Wykład 12

12. Zarządzanie blokami pamięci i łańcuchami

Wskaźniki umożliwiają wykonywanie operacji na ciągłych blokach pamięci. W szczególności można opracować funkcje przeznaczone do kopiowania oraz przesuwania obszarów pamięci, a także inicjowania bloków danymi lub odnajdywania sekwencji danych. Podobne operacje można wykonywać na łańcuchach. W praktyce są wykorzystywane standardowe funkcje przeznaczone do zarządzania blokami pamięci i tekstami.

12.1. Operacje na blokach pamięci

Biblioteki ANSI C <mem.h> i <string.h> zawierają funkcje umożliwiające wykonywanie różnorodnych operacji na blokach pamięci (np. kopiowanie bloku danych z miejsca przeznaczenia do miejsca docelowego – *memmove* lub przeszukiwanie bloku pamięci w celu odnalezienia danej o ustalonej wartości - *memchr*). W kompilatorach 16-bitowych, np. BC++3.1 niektóre z tych funkcji występują w wersjach bliskich (np. *memchr*) lub dalekich, zawierających przedrostek *_f* (np. *_fmemchr*). W przypadku braku przedrostka _f funkcje operują na wskaźnikach o rodzaju domyślnym dla wybranego modelu pamięci.

- Znajdowanie znaku: **memchr, _fmemchr**

```
void *memchr(const void *s, int z, size_t n);
void far* far _fmemchr(const void far *s, int z, size_t n);
```

Przeszukiwanie n bajtów w bloku pamięci wskazywanym przez s w celu odnalezienia adresu pierwszego bajtu (znaku) o wartości określonej przez z. Wartością funkcji jest adres napotkanego bajtu lub NULL jeśli bajt nie został odnaleziony.

Przykład 12.1. Wykorzystanie funkcji memchr.

```
char x[ ] = { 0, -1, 2, -3, 4, -5 };
char *wsk;
// wartość 0   - pozycja x+0          // wartość -1  - pozycja x+1
// wartość 2   - pozycja x+2          // wartość -3  - pozycja x+3
wsk = (char *) memchr(x, -3, 6*sizeof(char) );

if (wsk)  printf("Wartość -3 jest na pozycji: %d\n", wsk - x);   // 3
else      printf("Bajt danych nie został znaleziony \n");
if (wsk) printf("%d \n", *wsk);         // dana na pozycji wsk ma wartość -3
```

- Kopiowanie rozłącznych obszarów pamięci **memcpy**, **fmemcpy** lub do przeniesienia określonego znaku: **memccpy, _fmemccpy**

```
void *memcpy(void *d, const void *s, size_t n);
void far* far _fmemcpy(void far *d, const void far *s, size_t n);

void *memccpy(void *d, const void *s, int z, size_t n);
void far* far _fmemccpy(void far *d, const void far *s, int z, size_t n);
```

Kopiowanie n bajtów z obszaru początkowego s do obszaru przeznaczenia d. W przypadku funkcji memccpy kopiowanie jest przerywane jeśli przekopiowano n bajtów lub został przeniesiony bajt o wartości z. Wartością funkcji memcpy jest adres początku obszaru przeznaczenia, tj. d. Natomiast wartością funkcji memccpy jest adres bajtu w obszarze docelowym (d), który znajduje się bezpośrednio za poszukiwanym bajtem, lub NULL, jeśli bajt nie został znaleziony. *W przypadku, gdy d i s mają część wspólną d nie musi zawierać dokładnej kopii s.*

Przykład 12.2. Zastosowanie funkcji memccpy.

```
char x[ ] = { 0, -1, 2, -3, 4, -5 };
char *wsk;

wsk = (char *) memccpy(x, x+1, -3, 5*sizeof(char));

for (int i=0; i<6; i++) printf("%3d", x[i]);
if (wsk) {     printf("\nBajt zostal znaleziony \n");
               printf("%d \n", *(wsk -1) );     }          // -3
```

Zawartość tablicy x po kopiowaniu: { -1, 2, -3, | -3, 4, -5 }.

- Porównywanie bloków pamięci: **memcmp, _fmemcmp**

```
int memcmp(const void *a1, const void *a2, size_t n);
int far _fmemcmp(const void far *a1, const void far *a2, size_t n);
```

Porównywanie dwóch bloków pamięci o rozmiarze n. Bajty są traktowane jako dane typu unsigned char. Dane są porównywane aż do momentu napotkania różnych bajtów. Funkcja zwraca 0 jeśli wszystkie bajty bloku wskazywanego przez a1 są równe odpowiednim bajtom w bloku wskazywanym przez a2. Jeśli pojawił się bajt w bloku 1, który jest mniejszy od swego odpowiednika w bloku 2, to funkcja zwraca wartość mniejszą od 0, w przeciwnym przypadku wartość większą od 0.

