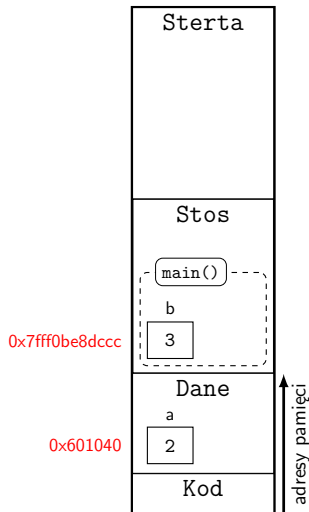


Wskaźniki

ADRESY ZMIENNYCH

```
1  #include<stdio.h>
2
3  int a = 2;
4
5  int main()
6  {
7      int b = 3;
8
9      printf("adres zmiennej a %p\n", &a);
10     printf("adres zmiennej b %p\n", &b);
11
12     return 0;
13 }
```

adres zmiennej a 0x601040
adres zmiennej b 0x7fff0be8dccc



WSKAŹNIK (*pointer*)

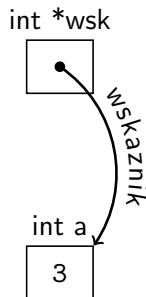
adres zmiennej w pamięci (np. `&a`)

```
int a = 5;
printf("%p\n", &a);
```

ZMIENNA WSKAŹNIKOWA

zmienna przechowująca adres

```
int *wsk;
wsk = &a;
```



TYP WSKAŹNIKOWY

typ zmiennej znajdujący się pod wskazanym adresem (np. `int`)

DEKLARACJA

`typ *identyfikator;`

PRZYKŁAD

```
int *wa;           /* wskaznik na zmienna typu int */
float *wx;         /* wskaznik na zmienna typu float */
char *wz;
void *w;           /* wskaznik na zmienna dowolnego typu */
int *t[10];        /* tablica zmiennych wskaznikowych */
int **ww;          /* wskaznik na zmienna wskaznikowa */
```

OPERATOR POBRANIA ADRESU

Operator referencji & zwraca adres zmiennej

PRZYKŁAD

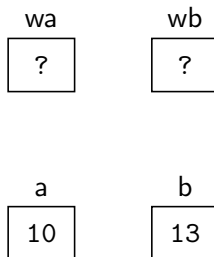
```
1  int a=10;
2  int b=13;
3  int *wa;
4  int *wb;
5
6  wa = &a;
7  wb = &b;
8
9  wb = wa;
```

OPERATOR POBRANIA ADRESU

Operator referencji & zwraca adres zmiennej

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa;  
4  int *wb;  
5  
6  wa = &a;  
7  wb = &b;  
8  
9  wb = wa;
```

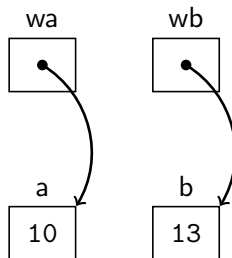


OPERATOR POBRANIA ADRESU

Operator referencji & zwraca adres zmiennej

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa;  
4  int *wb;  
5  
6  wa = &a;  
7  wb = &b;  
8  
9  wb = wa;
```

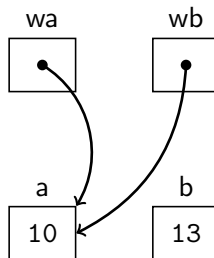


OPERATOR POBRANIA ADRESU

Operator referencji & zwraca adres zmiennej

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa;  
4  int *wb;  
5  
6  wa = &a;  
7  wb = &b;  
8  
9  wb = wa;
```



OPERATOR DOSTĘPU DO ADRESU

Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

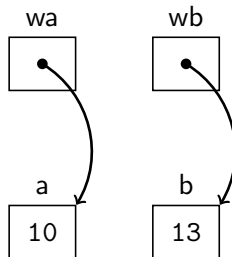
```
1  int a=10;
2  int b=13;
3  int *wa = &a;
4  int *wb = &b;
5
6  *wb = 5;
7
8  *wa = *wb;
9
10 wa = wb;
11 *wb = *wb + 1;
```

OPERATOR DOSTĘPU DO ADRESU

Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;  
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```

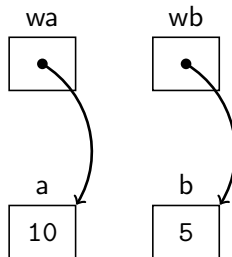


OPERATOR DOSTĘPU DO ADRESU

Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

