

Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>


Aktualizacja: 05.04.2020

6. Wskaźniki

Wskaźniki – wprowadzenie (1)

- Podczas wykonania programu wszystkie jego elementy (zmienne, wartości stałych, funkcje) umieszczone są w pamięci.
- Każdy z nich ma adres będący nieujemną liczbą całkowitą
- Adres jest pojęciem niskopoziomowym. Adresy są argumentami rozkazów procesora.

Adres	Zawartość
2026708	01110010
2026709	00001110
2026710	11111110
2026711	11111111
2026712	11111111
2026713	11111111
2026714	00011110



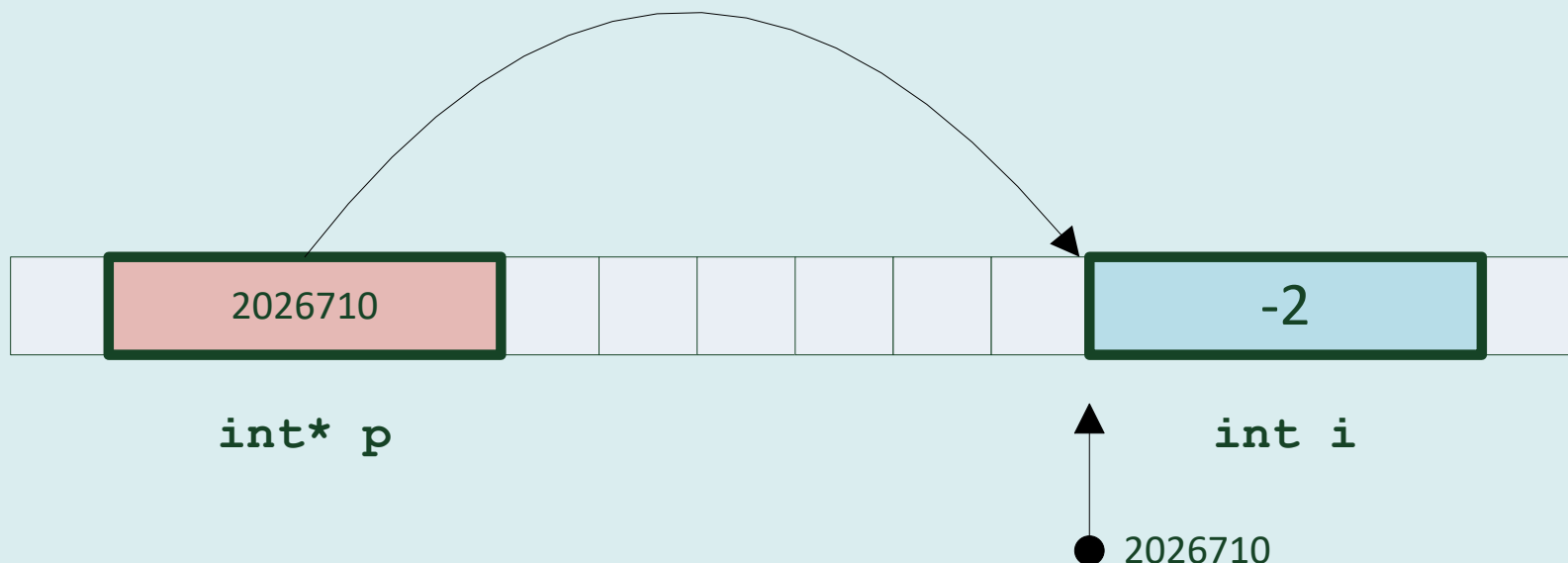
`int i`

Wskaźniki – wprowadzenie (2)

Wskaźniki są to zmienne, których wartościami są adresy. Korzystając ze wskaźników możemy:

- odczytać lub zmodyfikować wartość zmiennej zajmującą pamięć identyfikowaną przez adres
- wywołać funkcję

Zmienne wskaźnikowe **mają określone typy**. Informacje o typie są uzupełnieniem informacji o adresie. Dzięki znajomości typu kompilator może określić ile bajtów zajmuje wskazywany element i w jaki sposób należy interpretować dane (np.: jako `float` albo `int`).



Wskaźniki - deklaracje

Składnia deklaracji:

`type-specifier * pointer`

`type-specifier`

definiuje typ wskazywanego obiektu

`pointer`

identyfikator zmiennej

```
int *pi, tab[10];  
double *pd;  
float*px, *py, x, y;
```

Operatory adresu i dereferencji (1)

Język C definiuje dwa operatory umożliwiające posługiwanie się wskaźnikami:

- Jednoargumentowy operator adresu `&` (ang. *address operator*)

```
int x;  
int *px;  
px=&x;  
printf("%p ", &x);
```

- Dereferencji `*` (ang. *dereference, indirection operator*)

```
*px=7;  
printf("%d ", *px+3)
```


Operatory adresu i dereferencji (2)

Operator adresu &

- Operator adresu & pobiera adres obiektu będącego jego argumentem i zwraca wskaźnik zgodny z typem argumentu;
- Argumentem operatora adresu musi być obiekt, który ma przypisaną lokalizację w pamięci (zmienna, identyfikator funkcji).
- Nie można pobrać adresów zmiennych rejestrowych lub pól bitowych.

```
TYPE a;  
&a;
```

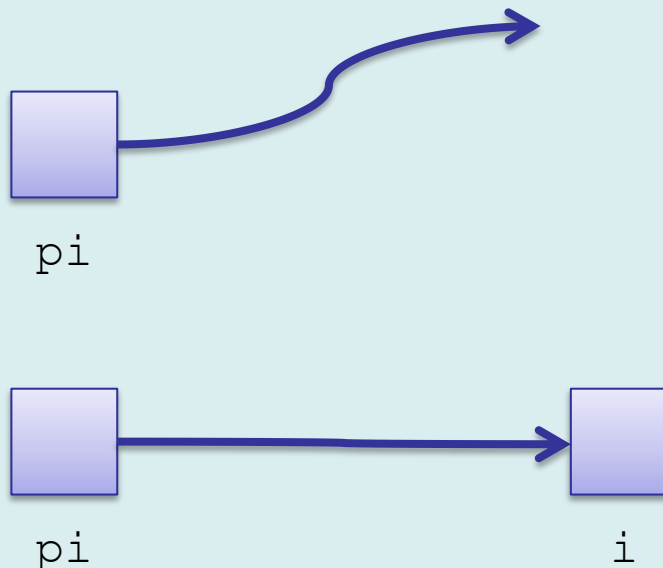
wskaźnik typu `TYPE*` o
wartości będącej adresem `a`