Przykład 12.3. Wykorzystanie funkcji memcmp.

```
double a1 = 5;    double a2 = 5;
i = -1;     i = memcmp(&a1, &a2, sizeof(double));          // a1 == a2
printf("%d\n", i);                                         // i == 0

char x[ ] = { 0, -1, 4, -3, 4, -5 };          // -1 == 255 jako unsigned char
char y[ ] = { 0, -1, 2, -3, 4, -5 };

i = memcmp(x, y, sizeof(x) );  // porównanie tablic x i y;   sizeof(x) == 6
printf("%d\n", i);                  //  4 > 2;   i  > 0
```

- Kopiowanie „zachodzących” bloków pamięci: **memmove**

```
void *memmove(void *d, const void *s, size_t n);
```

Kopiowanie n bajtów z obszaru początkowego s do obszaru przeznaczenia d. Kopiowanie jest poprawne nawet jeśli obszary nakładają się na siebie. Wartością funkcji jest adres początku obszaru przeznaczenia.

Przykład 12.4. Zastosowanie funkcji memmove.

```
char x[ ] = { 0, -1, 2, -3, 4, -5 };   char *wsk;

wsk = (char *) memmove(x+1, x, 5*sizeof(char));
// obszary nachodzą się  0 -1 2 -3 ...
        //                   -1 2 -3 4 ...
for (int i=0; i<6; i++) printf("%3d", x[i]);
```

Zawartość tablicy x po kopiowaniu: { 0, 0, -1, 2, -3, 4 }.

- Wypełnianie obszaru stałą wartością: **memset, _fmemset**

```
void *memset(void *s, int z, size_t n);
void far* far _fmemset(void far *s, int z, size_t n);
                // daleka funkcja (far) zwracająca daleki wskaźnik
```

Nadawanie pierwszym n bajtom bloku pamięci o adresie s wartości określonej parametrem z. Wartością funkcji jest s.

Przykład 12.5. Zastosowanie funkcji memset.

```
char x[ ] = { 0, -1, 2, -3, 4, -5 };
char *wsk;

wsk = (char *) memset(x, 0, 6*sizeof(char));

for (int i=0; i<6; i++) printf("%3d", x[i]);
```

Zawartość tablicy x po kopiowaniu: { 0, 0, 0, 0, 0, 0 }.

Operacja blokowej inicjacji tablicy jest wykonywana szybciej niż analogiczna operacja polegająca na wpisywaniu wartości do kolejnych jej elementów.

```
for (i=0; i<6; i++) x[i] = 0;
```

12.2. Operacje na łańcuchach

W języku C nie istnieje oddzielny typ reprezentujący łańcuchy znakowe, gdyż definicja łańcucha jest tożsama z definicją tablicy znaków, w której ostatnim znakiem jest '\0'. Biblioteki języka zawierają szereg funkcji umożliwiających wykonywanie operacji na łańcuchach. Prototypy tych funkcji są zawarte w zbiorze nagłówkowym <string.h>.

W programie można bezpośrednio zainicjować wskaźnik do zmiennej typu char adresem stałej łańcuchowej.

```
char *wznak;    // wskaźnik na znak – wskaźnik, który może
                // wskazywać na początek łańcucha znaków
wznak = "To jest C";      // przypisanie zmiennej wznak adresu początku
              // łańcucha znaków – łańcuch pamiętany w miejscu,
              //  w którym kompilator umieszcza inne stałe
```

Podobne przypisanie w przypadku tablicy tab prowadzi do błędu.

```
char tab[30];
tab = "To jest C";        // błąd kompilatora !
                          // poprawne tylko na etapie definicji tablicy
```

Kopiowanie można zrealizować wykorzystując funkcję memmove lub funkcję strcpy, która kopiuje łańcuch łącznie ze znakiem końca.

