaczycie, że jednak pozwalają one tworzyć zupełnie nowe struktury danych. Ale już za chwilę pokażemy Wam nowy rodzaj tablic, który zawdzięczamy wskaźnikom.

Teraz jeszcze program, który jest schowany - już tylko dla ciekawskich:

Chcesz więcej wiadomości ?

Wspomnieliśmy, że możliwe jest odwoływanie się do zmiennych statycznych poprzez wskaźnik oraz do dynamicznych poprzez nazwę. Tutaj pokażemy, jak to zrobić:

```
#include <iostream>
#include <cstdlib>

using namespace std;

// Wiemy, że możliwe jest odwoływanie się  poprzez
// wskaźnik do zmiennej statycznej, i poprzez nazwę do
// zmiennej dynamicznej. Ten programik demonstruje, jak
// to zrobić. Niemniej jednak ostrzegamy - nie stosujcie
// tych technik, dopóki nie będziecie dokładnie wiedzieli,
// co robicie...
int main(int argc, char *argv[])
{
  cout << "Prezentacja operacji na wskaznikach\n\n";
  // najpierw stworzymy sobie zmienną statyczną
  int zm;
  // i wskaźnik
  int *w;
  // zaalokujemy też pamięć na zmienną wskazywaną przez w1
  w = new int;

  // no i przypiszemy im wartosci poczatkowe:
  zm = 1;
  *w = 10;

  // teraz stworzymy sobie wskaźnik przez który będziemy
  // odwoływali się do zmiennej statycznej:
  int *w_zm;

  // aby uzyskać adres zmiennej zastosujemy operator &
  w_zm = &zm;

  // teraz w_zm wskazuje na zmienną zm. Zmiana wartości tej
  // zmiennej jest możliwa zarówno klasycznie, jak i poprzez
  // wskaźnik:
  cout << "Zmienna statyczna:\n";
  cout << "Wartosc uzyskana przez:\n";
  cout << "  nazwe   : " << zm << endl;
  cout << "  wskaznik: " << *w_zm << endl;

  // zmienmy więc zawartość tej zmiennej korzystając ze wskaźnika:
  *w_zm += 5;
  cout << "Po modyfikacji:\n";
  cout << "Wartosc uzyskana przez:\n";
  cout << "  nazwe   : " << zm << endl;
  cout << "  wskaznik: " << *w_zm << endl;


  // teraz zajmiemy się uzyskaniem dostępu przez nazwę do
  // zmiennej dynamicznej
  int &zm_w = *w;

  // podobnie jak w poprzednim przypadku - mamy możliwość
  // dostępu do tej samej zmiennej na dwa sposoby:
  cout << "Zmienna dynamiczna:\n";
  cout << "Wartosc uzyskana przez:\n";
  cout << "  nazwe   : " << zm_w << endl;
  cout << "  wskaznik: " << *w << endl;

  zm_w += 5;
  cout << "Po modyfikacji:\n";
  cout << "Wartosc uzyskana przez:\n";
  cout << "  nazwe   : " << zm_w << endl;
```

```
64.    cout << "  wskaznik: " << *w << endl;
65.
66.    // pozostaje jeszcze posprzątanie po sobie.
67.    // musimy skasować zmienną zaalokowaną dynamicznie:
68.    delete w;
69.
70.    // od tego momentu nie można odwoływać się ani do wskaźnika w,
71.    // ani do zmiennej zm_w...
72.
73.    // no i jeszcze jedno: absolutnie (!) nie wolno zwolnić pamięci
74.    // wskazywanej przez w_zm: polecenie  delete w_zm;
75.    // jest najprostszą drogą do awaryjnego zakończenia tego programu ;)
76.
77.    return 0;
78.  }
```

Tablice dynamiczne

Wszystkie tablice, których używaliśmy dotychczas, są tablicami statycznymi, gdyż pamięć na nie jest rezerwowana na etapie kompilacji, zanim program zacznie się wykonywać. Dlatego właśnie rozmiar ich jest określany za pomocą stałych, w dobrze znany Wam sposób:

```
int main() {
const int n=200;
int T [n] ;
...
}
```

Jeśli jednak nie wiemy z góry, ile elementów będziemy umieszczać w tablicy, musimy ją rezerwować niepotrzebnie dużą, na zapas (podając dużą wartość stałej n) - niekiedy nam się to przyda, ale często elementy tablicy zajmą w niej mały fragment i cały duży obszar pamięci pozostanie niepotrzebnie zarezerwowany. Aby temu zaradzić, aż się prosi, by najpierw dowiedzić się od użytkownika, ile elementów ma mieć tablica, a potem dopiero przydzielić jej pamięć. To oznacza jednak, że taką rezerwację należy przeprowadzić w trakcie działania programu - będzie to więc **dynamiczna rezerwacja pamięci**. Nie możemy jej jednak wykonać w taki sposób:

```
int main() {
int n;
cin >> n;
int T [n] ;  // TAK NIE WOLNO !!!  taki zapis oznacza tablicę statyczną
...
}
```

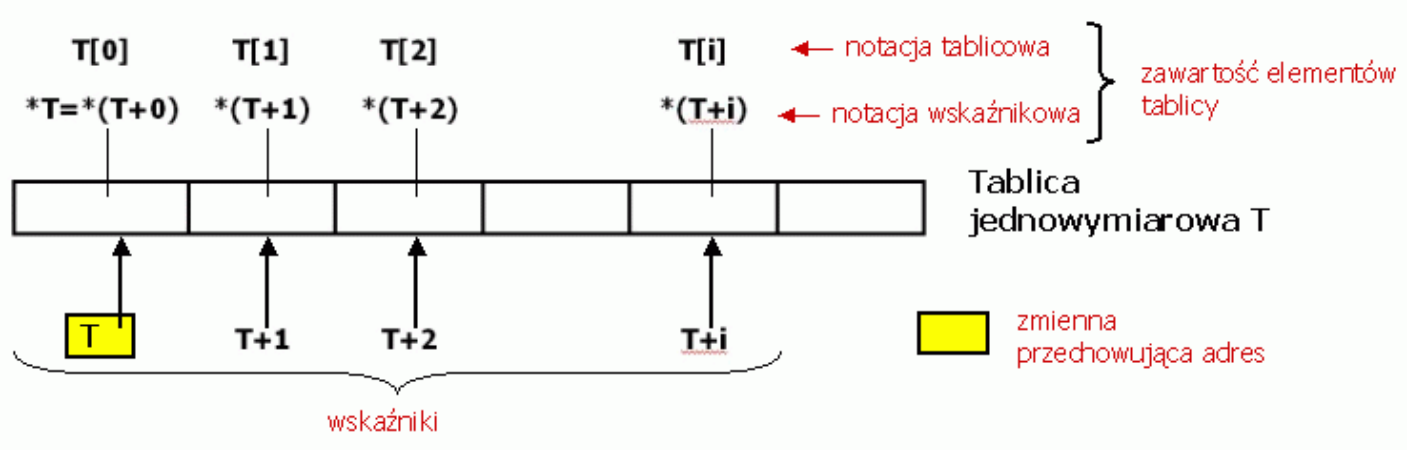
Z pomocą przychodzą nam tu właśnie - wskaźniki. To one przecież umożliwiają dostęp do pamięci rezerwowanej dynamicznie.Zobaczcie, jak to zrobić:

```
int main( ) {
// zarezerwujemy tablicę T typu int
int *T ; // to oznacza, że nazwa T jest wskaźnikiem do pierwszego elementu tablicy
int n;
cout << "ile elementow ma być w tablicy ? " << endl;
cin >> n;
T = new int [n]; // rezerwujemy pamięć  dla n elementów tablicy typu int
// i adres pierwszego z tych elementów  zapamiętujemy jako T
...
for (int i=0; i<n; i++) cin >> T[i]; // wykonujemy zwykłe działania na tablicy
...
delete [] T; // na końcu  zwalniamy pamięć  przydzieloną na tablicę
...
return 0;
};
```

Jak widzicie, jak tylko dokonacie rezerwacji pamięci, możecie takiej tablicy używać dokładnie tak jak zwykłej tablicy statycznej, w dobrze znany Wam sposób. Musicie tylko na końcu pamiętać o zwolnieniu pamięci. Jeśli przy kolejnych uruchomieniach programu

pamięć będzie tylko alokowana, a nie zwalniana, będzie dochodzić do powolnego wycieku pamięci i w końcu może nam jej zabraknąć. Nie zapominajcie więc o tzw. "sprzątaniu po sobie".

Oprócz możliwości traktowania tablic dynamicznych z użyciem zwykłej notacji tablicowej, istnieje sposób częściej wykorzystywany przez zawodowych programistów - używanie nazwy tablicy jak wskaźnika do pierwszego jej elementu (tego o indeksie 0), o czym już wspominaliśmy. W naszym przykładzie nazwa tablicy `T` oznacza więc adres elementu `T[0]`. W takim razie zamiast `T[0]` możemy napisać `*T`, zamiast `T[1]` - `*(T+1)` - itd. Dodanie wartości 1 do adresu `T` oznacza tu bowiem powiększenie adresu o rozmiar elementu pamiętanego w tablicy, czyli wyznaczenie adresu elementu następnego w tablicy - wiemy przecież, że są one poukładane za sobą w spójnym obszarze pamięci. Zasadę tej tzw. notacji wskaźnikowej w odniesieniu do tablic pokazuje poniższy rysunek:



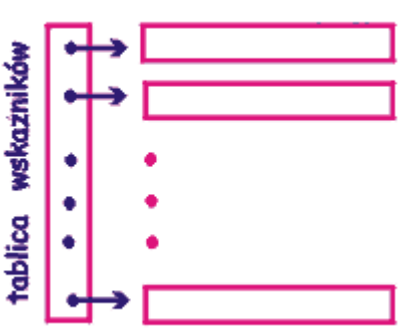
Dodajmy jeszcze przykład, który pokaże, jak za pomocą wskaźników można szybko przemieszczać się po tablicy:

```
int main() {
const int n=200;
int T [n] ;
int *wt;   // dodatkowy wskaźnik
wt=T;     // wt jest tu adresem pierwszego elementy tablicy
wt++;    // teraz wskaźnik przesunął się do następnego elementu tablicy
...     // w analogiczny  sposób przeskakujemy  do dowolnych elementów tablicy
wt--;  // w razie potrzeby możemy szybko cofnąć się do elementu poprzedniego
}
```

Jeśli polubicie notację wskaźnikową, możecie jej używać. To jest szczególnie ważne, gdy chcemy przyspieszyć działanie programu - za pomocą bezpośrednich adresów szybciej on "wskakuje" do odpowiednich miejsc w tablicy, więc kod maszynowy jest wtedy bardziej wydajny. Nie musicie tego jednak stosować, dopóki nie odczujecie zbyt powolnego działania programu - a i obecne kompilatory znacznie lepiej sobie teraz radzą z notacją tablicową, niż kiedyś. Możecie więc spokojnie pozostać przy dobrze Wam znanym zapisie z nawiasami kwadratowymi. Czytelność i jasność kodu jest zawsze sprawą nadrzędną.

Skoro wiecie już, jak sobie radzić z dynamicznymi tablicami jednowymiarowymi, możemy Wam zaprezentować strukturę, która na tych samych zasadach pozwoli Wam dynamicznie rezerwować dużą tablicę dwuwymiarową bez niepotrzebnego deklarowania z dużym zapasem tradycyjnej tablicy statycznej (na ogół w języku C++ nie stosuje się tablic statycznych o więcej niż jednym wymiarze - mimo że jest to możliwe, jak pokazywaliśmy w przykładach zamieszczonych do tej pory).

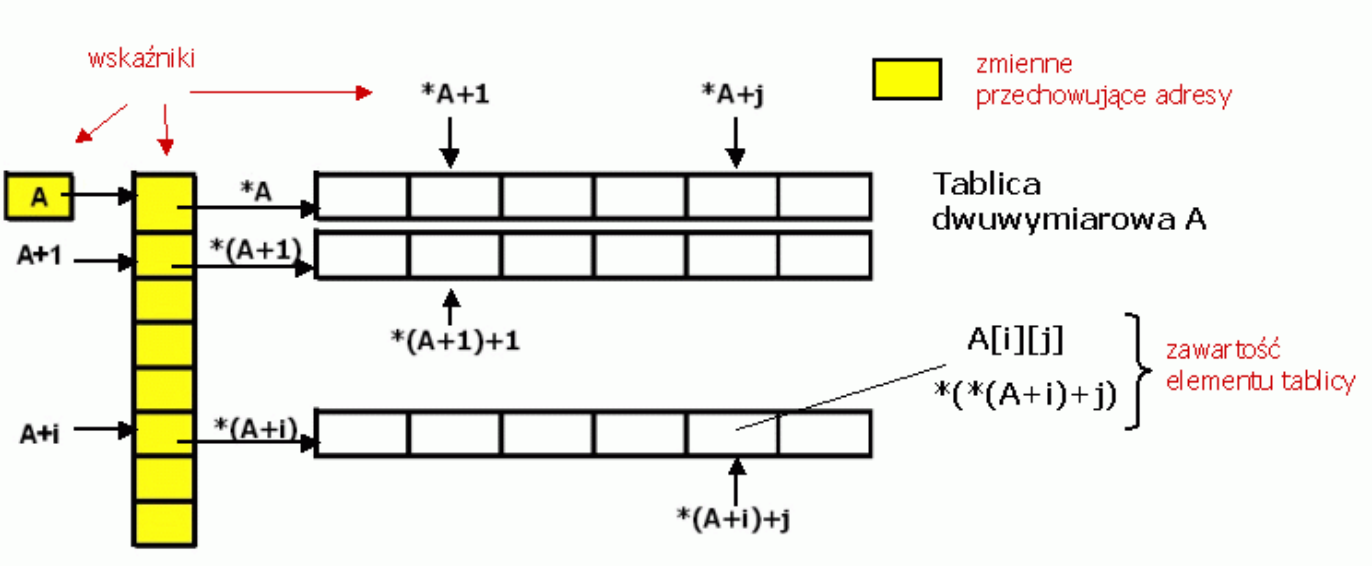
Możecie ją zadeklarować, utworzyć i wykorzystywać w sposób pokazany schematycznie na obrazku i opisany poniżej:



```
int main(int argc, char *argv[])
{
  // skoro tablica ma być dwuwymiarowa, uciekniemy się do prostego chwytu.
  // zadeklarujemy bowiem jednowymiarową tablicę wskaźników zawierającą
  // wskaźniki do jednowymiarowych tablic. Czyli będziemy mieli zmienną
  // wskaźnikową, która będzie zawierała wskaźnik do wskaźnika do liczby
  // rzeczywistej. W C++ zapisujemy to następująco:
  double **A;
  int w, k;
  cin >> w >> k;  // wczytujemy rozmiary tablicy;  w,k mogą być też stałymi - ale nie muszą

  // alokacja pamięci na taką tablicę także przebiega dwuetapowo. Najpierw
  // musimy stworzyć wektor na wskaźniki:
  A = new double*[w];

  // potem zaś stworzyć poszczególne tablice liczb rzeczywistych i umieścić
  // ich adresy w tym wektorze:
  for (int i = 0; i < w; i++)
    A[i] = new double[k];

  // teraz możemy korzystać z naszej tablicy podobnie jak z tablicy statycznej
  A[1][50] = 123.45;
  ...

  // usuwamy tablicę dwuwymiarową w sposób odwrotny do jej tworzenia.
  // tak więc najpierw kasujemy tablice liczb rzeczywistych:
  for (int i = 0; i < w; i++)
    delete[] A[i];

  // a na koniec kasujemy tablicę wskaźników
  delete[] A;
  ...
};
```

Oczywiście wczytywanie dużej liczby danych tak naprawdę powinno odbywać się z pliku, ale nie chcieliśmy wprowadzać tu zbędnego zamieszania.

Na koniec rysunek, który pokazuje, jak zastosować notację wskaźnikową do tablic dwuwymiarowych. Notacja tablicowa jest w tym przypadku jednak znacznie bardziej przejrzysta i łatwiejsza do stosowania...



To, co poznaliście do tej pory, to dopiero wstęp do operacji na wskaźnikach. W C++ wskaźniki i bezpośredni dostęp do pamięci to potężne i często wykorzystywane narzędzie. Pokazaliśmy to na przykładzie notacji wskaźnikowej tablic. Niemniej jednak - ponieważ nie jest to kurs języka C++ - to na tym zakończymy. Zainteresowani będą musieli uzupełnić swoją wiedzę korzystając z innych źródeł.

Lista

Skoro już wyjaśniliśmy sobie pojęcie wskaźnika i wiemy, jak go używać, pora na kolejne jego zastosowanie. Tym razem będziemy chcieli zdefiniować strukturę dynamiczną o nieokreślonej z góry liczbie elementów - czyli listę.

Co to jest?

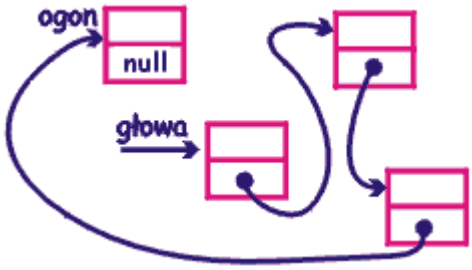
Listą będziemy nazywali łańcuch liniowo powiązanych ze sobą elementów dowolnego typu.

Prościej: lista jest w pewnym sensie odpowiednikiem wektora. Obie struktury są jednowymiarowe i mają liniowe uporządkowanie - zarówno w liście, jak i w wektorze każdy element z wyjątkiem pierwszego i ostatniego ma poprzednika i następnika.

Różnice: z punktu widzenia użytkownika wektor jest strukturą danych o swobodnym dostępie (możemy odwołać się bezpośrednio do dowolnego elementu poprzez jego numer), natomiast dostęp do danych pamiętanych na liście jest sekwencyjny - można odwołać się jedynie do elementów sąsiadujących z aktualnym. Z punktu widzenia systemu operacyjnego, wektor jest ciągłym obszarem pamięci; przypominamy z lekcji 3, że położenie wektora w pamięci możemy zilustrować następująco:

Dlatego też kompilator napotykając na definicję wektora (tablicy) może bezproblemowo obliczyć rozmiar pojedynczego jej elementu, i znając położenie pierwszego elementu i rozmiar, automatycznie obliczyć adres dowolnego elementu do niego należącego.