Operatory adresu i dereferencji (3)

Deklaracja zmiennej wskaźnikowej przydziela dla niej pamięć, ale wskaźnik nie musi wskazywać jakiegokolwiek obiektu.

```
int *pi;  
int i;  
  
pi=&i;
```



Deklarując wskaźnik można nadać mu wartość będącą adresem istniejącego obiektu

```
int i, *pi=&i;
```


Operatory adresu i dereferencji (4)

Operator dereferencji *

- W specyfikacji języka C terminem *obiekt* określany jest obszar pamięci, którego zawartość może być odczytywana/modyfikowana.
- *Lvalue* to wyrażenie identyfikujące taki obiekt. (Rozróżnienie lvalue i rvalue pochodzi z definicji operatora przypisania *lvalue = rvalue*)
- Operator dereferencji zwraca *lvalue* – wyrażenie identyfikujące wskazywany obiekt (mieszczący się pod wskazanym adresem)
- Typ argumentu określa typ zwracanego wyrażenia: jeżeli wskaźnik jest typu `TYPE*` zwracane wyrażenie jest typu `TYPE`

```
int i=7, j, *pi=&i;  
printf("%d ", *pi);
```

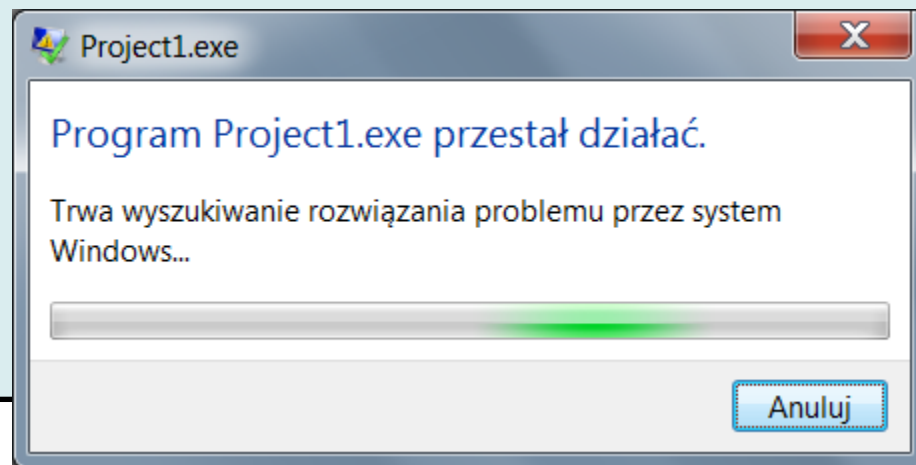
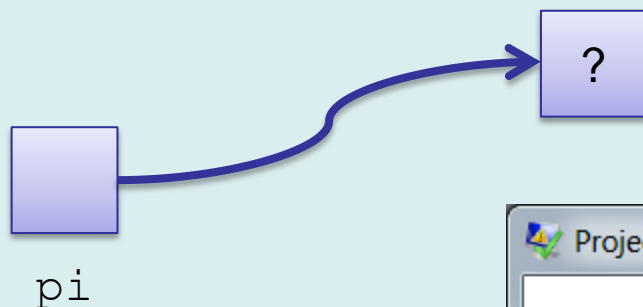
```
* (int*) 20000=7;  
// ale nie *20000
```

```
j=*&i; // znoszące się operatory  
// ale nie j=&*i;  
j=(int) &* (char*) i; //ok
```

zmienna typu `int` mieszcząca się pod adresem 20000

Operatory adresu i dereferencji (5)

Nigdy nie należy stosować operatora dereferencji do niezainicjowanych zmiennych (albo mających takie wartości jak 0 lub NULL) ...



```
int *pi, *pj=NULL;
printf("%d ", *pi);
*pj=7;
```

Operatory adresu i dereferencji (5)

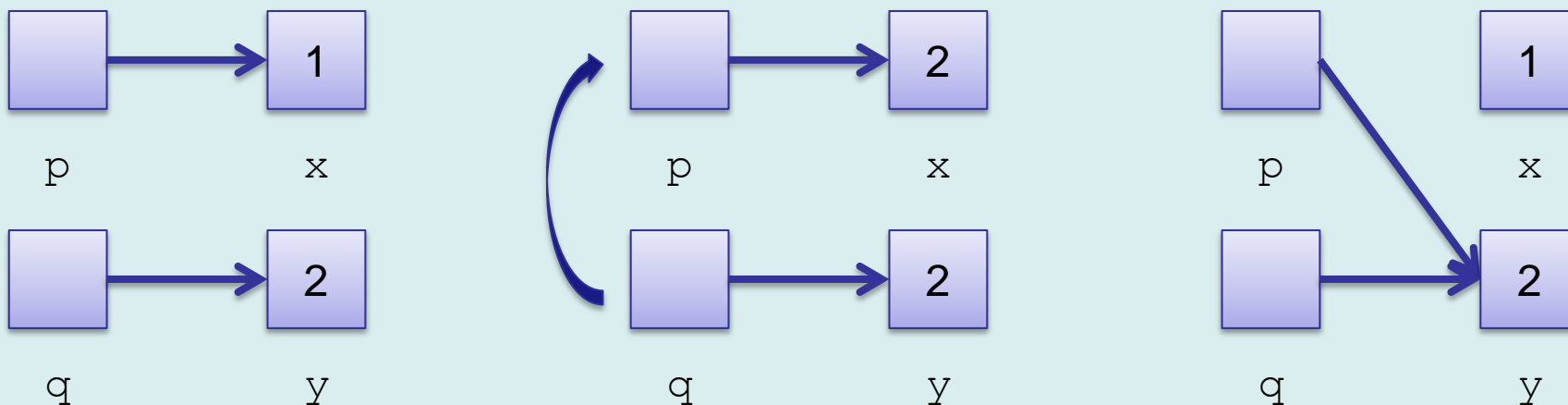
Przypisania:

```
int x=1, *p = &x, y=2, *q = &y;
```

```
*p=*q; // równoważne x = y  
// spełniona jest równość *p==*q
```

```
x=1;
```

```
p=q; // wskaźniki wskazują ten sam obiekt (y)  
// spełnione jest p==q i *p==*q
```



Dostęp do pól struktur i unii (1)