```
1  int a=10;
2  int b=13;
3  int *wa = &a;
4  int *wb = &b;
5
6  *wb = 5;
7
8  *wa = *wb;
9
10 wa = wb;
11 *wb = *wb + 1;
```

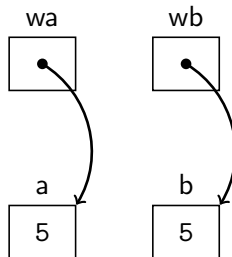


OPERATOR DOSTĘPU DO ADRESU

Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;  
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```

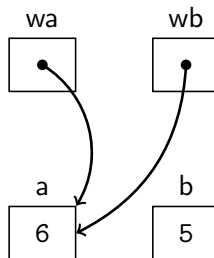


OPERATOR DOSTĘPU DO ADRESU

Operator dereferencji * daje dostęp do wskazanego adresu

PRZYKŁAD

```
1  int a=10;  
2  int b=13;  
3  int *wa = &a;  
4  int *wb = &b;  
5  
6  *wb = 5;  
7  
8  *wa = *wb;  
9  
10 wa = wb;  
11 *wb = *wb + 1;
```



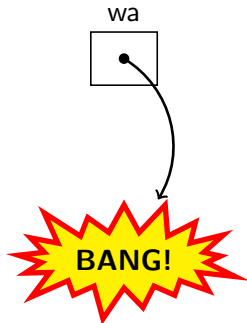
Uważaj na niezainicjowane zmienne wskaźnikowe.

PRZYKŁAD

```
int *wa;  
*wa = 5;
```

NULL TO ADRES 0

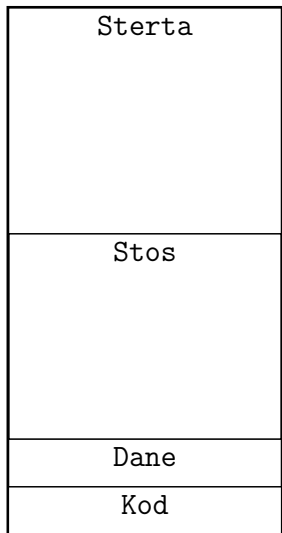
```
int *wa;  
wa = 0;  
wa = NULL;      /* stdlib.h */
```



- Nigdy nie używaj operatora `*` na niezainicjowanej zmiennej.
- Wskazanie puste, adres 0, NULL - informacja, że wskaźnik nic nie pokazuje

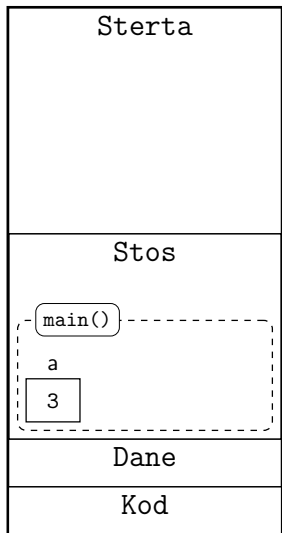
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n",a);
15
16     return 0;
17 }
```