Natomiast lista jest zestawem powiązanych ze sobą, nieciągłych fragmentów - zmiennych dynamicznych - umieszczonych w różnych obszarach pamięci RAM:



Głowa oznacza tu adres pierwszego elementu listy, ogon - adres ostatniego (oczywiście są to nazwy dowolne; zamiast nich mogą to być adresy o nazwach poczatek i koniec albo cokolwiek innego). Naszą listę możemy narysować inaczej:

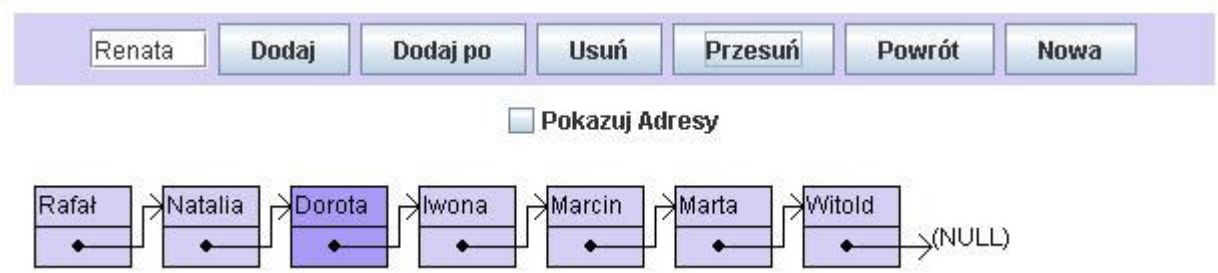
Powiązanie pomiędzy elementami listy uzyskujemy w ten sposób, że informację o położeniu następnego elementu umieszczamy w elemencie go poprzedzającym:

Jak widzicie, każdy element listy oprócz danych zawiera także informację, gdzie należy szukać następnego elementu (pole nast). Informacja ta jest zawsze zapamiętywana jako wskaźnik do następnego elementu. Przedstawiona na rysunku powyżej struktura danych nazywa się **listą jednokierunkową** - dlaczego, każdy chyba się domyśla (wąż jaki jest, każdy widzi :-)).

Koniec listy oznacza specyficzna wartość wskaźnika, znana już Wam z poprzedniego segmentu, nazwana `NULL`, co po angielsku oznacza *zero* (możecie nawet używać liczby zero zamiast NULL). A więc `NULL` oznacza, że dalej już nic nie ma.

Podczas korzystania z listy jednokierunkowej, użytkownik może ją przeglądać jedynie w jednym kierunku - od początku do końca, nie ma natomiast żadnej możliwości, aby się cofnąć. Jest to oczywiste ograniczenie, dlatego też udoskonalono ją dodając drugie pole na wskaźnik i umożliwiając w ten sposób przeglądanie listy w obu kierunkach (**lista dwukierunkowa**). Innym udoskonaleniem jest połączenie pierwszego elementu z ostatnim - uzyskuje się w ten sposób **listę cykliczną** (jedno- lub dwukierunkową).

My jednak w tej lekcji będziemy się zajmowali wyłącznie listami jednokierunkowymi. Na początek proponujemy aplet, który pomoże Wam zrozumieć własności takich list. Aplet uruchomi się w osobnym oknie, gdy klikniecie w poniższy obrazek. Koniecznie wykonajcie wszystkie ćwiczenia, które są tam opisane. To bardzo ułatwi zrozumienie zasady korzystania z tej nowej struktury danych.



Po tych ćwiczeniach z apletem listy już chyba stały się zrozumiałe, jednak Waszym celem jest też nauczyć się kodowania takiej struktury danych w C++. Przejdźmy więc do przykładu: stworzymy listę jednokierunkową, która będzie pamiętać elementy całkowite. Najpierw zdefiniujemy pojedynczy element umieszczany na liście. Jak widać na rysunku powyżej, każdy element musi zawierać, oprócz danych, które chcemy zapamiętać, pewne pola (co najmniej jedno), będące wskaźnikami i służące do budowy struktury danych, czyli do łączenia elementów ze sobą. Skoro element musi zawierać dane różnych typów, to na pewno musi być rekordem. A więc:

Lista dynamiczna jednokierunkowa zbudowana jest z rekordów, w których co najmniej jedno pole zawiera wskaźnik na rekord następny.

Ściślej rzecz ujmując - można napotkać implementację list, które na pierwszy rzut oka nie zawierają żadnych dodatkowych elementów (tak jak np. klasa `list` z STL. Ale to tylko na pierwszy rzut oka ... zawsze musi być gdzieś ów magiczny wskaźnik. Albo jest on zawarty bezpośrednio w danych pamiętanych na liście (jak to ma miejsce w naszych przykładach), albo w odpowiedniej strukturze "opakowującej" nasze dane (jak to jest w przypadku listy z STL).

Żeby korzystać z listy, musimy najpierw więc zdefiniować strukturę, która będzie pamiętała elementy na niej umieszczone wraz z odpowiednimi powiązaniami:

```
struct element_listy
{
  int dane;
  element_listy *wsk_nastepnika;
};
```

Zauważcie, że w C++ można stosować typ zmiennej strukturalnej wewnątrz jej definicji (pole `wsk_nastepnika` jest typu wskaźnik na `element_listy`).

W podany wyżej sposób będziemy definiowali zawsze typy potrzebne do skonstruowania listy.

Skoro już stworzyliśmy typy danych, spróbujmy coś z danymi na tej liście zrobić. Najprościej jest je zawsze po prostu wydrukować. Więc napiszmy funkcję do wydruku zawartości listy:

```
// Iteracyjna funkcja drukująca zawartość listy
void drukuj_liste_it(element_listy *adres)
{
  while (adres!=NULL)
  {
    // wypisujemy zawartość elementu
    cout << (*adres).dane << " ";
    // i przechodzimy do następnego
    adres = adres->wsk_nastepnika;
  }
};
```

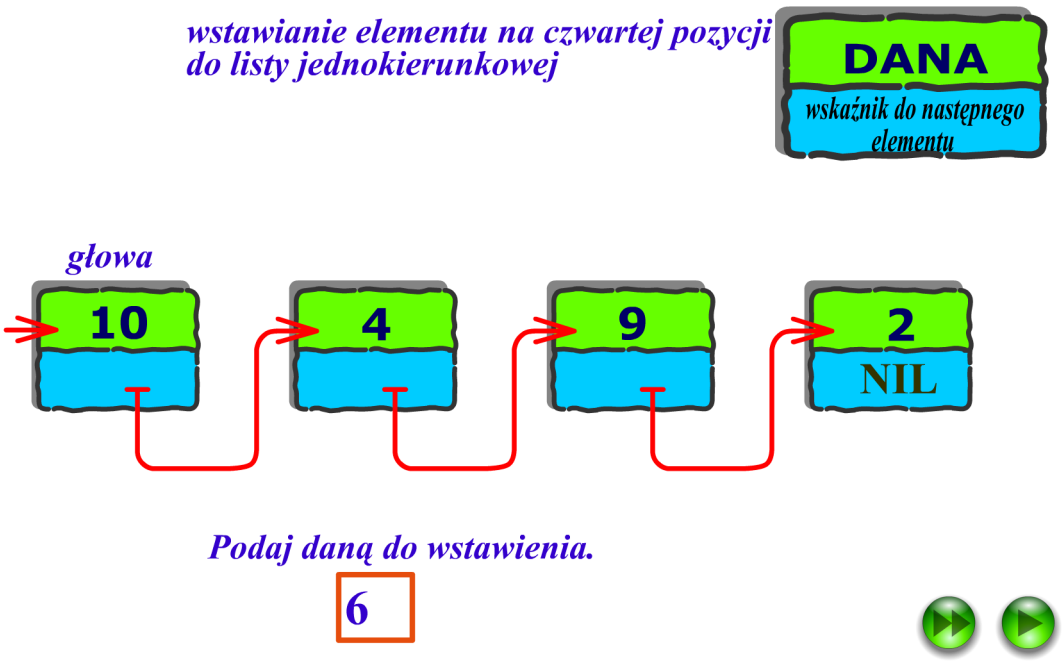
Chcesz więcej wiadomości ?

Jak widzicie, była to iteracyjna procedura drukująca listę. Lecz zauważcie, że każdy element listy ma odnośnik (wskaźnik) do innego takiego samego elementu. Czy nie przypomina to Wam czegoś? Lista jest również najprostszą rekurencyjną strukturą danych. Dlatego też zastosowanie procedur rekurencyjnych do jej obsługi jest jak najbardziej naturalne. Zobaczcie więc, jak wyglądałoby drukowanie listy przy wykorzystaniu procedury rekurencyjnej:

```
// Rekurencyjna funkcja drukująca zawartość listy
void drukuj_liste_rek(element_listy *adres)
{
  // warunek zakończenia rekurencji
  if (adres != NULL)
  {
    // wypisujemy zawartość elementu
    cout << adres->dane << " ";
    // przechodzimy do następnego
    adres=adres->wsk_nastepnika;
    // i wywołujemy procedurę dla następnego elementu
    drukuj_liste_rek(adres);
  };
};
```

Wstawianie i usuwanie elementów

Przyszła więc pora na pokazanie, jak do listy wstawiamy i jak z niej usuwamy elementy. W zasadzie czynność wstawienia elementu sprowadza się do dwu kroków: po pierwsze musimy wstawiany element utworzyć (za pomocą operatora **new**), a następnie zmodyfikować istniejące powiązania w liście:



Przykładowo, funkcja wykonująca umieszczanie elementów na liście mogłaby wyglądać następująco:

```
// Funkcja wstawiająca element na listę
void wstaw_element(element_listy *gdzie, element_listy *co)
{
  element_listy *tmp;
  // zapamiętajmy element umieszczony za tym wskazywanym przez gdzie
  tmp = gdzie->wsk_nastepnika;
```

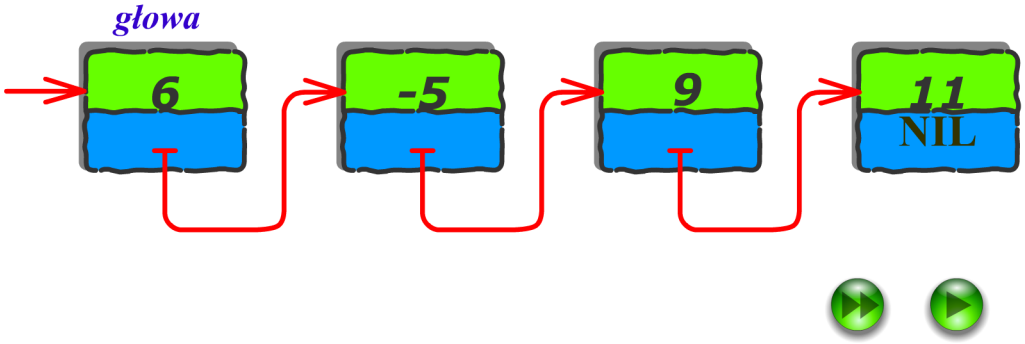
```
  // umieśćmy odwołanie do wstawianego
  gdzie->wsk_nastepnika = co;
  // i na koniec odtwórzmy dowiązanie do dotychczasowego natępnika
  co->wsk_nastepnika = tmp;
};
```

Podobnie z usuwaniem elementów. Tym razem musimy jedynie odwrócić kolejność działań, czyli najpierw zmienić powiązania, a potem element usunąć przy pomocy operatora **delete**. Przykładowy kod usuwający z naszej listy elementy zawierające dane mniejsze od 0 znajdziecie poniżej:

```
// USUWANIE ELEMENTÓW mniejszych od zera
  aktualny=glowa;
  poprzedni= NULL;
  while (aktualny != NULL)
  {
    if (aktualny->dane < 0)
    {
      if (aktualny == glowa)
        // jeśli usuniemy glowę, to nowa glowa będzie o jeden element dalej
        glowa = glowa->wsk_nastepnika;
      else
        // a jeśli usuwamy coś w środku lub na końcu listy, to
        // tworzymy połączenie z pominięciem kasowanego elementu
        poprzedni->wsk_nastepnika = aktualny->wsk_nastepnika;
      // w obu przypadkach zapamiętujemy adres elementu do usuniecia
      tmp = aktualny;
      // przesuwamy się o jeden element dalej
      aktualny = aktualny->wsk_nastepnika;
      // ...i zwalniamy pamięć zajętą przez usuwany element
      delete tmp;
    }
    else
    {
      // jeśli liczba nie jest ujemna, to przechodzimy dalej i uaktualniamy adresy
      poprzedni = aktualny;
      aktualny = aktualny->wsk_nastepnika;
    };
  };
```

Zwróćcie uwagę, że usuwając elementy musieliśmy inaczej potraktować element początkowy (modyfikowaliśmy adres początku), a inaczej pozostałe elementy listy (trzeba było utworzyć powiązanie elementu poprzedniego z następnym względem aktualnego).

usuwanie elementu o ujemnej danej z listy jednokierunkowej



Przykładowa implementacja

Na koniec zamieszczamy prosty program, w którym zaimplementowaliśmy taką właśnie listę jednokierunkową. Prześledźcie sobie jego działanie.

```
1.  #include <iostream>
2.  #include <cstdlib>
3.
4.  using namespace std;
5.
6.  struct element_listy
7.  {
8.    int dane;
9.    element_listy *wsk_nastepnika;
10. };
11.
12.
13. // Iteracyjna funkcja drukująca zawartość listy
14. void drukuj_liste_it(element_listy *adres)
15. {
16.   while (adres!=NULL)
17.   {
18.     // wypisujemy zawartość elementu
19.     cout << (*adres).dane << " ";
20.     // i przechodzimy do następnego
21.     adres = adres->wsk_nastepnika;
22.   }
23. };
24.
25. // Rekurencyjna funkcja drukująca zawartość listy
26. void drukuj_liste_rek(element_listy *adres)
27. {
28.   // warunek zakończenia rekurencji
29.   if (adres != NULL)
30.   {
31.     // wypisujemy zawartość elementu
32.     cout << adres->dane << " ";
33.     // przechodzimy do następnego
```

```
34.       adres=adres->wsk_nastepnika;
35.       // i wywołujemy procedurę dla następnego elementu
36.       drukuj_liste_rek(adres);
37.     };
38.   };
39.
40.   // Funkcja wstawiająca element na listę
41.   void wstaw_element(element_listy *gdzie, element_listy *co)
42.   {
43.     element_listy *tmp;
44.     // zapamiętajmy element umieszczony za tym wskazywanym przez gdzie
45.     tmp = gdzie->wsk_nastepnika;
46.     // umieśćmy odwołanie do wstawianego
47.     gdzie->wsk_nastepnika = co;
48.     // i na koniec odtwórzmy dowiązanie do dotychczasowego natępnika
49.     co->wsk_nastepnika = tmp;
50.   };
51.
52.   int main(int argc, char *argv[])
53.   {
54.     // zmienne do obsługi listy - wszystkie to wskaźniki
55.     element_listy *glowa, *aktualny, *poprzedni, *tmp;
56.     // dodatkowe dane pomocnicze
57.     int dana, ile, ilew, i;
58.
59.     cout << "Program lista_1 - prezentacja listy jednokierunkowej\n\n";
60.
61.     // Tworzenie listy - kolejno wczytywane rekordy
62.     // dopisywane są na koniec listy
63.     cout << "0 konczy wpisywanie\n";
64.     aktualny = NULL;
65.     poprzedni = NULL;
66.     glowa = poprzedni;
67.     cout << "Podaj liczbe do wstawienia: ";
68.     cin >> dana;
69.     while (dana!=0)
70.     {
71.       // zapamiętujemy dotychczasowy koniec listy
72.       poprzedni = aktualny;
73.       // tworzymy nowy element
74.       aktualny = new element_listy;
75.       // zapisujemy do niego odczytane dane
76.       aktualny->dane = dana;
77.       // teraz jest to ostatni element listy
78.       aktualny->wsk_nastepnika = NULL;
79.       // natomiast poprzedni ostatni już nie jest ostatnim
80.       if (poprzedni != NULL) // Zabezpieczenie na początek
81.         poprzedni->wsk_nastepnika = aktualny;
82.       else
83.         glowa = aktualny;
84.
85.       // i odczytujemy nowe dane
86.       cout << "Podaj liczbe do wstawienia: ";
87.       cin >> dana;
88.     };
89.
90.     cout << "\nPo wprowadzeniu:\n";
91.     drukuj_liste_it(glowa);
92.
93.     // PRZEGLADANIE LISTY - sprawdzanie, ile na liście jest liczb większych od 5
94.     // i zliczanie wszystkich elementów listy
95.     ile = 0;
96.     ilew = 0;
97.     aktualny = glowa;
98.     while (aktualny!=NULL)
99.     {
100.      ile++;
101.      if (aktualny->dane > 5)
```

```
102.       ilew++;
103.     aktualny = aktualny->wsk_nastepnika;
104.   };
105.   cout << "\nLiczb większych od 5 jest " << ilew << endl;
106.
107.   // DOPISANIE ELEMENTU NA CZWARTEJ POZYCJI - jeśli na liście są już
108.   // przynajmniej trzy elementy
109.   if (ile>=3)
110.   {
111.     cout << "Podaj liczbe do wstawienia: ";
112.     aktualny = new element_listy;
113.     cin >> aktualny->dane;
114.     // przejdźmy do trzeciego elementu
115.     poprzedni = glowa;
116.     for (int i = 0; i < 2; i++)
117.       poprzedni = poprzedni->wsk_nastepnika;
118.
119.     // wstawmy element
120.     wstaw_element(poprzedni, aktualny);
121.
122.     aktualny = glowa;
123.     cout << "\nPo dopisaniu elementu na czwartej pozycji\n";
124.     // tym razem do wydruku wykorzystamy funkcję rekurencyjną
125.     drukuj_liste_rek(aktualny);
126.   }
127.
128.   // USUWANIE ELEMENTÓW mniejszych od zera
129.   aktualny=glowa;
130.   poprzedni= NULL;
131.   while (aktualny != NULL)
132.   {
133.     if (aktualny->dane < 0)
134.     {
135.       if (aktualny == glowa)
136.         // jeśli usuniemy glowę, to nowa glowa będzie o jeden element dalej
137.         glowa = glowa->wsk_nastepnika;
138.       else
139.         // a jeśli usuwamy coś w środku lub na końcu listy, to
140.         // tworzymy połączenie z pominięciem kasowanego elementu
141.         poprzedni->wsk_nastepnika = aktualny->wsk_nastepnika;
142.       // w obu przypadkach zapamietujemy adres elementu do usuniecia
143.       tmp = aktualny;
144.       // przesuwamy się o jeden element dalej
145.       aktualny = aktualny->wsk_nastepnika;
146.       // ...i zwalniamy pamięć zajętą przez usuwany element
147.       delete tmp;
148.     }
149.     else
150.     {
151.       // jeśli liczba nie jest ujemna, to przechodzimy dalej i uaktualniamy adresy
152.       poprzedni = aktualny;
153.       aktualny = aktualny->wsk_nastepnika;
154.     };
155.   };
156.
157.   cout << "\nPo usunieciu liczb ujemnych\n";
158.   drukuj_liste_it(glowa);
159.
160.
161.   // ZWALNIANIE PAMIECI
162.   aktualny = glowa;
163.   while (aktualny != NULL)
164.   {
165.     poprzedni=aktualny;
166.     aktualny=aktualny->wsk_nastepnika;
167.     delete poprzedni;
168.   }
169.
```

```
170.     cout << "\nPamiec zwolniona\n";
171.
172.     return 0;
173.   }
```

Sortowanie kubełkowe

Jako przykład zastosowania list jednokierunkowych zaprezentujemy jeszcze jedną metodę sortowania, której siła tkwi właśnie w wykorzystaniu list i dlatego możemy ją omówić dopiero teraz. Wcześniej jednak na chwilę powrócimy do sortowania przez scalanie, bo jak już wspominaliśmy, ta właśnie metoda ma swoje uzasadnienie w implementacji listowej.