Operator kropkowy dostępu do pól struktur ma większy priorytet niż operator dereferencji.

```
struct complex {double re; double im;};  
struct complex vx={1,0};  
struct complex *pc=&vx;  
printf("( %f, %f) ", *pc.re, *pc.im) ;
```

21 main.c request for member `re' in something not a structure or union
21 main.c request for member `im' in something not a structure or union

Rozwiązania:

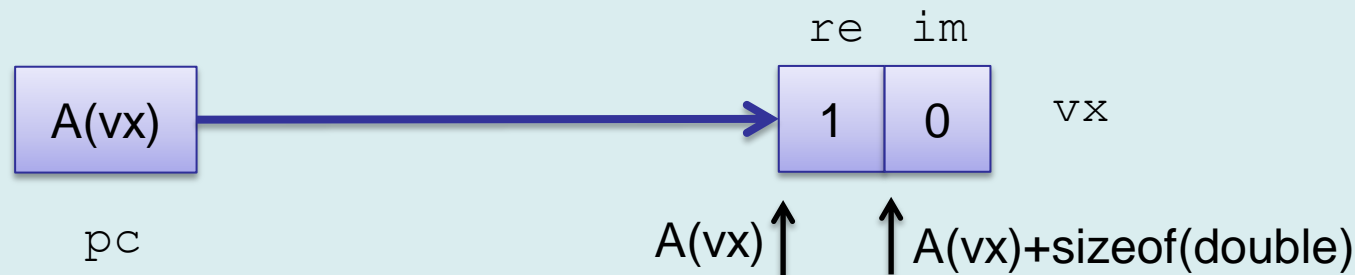
- Można zastosować nawiasy `(*pc).re`
- lub specjalny operator `->` : `pc->re`

```
printf("( %f, %f) ", (*pc).re, (*pc).im) ;  
pc->re=0;pc->im=1;
```

Dostęp do pól struktur i unii (2)

```
struct complex {double re; double im;};  
struct complex vx={1,0};  
struct complex *pc=&vx;
```

- **(*pc)** to *lvalue* identyfikująca zadeklarowaną wcześniej zmienną **vx**;
- **pc->im** to również wyrażenie *lvalue* równoważne **(*pc).im** oraz **vx.im**
- W wygenerowanym kodzie kompilator posługuje się adresami.
(Zapewne pole **re** ma adres początku struktury, natomiast pole **im** adres przesunięty o 8B)



Zastosowania wskaźników (1)

Podstawowe zastosowania wskaźników to:

- Możliwość modyfikacji obiektu zdefiniowanego na zewnątrz funkcji
- Ustalanie powiązań pomiędzy obiektami

Inne zastosowania to:

- Zarządzanie danymi tworzonymi dynamicznie (tablicami, listami, drzewami)
- Realizacja polimorfizmu w C++

Zastosowania wskaźników (2)

Modyfikacja zewnętrznych obiektów

- Standardowo, zmienne przekazywane są do funkcji *przez wartość*. Oznacza to, że wartością parametru funkcji jest kopia argumentu. Działania na parametrze funkcji nie modyfikują oryginalnego obiektu.
- Jeżeli do funkcji przekazany zostanie wskaźnik zawierający adres zewnętrznego obiektu, możliwa jest modyfikacja jego zawartości.

```
void foo(int * x){  
    (*x)++;  
    printf("x in foo=%d\n", *x);  
}
```

```
int main(){  
    int x = 2;  
    printf("x in main=%d\n", x);  
    foo(&x);  
    printf("x in main=%d\n", x);  
    return 0;  
}
```

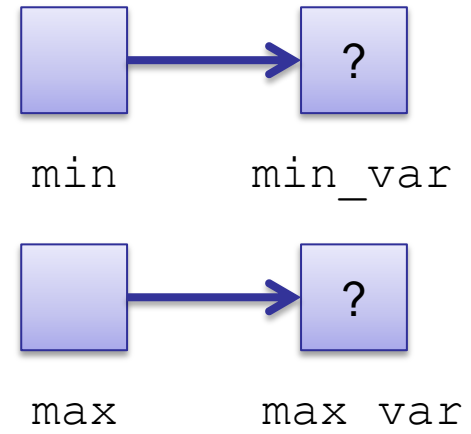
```
x in main=2  
x in foo=3  
x in main=3
```

Zastosowania wskaźników (3)

Przykład – obliczanie minimalnej i maksymalnej wartości elementu tablicy

```
void min_max(int tab[], int n, int*min, int*max) {  
    int i;  
    *max=*min=tab[0];  
    for(i=0; i<n; i++) {  
        if(*min>tab[i]) *min=tab[i];  
        if(*max<tab[i]) *max=tab[i];  
    }  
}
```

```
int main()  
{  
    int min_var, max_var;  
    int t[]={3,4,7,2,-2,234};  
    min_max(t, sizeof t/sizeof t[0], &min_var, &max_var);  
    printf("min = %d max = %d\n", min_var, max_var);  
    return 0;  
}
```

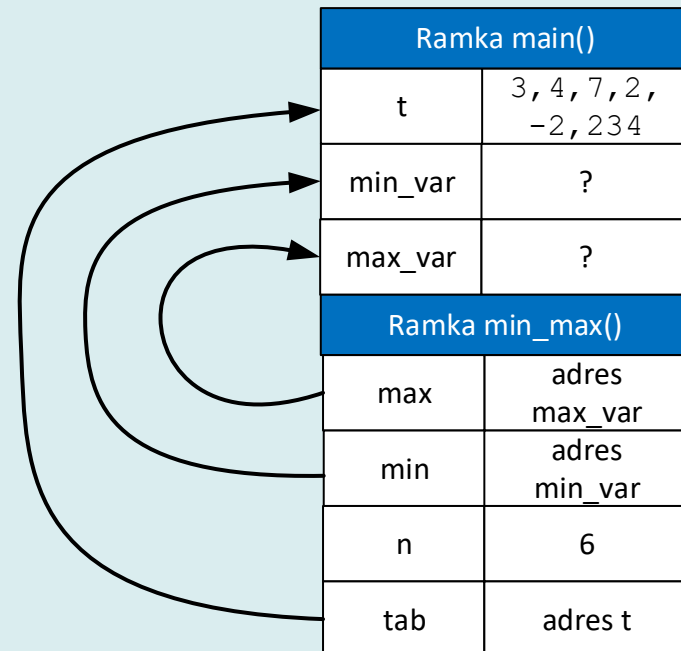


min = -2 max = 234

Ramki funkcji?

```
void min_max(int tab[], int n, int*min, int*max);
int main() {
    int min_var, max_var;
    int t[]={3,4,7,2,-2,234};
    min_max(t, sizeof t/sizeof t[0], &min_var, &max_var);
    printf("min = %d max = %d\n", min_var, max_var);
    return 0;
}
```

Po wywołaniu funkcji `min_max()` jej ramka zawiera wskaźniki na zmienne `max_var` i `min_var`, wyznaczoną liczbę elementów tablicy oraz adres tablicy `t`.