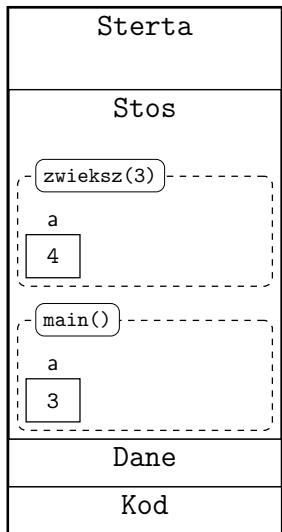
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\\n", a);
15
16     return 0;
17 }
```



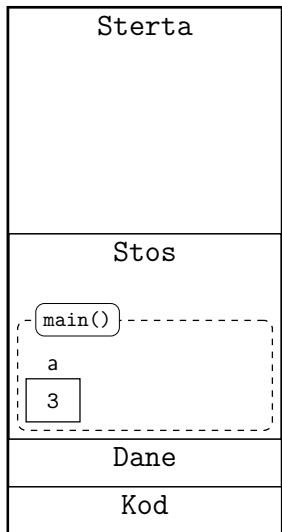
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



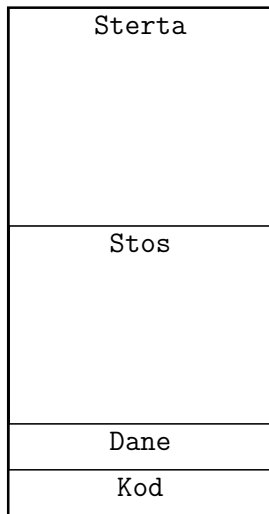
WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int a)
4  {
5      a = a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```

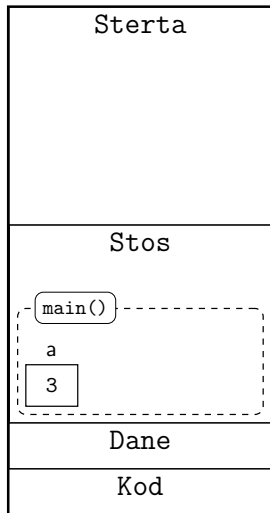


WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```

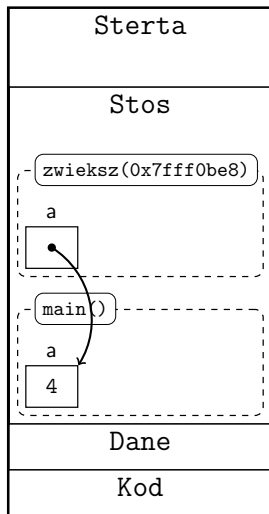


0x7fff0be8



WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

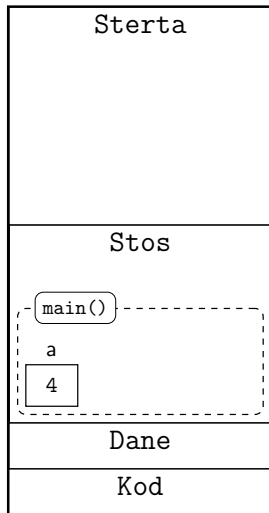
```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```



WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

```
1  #include<stdio.h>
2
3  void zwieksz(int *a)
4  {
5      *a = *a + 1;
6  }
7
8
9  int main()
10 {
11     int a = 3;
12
13     zwieksz(&a);
14     printf("%d\n", a);
15
16     return 0;
17 }
```

0x7fff0be8



WSKAŹNIKI JAKO ARGUMENTY FUNKCJI

- poprzez wskaźnik (adres zmiennej) funkcja może zwrócić dodatkową wartość
- `scanf("%d", &x) ←` funkcja modyfikuje zmienną `x`, stąd argumentem musi być adres



ZNAJDOWANIE MINIMUM I MAKSYMUM

Problem: znajdź wartość minimalną i maksymalną w zbiorze liczb

Algorytm 1 Wyznaczanie maksimum i minimum

Dane wejściowe: ciąg n liczb $\{x_1, x_2, \dots, x_n\}$

Wynik: wartość x_{min} i x_{max} , odpowiednio najmniejszy i największy element podanego ciągu

1: $x_{min} \leftarrow x_1$

2: $x_{max} \leftarrow x_1$

3: **dla każdego** $x \in \{x_2, \dots, x_n\}$ **wykonuj**

4: **jeżeli** $x < x_{min}$ **wykonaj**

5: $x_{min} \leftarrow x$

6: **jeżeli** $x > x_{max}$ **wykonaj**

7: $x_{max} \leftarrow x$

8: **zwróć** x_{min}, x_{max}

ELEMENT MINIMALNY I MAKSYMALNY

PRZYKŁAD W C

```
1 void minmax(const float t[], int n, float *min, float *max)
2 {
3     int i = 1;
4     *min=t[0];
5     *max=t[0];
6
7     while( i < n)
8     {
9         if( *min > t[i] ) *min = t[i];
10        if( *max < t[i] ) *max = t[i];
11        i = i + 1;
12    }
13 }
```

 minmax1.c

ELEMENT MINIMALNY I MAKSYMALNY

PRZYKŁAD W C

```
1  int main()  
2  {  
3      int n;  
4      float t[MAX], max, min;  
5  
6      n = wczytaj(t, MAX);  
7      minmax(t, n, &min, &max);  
8      printf("min=%f\nmax=%f\n", min, max);  
9  
10     return 0;  
11 }
```

 minmax1.c

ZŁOŻONOŚĆ PRZESZUKIWANIA

- Ilość porównań: $2(n - 1)$
- Czy istnieje szybszy sposób?
- Dziel i zwyciężaj !