MergeSort na listach dynamicznych

Wiemy już, że oprócz sortowania wewnętrznego istnieje również problem sortowania danych znajdujących się poza pamięcią podręczną komputera - a konkretnie zapisanych w plikach. Dostęp do nich jest wówczas sekwencyjny. Najczęściej konieczność zapisania danych w plikach występuje w przypadku przetwarzania bardzo dużych ilości danych - gdy zbiór danych, który chcemy posortować, nie mieści się w pamięci RAM - wtedy jesteśmy "zmuszeni" do wykorzystania pamięci "wolniejszej" - zazwyczaj jest nią twardy dysk komputera.

W przypadku właśnie takiego rodzaju dostępu do danych najczęściej używa się do ich sortowania struktury dynamicznych list jednokierunkowych. W pierwszym kroku dane są kopiowane z pliku do pamięci komputera, na listę dynamiczną. Następnie wywoływana jest procedura MergeSort działająca w analogiczny sposób do tej, która wykonywała sortowanie przez scalanie na tablicach (zgodnie z zasadą dziel i zwyciężaj):

- dzielimy listę na dwie listy - musimy ustawić ogon pierwszej listy "w środku" dużej listy, natomiast głowa drugiej listy będzie wskazywać na element następny po tymże ogonie
- wykonujemy rekurencyjnie sortowanie MergeSort dla pierwszej oraz dla drugiej listy - aż do uzyskania list jednoelementowych, które z natury rzeczy są posortowane; wtedy:
- finalnie scalamy obie posortowane listy procedurą Merge - działającą w analogiczny sposób, jak procedura używana do scalania dwóch posortowanych tablic - tylko ze zamiast kopiować elementy z poszczególnych komórek tablicy, następuje "przepinanie" wskaźników między poszczególnymi węzłami na liście.

Złożoność obliczeniowa dla powyższego sortowania na listach jednokierunkowych wynosi **O(n lg n)**. Dotyczy to wersji rekurencyjnej algorytmu - takiej, jaką przedstawiliśmy powyżej. Istnieje również możliwość zapisania omawianej metody sortowania bez użycia rekurencji - szybkość działania w takiej wersji w dużej mierze zależy od strategii podziału listy na mniejsze listy oraz jakości wykonania łączenia tych list.

BucketSort

Sortowanie kubełkowe (*BucketSort*) jest przykładem takiego algorytmu porządkującego dane, który działa w czasie liniowym. Do tej grupy algorytmów należy również sortowanie pozycyjne (*RadixSort*) oraz sortowanie przez zliczanie (*CountingSort*). W tej lekcji zajmiemy się tylko sortowaniem kubełkowym, jako najbardziej popularnym.

Naukowcy wykazali, że w przypadku pesymistycznego układu danych należy wykonać liczbę (n lgn) porównań, aby posortować zbiór złożony z n elementów. Z kolei w omawianych w lekcji 3 algorytmach sortowania (elementarnych i nieelementarnych) cała idea sortowania opierała się na porównywaniu elementów i przestawianiu ich (metody te określamy mianem algorytmów sortujących za pomocą porównań). Wniosek jest z tego taki, że nawet tak skuteczne techniki, jak *QuickSort* czy *MergeSort*, nie będą się wykonywać w czasie krótszym niż n lgn. Sortowanie kubełkowe może się wykonywać w czasie krótszym (nawet liniowym), gdyż nie opiera się wyłącznie na porównywaniu elementów.

BucketSort, bo taka jest powszechnie używana angielska nazwa sortowania kubełkowego, dla losowych danych porządkuje je w czasie liniowym. Jest to możliwe dzięki przyjęciu ściślejszych założeń co do danych, które chcemy uporządkować. Takim założeniem może być np. określenie przedziału wartości, do jakich należą liczby będące danymi wejściowymi - przykładowo mogą to być liczby całkowite z przedziału [a,b] lub liczby rzeczywiste z przedziału [0,1). Często są to liczby losowo wybierane zgodnie z rozkładem jednostajnym – wtedy sortowanie kubełkowe jest najskuteczniejsze.

Idea algorytmu *BucketSort* jest stosunkowo przejrzysta. Zakładamy, że mamy wiedzę o przedziale wartości, do którego należą elementy. Następnie wykonujemy poniższe czynności:

- dzielimy ten przedział liczbowy na m równych podprzedziałów, określanych potoczenie kubełkami (stąd nazwa metody)
- przenosimy liczby z sortowanego zbioru do właściwych kubełków

- dokonujemy operacji sortowania elementów wewnątrz podprzedziałów
- przekazujemy zawartości kubełków (w kolejności od „najmłodszego" do „najstarszego) do wynikowej struktury danych - w tym momencie całość danych została posortowana.

Do sortowania kubełkowego najczęściej wykorzystuje się listy, które pełnią rolę kubełków. W celu pokazania sposobu funkcjonowania algorytmu założymy teraz, że nasze dane wejściowe znajdują się w tablicy T[n]. Oprócz tej tablicy potrzebna jest również tablica wskaźników do początków list jednokierunkowych, które pełnią rolę kubełków. Do list tych będą zapisywane ("wrzucane" jak do kubełków) elementy tablicy T. Oznaczmy tę tablicę jako B[m], gdyż mamy stworzyć m kubełków.

Pseudokod sortowania kubełkowego będzie w tym przypadku miał następującą postać:

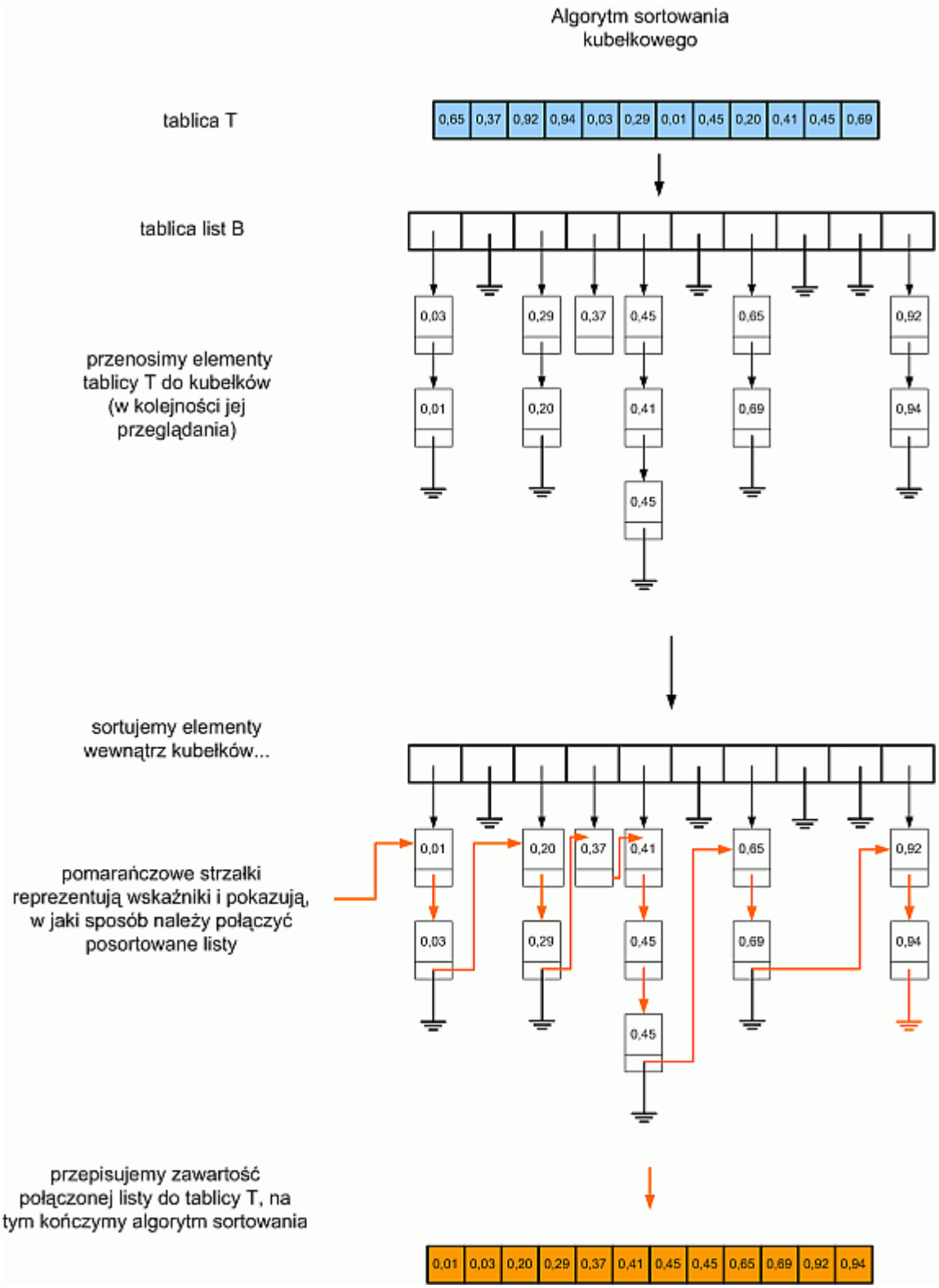
Procedura **BucketSort** (tablica **T, m**)

```
1. // n - rozmiar tablicy T, o indeksach od 0 do n-1
2. // m - liczba kubełków, rozmiar tablicy B, o indeksach od 0 do m-1
3. for (i=0;i< n; zwiększaj i o 1)
4.     wrzuć  T[i] do odpowiedniego kubełka - dołącz na koniec listy
5. for (i=0;i<m; zwiększaj i o 1)
6.     posortuj listę B[i] przez wstawianie
7. połącz listy B[0]..B[m-1] i przepisz zawartość tak połączonej listy do tablicy T
```

Na jakiej zasadzie algorytm przydziela elementy do właściwych kubełków? Bywa to o tyle ciekawie rozwiązane, że nie trzeba wykonywać zbędnych porównań. Jeśli na przykład liczby, które chcemy posortować, należą do przedziału [0,1), to możemy podzielić ten zakres wartości na 10 równych podprzedziałów: [0; 0.1), [0.1; 0.2) ... [0.9; 1). Wtedy element T[i] będzie w sposób natychmiastowy przenoszony do kubełka o indeksie $\lfloor 10*T[i] \rfloor$ (dla przypomnienia, $\lfloor \rfloor$ oznacza część całkowitą liczby; ilość kubełków m=10).

W powyższym pseudokodzie użyliśmy do sortowania wewnątrz kubełków metody *InsertionSort*, ale równie dobrze można użyć innej procedury, dobrze działającej na listach. Oprócz tego dosyć często stosuje się jeszcze jedną modyfikację algorytmu: elementy z tablicy T wstawia się nie na koniec listy, lecz na jej początek – nie wymaga to dostępu do końca listy. W takim wariancie zalecane jest przeglądanie tablicy T od końca, by nie utracić właściwości stabilności sortowania (**sortowanie** stabilne to takie, w którym elementy o tej samej wartości przed i po sortowaniu znajdują się w tym samym porządku względem siebie).

Polecamy zapoznanie się z poniższym rysunkiem, z pewnością rozwieje on wszelkie wątpliwości co do działania algorytmu *BucketSort*:



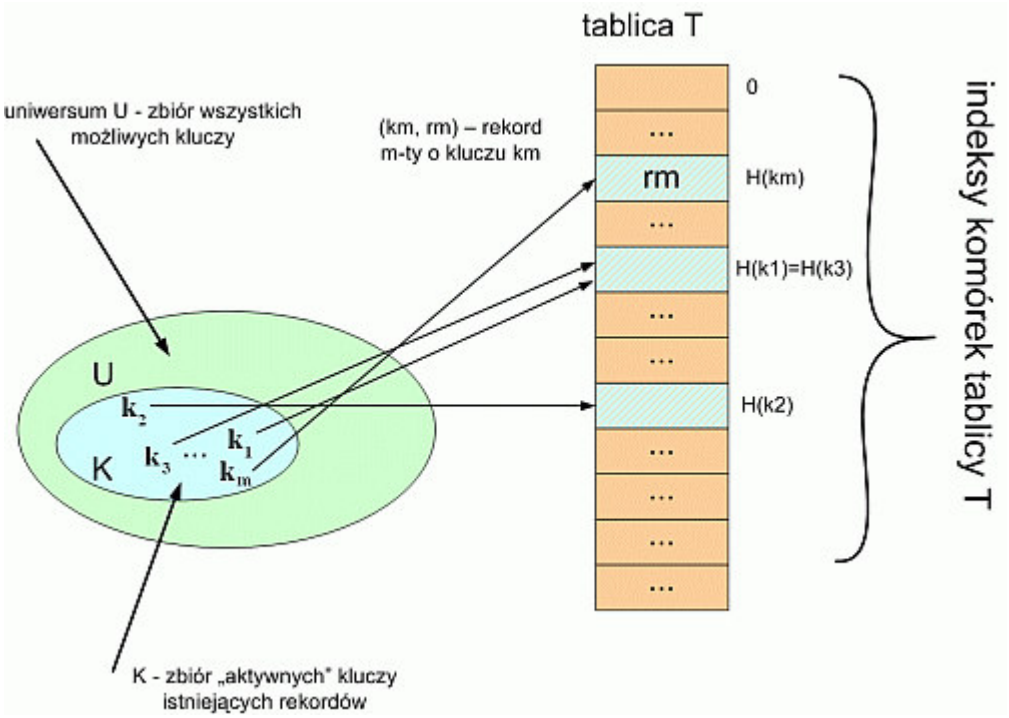
Tradycyjnie na koniec omawiania algorytmu sortowania przyjrzyjmy się jego złożoności obliczeniowej. Tworzenie tablicy B oraz łączenie list wymaga czasu O(m), natomiast przenoszenie elementów T do kubełków, a także przepisywanie zawartości listy posortowanej z powrotem do T wymaga czasu O(n). Wniosek – złożoność obliczeniowa całego algorytmu *BucketSort* wynosi **O(m+n)**, co najczęściej jest równoważne O(n) (gdyż m jest wielokrotnie mniejsze od n). Tym samym potwierdziliśmy, że algorytm sortowania kubełkowego rzeczywiście może się wykonywać w czasie liniowym.

Haszowanie

W tym rozdziale listy dynamiczne (i nie tylko) przydadzą się nam do innych zastosowań. Zajmiemy się problemem wyszukiwania elementów w sytuacji, gdy kluczowym ograniczeniem jest pojemność pamięci komputera, a istnieje ryzyko jej przekroczenia. Otóż wyobraźmy sobie, że liczba elementów pewnego zbioru danych (rekordów) jest znana i możliwa do zapisania w pamięci, natomiast teoretyczna liczba wszystkich możliwych do utworzenia elementów w danej przestrzeni absolutnie uniemożliwia zapamiętanie ich wszystkich. O zbiorze kluczy wszystkich możliwych rekordów będziemy dalej mówić jako o **uniwersum kluczy U**. Oprócz tego, jeśli liczba faktycznie wykorzystywanych kluczy byłaby tak mała w porównaniu z całą przestrzenia możliwych kluczy, to w takiej sytuacji duża część pamięci służąca do przechowywania elementów mogłaby się niepotrzebnie marnować – za sprawą dużych odległości między najbliższymi kluczami, których elementy byłyby zachowane w pamięci.

Z powodu wspomnianych powyżej problemów, wielu naukowców zajmujących się opracowywaniem nowych algorytmów możliwych do wdrożenia w praktyce zastanawiało się, jak uskutecznić operacje na danych (wyszukiwanie, wstawianie i usuwanie) przy wykorzystaniu mniejszej ilości pamięci. Doprowadziło to do stworzenia idei transformacji kluczowej, wykorzystującej funkcję mieszającą (ang. hashing). W skrócie polega ona na wyznaczeniu takiej **funkcji haszującej H**, która dla argumentu będącego kluczem elementu w zbiorze danych wyznaczy nam wartość oznaczająca indeks elementu w tablicy danych (którą dalej będziemy oznaczać jako tablicę **T** o rozmiarze **m**), pod którym znajduje się poszukiwany przez nas element. Jeśli pod tym indeksem nie znajduje się pożądany przez nas rekord, to oznacza, że nie znajduje się również nigdzie indziej i na tym etapie wyszukiwanie należy zakończyć. Jak widać, sama idea transformacji kluczowej jest stosunkowo prosta, jednak, jak łatwo się domyślić, niepozbawiona kilku istotnych wad.

W celu przybliżenia ogólnej zasady działania funkcji mieszającej przedstawiamy poniższy schemat:



Z rysunku wynika fakt, który obniża efektywność algorytmu. Jak wiemy, cały zbiór danych przewidzianych do zapisania w pamięci może być bardzo duży, a tablica, w której mamy zamiar zapisywać i wyszukiwać te dane, ma rozmiar znacznie mniejszy niż właśnie potencjalna liczba danych. Funkcja H nie jest funkcją różnowartościową. Dlatego może się zdarzyć, że wynikiem działania funkcji H na różnych argumentach kluczy będzie ta sama wartość (czyli ten sam indeks w tablicy T przechowującej dane).