Zastosowania wskaźników (4)

Przykład – zbiór funkcji działających na strukturze complex

W języku C często wykorzystuje się wskaźniki przy tworzeniu bibliotek funkcji działających na określonych typach danych.

```
struct complex {double re,im;} ;

void init(struct complex*pc,double x,double y) {
    pc->re=x;
    pc->im=y;
}

void add(struct complex*c,
         struct complex*a, struct complex*b){
    c->re=a->re + b->re;
    c->im=a->im + b->im;
}

double absoluteValue(struct complex*c) {
    return(sqrt(c->re * c->re + c->im * c->im));
}
```

Zastosowania wskaźników (5)

Przykład – kontynuacja

```
void dump(struct complex*c) {  
    printf("(%.2f, %.2f) ", c->re, c->im);  
}  
int main()  
{  
    struct complex c1, c2, c3;  
    init(&c1, 12.34, 1.5);  
    dump(&c1);  
    init(&c2, -12.34, 1.5);  
    dump(&c2);  
    add(&c3, &c1, &c2);  
    dump(&c3);  
    return 0;  
}
```

```
(12.340000, 1.500000) (-12.340000, 1.500000)  
(0.000000, 3.000000)
```

Zastosowania wskaźników (6)

- Używając wskaźników, jako argumentów funkcji należy wystrzegać się błędów: wartość wskaźnika może być adresem nieokreślonym.
- Poprawność adresów nie jest sprawdzana w trakcie wykonania.

```
void foo(int * x)
{
    (*x)++;
    printf("x in foo=%d\n", *x);
}

int main()
{
    int *px;
    foo(px); // błąd px nie wskazuje żadnej zmiennej
    return 0;
}
```

Zastosowania wskaźników (7)

- Szczególną wartością adresu jest **wartość zerowa**. W praktyce żadna zmienna nie może zajmować obszaru pamięci o zerowym adresie, stąd wartość zerowa jest często traktowana jako znacznik błędu.
- Typowym zabezpieczeniem funkcji działających na wskaźnikach jest testowanie, czy argument nie jest równy 0 (NULL).

```
int add(struct complex*c,  
        struct complex*a, struct complex*b)  
{  
    if(!a) return 0;  
    if(!b) return 0;  
    if(!c) return 0;  
    c->re=a->re + b->re;  
    c->im=a->im + b->im;  
    return 1;  
}
```

Wskaźniki – modyfikator `const` (1)

- Projektując interfejs funkcjonalny często z góry potrafimy określić, że dana funkcja może modyfikować wartość zewnętrznego obiektu, albo też powinna jedynie mieć prawo do jej odczytu.
- Ten typ prawa dostępu może być sprawdzany w trakcie kompilacji.
- Modyfikator `const` użyty przy deklaracji wskaźnika umożliwia ograniczenie prawa dostępu wyłącznie do odczytu.

Przykład 1

```
int x=7;
const int*px=&x;

printf("x = %d\n", *px); //ok.

*px=2; /* błąd! za pośrednictwem px nie wolno
        modyfikować wartości wskazywanej zmiennej x */
```

Wskaźniki – modyfikator const (2)

- Zazwyczaj obecność modyfikator `const` jest informacją dla programisty, że parametr jest parametrem wejściowym.
- Brak tego modyfikatora oznacza, że parametr może być obliczany wewnątrz funkcji (jest jej dodatkowym rezultatem).

Przykład 2

```
int add( struct complex*c,          /* out (inout) */
        const struct complex*a,    /* in */
        const struct complex*b    /* in */);
```

Prototyp deklaruje funkcję, `add` która ma prawo modyfikować zmienną wskazywaną przez `c`, ma prawo odczytywać zawartość zmiennych wskazywanych przez `a` i `b`.

Przykład 3

```
void foo( const struct complex*in){
    struct complex a={0,1};
    struct complex b={1,0};
    add(in,&a,&b); // błąd add może modyfikować in
}
```

Wskaźniki – jako zwracane wartości (1)

Funkcja może również zwracać wskaźnik do obiektu. Problemem jest jednak lokalizacja obiektu, do którego jest zwracany wskaźnik.

```
int*max(int*a, int*b)
{
    if(*a>*b) return a;
    return b;
}
```

Jeden ze wskaźników
dostarczonych z zewnątrz

```
struct complex* add2( struct complex*c,
struct complex*a, struct complex*b)
{
    c->re=a->re + b->re;
    c->im=a->im + b->im;
    return c;
}
```


Wskaźniki – jako zwracane wartości (1)

- Funkcja **nie może** zwracać wskaźnika do zmiennej automatycznej (zadeklarowanej wewnątrz funkcji, dla której pamięć zostanie przydzielona na stosie).
- Po wyjściu z funkcji wskaźnik stosu zostanie przesunięty i pamięć zmiennej zniknie!
- Kompilatory zazwyczaj raportują ostrzeżenia lub błędy

```
struct complex* add3(struct complex*a, struct complex*b){  
    struct complex result;  
    result.re=a->re + b->re;  
    result.im=a->im + b->im;  
    return &result;  
}
```

122 main.c [Warning] function returns address of local variable

Wskaźniki – jako zwracane wartości (3)

- Funkcja może zwrócić wskaźnik do zmiennej o statycznym czasie życia (globalnej lub zadeklarowanej jako `static`).
- Niebezpieczeństwem jest przechowywanie wskaźnika – kolejne wywołanie funkcji zmodyfikuje wartość zmiennej.

```
struct complex result;
struct complex* add4( struct complex*a, struct complex*b){
    result.re=a->re + b->re;
    result.im=a->im + b->im;
    return &result;
}

struct complex* add5(struct complex*a, struct complex*b){
    static struct complex result;
    result.re=a->re + b->re;
    result.im=a->im + b->im;
    return &result;
}
```

Wskaźniki – jako zwracane wartości (4)