Algorytm 2 Wyznaczanie maksimum i minimum

Dane wejściowe: ciąg n liczb $\{x_1, x_2, \dots, x_n\}$

Wynik: wartość x_{min} i x_{max} , odpowiednio najmniejszy i największy element podanego ciągu

- 1: $\mathcal{A} \leftarrow \emptyset, \quad \mathcal{B} \leftarrow \emptyset$
 - 2: **dla** $i = 1, 2, \dots, \lfloor \frac{n}{2} \rfloor$ **wykonuj**
 - 3: **jeżeli** $x_{2i-1} < x_{2i}$ **wykonaj**
 - 4: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_{2i-1}\}, \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_{2i}\}$
 - 5: **w przeciwnym wypadku**
 - 6: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_{2i}\}, \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_{2i-1}\}$
 - 7: **jeżeli** n jest nieparzyste **wykonaj**
 - 8: $\mathcal{A} \leftarrow \mathcal{A} \cup \{x_n\}, \quad \mathcal{B} \leftarrow \mathcal{B} \cup \{x_n\}$
 - 9: $x_{min} \leftarrow \min(\mathcal{A})$
 - 10: $x_{max} \leftarrow \max(\mathcal{B})$
 - 11: **zwróć** x_{min}, x_{max}
-

```

1 void minmax(const float t[], int n, float *min, float *max)
2 {
3     int i;
4     float tmin, tmax;
5
6     if( n==1 ){
7         *min=t[0];
8         *max=t[0];
9         return;
10    }
11
12    if( t[0]<t[1] ){
13        *min=t[0];
14        *max=t[1];
15    }
16    else{
17        *min=t[1];
18        *max=t[0];
19    }
20
21    for( i=2; i < n-1; i=i+2 ){
22        if( t[i]<t[i+1] ) {
23            tmin=t[i];

```

```

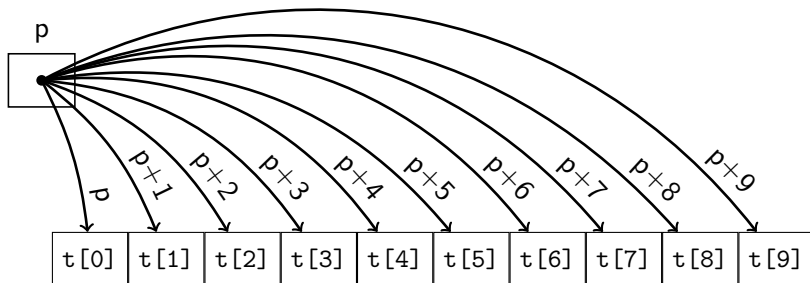
24         tmax=t [ i +1];
25     }
26     else {
27         tmin=t [ i +1];
28         tmax=t [ i ];
29     }
30     if( *max<tmax ) *max=tmax;
31     if( *min>tmin ) *min=tmin;
32 }
33 if( i == n-1 ){
34     if( *max<t [ i ] ) *max=t [ i ];
35     if( *min>t [ i ] ) *min=t [ i ];
36 }
37 }

```

 minmax2.c

WSKAŹNIKI DO ELEMENTÓW TABLIC

```
p = &t[0];
```



ARYTMETYKA NA WSKAŹNIKACH

```
1  int    a;  
2  char    b;  
3  int    *pa = &a;  
4  char    *pb = &b;  
5  
6  printf("pa    = %p %lu\n", pa    , pa    );  
7  printf("pa+1 = %p %lu\n", pa+1  , pa+1);  
8  printf("pb    = %p %lu\n", pb    , pb    );  
9  printf("pb+1 = %p %lu\n", pb+1  , pb+1);
```

 pointer.c

Przykładowy wynik:

```
pa    = 0x7fff44cb239c 140734347551644  
pa+1 = 0x7fff44cb23a0 140734347551648  
pb    = 0x7fff44cb239b 140734347551643  
pb+1 = 0x7fff44cb239c 140734347551644
```