Przykład 12.6. Kopiowanie łańcuchów z wykorzystaniem funkcji memove oraz strcpy .

```
cout << sizeof("To jest C") << endl;       //  9 + 1 == 10

memmove(tab, "To jest C", sizeof("To jest C") );
cout << tab << endl;

strcpy(tab, "To jest C");        // to samo za pomocą strcpy
cout << tab << endl;    // funkcji przeznaczonej do obsługi łańcuchów
```

Wybrane funkcje operujące na łańcuchach

- Wyznaczanie długości łańcucha: **strlen**

```
size_t  strlen(const char *s);
```

Funkcja zwraca długość łańcucha s (bez znaku '\0').

Przykład 12.7. Implementacja funkcji strlen - wersja wskaźnikowa.

```
int strlen(const char *s)
{
  char *x = s;
       while (*x) x++;
       return (x - s);
}
```

Wywołanie dla:

```
char tab[ ] = "To jest C";

coutt << strlen(tab) << endl;          // wynik = 9;   bez znaku '\0'
```

- Kopiowanie łańcuchów: **strcpy, _fstrcpy**

```
char *strcpy(char *d, const char *s);
char far * _fstrcpy(char far *d, const char far *s);
```

Kopiowanie łańcucha z obszaru s do obszaru d łącznie ze znakiem końca '\0'. Wartością funkcji jest adres d.

Przykład 12.8. Implementacja mstrcpy - wersja wskaźnikowa.

```
char* mstrcpy(char *d, const char *s)
{   char *pom = d;
       while(*d++ = *s++);   return pom;    }
```

Wywołanie dla:

```
char *x = "To jest C";            // łańcuch statyczny; system alokuje pamięć
char tab[30];

cout << mstrcpy(tab, x) << endl;
cout << tab << endl;
```

Nie wolno kopiować tekstu do obszaru, którego rozmiar jest mniejszy niż długość tekstu łącznie ze znakiem końca. Kompilator nie wykrywa błędów tego typu.

```
char buf[7];
strcpy(buf, x);             //  rozmiar obszaru buf jest zbyt mały !
```

W programie można dokonać podstawienia char *x = "To jest C". Wówczas, do x jest wpisywany adres początku obszaru, w którym znajduje się łańcuch "To jest C". Nie można jednak kopiować danych do wskaźnika, który nie zawiera adresu odpowiednio dużego obszaru pamięci.

```
char *pom;                  // wskaźnik nie jest zainicjowany
strcpy(pom, "To jest C");
                            // kopiowanie w obszar o przypadkowym adresie !
```

- Porównywanie łańcuchów: **strcmp, _fstrcmp**

```
int strcmp(const char *s1, const char*s2);
int far _fstrcmp(const char far *s1, const char far *s2);
```

Porównywanie tekstów s1 i s2 przez porównywanie odpowiadających sobie znaków należących do tych łańcuchów traktowanych jako liczby bez znaku. Wartość funkcji jest mniejsza od 0 - jeśli napotkane zostaną różne znaki o kodach s1 < s2, równa 0 - jeśli dla wszystkich znaków zachodzi s1 = s2, większa od 0 - jeśli napotkane zostaną znaki o kodach s1 > s2.

Przykład 12.9. Implementacja funkcji strcmp - wersja wskaźnikowa.

```
int strcmp(const char *s1, const char * s2)
{
  while (*s1 == *s2)  {
          if   (*s1 == '\0') return 0;
          s1++ ;    s2++;          }
   return (*s1 - *s2);
}
```

Wywołanie dla:

```
char *s1 = "Alb";    char *s2 = "Ala";
cout << strcmp(s1,s2) << endl;    // 1   s1 > s2
```

- Łączenie łańcuchów: **strcat, _fstrcat**

```
char *strcat(char *d, const char *s);
char far * far _fstrcat(char far *d, const char far *s);
```

Dołączanie łańcucha s na koniec łańcucha d. Obszar wskazywany przez d powinien być na tyle duży, aby zmieścił się w nim dołączany łańcuch. Długość otrzymanego tekstu jest równa sumie długości tekstów d i s.

Przykład 12.10. Wykorzystanie funkcji strcat.

```
char d[20] = "To";                      // rozmiar bufora 20 bajtów
char *sp = " " ,  *s = "jest C";

strcat(d, sp);    strcat(d, s);
printf("%s\n", d);                      //  "To jest C";
```

- Poszukiwanie pierwszego wystąpienia znaku: **strchr, _fstrchr**

```
char *strchr(const char *s, int z);
char far * far _fstrchr(const char far *s, int z);
```

Znajdowanie pierwszego wystąpienia znaku z w łańcuchu s. Wartością funkcji jest adres pierwszego znalezionego znaku lub NULL, jeśli poszukiwany znak nie został znaleziony.