W naszym przypadku , jeśli

$$\exists_{k_i,k_j,i\neq j} H(k_i) = H(k_j)$$

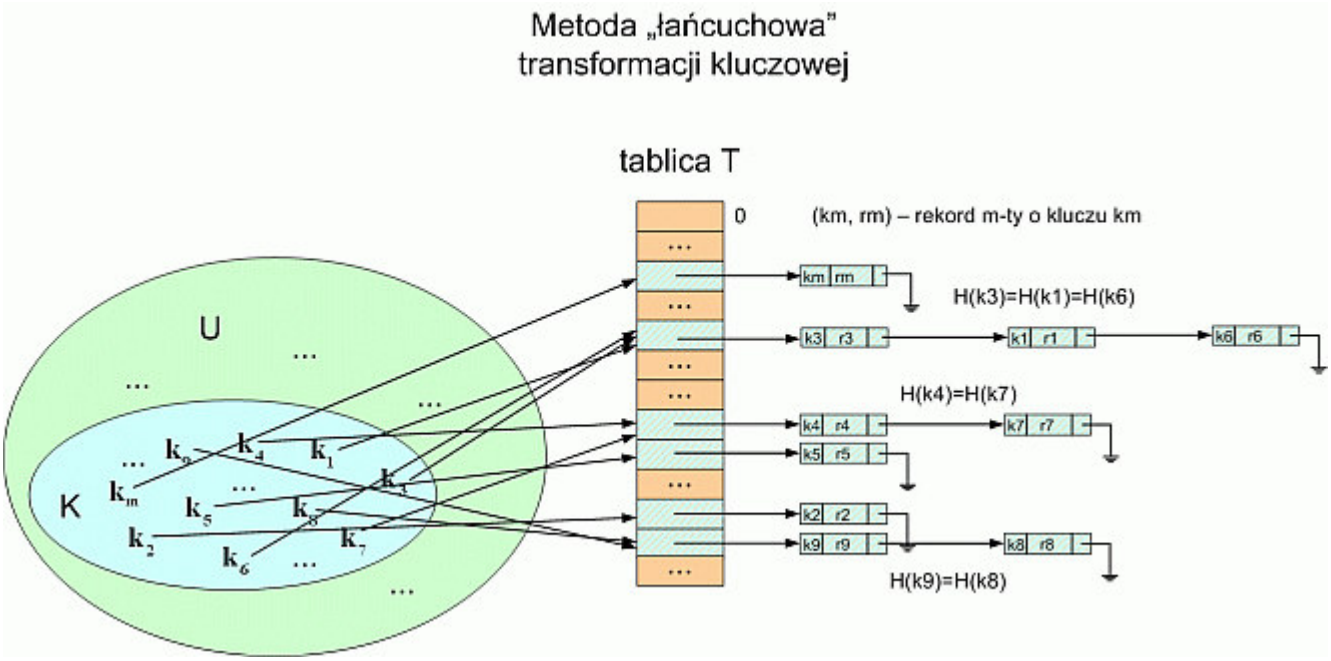
to oznacza, że funkcja haszująca działająca na tym konkretnym zbiorze danych powoduje **kolizje**.

Wydawałoby się, że najprostszą rzeczą, jak należałoby zrobić, jest takie sformułowanie algorytmu operacji wstawiania danych, które w wyniku powodowałoby unikanie kolizji. Jednak nie jest to do końca wykonalne, ponieważ odwzorowujemy przestrzeń danych z uniwersum na tablicę T o rozmiarze m, a wiemy że często liczność uniwersum jest o wiele większa od maksymalnej wartości m (która wynika z dostępnego miejsca w pamięci). Oczywiście ilość wystąpień kolizji zależy od odpowiedniego doboru funkcji haszującej. Wybranie takiej funkcji H, dla której przydzielanie indeksów (wynikających z wartości H(k)) w tablicy T kolejnym elementom będzie jak najbardziej równomierne (czyli T będzie równomiernie zapełniona), oznaczać będzie, że taka funkcja efektywnie „miesza" elementami z przestrzeni U i tym samym minimalizuje ilość możliwych wystąpień kolizji. Dobra funkcja H to taka, gdy dla typowego losowego zbioru danych kolizje zaczynają występować w momencie, gdy większość indeksów z T jest już wykorzystana. Niemniej

jednak nawet przy użyciu funkcji mieszającej najlepszej dla danego zbioru danych liczność uniwersum **#(U)>>m** , a więc niestety jesteśmy zmuszeni do radzenia sobie z sytuacją, gdy kolizja w przydzielaniu adresów przy użyciu haszowania jednak nastąpi.

Najpopularniejsza metoda polega na idei grupowania elementów i tworzenia list danych „podczepionych" pod ten indeks w tablicy T, który jest wspólną wartością funkcji mieszającej działającej na kluczach rekordów właśnie z tej listy. W ten sposób w tablicy umieszcza się nie konkretne rekordy danych z przestrzeni U, ale nagłówki list. Jeżeli zamierzamy przeszukiwać zbiór danych w celu stwierdzenia obecności konkretnego elementu, to po obliczeniu wartości funkcji haszującej jesteśmy zmuszeni do przeszukania tylko 1 listy. Wadą takiej metody jest wprowadzenie listowego zapisu danych, co zmusza do uważania na wszelkie pułapki, jakie się z tym wiążą.

W celu zobrazowania idei tworzenia list przy tabeli T przedstawiamy poniższy schemat:



Podstawowy problem polega na tym, by funkcja H dość równomiernie przydzielała indeksy tablicy T - a nie jest to łatwe przy danych, które mogą się okazać wybitnie nielosowe. W takim przypadku może się zdarzyć, że i owszem, dzięki metodzie łańcuchowej zareagujemy w odpowiedni sposób na kolizje, ale powstaną przez to bardzo długie listy – niewiele krótsze, niż gdybyśmy stosowali pojedynczą listę do przechowywania wszystkich danych. Zysk z zastosowania transformacji kluczowej w pesymistycznym przypadku okazałby się znikomy.

Warto więc zobaczyć, w jaki inny jeszcze sposób można rozbudować ideę haszowania. Otóż kolejna koncepcja polega na zrezygnowaniu z tworzenia list i pozostaniu przy deklaracji jedynie statycznej tablicy T.

W tej metodzie „dzielimy" T na 2 obszary:

- część podstawową, w której komórki są wypełniane zgodnie z wartością funkcji H
- część rezerwową, w której komórki są wypełniane liniowo (czyli po kolei) - do tej części trafiają rekordy, których wartość funkcji H wskazuje na już zapełnioną komórkę w części podstawowej – w celu uniknięcia kolizji umieszczamy takie rekordy właśnie w części rezerwowej tablicy T

Niestety, również taka metoda ma wyraźne wady. Głównym problemem jest oszacowanie rzędu liczności napływających danych oraz wyznaczenie odpowiedniej proporcji wielkości części podstawowej i rezerwowej. Możemy sobie przecież wyobrazić, że część podstawowa zostanie zapełniona, ale również i w części rezerwowej zabraknie już miejsca na zapisywanie danych. Co wtedy powinniśmy zrobić? Implementacja takiego algorytmu powinna posiadać tak dobrane rozmiary wydzielonej pamięci na dane, by właśnie do tak niebezpiecznej sytuacji nie dopuścić.

Jest jeszcze jedna grupa metod wypełniania tablicy T – adresowanie otwarte. W tej klasie wszystkie elementy są przechowywane w tablicy, beż użycia dołączanych list. Spośród najczęściej poszukiwanych metod tej grupy możemy przede wszystkim wymienić:

- adresowanie liniowe
- adresowanie kwadratowe
- haszowanie podwójne

Wszystkie wymienione powyżej działają w podobny sposób: w momencie wystąpienia kolizji obliczany jest nowy adres w tablicy, pod którym ma być zapisany nowy rekord. Zostaje wtedy użyta tzw. **funkcja przyrostu p(i)**, która zostaje dodana do podstawowej wartości funkcji haszującej H(k). Wartość i określa, ile razy próba wstawienia nowego elementu się nie powiodła – mianowicie, ile razy komórka w tabeli o wyznaczonym indeksie dla nowego rekordu okazywała się już zapełniona. Ogólny wzór na wartość indeksu tabeli T możemy określić wzorem:

$$H(k,i) = (H'(k) + p(i)) \bmod m$$

gdzie m oznacza rozmiar tablicy T.

Podane metody adresowania otwartego różnią się właśnie sposobem wyznaczania funkcji p(i). Przedstawiają się one następująco:

$$p(i) = \begin{cases} i & \textbf{adresowanie liniowe} \\ a_1 i + a_2 i^2 & \textbf{adresowanie kwadratowe} \\ iH''(k) & \textbf{haszowanie dwukrotne} \end{cases}$$

Funkcja H'' jest funkcją haszującą różną od H'.

Przyglądając się adresowaniu liniowemu stwierdzimy, że w momencie stwierdzenia kolizji algorytm używający tej metody zacznie sprawdzać kolejne komórki tablicy T - dopóki nie znajdzie pierwszej wolnej pozycji, nie zajętej przez wcześniej zapisany rekord. W ostateczności zostanie sprawdzona, komórka po komórce, cała tablica. Niestety takie rozwiązanie poszukiwania wolnego miejsca w tabeli posiada dużą wadę – nadmierne, nierównomierne grupowanie się zajętych pozycji. Jest znacznie większe prawdopodobieństwo, że dana komórka tabeli zostanie zapełniona, jeśli komórki „poprzedzające" ją zostały już wcześniej wypełnione, niż gdy bezpośrednio przed nią znajduje się wolne miejsce na zapisanie nowego rekordu – wtedy w pierwszej kolejności tam właśnie zostanie zapisany element o nowym kluczu. Łatwo jesteśmy sobie w stanie wyobrazić, że zaczynając od dwóch, trzech kolejnych zajętych komórek, ciąg ten może w szybkim tempie znacząco urosnąć. Zatem jest to cecha adresowania liniowego, która nie spełnia wymagań potrzebnych do równomiernego haszowania.

Adresowanie kwadratowe i haszowanie podwójne radzą sobie z tym problemem znacznie lepiej i dlatego też są nieporównywalnie częściej stosowane w praktyce (szczególnie używanie 2 funkcji haszujących).

Wymieniliśmy już kilka metod zastosowania transformacji kluczowej. Nie pokazaliśmy natomiast różnych przykładów funkcji haszującej. Teraz jest pora, by się tym zająć.

Do konstrukcji funkcji H oczywistą rzeczą wydaje się użycie funkcji matematycznych – zarówno pojedynczo, jak i w kombinacji tworzącej wspólnie jedną funkcję haszującą. Dowiedzieliśmy się już wcześniej, że cechą dobrej funkcji mieszającej jest równomierne przydzielanie indeksów w tablicy T (czyli wartości funkcji H) dla losowych danych. Chodzi o to, by tablica T była jak najbardziej równomiernie wypełniana.

Obecnie nie poszukuje się jednej idealnej funkcji H. Dobór optymalnej funkcji wynika z rodzaju danych, które mamy zapisać, a w szczególności z informacji o sposobie przydzielania im kluczy k.

Może się zdarzyć, że znamy rozkład prawdopodobieństwa przydzielania kluczy. W takiej sytuacji jesteśmy w stanie zastosować taką funkcję mieszającą, która pozwoli uzyskać bardzo dobre efekty rozmieszczenia danych w tablicy T. Oczywiście znacznie częściej nie znamy dokładnych informacji do sposobu generowania kluczy dla nowych danych. Okazuje się, że wtedy najlepszym wyjściem jest zastosowanie różnorodnych metod heurystycznych. Poprzez mniej lub bardziej złożone analizy jesteśmy w stanie z dobrym wyczuciem wybrać taką funkcję H, która będzie w tym konkretnym przypadku stosunkowo użyteczna.

Poniżej przedstawiamy przykłady kilku konkretnych typów funkcji haszujących:

- funkcja modulo
- funkcja mnożenia
- haszowanie uniwersalne

Funkcja modulo stosowana do haszowania przedstawia się prostym, intuicyjnym wzorem, przedstawionym poniżej:

$H(k)$ = **k mod m**, gdzie m jest rozmiarem tablicy T.

Przy takiej postaci funkcji H ważne jest dobranie odpowiedniego rozmiaru tablicy T, w celu skutecznego i w miarę losowego przydzielania komórek tablicy. Udowodniono, że należy unikać wartości m równym potęgom liczby 2, a do tego zalecane jest, by wartość określająca rozmiar tablicy T była liczbą pierwszą.

$H(k) = \lfloor m(A\ k \bmod 1) \rfloor$, gdzie A jest liczbą z przedziału (0,1).

Wartość funkcji H jest obliczana poprzez wyznaczenie części ułamkowej iloczynu A*k, następnie przemnożenie tej części przez m (rozmiar tablicy T) i następnie obliczenie części całkowitej końcowego iloczynu. W przypadku takiego wyboru funkcji mieszającej wybór parametru m jest znacznie bardziej dowolny niż w przypadku haszowania modularnego, aczkolwiek najczęściej m jest ustawiane jako potęga 2 - w celu łatwiejszej implementacji algorytmu w programie komputerowym.

Kolejnym problemem jest dobór stałej A. Mimo, że algorytm wykonuje się poprawnie dla dowolnej wartości, to wśród nich są takie, dla których „mieszanie" wartości funkcji H przebiega wyjątkowo sprawnie. Najpopularniejszą wartością A jest $(\sqrt{5}-1)/2$ znana z zasady „złotego podziału".

Ostatni wymieniony typ funkcji H nie różni się wzorem od pozostałych. Haszowanie uniwersalne jest stosowane wtedy, gdy możemy z dużą dozą prawdopodobieństwa stwierdzić, że dane napływające do algorytmu (a więc rekordy i ich klucze) będą wyjątkowe „nielosowe". W takiej sytuacji przydaje się właśnie podejście haszowania uniwersalnego, polegającego na losowym dobieraniu funkcji haszującej z pewnej grupy – wyniki przydzielania rekordów do komórek tablicy T są wtedy zaskakująco dobre, mianowicie liczba kolizji zostaje w ten sposób mocno ograniczona.

Kalkulator

Tym razem rozbudujemy nasz kalkulator o możliwość zapamiętywania zmiennych. Użytkownik będzie mógł w prosty sposób definiować swoje własne zmienne oraz przypisywać im wartości, a następnie z nich korzystać dokładnie tak, jak to się robi podczas programowania. Wykorzystamy w tym celu listę jednokierunkową.

Skoro będziemy chcieli wykorzystywać listę, musimy stworzyć typ danych, który będzie odpowiadał pojedynczemu elementowi. Zakładając, że możemy pamiętać jedynie zmienne typu rzeczywistego, do opisania każdej zmiennej wystarczą nam dwa pola: jej nazwa i wartość. Czyli definicja elementu listy będzie wyglądać następująco:

```
// Zdefiniujmy typ danych który będzie wykorzystywany do pamiętania zmiennych
// element listy
struct SZmienna
{
  // nazwa zmiennej
  string n;
  // wartość zmiennej
  double v;
  // wskaźnik do następnego elementu na liście
  SZmienna* nast;
};
```

Skoro lista ma być jedynie jednokierunkowa, wystarczy nam jedno pole w rekordzie wskazujące na następny element.

Następnym krokiem powinno być napisanie funkcji obsługujących listę. Tutaj możliwe są trzy podejścia:

1. Możemy nie tworzyć żadnych funkcji do obsługi listy, zamieszczając potrzebny do jej obsługi kod bezpośrednio w ciele podprogramów korzystających z niej. Lecz nie jest to eleganckie i odporne na błędy rozwiązanie. Zawsze powinniśmy się starać tak podzielić program na podprogramy, aby te ostatnie miały ściśle zdefiniowane i pojedyncze zadania.
2. Możemy również stworzyć zestaw maksymalnie uniwersalnych funkcji, pozwalających na swobodny dostęp do poszczególnych elementów listy. Rozwiązanie to jest godne polecenia, lecz bardzo pracochłonne.
3. My zastosujemy strukturę funkcji obsługujących listę zoptymalizowaną do określonego zadania, w naszym przypadku do obsługi kalkulatora.