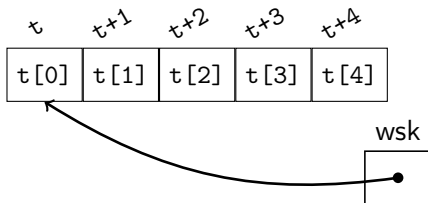
Tablica jest wskaźnikiem !

i -ty element $t[i] \Leftrightarrow *(t+i)$
 adres i -tego elementu $\&(t[i]) \Leftrightarrow t+i$

PRZYKŁAD

```

1  int t[5];
2  int *wsk;
3
4  wsk=t;
5  *t = 5;
6  *(t+2) = 6;
7  wsk = wsk + 1;
8  t = t + 1;
    
```



Adres tablicy jest stały

Źle !

PONOWNIE ŚRODEK MASY 2 PUNKTÓW

```
float* srodek(float p1[], float p2[])
{
    float sm[4];
    int i=0;
    sm[3]=p1[3]+p2[3];
    while(i<3)
    {
        sm[i]=(p1[3]*p1[i]+p2[3]*p2[i])/sm[3];
        i = i + 1;
    }
    return sm;
}
```

zmienna lokalna

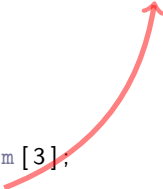
zwracany adres zmiennej lokalnej

Źle!

PONOWNIE ŚRODEK MASY 2 PUNKTÓW

- Deklaracja `const` zmiennej wskaźnikowej - kompilator wykryje próbę modyfikacji.
- Zmienna `sm` zawiera adres zmiennej, która zostanie zmodyfikowana przez funkcję `srodek()`.

```
void srodek(const float p1[], const float p2[], float sm[])
{
    int i=0;
    sm[3]=p1[3]+p2[3];
    while(i<3)
    {
        sm[i]=(p1[3]*p1[i]+p2[3]*p2[i])/sm[3];
        i = i + 1;
    }
}
```



Wskaźnik

WSKAŹNIK JAKO WARTOŚĆ ZWRACANA Z FUNKCJI

```
1 float *wczytaj(float *t, int n)
2 {
3     float *p=t;
4     printf("Wprowadz %d liczb\n", n);
5     while( n >0 )
6     {
7         scanf("%f",t);
8         t = t + 1;
9         n = n - 1;
10    }
11    return p;
12 }
13
14 int main()
15 {
16     float t[100];
17     float min,max;
18     minmax(wczytaj(t,10), &min, &max);
19 }
```

ROZSZYFRUJ DEKLARACJE

PRZYKŁADOWE DEKLARACJE FUNKCJI

```
/* Miejsca zerowe paraboli */
int pierwiastki(float a, float b, float c, float *x1, float *x2);

/* Wyszukiwanie binarne */
int szukaj(const int *t, int n, int x);

/* Srodek masy ukladu punktow */
struct punkt srodek(const struct ogromna_chmura_punktow *u);

/* Dekompozycja liczby zmiennopozycyjnej (math.h) */
float modf(float num, float *i);

/* Alokacja pamieci (stdlib.h) */
void* malloc(int size);

/* Kopiowanie tablic (stdlib.h) */
void* memcpy(void *dest, const void *src, int count);

/* Otwieranie pliku (stdio.h) */
FILE *fopen( const char *filename, const char *mode );
```

PRZYKŁAD: WYSZUKIWANIE DOMINANTY

Problem: znajdź wartość występującą najczęściej razy w zbiorze

8	1	-6	3	5	7	4	9	2	10	-4	88	6	3	1	3	332	2
---	---	----	---	---	---	---	---	---	----	----	----	---	---	---	---	-----	---

Algorytm 3 Wyznaczanie dominanty (mody) - algorytm naiwny

Dane wejściowe: ciąg n elementów $\{x_1, x_2, \dots, x_n\}$

Wynik: wartość dominanty x_{moda} oraz ilość wystąpień l_{moda} . Jeżeli istnieje więcej niż jedna wartość dominującą to zwracana jest pierwsza znaleziona.