Przykład 12.11. Zastosowanie funkcji strchr.

```
char s[20] = "To jest C";
char *wsk;

int z = 'j';

wsk = strchr(s, z);

if (wsk) printf("%s\n%c", wsk, *wsk);      //  "jest C";  'j'
```

- Poszukiwanie ostatniego wystąpienia znaku: **strrchr, _fstrrchr**

```
char *strrchr(const char *s, int z);
char far * far _fstrrchr(const char far *s, int z);
```

Poszukiwanie ostatniego wystąpienia znaku z w łańcuchu s (inaczej - poszukiwanie pierwszego wystąpienia znaku podczas przeszukiwania łańcucha od końca). Wartością funkcji jest adres znalezionego znaku lub NULL, jeśli poszukiwany znak nie został znaleziony.

Przykład 12.12. Wykorzystanie funkcji strrchr.

```
char s[20] = "To jest moje C";
char *wsk;

int z = 'j';

wsk = strrchr(s, z);

if (wsk) printf("%s\n%c", wsk, *wsk);      //  "je C";  'j'
```

- Poszukiwanie podłańcucha w łańcuchu: **strstr, _fstrstr**

```
char *strstr(const char *s1, const char *s2);
char far * far _fstrstr(const char far *s1, const char far *s2);
```

Przeszukiwanie łańcucha s1 w poszukiwaniu podłańcucha s2. Zwracanie adresu pierwszego znaku podłańcucha lub NULL, jeśli podłańcuch s2 nie został znaleziony.

Przykład 12.13. Zastosowanie funkcji strstr.

```
char *s1 = "To jest moje C";
char *s2 = "moje", *wsk;

wsk = strstr(s1, s2);
if (wsk) cout << wsk << endl;                //  "moje C"
```

- Poszukiwanie znaczników: **strtok, _fstrtok**

```
char *strtok(char *s1, const char *s2);
char far * far _fstrtok(char far *s1, const char far *s2);
```

Przeszukiwanie łańcucha s1 w celu znalezienia ciągu znaków (wyrazu) rozdzielonego dowolną liczbą separatorów występujących w łańcuchu s2. Po znalezieniu początku pierwszego wyrazu w s1 jest za nim wstawiany NULL. Kolejne wywołania funkcji z parametrem NULL na pierwszej pozycji zwracają adresy kolejnych wyrazów w łańcuchu s1 zakończonych NULL i rozdzielonych separatorami. Funkcja może być wykorzystana do podziału tekstu na wyrazy. Jeśli separator nie zostanie odnaleziony, to zwracana jest wartość NULL.

Przykład 12.14. Zastosowanie funkcji strtok.

```
char *s1 = ";;;ABC,,123;;;;,,,XYZ";  // dowolna liczba separatorów typu , ;
char *wsk;

wsk = strtok(s1, ",;");    // pobranie do wsk adresu pierwszego wyrazu
if (wsk) printf("%s\n", wsk);    // wsk = "ABC"
                                 // w s1 za znakiem 'C' jest wstawiane '\0'
while (wsk) {                    // kolejne wywołania
  wsk = strtok(NULL, ",;");
  if (wsk)   printf("%s\n", wsk);       //  "123\nXYZ"
}
```

Wyniki:
ABC
123
XYZ

Do poszukiwania wyrazów w tekście można wykorzystać również funkcję strchr, ale może to wymagać konstrukcji skomplikowanego algorytmu.

Przykład. 12.15. Szukanie wyrazów w zdaniu w oparciu o strchr, strcpy oraz memmove. Separatorami są spacje.

```
char t[ ] = "To jest      tekst";
char tab[3][20];      // tablica pomocnicza 3 tablic po 20 znaków
int k=0;
char *p=t;           // pomocniczy 1
char *pom=t;       // pomocniczy 2

for (i=0; i<strlen(t); i++)
{
  p=strchr(p,' ');            // pozycja pierwszej spacji za wyrazem
  if (!p) break;
  memmove(tab[k], pom, p-pom);          // kopiowanie wyrazu
  tab[k][p-pom] = '\0';                 // dodanie 0
  k++;                                  // następny wyraz
  while (*p==' ') p++;                  // eliminowanie spacji między wyrazami
  pom=p;                                // początek następnego wyrazu
 }
  strcpy(tab[k], pom);    // kopiowanie ostatniego wyrazu łącznie z zerem

 for (i=0; i<=k; i++)  printf("%s\n", tab[i]);   // wydruk wyrazów
```

Wyniki:
To
jest
tekst

To samo zadanie można zrealizować za pomocą funkcji strtok.