Zastanówmy się więc, jakiej funkcjonalności od listy będą wymagać funkcje naszego kalkulatora. Załóżmy, że kalkulator ma umożliwiać wykonania przypisania w postaci:

nazwa_zmiennej = wyrażenie

Oraz że w wyrażeniu wszędzie tam, gdzie możemy wstawić wartość liczbową, możemy też wykorzystać zmienną. Przy czym załóżmy, żeby nie komplikować programu, że zmienne nie zdefiniowane wcześniej będą definiowane w momencie pierwszego wystąpienia i inicjowane wartością zerową. Czyli prawidłowy będzie zapis typu:

10*(*nazwa_zmiennej*-8)

Żeby spełnić powyższe wymagania, potrzebujemy funkcji odczytującej wartość zmiennej o danej nazwie z listy. Jeśli zmiennej o takiej nazwie na liście jeszcze nie ma, funkcja ma ją utworzyć oraz przypisać jej wartość 0. Przykładowa implementacja będzie wyglądała więc następująco:

```
double znajdz_zmienna(string n)
{
  // tymczasowy wskaźnik wykorzystywany do przeszukania listy
  SZmienna* t;
  // jeśli lista nie ma jeszcze elementów, musimy utworzyć pierwszy z nich
  if (poczatek == NULL)
  {
    poczatek = new SZmienna();
    poczatek->n = n;
    poczatek->nast = NULL;
    poczatek->v = 0.0;
    // w tym przypadku, skoro utworzyliśmy właśnie listę, nie ma co jej
    // przeszukiwać - możemy zwrócić wartość 0 i zakończyć procedurę
    return 0.0;
  };

  // zaczynamy szukać od początku
  t = poczatek;
  // przesuwamy się po liście aż do napotkania końca lub elementu o
  // nazwie podanej jako parametr
  while (t->nast && t->n != n)
    t = t->nast;

  // teraz musimy sprawdzić czy została znaleziona zmienna czy też napotkany
  // koniec listy. Jeśli jest to koniec listy, musimy stworzyć nowy element }
  if (t->nast == NULL)
  {
    // pozostaje nam jeszcze sprawdzić czy ostatni element nie jest przypadkiem
    // tym szukanym
    if (t->n != n)
    {
      // jeśli nie jest, musimy utworzyć nowy element, umieścić go na liście
      // i zaznaczyć że teraz on jest tym ostatnim
      t->nast = new SZmienna();
      t = t->nast;
      t->nast = NULL;
      t->n = n;
      t->v = 0;
    }
  }

  // teraz możemy być już pewni, że t wskazuje na element o nazwie n. Wystarczy
  // zwrócić jego wartość
  return t->v;
} // znajdz_zmienna
```

Drugim potrzebnym podprogramem będzie funkcja zmieniająca wartość pamiętaną w zmiennej o danej nazwie.

```
void zmien_wartosc(string n, double x)
{
  // wskaźnik pomocniczy
  SZmienna* t;
  // jak zwykle szukamy od początku
  t = poczatek;

  // główna pętla
```

```
  while (t)
  {
    // sprawdzamy czy nie znaleźliśmy elementu
    if (t->n == n)
    {
      // jeśli tak, to zmieniamy jego wartość
      t->v = x;
      // i opuszczamy procedurę
      return;
    }
    // przechodzimy do następnego elementu
    t = t->nast;
  };

  // skoro doszliśmy aż tutaj, to mamy problem - zmiennej nie ma na liście.
  // co prawda to nigdy się nie zdarzy (tak skonstruowaliśmy algorytm) lecz
  // dobrym stylem jest reakcja na takie błędy również
  string msg = "Zmiennej "+n+" nie ma na liście !";
  blad(msg);

}; // zmien_wartosc
```

Dodatkowo, zamieścimy jeszcze funkcję kasującą zawartość listy. Nie jest ona niezbędna, ponieważ nie przwidujemy kasowania listy zmiennych podczas pracy programu, a po jego zakończeniu i tak zawsze jest zwalniana cała pamięć przez niego zajmowana, lecz do kanonu dobrego stylu należy każdorazowe zwolnienie pamięci przeznaczonej na zmienne dynamiczne alokowane przez nas. Ubocznym zyskiem dla Was będzie prezentacja rekurencyjnej metody kasowania listy:

```
// Rekurencyjna procedura kasująca całą listę
void kasuj_liste(SZmienna* p)
{
  if (p)
  {
    kasuj_liste(p->nast);
    cout << "Kasuje zmienna " << p->n << endl;
    delete p;
  }
};
```

Procedura ta zawiera zupełnie niepotrzebną linijkę wypisującą nazwę kasowanej zmiennej. My ją zamieściliśmy, aby zaprezentować Wam pierwotnie najstarszą metodę śledzenia toku wykonania programu. Dzięki temu będziecie wiedzieli, że ta procedura naprawdę skasuje całą listę.

Pozostaje nam jeszcze zadeklarowanie wskaźnika do pierwszego elementu listy i problem przechowywania zmiennych mamy z głowy.

```
// głowa listy zawierającej nasze zmienne - inicjowana jako NULL
// przy starcie programu
SZmienna* poczatek = NULL;
```

Jak się zapewne domyślacie, należy jeszcze zmodyfikować kod procedur naszego analizatora składni tak, by zaczął rozumieć pojęcie zmiennej i nauczył się go obsługiwać. Pierwszą narzucającą się rzeczą jest konieczność rozszerzenia listy obsługiwanych symboli o nazwę zmiennej oraz operator przypisania. W tym celu po pierwsze - rozszerzamy listę symboli które obsługuje kalkulator:

```
enum TSymbol { sPLUS, sMINUS, sMNOZENIE, sDZIELENIE, sLN, sPN,
               sLICZBA, sKONIEC, sPRZYPISANIE, sNAZWA };
```

dodajemy do zmiennych globalnych modułu nową pozycję - będzie ona wykorzystywana do przechowywania nazwy ostatnio rozpoznanej zmiennej:

```
// nazwa ostatnio rozpoznanej zmiennej
string nazwa_zmiennej;
```

i modyfikujemy procedurę `daj_symbol` do postaci:

```
void daj_symbol()
{
  // długość symbolu
  int usunac_znakow = 0;
  // zmienna pomocnicza
  int tmp;

  /* najpierw usuwamy z poszątku wszystkie odstępy zwane niekiedy białymi
    spacjami.*/

  while (isspace(akt_wyrazenie[usunac_znakow]) && usunac_znakow < akt_wyrazenie.size())
    usunac_znakow++;

  akt_wyrazenie.erase(0, usunac_znakow);

  // zakładamy że do usunięcia będzie jedynie jeden znak
  usunac_znakow = 1;

  // jeśli wyrażenie się nam skończyło, bieżącym symbolem jest koniec
  if (akt_wyrazenie.empty())
  {
    biezacy_symbol = sKONIEC;
  // w przeciwnym wypadku
  } else {

    // rozpoznanie na podstawie pierwszego znaku
    switch (akt_wyrazenie[0])
    {
      case '+' : biezacy_symbol = sPLUS; break;
      case '-' : biezacy_symbol = sMINUS; break;
      case '*' : biezacy_symbol = sMNOZENIE; break;
      case '/' : biezacy_symbol = sDZIELENIE; break;
      case '(' : biezacy_symbol = sLN; break;
      case ')' : biezacy_symbol = sPN; break;
      case '=' : biezacy_symbol = sPRZYPISANIE; break;
      // jeśli jest to cyfra
      case '0' : case '1' : case '2' : case '3' : case '4' :
      case '5' : case '6' : case '7' : case '8' : case '9' :
        /* konwertujemy napis na liczbę korzystając z funkcji bibliotecznej
          strtod. W przypadku jej wykorzystania, drugi argument funkcji będzie
          zawierał wskaźnik do znaku na którym funkcja skończyła przetwarzanie.
          dzięki temu dowiemy się jaka jest długość liczby */
        char *koniec;
        wartosc_liczby = strtod(akt_wyrazenie.c_str(), &koniec);
        biezacy_symbol = sLICZBA;
        /* w C i C++ wartości wskaźników można dodawać i odejmować, więc
         długość odczytanej liczby obliczamy następująco: */
        usunac_znakow = koniec - akt_wyrazenie.c_str();
      break;
      // teraz już wiemy że jest to nazwa lub błąd
      default:
        // spróbujmy więc odczytać nazwę przy założeniu, że zaraz po niej
        // musi wystąpić spacja lub jeden z operatorów
        size_t tmp = akt_wyrazenie.find_first_of("'+-*/()= ");
        nazwa_zmiennej = akt_wyrazenie.substr(0, tmp);
        usunac_znakow = tmp;
        biezacy_symbol = sNAZWA;
      break;
    }

    // na koniec usunięcie rozpoznanego symbolu z wyrażenia
    akt_wyrazenie.erase(0, usunac_znakow);
  }; // koniec else
}; // koniec funkcji
```

Rozpoznanie faktu istnienia zmiennej w naszym kodzie to jeszcze nie wszystko. Musimy również tą informację wykorzystać - czyli wrowadzić zmiany do funkcji `czynnik`

```
double czynnik()
{
  // zmienna pomocnicza
  double tmp;
  // na początek przypisujemy bezpieczną wartość czynnika
  double wynik = 1.0;
  // następnie w zależności od bieżącego symbolu
  switch (biezacy_symbol)
  {
    // jeśli jest to liczba
    case sLICZBA :
      // odczytujemy następny czynnik
      daj_symbol();
      // i zwracamy jej wartość
      wynik = wartosc_liczby;
    break;

    // jeśli jest to zmienna
    case sNAZWA :
      // odczytujemy wartość z listy
      wynik = znajdz_zmienna(nazwa_zmiennej);
      daj_symbol();
      // jeśli wystąpił znak przypisania
      if (biezacy_symbol == sPRZYPISANIE)
      {
        // zapamiętujemy nazwę zmiennej
        string nazwa = nazwa_zmiennej;
        // czytamy następny symbol
        daj_symbol();
        // obliczamy wartość wyrażenia
        tmp = wyrazenie();
        // zapamiętujemy ją pod podaną nazwą
        zmien_wartosc(nazwa, tmp);
        // i zwracamy obliczoną wartość
        wynik = tmp;
      };
    break;

    // jeśli jest to minus jednoargumentowy
    case sMINUS :
      // odczytujemy następny czynnik
      daj_symbol();
      // i obliczamy wartość
      wynik  = -czynnik();
    break;

    // jeśli jest to lewy nawias (otwierający)
    case sLN :
      // odczytujemy następny czynnik (w ten sposób pozbyliśmy się nawiasu
      // otwierającego)
      daj_symbol();
      // obliczamy wartość wyrażenia w nawiasie
      tmp = wyrazenie();
      // jeśli po tym obliczeniu nie napotkamy nawiasu zamykającego
      if (biezacy_symbol != sPN)
      {
        // to musimy zgłosić błąd
        blad("Spodziewany prawy nawias");
      } else { // w przeciwnym wypadku
        // zwracamy wartość wyrażenia w nawiasie
        wynik = tmp;
        // i odczytujemy następny czynnik
        daj_symbol();
      };
```

```
    break;

    // jeśli to koniec wyrażenia, to zgłaszamy błąd !
    case sKONIEC :
      blad("Nieoczekiwany koniec wyrazenia !");
    break;

    // jeśli nie napotkaliśmy żadnego z wymienionych symboli
    // wyrażenie zawiera błąd składniowy
    default:
      blad("Spodziewany czynnik");
  };

  return wynik;
};
```

I to wszystko ... Zobaczcie jak niewiele kodu jest czasem potrzeba, aby uzyskać zupełnie nową jakość :)

Poniżej zamieściliśmy pełen kod pliku nagłówkowego i implementacji modułu parsera oraz nieznacznie zmodyfikowany program główny. Pozostałe moduły nie zostały zmienione, więc nie zamieszczamy ich tutaj.

Nagłówek modułu parsera:

```
1.  #ifndef kalk_parserH
2.  #define kalk_parserH
3.
4.  #include <string>
5.
6.  using namespace std;
7.
8.  // deklaracja zmiennej zawierającej opisy błędów występujących w wyrażeniu
9.  extern string blad_opis;
10. // deklaracja zmiennej zawierającej liczba błędów które wystąpiły podczas
11. // interpetacji
12. extern int blad_l;
13.
14. /* pierwsza funkcja eksportowana będzie obliczać wartość wyrażenia
15.   przekazywanego jej jako parametr w. Wynik będzie zamieszczony w parametrze
16.   v, natomiast funkcja będzie zwracać liczbę błędów napotkanych w wyrażeniu */
17. int policz_wyrazenie(string w, double& v);
18.
19. /* druga funkcja będzie wywoływana przed zakończeniem programu aby skasować
20.   zawartość listy */
21. void czysc_liste();
22.
23.
24. #endif
```