- Funkcja może zwrócić wskaźnik do obiektu (zmiennej, struktury, tablicy), dla którego pamięć została przydzielona na stercie.
- Problemem jest równoczesne użycie obiektów, które nie wymagają zwolnienia pamięci (pamięć przydzielona na stosie) oraz tych, które należy usunąć jawnie (pamięć przydzielona na stercie).

```
struct complex* add6(struct complex*a, struct complex*b){  
    struct complex*result=malloc(sizeof(struct complex));  
    result->re=a->re + b->re;  
    result->im=a->im + b->im;  
    return result;  
}  
  
int main() {  
    struct complex a,b;  
    init(&a,1,2);  
    init(&b,3,4);  
    struct complex*r=add6(&a,&b);  
    dump(r);  
    free(r);  
}
```

Wskaźniki – powiązania obiektów (1)

- Wskaźniki mogą być wykorzystane do ustalania powiązań (asocjacji) pomiędzy obiektami.
- Zazwyczaj tymi obiektami są struktury, a wskaźniki ich polami.

```
struct person
{
    char name[32];
    struct person*father;
    struct person*mother;
};

struct person adam = {"Adam", NULL, NULL};
struct person ewa = {"Ewa", NULL, NULL};
struct person kain = {"Kain", &adam, &ewa};
struct person abel = {"Abel", &adam, &ewa};
struct person set = {"Set", &adam, &ewa};
struct person enosh = {"Enosh", &set, NULL};
```

Wskaźniki – powiązania obiektów (2)

Korzystanie z informacji o powiązaniach

```
void about(struct person*p) {
    printf("%s: ", p->name);
    if(p->father!=NULL)
        printf("Ojciec: %s ", p->father->name);
    else printf("Ojciec: nieznany ");
    if(p->mother!=NULL)
        printf("Matka: %s ", p->mother->name);
    else printf("Matka: nieznana");
    printf("\n");
}

int main() {
    about(&adam); about(&ewa); about(&kain); about(&abel);
    about(&set); }
```

```
Adam: Ojciec: nieznany Matka: nieznana
Ewa: Ojciec: nieznany Matka: nieznana
Kain: Ojciec: Adam Matka: Ewa
Abel: Ojciec: Adam Matka: Ewa
Set: Ojciec: Adam Matka: Ewa
```

Wskaźniki – powiązania obiektów (3)

Drzewo genalogiczne

Rekurencyjne wyszukiwanie przodków

```
void indent(int level){
    while(level>0){printf("  ");level--;}
}
void genalogy(struct person*p,int level)
{
    indent(level);printf("%s:\n",p->name) ;

    indent(level+1);printf("Ojciec:");
    if(p->father!=NULL) genalogy(p->father,level+1);
    else printf("nieznany\n");

    indent(level+1);printf("Matka:");
    if(p->mother!=NULL) genalogy(p->mother,level+1);
    else printf("nieznana\n");

    printf("\n");
}
```

Wskaźniki – powiązania obiektów (4)

Drzewo genalogiczne

Funkcja rekurencyjne wyszukuje przodków (wpierw od strony ojca, potem od strony matki).

```
int main()
{
    genalogy(&enosh, 0);

    genalogy(&abel, 0);
    return 0;
}
```

Enosh:

Ojciec: Set:

Ojciec: Adam:

Ojciec:nieznany

Matka:nieznana

Matka: Ewa:

Ojciec:nieznany

Matka:nieznana

Matka:nieznana

Abel:

Ojciec: Adam:

Ojciec:nieznany

Matka:nieznana

Matka: Ewa:

Ojciec:nieznany

Matka:nieznana

Wskaźniki – powiązania obiektów (5)

Relacja przodek-potomek

- Funkcja rekurencyjnie sprawdza, czy pomiędzy dwiema osobami `parent` i `ch` zachodzi relacja pokrewieństwa (czy `parent` jest przodkiem `ch`)
- Wyrażenie **`isAncestor`** (`parent, ch->mother`) || **`isAncestor`** (`parent, ch->father`) – jeżeli pierwszy warunek alternatywy jest prawdziwy, drugi nie jest sprawdzany

```
enum {false=0,true=1};
```

```
int isAncestor(struct person*parent, struct person*ch) {  
    if(ch==NULL) return false;  
    if(parent==ch->mother || parent==ch->father)  
        return true;  
    return isAncestor(parent, ch->mother) ||  
           isAncestor(parent, ch->father);  
}
```


Wskaźniki – powiązania obiektów (6)

Wywołanie

```
void checkIfAncestor(struct person*parent,struct person*ch) {
    if(isAncestor(parent,ch))
        printf("%s jest przodkiem %s\n", parent->name,ch->name);
    else
        printf("%s nie jest przodkiem %s\n",parent->name,ch->name);
}

int main()
{
    checkIfAncestor(&abel,&enosh);
    checkIfAncestor(&ewa,&enosh);
    return 0;
}
```

Abel nie jest przodkiem Enosh
Ewa jest przodkiem Enosh

```
struct person adam ={"Adam",NULL,NULL};
struct person ewa = {"Ewa",NULL,NULL};
struct person kain ={"Kain",&adam,&ewa};
struct person abel ={"Abel",&adam,&ewa};
struct person set = {"Set",&adam,&ewa};
struct person enosh = {"Enosh",&set,NULL};
```

Pamięć dla struktur

- Czy można w jakiś inny sposób przydzielić pamięć dla struktur opisujących osoby?

```
struct person*create_person(const char*name,struct person*f,struct person*m){  
    struct person *p = malloc(sizeof(struct person));  
    strcpy(p->name,name);  
    p->father = f;  
    p->mother = m;  
    return p;  
}
```

- Można przydzielić pamięć za pomocą funkcji malloc().
- Koniecznie należy ją zwolnić za pomocą free()

```
int main(){  
    struct person*adam = create_person("Adam",NULL,NULL);  
    struct person*ewa=create_person("Ewa",NULL,NULL);  
    struct person*kain=create_person("Kain",adam,ewa);  
    struct person*abel=create_person("Abel",adam,ewa);  
    struct person*set=create_person("Set",adam,ewa);  
    struct person*enosh=create_person("Enosh",set,NULL);  
    genalogy(enosh,0);  
    free(adam);free(ewa);free(kain);free(abel);free(set);free(enosh);  
}
```