Przykład. 12.16. Szukanie wyrazów w zdaniu w oparciu o strtok.

```
char *s1 = "To jest     tekst";
char *wsk;

 wsk = strtok(s1, " ");            // pierwsze wywołanie - separator spacja
 if (wsk) printf("%s\n", wsk);

        while (wsk) {                           // kolejne wywołania
          wsk = strtok(NULL, " ");
            if (wsk)  printf("%s\n", wsk);      //  "To\njest\ntekst"
            }
```

12.3. Dynamiczna alokacja bloków pamięci

Tablica jednowymiarowa jest ciągiem jednakowych elementów. W przypadku tablic zewnętrznych (globalnych) oraz tablic statycznych elementy tablicy są pamiętane w segmencie danych programu. Programista nie ma możliwości zwolnienia w programie pamięci przydzielonej dla tych tablic. Tablice automatyczne (lokalne) zdefiniowane wewnątrz funkcji są pamiętane na stosie. W momencie zakończenia funkcji są one automatycznie usuwane ze stosu.

W języku C istnieje możliwość dynamicznej alokacji bloków pamięci, o rozmiarze dowolnie zadawanym przez użytkownika, na stercie i korzystania z nich w programie w taki sam sposób jak z tablic. Służą do tego funkcje standardowe malloc, calloc. Tablice dynamiczne mają jednak tą zaletę, że mogą być usunięte z pamięci jeśli nie są już potrzebne.

W języku C++ można przydzielać pamięć dynamicznie za pomocą operatora new oraz standardowych funkcji malloc i calloc. Do zwalniania pamięci służą odpowiednio operator delete (dla new) oraz funkcja free (dla malloc, calloc). Prototypy funkcji:

```
char *t = new char [K];         // alokacja K bajtów (danych typu char)
                                // (sterta bliska lub daleka)
                                // akceptowane dane K typu long
void *malloc(size_t  K);        //  alokacja K bajtów
void *calloc(size_t  N, size_t K);    // alokacja N razy po K bajtów

// w BC++ 3.1 wersje dla wskaźników dalekich
void far *farmalloc(long K);            // alokacja w obszarze sterty dalekiej
void far *farcalloc(long N, long K);  // alokacja w obszarze sterty dalekiej

delete [ ] t;                  // zwolnienie obszaru wskazywanego przez t
                               // operator [ ] oznacza zwolnienie bloku pamięci
                               // związanego ze wskaźnikiem t
void free(void *x);                 // zwolnienie obszaru

void farfree(void far* x);        //zwolnienie obszaru
```

W BC++3.1 typ **size_t** jest zdefiniowany jako **unsigned**.

Operator new zwraca wskaźnik do typu elementu przydzielanego.

Na przykład: new char [K] zwraca wskaźnik typu (char *), natomiast new long [10] – wskaźnik typu (long *).

Funkcje malloc i calloc przydzielają spójne obszary pamięci o podanym rozmiarze i zwracają wskazanie do przydzielonego obszaru. Jeżeli alokacja nie jest możliwa, to zwracany jest wskaźnik NULL. Funkcja calloc dodatkowo zeruje przydzieloną pamięć. Funkcja free zwraca do systemu przydzieloną pamięć.

Funkcje malloc i calloc zwracają wskaźniki do typu void dlatego niezbędne są konwersje typu przy podstawieniach do wskaźników innych typów, np. double *p = (double *) malloc(10*sizeof(double)) prowadzi do alokacji bloku pamięci o rozmiarze 10 danych typu double.

Należy uważać, aby za pomocą funkcji free lub operatora delete nie zwalniać pamięci, która nie została przydzielona. Wskaźnik, dla którego zwolniono pamięć zachowuje swoją wartość, ale nie można za jego pomocą uzyskiwać dostępu do adresowanej pamięci (w trybie chronionym nawet nie da się czytać danych). Aby zapobiec przypadkowym odwołaniom do zwolnionej pamięci można przypisać wskaźnikowi, wartość NULL, np. delete p; p = NULL, lub free(p); p = NULL.