Implementacja parsera

```
1.  #include <iostream>
2.
3.  #pragma hdrstop
4.
5.  #include "kalk_parser.h"
6.
7.  #pragma package(smart_init)
8.
9.
10. // Zdefiniujmy typ danych który będzie wykorzystywany do pamiętania zmiennych
11. // element listy
12. struct SZmienna
13. {
14.   // nazwa zmiennej
15.   string n;
16.   // wartość zmiennej
```

```
17.   double v;
18.   // wskaźnik do następnego elementu na liście
19.   SZmienna* nast;
20. };
21. 
22. 
23. // zmienna zawierająca opisy błędów występujących w wyrażeniu
24. string blad_opis;
25. // liczba błędów które wystąpiły podczas interpetacji
26. int blad_l;
27. 
28. 
29. // Zdefiniujemy sobie typ wyliczeniowy który będzie opisywał rozpoznawane
30. // przez nasz kalkulator symbole
31. enum TSymbol { sPLUS, sMINUS, sMNOZENIE, sDZIELENIE, sLN, sPN,
32.        sLICZBA, sKONIEC, sPRZYPISANIE, sNAZWA } ;
33. 
34. // definicje zmiennych lokalnych, tzn. widocznych dla wszystkich porcedur w
35. //  module natomiast niewidoczne dla reszty programu
36. 
37. // Aktualny symbol
38. TSymbol biezacy_symbol;
39. // przetwarzane wyrażenie
40. string akt_wyrazenie;
41. // wartość liczbowa symbolu (o ile jest on liczbą)
42. double wartosc_liczby;
43. // głowa listy zawierającej nasze zmienne - inicjowana jako NULL
44. // przy starcie programu
45. SZmienna* poczatek = NULL;
46. // nazwa ostatnio rozpoznanej zmiennej
47. string nazwa_zmiennej;
48. 
49. // funkcja zapewniająca obsługę błędów
50. void blad(string s)
51. {
52.   blad_l = blad_l + 1;
53.   if (blad_opis == "")
54.     blad_opis = s;
55.   else
56.     blad_opis += string(" | ") + s;
57. }
58. 
59. // Trzy funkcje których celem jest obsługa listy zmiennych
60. 
61. /* Najpierw funkcja umieszczająca daną zmienną na liście, lub odczytująca
62.   jej wartość, jeśli zmienna owa już na niej jest. Przekazywana do niej jest
63.   nazwa zmiennej. */
64. double znajdz_zmienna(string n)
65. {
66.   // tymczasowy wskaźnik wykorzystywany do przeszukania listy
67.   SZmienna* t;
68.   // jeśli lista nie ma jeszcze elementów, musimy utworzyć pierwszy z nich
69.   if (poczatek == NULL)
70.   {
71.     poczatek = new SZmienna();
72.     poczatek->n = n;
73.     poczatek->nast = NULL;
74.     poczatek->v = 0.0;
75.     // w tym przypadku, skoro utworzyliśmy właśnie listę, nie ma co jej
76.     // przeszukiwać - możemy zwrócić wartość 0 i zakończyć procedurę
77.     return 0.0;
78.   };
79. 
80.   // zaczynamy szukać od początku
81.   t = poczatek;
82.   // przesuwamy się po liście aż do napotkania końca lub elementu o
83.   // nazwie podanej jako parametr
84.   while (t->nast && t->n != n)
```

```
85.       t = t->nast;
86.
87.     // teraz musimy sprawdzić czy została znaleziona zmienna czy też napotkany
88.     // koniec listy. Jeśli jest to koniec listy, musimy stworzyć nowy element }
89.     if (t->nast == NULL)
90.     {
91.       // pozostaje nam jeszcze sprawdzić czy ostatni element nie jest przypadkiem
92.       // tym szukanym
93.       if (t->n != n)
94.       {
95.         // jeśli nie jest, musimy utworzyć nowy element, umieścić go na liście
96.         // i zaznaczyć że teraz on jest tym ostatnim
97.         t->nast = new SZmienna();
98.         t = t->nast;
99.         t->nast = NULL;
100.        t->n = n;
101.        t->v = 0;
102.      }
103.    }
104.
105.    // teraz możemy być już pewni, że t wskazuje na element o nazwie n. Wystarczy
106.    // zwrócić jego wartość
107.    return t->v;
108.  } // znajdz_zmienna
109.
110.  // Funkcja zmieniająca wartość pamiętaną na liście
111.  void zmien_wartosc(string n, double x)
112.  {
113.    // wskaźnik pomocniczy
114.    SZmienna* t;
115.    // jak zwykle szukamy od początku
116.    t = poczatek;
117.
118.    // główna pętla
119.    while (t)
120.    {
121.      // sprawdzamy czy nie znaleźliśmy elementu
122.      if (t->n == n)
123.      {
124.        // jeśli tak, to zmieniamy jego wartość
125.        t->v = x;
126.        // i opuszczamy procedurę
127.        return;
128.      }
129.      // przechodzimy do następnego elementu
130.      t = t->nast;
131.    };
132.
133.    // skoro doszliśmy aż tutaj, to mamy problem - zmiennej nie ma na liście.
134.    // co prawda to nigdy się nie zdarzy (tak skonstruowaliśmy algorytm) lecz
135.    // dobrym stylem jest reakcja na takie błędy również
136.    string msg = "Zmiennej "+n+" nie ma na liście !";
137.    blad(msg);
138.
139.  }; // zmien_wartosc
140.
141.  // Rekurencyjna procedura kasująca całą listę
142.  void kasuj_liste(SZmienna* p)
143.  {
144.    if (p)
145.    {
146.      kasuj_liste(p->nast);
147.      cout << "Kasuje zmienna " << p->n << endl;
148.      delete p;
149.    }
150.  };
151.
152.  // mając procedurę kasowania listy możemy zamieścić kod procedury czyszczącej
```

```
153. void czysc_liste()
154. {
155.   kasuj_liste(poczatek);
156.   poczatek = NULL;
157. }
158. 
159. /* Nasz parser będzie się składał z trzech funkcji dokonujących analizy
160.   składniowej, jednej dokonującej rozpoznania symboli oraz głównej funkcji
161.   będącej "oknem na świat" modułu. Dodatkowo zamieścimy procedurę
162.   zapamiętującą napotkane błędy */
163. 
164. /* funkcja dekodująca ciąg znaków na symbole. Pobiera ona symbol
165.   znajdujący się na początku łańcucha akt_wyrazenie, rozpoznaje go,
166.   a następnie umieszcza jego typ w zmiennej biezacy_symbol, wartość
167.   liczbową (o ile posiada takową) w zmiennej wartosc_liczby oraz
168.   __usuwa symbol z łańcucha__ */
169. void daj_symbol()
170. {
171.   // długość symbolu
172.   int usunac_znakow = 0;
173.   // zmienna pomocnicza
174.   int tmp;
175. 
176.   /* najpierw usuwamy z poszątku wszystkie odstępy zwane niekiedy białymi
177.     spacjami.*/
178. 
179.   while (isspace(akt_wyrazenie[usunac_znakow]) && usunac_znakow < akt_wyrazenie.size())
180.     usunac_znakow++;
181. 
182.   akt_wyrazenie.erase(0, usunac_znakow);
183. 
184.   // zakładamy że do usunięcia będzie jedynie jeden znak
185.   usunac_znakow = 1;
186. 
187.   // jeśli wyrażenie się nam skończyło, bieżącym symbolem jest koniec
188.   if (akt_wyrazenie.empty())
189.   {
190.     biezacy_symbol = sKONIEC;
191.   // w przeciwnym wypadku
192.   } else {
193. 
194.     // rozpoznanie na podstawie pierwszego znaku
195.     switch (akt_wyrazenie[0])
196.     {
197.       case '+' : biezacy_symbol = sPLUS; break;
198.       case '-' : biezacy_symbol = sMINUS; break;
199.       case '*' : biezacy_symbol = sMNOZENIE; break;
200.       case '/' : biezacy_symbol = sDZIELENIE; break;
201.       case '(' : biezacy_symbol = sLN; break;
202.       case ')' : biezacy_symbol = sPN; break;
203.       case '=' : biezacy_symbol = sPRZYPISANIE; break;
204.       // jeśli jest to cyfra
205.       case '0' : case '1' : case '2' : case '3' : case '4' :
206.       case '5' : case '6' : case '7' : case '8' : case '9' :
207.         /* konwertujemy napis na liczbę korzystając z funkcji bibliotecznej
208.           strtod. W przypadku jej wykorzystania, drugi argument funkcji będzie
209.           zawierał wskaźnik do znaku na którym funkcja skończyła przetwarzanie.
210.           dzięki temu dowiemy się jaka jest długość liczby */
211.         char *koniec;
212.         wartosc_liczby = strtod(akt_wyrazenie.c_str(), &koniec);
213.         biezacy_symbol = sLICZBA;
214.         /* w C i C++ wartości wskaźników można dodawać i odejmować, więc
215.          długość odczytanej liczby obliczamy następująco: */
216.         usunac_znakow = koniec - akt_wyrazenie.c_str();
217.       break;
218.       // teraz już wiemy że jest to nazwa lub błąd
219.       default:
```

```
220.          // spróbujmy więc odczytać nazwę przy założeniu, że zaraz po niej
221.          // musi wystąpić spacja lub jeden z operatorów
222.          size_t tmp = akt_wyrazenie.find_first_of("'+-*/()= ");
223.          nazwa_zmiennej = akt_wyrazenie.substr(0, tmp);
224.          usunac_znakow = tmp;
225.          biezacy_symbol = sNAZWA;
226.        break;
227.      }
228.
229.      // na koniec usunięcie rozpoznanego symbolu z wyrażenia
230.      akt_wyrazenie.erase(0, usunac_znakow);
231.    }; // koniec else
232.  }; // koniec funkcji
233.
234.  /* ponieważ mamy do czynienia z rekursją pośrednią (funkcja wyrażenie wywołuje
235.    pośerdnio funkcję czynnik, która znowuż może wywołać funkcję wyrażenie)
236.    musimy poinformować kompilator, że gdzieś dalej będzie zdefiniowana funkcja
237.    wyrażenie */
238.  double wyrazenie();
239.
240.  // Obliczenie wartości czynnika
241.  double czynnik()
242.  {
243.    // zmienna pomocnicza
244.    double tmp;
245.    // na początek przypisujemy bezpieczną wartość czynnika
246.    double wynik = 1.0;
247.    // następnie w zależności od bieżącego symbolu
248.    switch (biezacy_symbol)
249.    {
250.      // jeśli jest to liczba
251.      case sLICZBA :
252.        // odczytujemy następny czynnik
253.        daj_symbol();
254.        // i zwracamy jej wartość
255.        wynik = wartosc_liczby;
256.      break;
257.
258.      // jeśli jest to zmienna
259.      case sNAZWA :
260.        // odczytujemy wartość z listy
261.        wynik = znajdz_zmienna(nazwa_zmiennej);
262.        daj_symbol();
263.        // jeśli wystąpił znak przypisania
264.        if (biezacy_symbol == sPRZYPISANIE)
265.        {
266.          // zapamiętujemy nazwę zmiennej
267.          string nazwa = nazwa_zmiennej;
268.          // czytamy następny symbol
269.          daj_symbol();
270.          // obliczamy wartość wyrażenia
271.          tmp = wyrazenie();
272.          // zapamiętujemy ją pod podaną nazwą
273.          zmien_wartosc(nazwa, tmp);
274.          // i zwracamy obliczoną wartość
275.          wynik = tmp;
276.        };
277.      break;
278.      // jeśli jest to minus jednoargumentowy
279.      case sMINUS :
280.        // odczytujemy następny czynnik
281.        daj_symbol();
282.        // i obliczamy wartość
283.        wynik  = -czynnik();
284.      break;
285.
286.      // jeśli jest to lewy nawias (otwierający)
287.      case sLN :
```

```
288.       // odczytujemy następny czynnik (w ten sposób pozbyliśmy się nawiasu
289.       // otwierającego)
290.       daj_symbol();
291.       // obliczamy wartość wyrażenia w nawiasie
292.       tmp = wyrazenie();
293.       // jeśli po tym obliczeniu nie napotkamy nawiasu zamykającego
294.       if (biezacy_symbol != sPN)
295.       {
296.         // to musimy zgłosić błąd
297.         blad("Spodziewany prawy nawias");
298.       } else { // w przeciwnym wypadku
299.         // zwracamy wartość wyrażenia w nawiasie
300.         wynik = tmp;
301.         // i odczytujemy następny czynnik
302.         daj_symbol();
303.       };
304.     break;
305.
306.     // jeśli to koniec wyrażenia, to zgłaszamy błąd !
307.     case sKONIEC :
308.       blad("Nieoczekiwany koniec wyrazenia !");
309.     break;
310.
311.     // jeśli nie napotkaliśmy żadnego z wymienionych symboli
312.     // wyrażenie zawiera błąd składniowy
313.     default:
314.       blad("Spodziewany czynnik");
315.   };
316.
317.   return wynik;
318. };
319.
320. // Funkcja wykonuje mnożenie i dzielenie
321. double skladnik()
322. {
323.   //przydatne zmienne tymczasowe
324.   double lewa, dzielnik;
325.   bool koniec;
326.   // mnożymy przez siebie dwa czynniki. Więc odczytajmy najpierw pierwszy z
327.   // nich
328.   lewa = czynnik();
329.
330.   // następnie wchodzimy w pętlę, którą opuścimy dopiero po wykonaniu
331.   // wszystkich mnożeń i dzieleń na tym poziomie
332.   do
333.   {
334.     // w zależności od tego jaki jest bieżący symbol
335.     switch (biezacy_symbol)
336.     {
337.
338.       // jesli jest to mnożenie
339.       case sMNOZENIE :
340.         // odczytujemy następny symbol
341.         daj_symbol();
342.         // wykonujemy mnożenie
343.         lewa *= czynnik();
344.       break;
345.
346.       // jeśli to dzielenie
347.       case sDZIELENIE :
348.         // odczytujemy następny symbol
349.         daj_symbol();
350.         // najpierw obliczamy dzielnik
351.         dzielnik = czynnik();
352.         // jeśli dzielnik = 0
353.         if (dzielnik == 0)
354.         {
355.           // no to mamy błąd. Powiadommy o tym użytkownika
```

```
356.            blad("Dzielenie przez 0");
357.            // i przypiszmy dzielnikowi wartość neutralną
358.            dzielnik  = 1.0;
359.          }
360.          // wykonujemy dzielenie
361.          lewa /= dzielnik;
362.        break;
363.
364.        // jeśli natomiast nie było to ani dzielenie ani mnożenie, to nie mamy
365.        // już tu nic do roboty. Więc opuszczamy funkcję przypisując wynik
366.        default:
367.          return lewa;
368.      };
369.    }
370.    while (true); // przykład pętli bez końca
371.
372.  };
373.
374.  // Dodawanie i odejmowanie
375.  double wyrazenie()
376.  {
377.    // przydatne zmienne tymczasowe
378.    double lewa;
379.    // dodajemy / odejmujemy dwa składniki. Policzmy więc pierwszy z nich
380.    lewa = skladnik();
381.    // i wchodzimy w pętlę wykonującą wszystkie dodawania i odejmowania na
382.    // danym poziomie
383.    while (true)
384.    {
385.      // w zależności od bieżącego symbolu
386.      switch (biezacy_symbol)
387.      {
388.        // jeśli jest to dodawanie
389.        case sPLUS :
390.          // odczytujemy następny symbol
391.          daj_symbol();
392.          // wykonujemy dodawanie, obliczając "w locie" drugi składnik
393.          lewa += skladnik();
394.        break;
395.
396.        // jeśli to odejmowanie
397.        case sMINUS :
398.          // odczytujemy następny symbol
399.          daj_symbol();
400.          // i wykonajmy odejmowanie
401.          lewa -= skladnik();
402.        break;
403.        // jeśli natomiast nie było to ani dodawanie ani odejmowanie, to nie mamy
404.        // już tu nic do roboty. Więc opuszczamy funkcję przypisując wynik
405.        default:
406.          return lewa;
407.      }
408.    };
409.  }
410.
411.  // główna funkcja modułu
412.  int policz_wyrazenie(string w, double &v)
413.  {
414.    // zainicjujmy najpierw obsługę błędów w naszym module
415.    blad_l = 0;
416.    blad_opis = "";
417.
418.    // następnie przepiszmy do zmiennej lokalnej obliczane wyrażenie
419.    akt_wyrazenie = w;
420.    // zainicjujmy parser (pobierając pierwszy symbol)
421.    daj_symbol();
422.    // wykonanie obliczeń
423.    v = wyrazenie();
```

```
424.
425.     if (biezacy_symbol != sKONIEC)
426.       blad("Nierozpoznane znaki na koncu wyrazenia !");
427.
428.     // zwracamy liczbę błędów
429.     return blad_l;
430.   };
```

Główny program (jedyna zmiana to dodanie funkcji kasującej listę)

```
  1. #include <iostream>
  2. #include <cstdlib>
  3.
  4. #include "kalk_6_bibl.h"
  5. #include "kalk_6_iu.h"
  6. #include "kalk_parser.h"
  7.
  8.
  9. //*************************************************************************
 10. //
 11. //  Program główny
 12. //
 13. //*************************************************************************
 14.
 15. int main(int argc, char* argv[])
 16. {
 17.   // argumenty
 18.   SArgument x, y;
 19.   // Zmienna przechowująca wynik obliczeń
 20.   SArgument w;
 21.   // Zmienna przechowująca wybranie działanie
 22.   char dzialanie;
 23.   // Zmienne sterujące pracą programu
 24.   char wybor;
 25.   // Zmienna zawierająca wprowadzone wyrażenie
 26.   string wyr;
 27.
 28.   cout << "Program Kalkulator v. 6\n";
 29.
 30.   // Główna pętla progamu. Będzie się wykonywała dopóki użytkownik nie wybierze
 31.   // opcji "Koniec"
 32.   do
 33.   {
 34.     cout << endl;
 35.
 36.     do
 37.     {
 38.       cout << "Wybierz typ interfejsu [k-klasyczny, w-wyrazenie]: ";
 39.       cin >> wybor;
 40.       wybor = toupper(wybor);
 41.     }
 42.     while (wybor != 'K' && wybor != 'W');
 43.
 44.     if (wybor == 'W')
 45.     {
 46.
 47. // nowa wersja
 48.
 49.       // Zakładamy, że pętla ma zostać powtórzona w przypadku niepowodzenia w
 50.       // obliczeniach
 51.       wybor = 'n';
 52.
 53.       // Wczytajmy wyrażenie do obliczenia
 54.       cout << "\nWprowadz wyrazenie zawierajace liczby, operatory i nawiasy:\n";
 55.       // opróżniamy bufor przed wczytywaniem wyrażenia
 56.       cin.ignore();
 57.       // wczytujemy całą linię ze spacjami
```

```
58.       getline(cin, wyr);
59.
60.       // policzmy jego wartość od razu dokonując kontroli parametrów
61.       if (policz_wyrazenie(wyr, w.v[1].re) != 0)
62.       {
63.         // wystąpiły błędy w wyrażeniu
64.         cout << "Blady w wyrazeniu:\n";
65.         cout << blad_opis << endl;
66.       }
67.       else
68.       {
69.         // nie było błędów
70.         cout << wyr << " = " << w.v[1].re << endl;
71.       };
72.
73.     }
74.     else
75.     {
76.
77. // stara wersja
78.
79.       // Zakładamy, że pętla ma zostać powtórzona w przypadku niepowodzenia w
80.       // obliczeniach
81.       wybor = 'n';
82.
83.       // Do wprowadzenia argumentu wykorzystamy funkcję czytaj_argument
84.       if (!czytaj_argument(x))
85.         // jak wczytanie się nie powiodło - wracamy na początek pętli
86.         continue;
87.
88.       // Wczytanie wybranego działania. Kropka oznacza iloczyn skalarny
89.       cout << "Podaj dzialanie (+ - * / .): ";
90.       cin >> dzialanie;
91.
92.       // Wczytanie drugiego argumentu
93.       if (!czytaj_argument(y))
94.         // jak wczytanie się nie powiodło - wracamy na początek pętli
95.         continue;
96.
97.       cout << endl;
98.       // Wykonanie żądanej operacji - także wykorzystamy funkcję
99.       if (!policz(w, x, y, dzialanie))
100.        // jak obliczenia się nie powiodły - wracamy na początek pętli
101.        continue;
102.
103.      // wyświetlenie wyniku
104.      pisz_wynik(w,x,y,dzialanie);
105.    };
106.
107.    // zadajmy użytkownikowy pytanie, czy chce już zakończyć pracę z programem
108.    cout << "\nZakonczyc prace programu (n - nie, inny klawisz - tak) ? ";
109.
110.    // wczytajmy jego odpowiedź. Wykorzystamy ją do sprawdzenia warunku wyjścia
111.    // z pętli
112.    cin >> wybor;
113.
114.  // koniec pętli. Jeśli użytkownik wybrał 'n' - wracamy na jej początek, w
115.  // przeciwnym wypadku - opuszczamy pętlę i kończymy program
116.  }
117.  while (wybor == 'n' || wybor == 'N');
118.
119.  czysc_liste();
120.
121.  return 0;
122. }
```

Zadania

Zadania do lekcji 4:

Napisać programy, które realizują następujące zadania dla komputera:

1. Utworzyć listę jednokierunkową z 4 wczytanych liczb, w kolejności odwrotnej do wczytywania, po czym znaleźć największy co do wartości pola liczbowego element na tej liście.

```
#include <cstdlib>
#include <iostream>

using namespace std;


// lista będzie złożona z rekordów typu SElement
struct SElement
{
  int dane;         // pole zawierające daną liczbową
  SElement *nast;   // wskaźnik do elementu następnego
};

// funkcja drukuje zawartość listy zaczynającej się adresem adres
void pisz(SElement *adres)
{
  cout << "Zawartosc listy\n";
  while (adres)  // taki zapis jest równoważny zapisowi while (adres != NULL)
  {
    cout << adres->dane << endl;
    adres = adres->nast;
  }
};

int main(int argc, char* argv[])
{
  SElement *glowa, *aktualny;
  int liczba, naj;
  cout << "Napisz 4 liczby calkowite\n\n";
  glowa = NULL;
  for (int i=0; i < 4; i++)
  {
    cin >> liczba;
    aktualny = new SElement;  // przygotowujemy miejsce dla nowego rekordu
    aktualny->dane = liczba;    // wpisujemy wczytaną liczbę do pola dane
    // wpisujemy adres poprzednio wpisanej liczby do pola nast;
    // dla i=0 wpisze się NULL, czyli pierwsza liczba będzie na końcu listy
    aktualny->nast=glowa;
    glowa=aktualny;             /
/ adres ostatnio wpisanej liczby to nowa głowa listy
  };

  // wydrukujemy teraz listę zaczynając od głowy
  pisz(glowa);
  // poszukiwanie największej wartości na liście
  aktualny=glowa;
  naj=glowa->dane;            // początkowa wartość maksimum
  for (int i = 1; i < 4; i++) // trzeba przeanalizować jeszcze 3 kolejne elementy
  {
    aktualny=aktualny->nast;    // przesuwamy się do następnego elementu
    if (aktualny->dane > naj)   // sprawdzamy pole dane tego elementu
      naj=aktualny->dane;       // w razie potrzeby uaktualniamy max
  };
  cout << "najwiekszy element na liscie: " << naj << endl;

  // Na koniec zwolnimy dla porządku pamięć zajętą przez listę
  // choć nie było tego w treści zadania
```

```
57.   // jako tymczasowy wskaźnik wykorzystamy sobie zmienną glowa -
58.   // możemy tak zrobić, bo kasujemy listę i dostęp do głowy nie
59.   // będzie nam już dłużej potrzebny
60.   aktualny = glowa;                    // zaczynamy od glowy
61.   while (aktualny)
62.   {
63.     glowa=aktualny;              // zapamiętujemy adres kolejnego elementu
64.     aktualny=aktualny->nast;     // przesuwamy się do następnego
65.     cout << "Kasuje: " << glowa->dane << endl;
66.     delete glowa;                // i zwalniamy miejsce poprzedniego
67.   };
68.   glowa = NULL;
69.   return 0;
70. }
```

2. Utworzyć, w kolejności odwrotnej do wczytywania, listę jednokierunkową złożoną z liczb całkowitych, wczytywanych aż do napotkania zera, po czym wydrukować ją, a następnie usunąć pierwszy element z tej listy i ponownie ją wydrukować.

```
1.  #include <cstdlib>
2.  #include <iostream>
3.
4.  using namespace std;
5.
6.  struct SElement
7.  {
8.    int dane;          // pole zawierające daną liczbową
9.    SElement *nast;    // wskaźnik do elementu następnego
10. };
11.
12. // funkcja drukuje zawartość listy zaczynającej się adresem adres
13. void pisz(SElement *adres)
14. {
15.   cout << "Zawartosc listy\n";
16.   while (adres)  // taki zapis jest równoważny zapisowi while (adres != NULL)
17.   {
18.     cout << adres->dane << endl;
19.     adres = adres->nast;
20.   }
21. };
22.
23.
24. int main(int argc, char* argv[])
25. {
26.   SElement *glowa, *aktualny;
27.   int liczba;
28.   // na początku lista jest pusta, więc adres jej poczatku ustawiamy na NULL
29.   glowa = NULL;
30.
31.   cout << "Wprowadz ciag liczb calkowitych zakonczony zerem\n\n";
32.   cin >> liczba;
33.   while (liczba!=0)
34.   {
35.     aktualny = new SElement();  // przygotowujemy miejsce dla nowego elementu
36.     aktualny->dane = liczba;    // wpisujemy wczytaną liczbę do pola dane
37.     aktualny->nast = glowa;     // element następny to ten, co przedtem był głową
38.     glowa = aktualny;           // element aktualny staje się teraz głową
39.     cin >> liczba;
40.   };
41.
42.   // wydrukujemy teraz listę zaczynając od głowy;
43.   // parametr funkcji pisz nie jest przekazywany przez zmienną, więc nie
44.   // ulegnie zmianie po wykonaniu tej funkcji
45.   // lista może być pusta, jeśli od razu podamy zero
46.   pisz(glowa);
47.
48.   // zapamiętamy adres elementu, który należy zwolnić
```

```
49.   aktualny=glowa;
50.   // element możemy usunąć z listy tylko wtedy, gdy nie jest ona pusta
51.   if (glowa)
52.   {
53.     // nowa głowa listy to adres drugiego jej elementu
54.     glowa=glowa->nast;
55.     // teraz możemy zwolnić pierwszy element
56.     delete aktualny;
57.     // i ponownie drukujemy zawartość listy
58.     pisz(glowa);
59.   }
60.
61.   // Na koniec zwolnimy dla porządku pamięć zajętą przez listę,
62.   // choć nie było tego w treści zadania
63.   aktualny = glowa;                    // zaczynamy od glowy
64.   while (aktualny)
65.   {
66.     glowa=aktualny;              // zapamiętujemy adres kolejnego elementu
67.     aktualny=aktualny->nast;     // przesuwamy się do następnego
68.     cout << "Kasuje: " << glowa->dane << endl;
69.     delete glowa;                // i zwalniamy miejsce poprzedniego
70.   };
71.   glowa = NULL;
72.
73.   return 0;
74. }
```