Pamięć dla struktur

W tym przypadku wygodniejsze jest zebranie informacji o wszystkich strukturach w tablicy wskaźników i zwolnienie pamięci poprzez iterację po zawartości tablicy.

```
int main(){
    struct person* tab[100]; // tablica wskaźników
    int cnt=0;
    tab[cnt++]=create_person("Adam",NULL,NULL);
    tab[cnt++]=create_person("Ewa",NULL,NULL);
    tab[cnt++]=create_person("Kain",tab[0],tab[1]);
    tab[cnt++]=create_person("Abel",tab[0],tab[1]);
    tab[cnt++]=create_person("Set",tab[0],tab[1]);
    tab[cnt++]=create_person("Enosh",tab[4],NULL);
    about(tab[5]);
    genalogy(tab[5],0);
    // zwolnienie pamięci dla kolejnych obiektów
    for(int i=0;i<cnt;i++)free(tab[i]);
}
```

Wskaźniki void * (1)

- Deklarując wskaźnik podajemy zawsze jego typ. Umożliwia to poprawną realizację dostępu do wskazywanego obiektu (za pomocą operatora * lub ->).
- Istnieje jednak wiele funkcji, które nie realizują bezpośrednio dostępu do obiektu, ale raczej działają na pamięci zajmowanej przez obiekt.
- Są to funkcje odpowiedzialne za:
 - kopiowanie bloków pamięci
 - przydział pamięci o określonej wielkości
 - zapis i odczyt z dysku.

Wskaźniki void * (2)

Funkcje te posługują się specjalnym typem wskaźników deklarowanych jako `void *`.

- Wskaźniki typu `void *` mogą wskazywać element dowolnego typu.
- Na wskaźnikach `void *` nie można wykonywać operacji dereferencji, ponieważ typ wskazywanego obiektu nie jest znany.
- Aby uzyskać dostęp do rzeczywistego obiektu, którego adres jest wartością wskaźnika, należy dokonać konwersji typów – rzutowania.

Wskaźniki void * (3)

Rzutowanie

- Jeżeli `type-name` jest zadeklarowanym typem, operator `(type-name)` pozwala na zmianę typu dowolnego wyrażenia.
- Wyrażenie `(type-name) expression` ma wartość oryginalnego wyrażenia `expression` natomiast typ zmieniony na `type-name`.

Przykład

```
int x=7;
void *pv=&x;
int*pi=(int*)pv;    // konieczne rzutowanie
*pi=5 ;
* (int*)pv = 5;
printf("x=%d\n", x) ;
* (double*)pv = 5; // zapewne błąd
```

Przykład

Funkcja umożliwiająca wypisanie bitów dowolnego bloku pamięci. Parametr `void*` umożliwia przekazanie jako argumentu wskaźnika dowolnego typu.

```
void bitdump(void*block,size_t size){
    char*p=block;
    printf("|");
    for(char*p=(char*)block;p<(char*)block+size;p++){
        for(int i=7;i>=0;i--){
            if(*p&(1<<i))printf("1");
            else printf("0");
        }
        printf("|");
    }
    printf("\n");
}
```

Przykład

```
struct abc{char a; short b; int c};
```

```
int main(){
    int x = 4;
    bitdump(&x,sizeof(x));
    x=-4;
    bitdump(&x,sizeof(x));
    double z = 1;
    bitdump(&z,sizeof(z));
    struct abc s = {'1',-1,48};
    bitdump(&s,sizeof(s));
    printf("a:%p b:%p c:%p",&s.a,&s.b,&s.c);
}
```

Jaka jest kolejność bajtów --
big czy little endian?

Który bajt w strukturze jest
nieużywany?

```
|00000100|00000000|00000000|00000000| | | | |
|11111100|11111111|11111111|11111111|
|00000000|00000000|00000000|00000000|00000000|00000000|11110000|00111111|
|00110001|00000000|11111111|11111111|00110000|00000000|00000000|00000000|
a:0xffffcc08 b:0xffffcc0a c:0xffffcc0c
```


Arytmetyka wskaźników



ONE WORLD : ONE LANGUAGE. C UNITES WORKERS

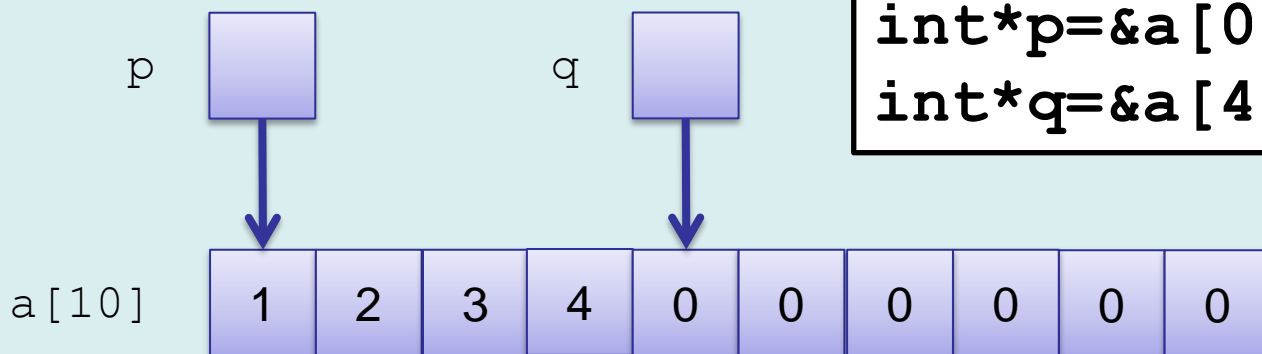
```
#include<stdio.h>
CHAR A[] = "\\";\nmain() {CHAR *B=A;printf(\"#include<stdio.h>\\nCHAR A[] = \\\"\\\"\\n\");
\"FOR(;B;B++) {SWITCH(*B){CASE '\\n': printf(\"\\\\\\\\\\\\\\\\n\\\\n\"); break;\\nCASE '\\\\\\\\\\\\\\': CASE '\\\\\\\\\\\\\\':\"
\"PUTCHAR(\"\\\\\\\\\\\\\\\\\"); default: putchar(*B);}} printf(A);\\n\"; main() {CHAR *B=A;
printf(\"#include<stdio.h>\\nCHAR A[] = \\\"\\\"\\n\"); for(;*B;B++) {switch(*B){case '\\n':
printf(\"\\\\\\\\n\"); break; case '\\\\\\\\\\\\\\': case '\\\\\\\\\\\\\\': putchar(\"\\\\\\\\\"); default: putchar(*B);}} printf(A);
```

STRENGTH THROUGH POINTER ARITHMETIC

Arytmetyka wskaźników (1)