Przykład. 12.17. Tworzenie dynamicznej tablicy liczb całkowitych o rozmiarze N zadawanym przez użytkownika za pomocą operatora new i funkcji malloc oraz calloc. Inicjacja tablic. Zwolnienie tablic.

```
randomize();                // inicjacja generatora liczb losowych
cout << "Rozmiar tablicy " << endl;

int N;                      // rozmiar tablicy
scanf("%d", &N);            // wczytanie N

int *tab1 = new int [N];                        // alokacja N elementów typu int
int *tab2 = (int *) malloc( N*sizeof(int) );  // alokacja N*2 bajtów
int *tab3 = (int *) calloc(N, sizeof(int) );  // alokacja N*2 bajtów i zerowanie

if (!tab1 || !tab2 || !tab3) { cout << "Blad alokacji"<< endl; exit(0); }

for (int i=0; i<N; i++) {
  tab1[i] = random(100);     // losowa liczba z przedziału [0,99]
  tab2[i] = tab1[i];
  tab3[i] = tab1[i];
}

for (i=0; i<N; i++)                // wydruk zawartości tablic
 cout <<  tab1[i]  << "  " << tab2[i] << "  "<< tab3[i] << endl;

// zwolnienie pamięci

free(tab3);                 // zwolnienie dla calloc
tab3=NULL;                  // zabezpieczenie

free(tab2);                 // zwolnienie dla malloc
tab2 = NULL;                // zabezpieczenie

delete [ ] tab1;            // zwolnienie dla new
tab1 = NULL;                // zabezpieczenie
```

Przykład. 12.18. Opracować program, który utworzy dynamicznie tablicę liczb całkowitych o rozmiarze n wczytywanym z klawiatury.

a) Utworzyć tablicę t1 za pomocą operatora new.
b) Utworzyć tablicę t2 za pomocą funkcji malloc.

Jeżeli operacja przydziału pamięci dla t1 i t2 zakończyła się pomyślnie (wskaźniki t1 i t2 są różne od NULL), to zainicjować tablicę t1 losowymi liczbami należącymi do przedziału [0, 100). Znaleźć minimalny i maksymalny element w tablicy. Wyprowadzić zawartość tablicy t1, a także znalezione minimum i maksimum, na ekran. Przekopiować tablicę t1 do t2 wykorzystując standardową funkcję memmove lub memcpy. Wyprowadzić zawartość tablicy t2 na ekran. Przed zakończeniem programu zwolnić pamięć przydzieloną t1 i t2.

```
#include <iostream.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*---------------------------------------------------------------------*/
void inicjuj(int *w , int rozmiar)   // inicjacja tablicy
{
int i;
        for (i=0 ; i<rozmiar ; i++)
        {
                w[i]=random(100);
              // lub *w++ = random(100);
        }
}

/*---------------------------------------------------------------------*/
void pisz(int *w , int rozmiar)   // szukanie min i max
{
int i;
int min=w[0];
int max=w[0];
        for ( i=0 ; i<rozmiar ; i++)
        {
                if (w[i]<min) min=w[i];
                if (w[i]>max) max=w[i];
                if (i%10==0) cout << endl;
                cout.width(4);  cout<<w[i];
        }
        if (min!=max) cout << endl << "min : "<< min <<
                                                "\tmax : "<<max<<endl;
        else cout<<"min i max : "<< max << endl;
}

/*---------------------------------------------------------------------*/
void main()
{
int n;
int *t1;
int *t2;

        clrscr();      randomize();
        cout << "Proszę podać rozmiar tablicy  n= ";
        cin>>n;

        t1=new int [n];    // alokacja tablicy t1

        if ( t1 )
        {
                cout<<"Alokacja tablicy t1["<<n<<"] powiodla sie !!!"<<endl;
                cout<<"dowolny klawisz kontynuacja ... "<<endl;
                getch();
        }
          else { cout<<"Brak pamieci !!"; getch(); exit(0); };

        t2=(int *)malloc( n*sizeof(int) );   // alokacja tablicy t2

        if ( t2 )
        {
                cout<<"Alokacja tablicy t2["<<n<<"] powiodla sie !!!"<<endl;
                cout<<"dowolny klawisz kontynuacja ... "<<endl;
                getch();
        }
          else { cout<<"Brak pamieci !!"; exit(0); }

        inicjuj(t1,n);   // inicjacja t1
        pisz(t1,n);          // min i max w t1

        memmove( t2, t1, n*sizeof(int));  // kopiowanie t1 do t2

        pisz(t2,n);          // min i max w t2

        getch();

        free(t2);                   // kasowanie t2 dla malloc
        t2 = NULL;

        delete [ ] t1;              // kasowanie t1 dla new
        t1 = NULL;

        getch();
}
```