3. Utworzyć listę jednokierunkową złożoną z n wczytanych napisów w kolejności odwrotnej do wczytywania, gdzie n należy najpierw wczytać. Wydrukować listę, po czym dołożyć do niej na drugiej pozycji wczytany dodatkowy napis i ponownie wydrukować.

```
1.  #include <cstdlib>
2.  #include <iostream>
3.  #include <string>
4.
5.  using namespace std;
6.
7.  // lista będzie złożona z rekordów zawierających pole typu string
8.  struct SElement
9.  {
10.   string dane;
11.   SElement *nast;
12. };
13.
14. // tym razem wykorzystamy wersję rekurencyjną funkcji drukującej
15. void pisz(SElement *adres)
16. {
17.   if (!adres) // warunek równoważny takiemu: if (adres == NULL)
18.     return;
19.   cout << adres->dane << endl;
20.   pisz(adres->nast);
21. }
22.
23. int main(int argc, char* argv[])
24. {
25.   SElement *glowa, *aktualny;
26.   int n; // liczba napisów do pobrania
27.   string napis;
28.
29.   cout << "Ile ma byc napisow?\n";
30.   cin >> n;
31.   cout << "wprowadz " << n << " napisow, kazdy od nowej linii\n\n";
32.   glowa = NULL;
33.   // czyścimy bufor klawiatury z entera
34.   cin.ignore();
35.   // listę tworzymy analogicznie jak w poprzednim zadaniu
36.   for (int i = 0; i < n; i++)
```

```
  {
    // do wczytywania napisów wykorzystamy getline, by umożliwić podawanie
    // spacji.
    getline(cin, napis);
    aktualny = new SElement; // przygotowujemy miejsce dla nowego elementu
    aktualny->dane = napis;  // wpisujemy wczytany napis do pola dane
    aktualny->nast = glowa;
    glowa = aktualny;        // adres ostatnio wpisanego napisu to nowa głowa listy

  };

  // wydrukujemy listę
  cout << "\nWczytana lista\n";
  pisz(glowa);

  cout << "podaj dodatkowy napis: " << endl;
  getline(cin, napis);
  aktualny = new SElement;  // przygotowujemy miejsce dla dodatkowego elementu
  aktualny->dane = napis;   // wpisujemy wczytany napis do pola dane tego elementu

  aktualny->nast = glowa->nast; /
/ następny el. za dodatkowym to ten, co był za głową
  glowa->nast = aktualny;       // a następny element za głową to ten dodatkowy

  cout << "\nRozszerzona lista\n";
  pisz(glowa);                // ponownie drukujemy listę od początku

  // Na koniec zwolnimy dla porządku pamięć zajętą przez listę
  // choć nie było tego w treści zadania
  aktualny = glowa;                   // zaczynamy od glowy
  while (aktualny)
  {
    glowa=aktualny;             // zapamiętujemy adres kolejnego elementu
    aktualny=aktualny->nast;    // przesuwamy się do następnego
    cout << "Kasuje: " << glowa->dane << endl;
    delete glowa;               // i zwalniamy miejsce poprzedniego
  };
  glowa = NULL;

  return 0;
}
```

4. Utworzyć listę jednokierunkową złożoną z n wczytanych liczb (n - stała) w kolejności wczytywania, po czym usunąć z niej k-ty element, gdzie k należy przedtem wczytać. Drukować listę przed i po usunięciu elementu.

Chcesz znać wskazówkę ?

Listę można utworzyć "na piechotę", zaś usuwanie należy uzależnić od numeru usuwanego elementu.

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

// zdefiniujemy sobie stałą oznaczającą liczbę elementów do wczytania
const int n = 5;

// typy definiujemy podobnie, jak w poprzednich zadaniach
struct SElement
{
  double dane;
  SElement *nast;
};

// znów wykorzystamy wersję rekurencyjną funkcji drukującej
void pisz(SElement *adres)
{
```

```
20.   if (!adres) // warunek równoważny takiemu: if (adres == NULL)
21.     return;
22.   cout << adres->dane << endl;
23.   pisz(adres->nast);
24. };
25.
26. int main(int argc, char* argv[])
27. {
28.   SElement *glowa, *aktualny, *ogon;
29.   int k;
30.
31.   cout << "podaj " << n << " liczb:\n";
32.
33.   // listę tworzymy inaczej niż w poprzednich zadaniach: w kolejności wczytywania
34.
35.   // najpierw osobno pierwszy element
36.   glowa = new SElement; // tworzymy miejsce dla pierwszego elementu
37.   cin >> glowa->dane;   // od razu wczytujemy daną liczbę do rekordu
38.   glowa->nast = NULL;   // żeby było bezpiecznie - oznaczamy koniec listy.
39.   ogon = glowa;         // na razie koniec listy jest też jej początkiem
40.
41.   // a następnie n-1 pozostałych elementów
42.   for (int i = 1; i < n; i++)
43.   {
44.     aktualny = new SElement; // przygotowujemy miejsce dla nowego elementu
45.     cin >> aktualny->dane;   // wpisujemy wczytaną liczbę do pola dane
46.     aktualny->nast = NULL;   // na razie to jest ostatni element
47.     ogon->nast = aktualny;   // dołączamy go do ogona
48.     ogon = aktualny;         // i ten element będzie odtąd ogonem
49.   };
50.
51.   // wydrukujemy listę zaczynając od głowy
52.   cout << "\nWprowadzona lista:\n";
53.   pisz(glowa);
54.
55.   cout << "ktory element usunac z listy? podaj liczbe od 1 do " << n << endl;
56.   do
57.     cin >> k;
58.   while (k <= 0 || k > n);
59.
60.   // algorytm usuwania zależy od tego, czy usuwamy głowę, czy coś w środku listy
61.   // zauważcie, że zastosowaliśmy nietypowy dla C sposób indeksowania - od jedynki
62.   if (k==1)
63.   {
64.     // usuwamy głowę listy
65.     aktualny = glowa; // zapamiętujemy adres pierwszego elementu
(korzystając ze zmiennej aktualny)
66.     glowa = glowa->nast;  // nowa głowa to adres drugiego elementu
67.     delete aktualny;      // zwalniamy miejsce zajęte przez pierwszy element
68.   }
69.   else
70.   {
71.     // usuwamy element w dalszej części listy
72.     aktualny = glowa;        // startujemy od początku
73.     // przesuwamy się do elementu poprzedzającego element usuwany
74.     for (int i=1; i != k-1; i++)
75.       aktualny = aktualny->nast;
76.     ogon = aktualny->nast;  // zapamiętujemy adres el. usuwanego
(tym razem korzystając ze zmiennej ogon)
77.     // wiążemy element poprzedzający z następnym po usuwanym
78.     aktualny->nast = ogon->nast;
79.     delete ogon;              // zwalniamy miejsce zajęte przez el. usuwany
80.   };
81.
82.   // ponownie drukujemy listę od początku
83.   cout << "\nPo usunieciu elementu\n";
```

```
84.   pisz(glowa);
85.
86.   // Na koniec zwolnimy dla porządku pamięć zajętą przez listę
87.   // choć nie było tego w treści zadania
88.   aktualny = glowa;                    // zaczynamy od głowy
89.   while (aktualny)
90.   {
91.     glowa=aktualny;              // zapamiętujemy adres kolejnego elementu
92.     aktualny=aktualny->nast;     // przesuwamy się do następnego
93.     cout << "Kasuje: " << glowa->dane << endl;
94.     delete glowa;                // i zwalniamy miejsce poprzedniego
95.   };
96.   glowa = NULL;
97.
98.   return 0;
99. }
```

5. Napisać program umieszczający na liście jednokierunkowej (w kolejności obliczeń) wartości funkcji:
y = (x+2)(x-1)-1 dla x <-5, 5> z krokiem 1.
Wyświetlić otrzymaną listę, usunąć z niej dwa pierwsze elementy i jeszcze raz wyświetlić.

```
1.  #include <iostream>
2.  #include <cstdlib>
3.
4.  using namespace std;
5.
6.  struct SElem {
7.    int v;
8.    SElem *n;
9.  };
10.
11. // pomocnicza funkcja wyliczająca wartość wyrażenia
12. int f(int x)
13. {
14.   return (x+2)*(x-1)-1;
15. }
16.
17. // drukowanie rekurencyjne
18. void drukuj(SElem *e)
19. {
20.   if (e)
21.   {
22.     cout << e->v << " ";
23.     drukuj(e->n);
24.   }
25. }
26.
27. int main(int argc, char *argv[])
28. {
29.   SElem *glowa, *e;
30.   glowa = new SElem();
31.   glowa->v = f(-5);
32.   e = glowa;
33.   for (int i = -4; i <=5; i++)
34.   {
35.     e->n = new SElem();
36.     e = e->n;
37.     e->v = f(i);
38.   }
39.   e->n = NULL;
40.   // wyświetlanie
41.   drukuj(glowa);
42.   cout << endl;
43.     // usuniecie dwóch pierwszych (jeśli są)
44.   for (int i = 0; i < 2; i++)
45.   {
46.     if (glowa)
```

```
    {
      e = glowa;
      glowa = glowa->n;
      delete e;
    }
  }
  // wyświetlanie
  drukuj(glowa);
  cout << endl;
  // usunięcie listy
  while (glowa)
  {
    e = glowa;
    glowa = glowa->n;
    delete e;
  }

  system("PAUSE");
  return 0;
}
```

6. Dla listy zdefiniowanej następująco:

```
struct TInt {
  int liczba;
  TInt* nast
```

napisać funkcję, która zwraca sumę co drugiego elementu na liście, poczynając od pierwszego. Parametrem funkcji jest adres początku listy.
Uwaga: Należy napisać tylko funkcję, a nie cały program.

Chcesz znać wskazówkę ?

Uwaga: Nie należy bez kontroli "przeskakiwać" przez co drugi element listy, bo można nie trafić na koniec!

```
/* Uwaga: to zadanie wymagało napisania jedynie funkcji (i tylko tyle należy
napisać na egzaminie przy tak sformułowanym zadaniu). Ale abyście mogli
sprawdzić, jak ona działa, zamieszczamy dodatkowo prosty program, który tworzy
przykładową listę i zadaną funkcję wywołuje. */

#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

struct SInt
{
  int liczba;
  SInt *nast;
};

/* Funkcja, która będzie rozwiązaniem tego zadania, będzie przyjmowała jeden
  parametr: wskaźnik do początku listy. Zwracana wartość będzie typu
  całkowitego - ponieważ  na liście zapamiętane są elementy typu całkowitego,
  więc ich suma także będzie liczbą całkowitą. Wskaźnik do początku listy
  przekażemy przez wartość, a nie zmienną, co pozwoli nam bezpiecznie go
  modyfikować wewnątrz funkcji, i sprawi, że nie będzie trzeba deklarować
  zmiennej lokalnej */
int suma(SInt *aktualny)
{
  // Jedyną zmienną lokalną w tej funkcji będzie dotychczasowa suma elementów
  int suma;
  // Na początku musimy zabezpieczyć się przed sytuacją, kiedy lista będzie pusta

  if (aktualny == NULL)
```

```
31.     // W tym przypadku suma elementów wynosi 0
32.     return 0;
33.   // skoro na liście jest co najmniej jeden element, początkowa wartość sumy
34.   // będzie mu równa
35.   suma = aktualny->liczba;
36.
37.   // I możemy zacząć pętlę zliczającą. Proszę zauważyć, jak skomplikowany jest
38.   // warunek zakończenia pętli. Taka jego postać zapewnia nas, że na liście
39.   // istnieją jeszcze co najmniej dwa elementy. Nie można napisać bezpośrednio
40.   // aktualny->nast->nast != NULL, ponieważ jeśli aktualny byłby ostatnim elementem
41.   // listy, to aktualny->nast == NULL i odwołanie aktualny->nast->nast staje się
42.   // błędne. Lecz ponieważ warunki są sprawdzane zawsze od lewej do prawej,
43.   // pierwszy warunek, (aktualny->nast != NULL) zapewni nam opuszczenie pętli w
44.   // przypadku, gdy aktualny jest ostatnim elementem na liście, natomiast jeśli
45.   // nie jest, możemy bezpiecznie sprawdzić, czy istnieje jeszcze jeden element }
46.   while (aktualny->nast != NULL && aktualny->nast->nast != NULL)
47.   {
48.     // Tutaj już bezpiecznie możemy przeskoczyć o dwa elementy do przodu
49.     aktualny = aktualny->nast->nast;
50.
51.     // I zwiększyć sumę
52.     suma += aktualny->liczba;
53.   };
54.
55.   // na koniec pozostaje nam jedynie zwrócenie wyniku
56.   return suma;
57. };
58.
59. // dla ciekawskich - ta sama funkcja rekurencyjnie:
60. int suma_rek(SInt *aktualny)
61. {
62.   if (aktualny == NULL)
63.     return 0;
64.
65.   if (aktualny->nast)
66.     return aktualny->liczba + suma_rek(aktualny->nast->nast);
67.
68. /
    / wbrew pozorom else jest tutaj niepotrzebny, bo każda poprzednia instrukcja kończy
     się return-em
69. /
    / zatem do miejsca poniżej dojdziemy tylko wówczas, gdy żaden z poprzednich warunkó
    w nie będzie spełniony
70. // ten sposób pisania  w przypadku funkcji z wieloma return-
    ami jest lepszy dla kompilatora :)
71. /
    / jeśli choć jeden return jest bezwarunkowy, nie wykazuje on zaniepokojenia, czy z
    tej funkcji w ogóle da się wyjść
72. // i nie drukuje nam niepotrzebnych ostrzeżeń (Warnings) pod oknem edytora
73.
74.   return aktualny->liczba;
75. };
76.
77. int main(int argc, char* argv[])
78. {
79.   SInt *g, *p, *a;
80.   // Utworzymy listę z kolejnych liczb od 0 do 12
81.   g = new SInt;
82.   g->liczba = 0;
83.   p = g;
84.   for (int i=1; i <= 12; i++)
85.   {
86.     a = new SInt;
87.     a->liczba = i;
88.     p->nast = a;
89.     p = a;
```

```
    }
    a->nast = NULL;

    // i wywołamy zadaną funkcję
    cout << "suma normalnie: " << suma(g) << ", rekurencyjnie: " << suma_rek(g)
   << endl;

    return 0;
}
```

7. Napisać program umieszczający na liście jednokierunkowej kolejne wprowadzane z klawiatury słowa, w kolejności zgodnej z wczytywaniem (jeden element listy – jeden wiersz), aż do momentu wprowadzenia samej kropki. Następnie wyświetlić co drugi element listy, zaczynając od pierwszego, usunąć pierwszy element oraz wyświetlić całą listę.

```
#include <cstdlib>
#include <string>
#include <iostream>

using namespace std;

struct SSlowo {
    string s;
    SSlowo *n;
};


int main(int argc, char* argv[])
{
    SSlowo *g, *e;
    string s;
    cin >> s;
    g = NULL;
    while (s != ".")
    {
        if (!g) {
            g = new SSlowo();
            e = g;
        } else {
            e->n = new SSlowo();
            e = e->n;
        }
        e->s = s;
        cin >> s;
    }

    e->n = NULL;

// wydruk co drugiego
    e = g;
    while (e) {
        cout << e->s << " ";
        e = e->n;
        if (e)
            e = e->n;
    }

//  usunięcie pierwszego
    e = g;
    if (g) {
        g = g->n;
        delete e;
    }

// wydruk listy
    e = g;
    while (e) {
        cout << e->s << " ";
```

```
54.         e = e->n;
55.     }
56.
57. // kasowanie nie jest potrzebne - ale dla elegancji zapiszemy je }
58.     e = g;
59.     while (e) {
60.         g = e;
61.         e = e->n;
62.         delete g;
63.     }
64.     return 0;
65. }
```

8. Napisać program, który umożliwi zapamiętanie na liście jednokierunkowej wprowadzanych przez użytkownika napisów wraz z ich długością, w kolejności wprowadzania. Wprowadzanie danych kończy się po podaniu przez użytkownika kropki. Następnie z listy usunąć napisy o parzystej liczbie znaków. Wydrukować listę przed i po usuwaniu napisów.