- Wskaźnikowi można przypisać adres elementu tablicy i za jego pośrednictwem zmodyfikować lub odczytać zawartość elementu.
- W języku C predefinedowano 3 (4) operacje arytmetyczne na wskaźnikach wiążące się ściśle z tablicami:
 - Dodanie do wskaźnika liczby całkowitej
 - Odjęcie od wskaźnika liczby całkowitej
 - Odjęcie wskaźników
 - Porównanie wskaźników

```
int a[10]={1,2,3,4};  
int*p=&a[0];  
int*q=&a[4];
```



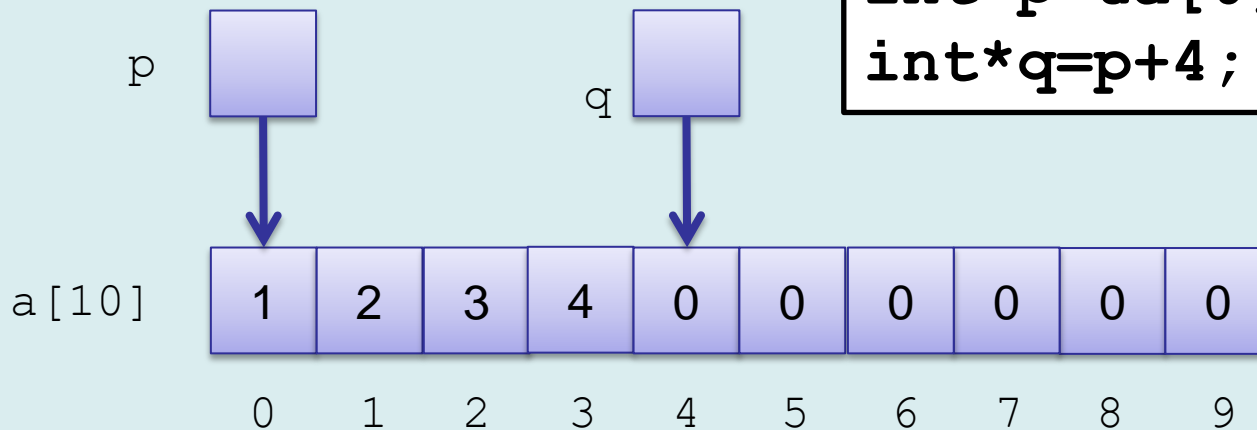
Arytmetyka wskaźników (2)

Dodawanie (odejmowanie) liczby całkowitej do (od) wskaźnika

Jeżeli wskaźnik wskazuje i -ty element tablicy

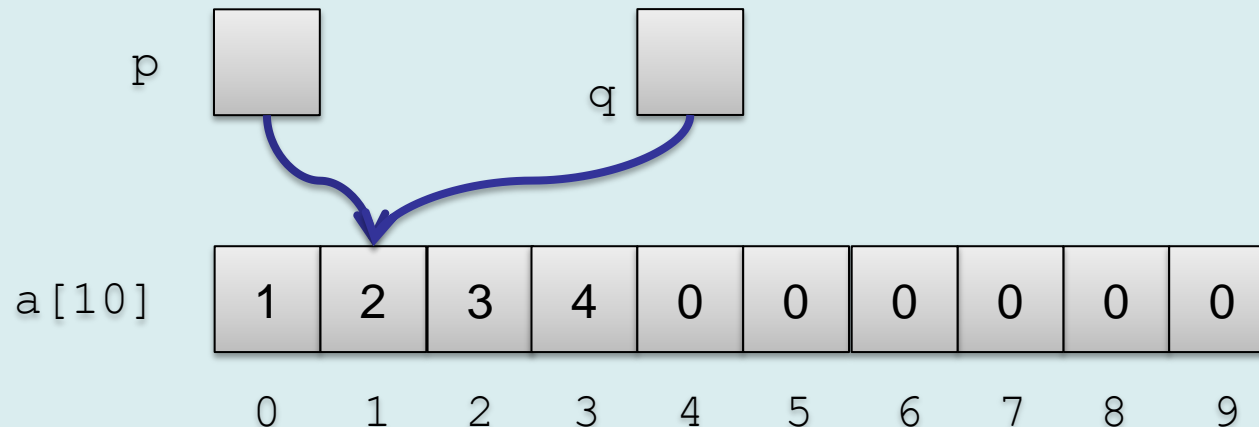
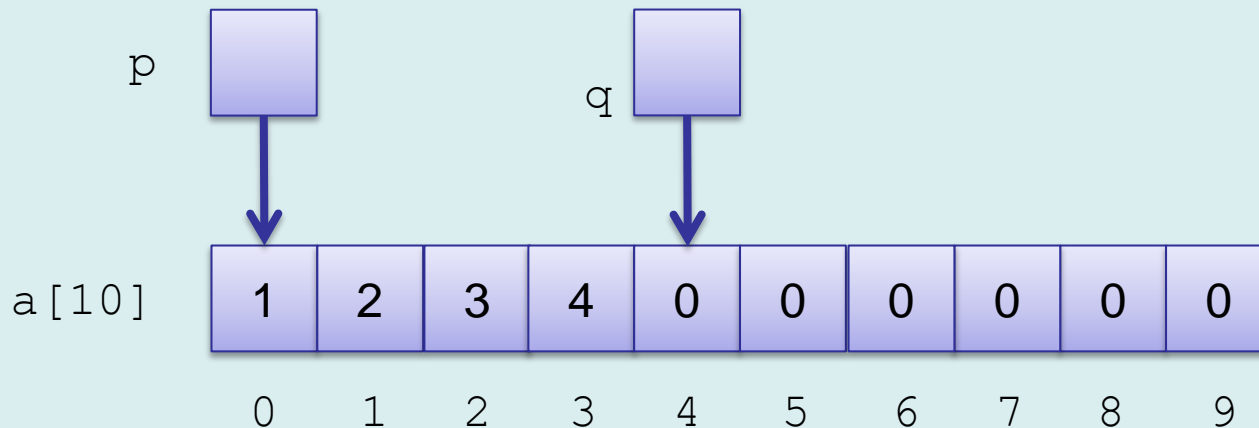
$p = \&a[i]$, wyrażenie $p + j$, wskazuje element tablicy $i + j$, czyli prawdziwe jest $p+j==\&a[i+j]$;

```
int a[10]={1,2,3,4};  
int*p=&a[0];  
int*q=p+4;
```



Arytmetyka wskaźników (3)

- W wyniku wykonania instrukcji $p=p+1$ wskaźnik p przesuwa się na następny element.
- W wyniku wykonania $q=q-3$ wskaźnik q cofa się o 3 elementy w tył.