- 1: $l_{moda} \leftarrow 0$
 - 2: **dla każdego** $x \in \{x_1, \dots, x_n\}$ **wykonuj**
 - 3: $k \leftarrow 0$
 - 4: **dla każdego** $y \in \{x_1, \dots, x_n\}$ **wykonuj**
 - 5: **jeżeli** $x = y$ **wykonaj**
 - 6: $k \leftarrow k + 1$
 - 7: **jeżeli** $k > l_{moda}$ **wykonaj**
 - 8: $l_{moda} \leftarrow k$
 - 9: $x_{moda} \leftarrow x$
 - 10: **zwróć** x_{moda}, l_{moda}
-

```

1  int dominanta(const int *t, int n, int *c)
2  {
3      int i, j, k, x;
4
5      *c = 0;
6      i=0;
7      while( i<n )
8      {
9          k=0;
10         j=0;
11         while( j<n )
12         {
13             if ( t[i]==t[j] ) k = k + 1;
14             j = j + 1;
15         }
16         if( k > *c )
17         {
18             *c = k;
19             x = t[i];
20         }
21         i = i + 1;
22     }
23     return x;
24 }

```

 dominanta1.c

ZŁOŻONOŚĆ ALGORYTMU

- ilość operacji rzędu n^2
- jeżeli pewien element został aktualnie zaznaczony jako dominujący to nie musimy powtarzać dla niego obliczeń
- zliczanie można rozpocząć od $i + 1$ miejsca, jeżeli wartość dominująca pojawiła się wcześniej to już została policzona
- jeśli aktualna wartość dominująca ma k wystąpień, to szukanie możemy przerwać na pozycji $n - k$ w zbiorze

```

1 int dominanta2(const int *t, int n, int *c)
2 {
3     int i, j, k, x;
4
5     *c=0;
6     x=t[0]-1;
7     i=0;
8     while( i<n-*c )
9     {
10         if( t[i]!=x )
11         {
12             k=1;
13             j=i+1;
14             while( j<n )
15             {
16                 if ( t[i]==t[j] ) k = k + 1;
17                 j = j + 1;
18             }
19             if( k > *c )
20             {
21                 *c = k;
22                 x = t[i];
23             }
24         }
25         i = i + 1;
26     }
27     return x;
28 }

```

 dominanta2.c

DYNAMICZNY PRZYDZIAŁ PAMIĘCI

ALOKACJA PAMIĘCI

```
void *malloc(int rozmiar);
```

Funkcja `malloc` zwraca adres przydzielonego bloku pamięci lub wartość 0 (NULL) w przypadku niepowodzenia.

ZWOLNIENIE PRZYDZIELONEJ PAMIĘCI

```
void free(void *wskaznik);
```

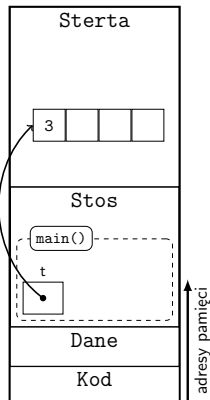
Argumentem funkcji `free` jest adres uzyskany wcześniej z funkcji `malloc`.

Funkcja `malloc` i `free` zadeklarowane są w pliku `stdlib.h`.

```

1  #include<stdlib.h>
2  #include<stdio.h>
3
4  int main()
5  {
6      float *t;
7
8      t = malloc( 4 * sizeof(int));
9      if( t == NULL ){
10         printf("Bład alokacji pamieci.\n");
11         exit(1);
12     }
13
14     *t = 3;
15
16     free(t);
17
18     return 0;
19 }

```



- **Wskaźnik** to adres zmiennej w pamięci a **zmienna wskaźnikowa** to zmienna przechowująca adres
- Operator referencji (adresowania) & daje adres zmiennej, &a to adres zmiennej a
- Operator dereferencji (wyłuskania) *
*b daje dostęp do wartości wskazywanej przez zmienną b
- Deklaracja zmiennej wskaźnikowej również zawiera znak *
`int *wsk;`
`float* wsk2;`
- Tablice to wskaźniki $t[i] \Leftrightarrow *(t+i)$
- Przekazanie wskaźnika do funkcji pozwala na modyfikację zmiennej wskazywanej
- Uważaj na co wskazujesz!

-  Maciej M. Sysło, „*Algorytmy*”, WSiP, Warszawa, 2002.
-  David Griffiths, Dawn Griffiths „*Rusz głową! C.*”, Helion, Gliwice, 2013.
-  „Kurs programowania w C”, WikiBooks,
<http://pl.wikibooks.org/wiki/C>