```
1.  #include <cstdlib>
2.  #include <iostream>
3.  #include <iomanip>
4.  #include <string>
5.
6.  using namespace std;
7.
8.
9.  // Na początek zdefiniujemy element listy jednokierunkowej.
10. struct SElement
11. {
12.   string napis;
13.   int dlugosc;
14.   SElement *nastepny;
15. };
16.
17. // Wykorzystamy rekurencyjną procedurę drukującą
18. void drukuj_liste(SElement *element)
19. {
20.   if (!element)
21.     return;
22.   cout << element->napis << " : " << element->dlugosc << endl;
23.   drukuj_liste(element->nastepny);
24. };
25.
26. // Dodamy także dodatkową funkcję usuwającą listę z pamięci - także w
27. // formie rekurencyjnej; nierekurencyjne usuwanie listy jest na końcu
28. // zadań wcześniejszych
29. void usun_liste(SElement *element)
30. {
31.   if (!element)
32.     return;
33.   usun_liste(element->nastepny);
34.   delete element;
35. };
36.
37. int main(int argc, char* argv[])
38. {
39.   SElement *pierwszy, *biezacy, *poprzedni;
40.
41.   cout << "Wprowadzaj napisy, kazdy z nowej linii. Kropka konczy dane\n\n";
42.
43.   // wychodząc z założenia, że kropka należy do listy, możemy stwierdzić,
44.   // że lista zawsze będzie składała się z co najmniej jednego elementu,
45.   // który możemy utworzyć przed pętlą
46.   pierwszy = new SElement;
47.   biezacy = pierwszy;
48.
49.   // Pętla wczytująca będzie tak prosta, jak się da. Jako warunek końca pętli
```

```
50.   // wykorzystamy fakt, że nie utworzono następnego elementu (bo napotkano kropkę)
51.   do
52.   {
53.     // Wczytanie nowego napisu bezpośrednio do pola napis elementu
54.     getline(cin, biezacy->napis);
55.     // Długość napisu obliczamy i wstawiamy do pola dlugosc elementu
56.     biezacy->dlugosc = biezacy->napis.size();
57.     // Następnie w zależności od wprowadzonego tekstu tworzymy bądź nie
58.     // nowy element
59.     if (biezacy->napis == ".")
60.       biezacy->nastepny = NULL;
61.     else
62.     // w poniższej instrukcji tworzymy nowy element i jego adres wpisujemy
63.     // od razu do pola nastepny elementu bieżącego - wiążemy oba elementy
64.       biezacy->nastepny = new SElement;
65.     // i przechodzimy do nowego elementu
66.     biezacy = biezacy->nastepny;
67.   } while (biezacy);
68.
69.   // wydrukujmy dane wprowadzone przez użytkownika
70.   cout << "Oryginalna lista: \n\n";
71.   drukuj_liste(pierwszy);
72.
73.   // następnie usuniemy elementy z napisami o parzystej liczbie znaków
74.   biezacy = pierwszy;
75.   poprzedni = NULL;
76.   while (biezacy)
77.   {
78.     // sprawdźmy, czy długość napisu, czyli liczba znaków, jest parzysta
79.     if (biezacy->dlugosc % 2 == 0)
80.       // 1-szy przypadek - usuwanie pierwszego elementu (poprzedni nie istnieje)
81.       // wymaga specjalnego potraktowania
82.       if (!poprzedni)
83.       {
84.         // przechodzimy do następnego elementu - on teraz będzie pierwszym
85.         biezacy = biezacy->nastepny;
86.         // kasujemy dotychczasowy pierwszy element
87.         delete pierwszy;
88.         // i wskaźnikowi na pierwszy element przypisujemy nową wartość
89.         pierwszy = biezacy;
90.       }
91.       else
92.       {
93.         // 2-gi przypadek - liczba znaków parzysta, lecz element nie jest pierwszy
94.         // na liście.
95.         // teraz najpierw zadbamy o spójność listy i przypiszemy powiązanie
96.         // "omijające" element do skasowania
97.         poprzedni->nastepny = biezacy->nastepny;
98.         // a dopiero potem usuwamy bieżący element
99.         delete biezacy;
100.        // i przechodzimy do elementu następnego
101.        biezacy = poprzedni->nastepny;
102.      }
103.    else
104.    {
105.      // jeśli napis zawiera nieparzystą liczbę znaków,
106.      // przechodzimy do następnego elementu na liście
107.      poprzedni = biezacy ;
108.      biezacy   = biezacy->nastepny;
109.    };
110.  }; // koniec pętli
111.
112.  // pozostało jeszcze wydrukować listę
113.  cout << "\nLista po usunieciu elementow: \n\n";
114.  drukuj_liste(pierwszy);
115.
```

```
116.     // i skasować ją
117.     usun_liste(pierwszy);
118.     pierwszy = NULL;
119.     return 0;
120. }
```

9. Napisać funkcję, która dla dla jakiegoś adresu typu Tadr wskazującego na początek listy jednokierunkowej zdefiniowanej nastepującymi typami:

```
struct SElem {
    ...
    SElem next;
};
```

tworzy listę z 4 ostatnich elementów listy danej, ułożonych w kolejności odwrotnej (czyli najpierw ostatni, potem przedostatni itd.), po czym zwraca adres początku nowej listy. Przyjąć dla uproszczenia, że lista dana zawiera co najmniej 4 elementy

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  #include <cmath>
4.
5.  using namespace std;
6.
7.  struct SElem {
8.      int d;
9.      SElem *next;
10. };
11.
12. SElem* tworzInaczej(SElem *g)
13. {
14.     // skoro mamy bezpieczne założenie, że lista ma co najmniej 4 elementy -
15.     // możemy poszukać czwartego od konca:
16.     while (g->next->next->next->next != NULL)
17.         g = g->next;
18.
19.     // tworzymy nową listę
20.     SElem* ng = new SElem();
21.     SElem* e;
22.     ng->next = NULL;
23.     ng->d = g->d;
24.     e = ng;
25.     while (g->next) {
26.         g = g->next;
27.         e = new SElem();
28.         e->d = g->d;
29.         e->next = ng;
30.         ng = e;
31.     }
32.     return e;
33. }
34.
35. /
    / reszta kodu nie jest konieczna przy tak sformułowanym zadaniu - tu tylko po to, b
    y pokazać, że
36. // działa
37. void drukuj(SElem *e)
38. {
39.     if (!e)
40.     {
41.         cout << endl;
42.     } else {
43.         cout << e->d << " ";
44.         drukuj(e->next);   // rekurencyjne wywołanie
45.     }
46. }
47.
```

```
48.
49. int main(int argc, char* argv[])
50. {
51.     SElem *g, *e;
52.     g = new SElem();
53.     g->d = 0;
54.     e = g;
55.     for (int i = 1; i < 10; i++)
56.     {
57.         e->next = new SElem();
58.         e=e->next;
59.         e->d = i;
60.     }
61.     e->next = NULL;
62.     drukuj(g);
63.     e = tworzInaczej(g);
64.     drukuj(e);
65.
66.     system("pause");
67.     return 0;
68. }
```

10. Napisać funkcję, która dla listy zdefiniowanej następująco:

```
struct TEl {
  double cos;
  TEl *dalej;
};
```

i zaczynającej się pod jakimś adresem (będącym parametrem funkcji) tworzy nową listę z 3 pierwszych elementów listy danej, ułożonych w kolejności odwrotnej (czyli najpierw trzeci, potem drugi, potem pierwszy). Na liście danej może być mniej niż 3 elementy i lista wynikowa będzie wtedy odpowiednio krótsza.
Uwaga: Należy napisać tylko funkcję, a nie cały program.

```
1.  // I znowu funkcja jest pokazana wraz z przykładowym programem
2.  #include <cstdlib>
3.  #include <iostream>
4.
5.  struct SElement
6.  {
7.    double cos;
8.    SElement *dalej;
9.  };
10.
11. using namespace std;
12.
13. /* Nasza funkcja będąca rozwiązaniem tego zadania jest funkcją rekurencyjną.
14.    Podajemy rozwiązanie bardzo eleganckie, pokazujące urok i sens rekurencji
15.    (zilustrowane dla lepszego zrozumienia wydrukami kontrolnymi).
16.    Gdyby ściśle się trzymać treści zadania, nie trzeba kasować elementów
17.    niepotrzebnych i nie jest konieczne stosowanie rekurencji.
18.
19.    Zanim przejdziemy do napisania funkcji, spróbujmy sprecyzować dokładnie, co
20.    ona ma robić. Przy czym jej działanie będzie się zmieniało w zależności od tego,
21.    jaki element będzie elementem aktualnie przetwarzanym. Zaczniemy ją opisywać
22.    od końca listy:
23.    1. jeśli przekazany parametr jest wskaźnikiem zerowym (NULL), funkcja ma nic
24.       nie robić.
25.    2. dla elementów od 4 do ostatniego funkcja powinna usunąć aktualny element
26.       i wywołać samą siebie dla następnego elementu, by usunąć resztę listy.
27.    3. Najbardziej skomplikowany przypadek: jeśli aktualny element jest z początku
28.       listy, musimy przestawić go z końca na początek listy. Inne wyjście
29.       to stworzenie nowej listy w odwrotnej kolejności z trzech pierwszych
30.       elementów. Wybór metody rozwiązania będzie determinował listę parametrów
31.       przekazywanych procedurze. Prostsze rozwiązanie byłoby w przypadku
```

```
32.       przestawiania elementów, i w tym wypadku potrzebowalibyśmy przekazywać
33.       procedurze wskaźnik do aktualnego elementu, wskaźnik do pierwszego elementu
34.       na liście oraz (pomocniczo) numer elementu. Lecz my wybraliśmy bardziej
35.       finezyjne rozwiązanie - będziemy tworzyć nową listę, wykorzystując przy tym
36.       tylko jeden wskaźnik jako parametr. Przy wywołaniu funkcji powinien on
37.       zawierać adres pierwszego elementu listy, po zakończeniu jej wykonania
38.       będzie zawierał adres nowego pierwszego elementu listy. I przy tym
39.       rozwiązaniu będzie nam potrzebny pomocniczo numer elementu listy. */
40.
41.  void przestaw(SElement **akt, int numer)
42.  {
43.    SElement *tmp, *tmp2;
44.
45.    // Odpowiednik punktu 1 z opisu procedury. W ciągu wywołań rekurencyjnych
46.    // dotarliśmy do końca listy, więc już nic nie mamy do roboty - opuszczamy
47.    // funkcję i jednocześnie jest to zakończenie ciągu wywołań rekurencyjnych.
48.    if (*akt == NULL)
49.      return;
50.
51.    for (int i=0; i<numer; i++)
52.      cout << " ";
53.
54.    cout << "Element nr " << numer << ", wart. " << (*akt)->cos;
55.
56.    // Punkt 2: usuwanie elementów, które mają numer większy niż 3.
57.    if (numer > 3)
58.    {
59.      cout << " - kasujemy.\n";
60.      // zapamiętujemy element do usunięcia
61.      tmp = *akt;
62.      // przechodzimy do następnego elementu
63.      *akt = (*akt)->dalej;
64.      // usuwamy zapamiętany
65.      delete tmp;
66.      // i procedura wywołuje samą siebie dla następnego elementu.
67.      przestaw(akt, numer + 1);
68.    }
69.    else
70.    // Punkt 3 - czyli to, co ma się wykonać dla pierwszych 3 elementów
71.    {
72.      // zapamiętamy aktualny element
73.      tmp = *akt;
74.      cout << " - zapamietujemy i usuwamy z listy. \n";
75.      // przejdziemy do następnego
76.      *akt = (*akt)->dalej;
77.      // a w elemencie zapamiętanym - zerwiemy połączenie z listą. Czyli możemy
78.      // go traktować w tej chwili jako "zawieszonego w nicości" - nie należy do
79.      // listy
80.      tmp->dalej = NULL;
81.
82.      // rekurencyjne wywołanie procedury dla nowego aktualnego elementu
83.      przestaw(akt, numer + 1);
84.
85.      // teraz po wykonaniu rekurencyjnym parametr akt zawiera adres nowego
86.      // pierwszego elementu listy. Nam pozostaje dostawić zapamiętany element na
87.      // jej koniec. W przypadku, gdy wywołanie rekurencyjne zmieniło nam zawartość
88.      // wskaźnika akt na NULL, lista jest pusta. Nasz zapamiętany element ma być
89.      // na liście pierwszy. Musimy to zaznaczyć poprzez przypisanie zmiennej akt
90.      // adresu tego właśnie zapamiętanego elementu .
91.      for (int i=1; i < numer; i++)
92.        cout << " ";
93.
94.      cout << "Element nr " << numer << ", wart. " << tmp->cos;
95.      if (*akt == NULL)
96.      {
97.        cout << " - nowy poczatek listy.\n";
98.        *akt = tmp;
```

```
99.       }
100.      // W przeciwnym przypadku wstawiamy zapamiętany element na koniec listy
101.      else
102.      {
103.        cout << " - wstawiamy na koniec.\n";
104.        // aby to zrobić, najpierw wyszukujemy ostatni element
105.        tmp2 = *akt;
106.        while (tmp2->dalej)
107.          tmp2 = tmp2->dalej;
108.
109.        // a następnie dowiązujemy na koniec nasz element
110.        tmp2->dalej = tmp;
111.      };
112.    };
113.  };
114.
115.  void drukuj(SElement *adres)
116.  {
117.    while (adres)
118.    {
119.      // wypisujemy zawartość elementu
120.      cout << adres->cos << endl;
121.      // i przechodzimy do następnego
122.      adres=adres->dalej;
123.    };
124.  };
125.
126.  int main(int argc, char* argv[])
127.  {
128.    SElement *a, *p, *g;
129.    double n;
130.
131.    cout << "Podawaj elementy liczbowe, koniec - 0\n";
132.    a = p = g = NULL;
133.    cin >> n;
134.    while (n!=0)
135.    {
136.      p = a;
137.      a = new SElement;
138.      a->cos = n;
139.      a->dalej = NULL;
140.      if (p)
141.        p->dalej = a;
142.      else
143.        g = a;
144.      cin >> n;
145.    };
146.    cout << "\n\nLista utworzona\n";
147.    drukuj(g);
148.    przestaw(&g, 1);
149.    cout << "\n\nLista po zamianie\n";
150.    drukuj(g);
151.    a = g;
152.    while (a)
153.    {
154.      p = a;
155.      a = a->dalej;
156.      delete p;
157.    };
158.    return 0;
159.  }
```

11. Dla listy zdefiniowanej następująco:

```
struct TNapis {
  string napis;
  TNapis* nast;
```

```
};
```

napisać funkcję, która usuwa ostatni element z listy zaczynającej się pod jakimś adresem (będącym parametrem funkcji) i wstawia go na początek listy. Uwzględnić przypadki, kiedy z listą nie należy nic robić.
Uwaga: Należy napisać tylko funkcję, a nie cały program.

```
1.  // I znowu zamieszczamy dodatkowo program, który listę tworzy, drukuje i wywołuje
2.  // zadaną funkcję
3.  #include <cstdlib>
4.  #include <iostream>
5.  #include <string>
6.
7.  using namespace std;
8.
9.  struct SNapis
10. {
11.   string napis;
12.   SNapis *nast;
13. };
14.
15. /* Rozwiązanie, które prezentujemy Państwu tutaj, jest trochę bardziej
16.   skomplikowane niż mogłoby być, gdybyśmy zdecydowali się na wprowadzenie
17.   jeszcze jednej zmiennej typu SNapis* do procedury. Niemniej jednak może warto
18.   zobaczyć, że czasem można ograniczyć liczbę zmiennych pomocniczych.
19.
20.   Funkcja przyjmuje jeden parametr - głowę listy. Adres początku listy zmieni
21.   się w wyniku działania tej funkcji. Tę "zmianę" w głównym programie można
22.   odebrać poprzez parametr przekazywany przez zmienną, bądź poprzez zwracanie
23.   nowej wartości głowy listy przez funkcję. My skorzystamy z pierwszego
24.   rozwiązania */
25. void zamien(SNapis **glowa)
26. {
27.   SNapis *aktualny;
28.   // zakładamy, że naszym aktualnym elementem jest pierwszy element listy
29.   aktualny = *glowa;
30.
31.   // Z listą nic nie robimy, gdy jest pusta, bądź ma jeden element
32.   if (aktualny == NULL || aktualny->nast == NULL)
33.     return;
34.
35.   // jeszcze jeden przypadek, kiedy nic nie robimy, to wtedy, gdy lista ma
36.   // dokładnie dwa jednakowe elementy. Wtedy przestawienie końca na początek nic
37.   // nie zmieni. Lista ma dokładnie dwa elementy, gdy aktualny->nast->nast == NULL
38.   if (aktualny->nast->nast == NULL && aktualny->napis == aktualny->nast->napis)
39.     return;
40.
41.   // Oczywiście dwie powyższe instrukcje if można byłoby zapisać w postaci jednej
42.   // instrukcji, lecz byłaby ona zdecydowanie mniej czytelna
43.
44.   // Wszystkie pozostałe przypadki wymagają już wykonania przekształcenia.
45.   // Ponieważ element ma być usunięty z końca i przeniesiony na początek,
46.   // nie musimy nic tworzyć ani usuwać. Musimy tylko znaleźć ostatni element.
47.   // A dokładniej - wygodniej nam będzie znaleźć przedostatni element - wtedy
48.   // będziemy mogli go zmodyfikować, aby zaznaczyć że to teraz będzie koniec
49.   // listy
50.   while (aktualny->nast->nast != NULL)
51.     aktualny = aktualny->nast;
52.
53.   // teraz zmienna aktualny zawiera przedostatni element listy. Skoro tak, to
54.   // najpierw przenieśmy ostatni element jako nową głowę listy (przypominamy,
55.   // że ostatni element listy jest teraz wskazywany jako aktualny->nast: }
56.
57.   // najpierw dowiązanie starej głowy listy do nowej:
58.   aktualny->nast->nast = *glowa;
```

```
59.
60.   // następnie zmiana zmiennej oznaczającej głowę listy
61.   *glowa = aktualny->nast;
62.
63.   // oraz zaznaczenie, że lista teraz kończy się na elemencie aktualny
64.   aktualny->nast = NULL;
65. };
66.
67. // funkcja drukowania zawartości listy, poczynając od adresu adres-
68. // posłuży do konroli poprawności działania programu
69. void drukuj(SNapis *adres)
70. {
71.   if (!adres)
72.     return;
73.   cout << adres->napis << endl;
74.   drukuj(adres->nast);
75. };
76.
77. int main(int argc, char* argv[])
78. {
79.   SNapis *g, *p, *a;
80.   string n;
81.   cout << "wprowadzaj kolejne napisy, pusty napis (sam Enter) konczy dane\n";
82.   a = p = g = NULL;
83.   getline(cin, n);
84.   while (n.size() > 0)
85.   {
86.     p = a;
87.     a = new SNapis;
88.     a->napis = n;
89.     a->nast = NULL;
90.     if (p != NULL)
91.       p->nast = a;
92.     else
93.       g = a;
94.     getline(cin, n);
95.   };
96.
97.   cout << "Lista wprowadzona\n";
98.   drukuj(g);
99.   zamien(&g);
100.  cout << "Lista po zamianie\n";
101.  drukuj(g);
102.
103.  return 0;
104. }
```