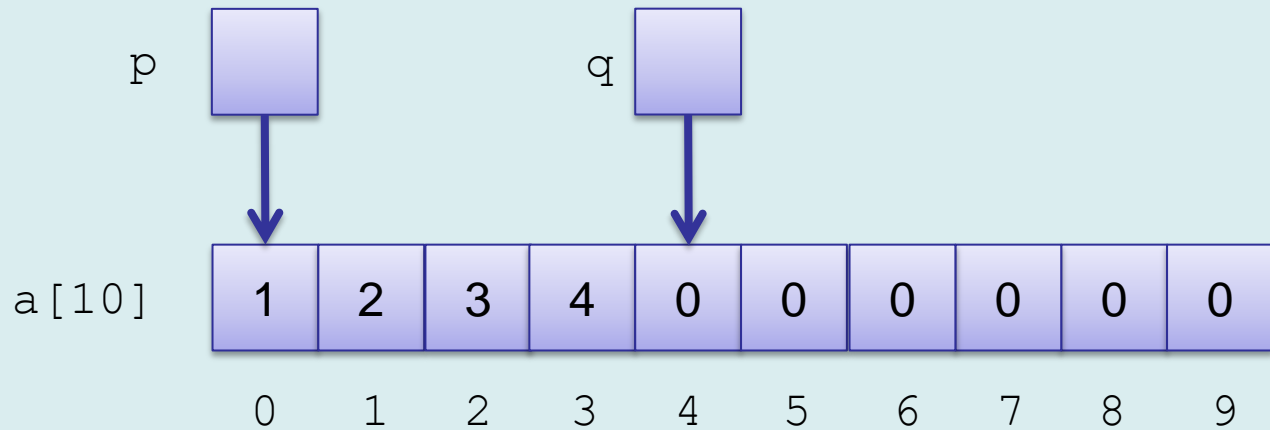


$p=p+1;$
 $q=q-3$

Arytmetyka wskaźników (4)

Odejmowanie wskaźników

- Zakładając, że wskaźniki p i q zawierają adresy elementów tablicy $a[i]$ oraz $a[j]$, ich różnica jest równa $i - j$, czyli liczbie elementów, które „zmieszczą się” w bloku pamięci pomiędzy wskaźnikami.



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

```
q=&a[4];
```

```
printf("%d", q-a); // 4
```

```
printf("%d", a-q); // -4
```

Arytmetyka wskaźników (5)

Podsumowanie arytmetyki wskaźników

- Wskaźniki `TYPE*p`, `*q`;
- $V(p)$ – wartość zmiennej (adres)
- a – wyrażenie całkowitoliczbowe (na przykład stała)

Wyrażenie	Wartość wyrażenia
$p + a$	$V(p) + \text{sizeof}(\text{TYPE}) * a$
$p - a$	$V(p) - \text{sizeof}(\text{TYPE}) * a$
$p++$	$V(p) + \text{sizeof}(\text{TYPE})$ p ma wartość $V(a1) + \text{sizeof}(\text{TYPE})$
$p--$	$V(p) - \text{sizeof}(\text{TYPE})$ p ma wartość $V(a) - \text{sizeof}(\text{TYPE})$
$p+=a$	$V(p) + \text{sizeof}(\text{TYPE}) * a$ p ma wartość $V(p) + \text{sizeof}(\text{TYPE}) * a$
$p-=a$	$V(p) - \text{sizeof}(\text{TYPE}) * a$ p ma wartość $V(p) - \text{sizeof}(\text{TYPE}) * a$
$p-q$	$(V(p) - V(q)) / \text{sizeof}(\text{TYPE})$

Arytmetyka wskaźników (6)

- Wartości wskaźników są adresami. Podobnie, jak inne liczby całkowite można je porównywać za pomocą operatorów relacyjnych:
`== != < <= > >=`
- Porównywanie wskaźników różnych typów jest podejrzane (podczas kompilacji pojawia się ostrzeżenie).
- Nie można odejmować wskaźników różnych typów
- W zasadzie – wszystkie operacje na wskaźnikach powinny dotyczyć zmiennych wskazujących elementy **jednej** tablicy. W przeciwnym przypadku zachowanie jest nieokreślone.
- Dla wskaźników `void*` niemających informacji o typie – operatory arytmetyczne przesuwają wskaźniki o wielokrotności bajtów.

Wskaźniki i tablice (1)

Kompilator języka C/C++ traktuje identyfikator tablicy tak samo jak **niemodyfikowalny** wskaźnik do jej pierwszego elementu (elementu o indeksie zerowym).

```
int main()
{
    int a[10]={1,2,3,4};
    printf("a=%p\n", a);
    printf("&a[0]=%p\n", &a[0]);
    printf("*a=%d a[0]=%d\n", *a, a[0]);
    return 0;
}
```

```
a=0028FF10
&a[0]=0028FF10
*a=1 a[0]=1
```


Wskaźniki i tablice (2)

Identyfikatory tablic i wskaźników mogą być używane wymiennie (wyjątek: symbol zadeklarowanej tablicy nie jest lvalue).

```
int max1(int* p, int size)
{
    int max=p[0];
    int i;
    for(i=1;i<size;i++)
        if(max<p[i]) max=p[i];
    return max;
}
```

```
int max3(int t[], int size)
{
    int max=t[0];
    int* p=t;
    int i;
    for(i=1;i<size;i++)
        if(max<p[i]) max=p[i];
    return max;
}
```

Wskaźniki i tablice (9)

```
int sum2(int t[],int size)
{
    int sum=0,i;
    for(i=0;i<size;i++) sum+=* (t++);
    return sum;
}
```

Poprawne. `t` jest kopią adresu tablicy przekazaną do funkcji poprzez stos

```
int main()
{
    int tab[]={4,3,7,5};
    int sum=0,i;
    for(i=0;i<4;i++) sum+=* (tab++);
    printf("%d\n",sum);
    return 0;
}
```

Niepoprawne. `tab` jest adresem tablicy. Gdyby adres został zmodyfikowany – zostałaby utracona informacja o jej położeniu.

main.c:56: error: wrong type argument to increment

Przykład: tablice, struktury, wskaźniki (1)

```
#define TSIZE 4
struct complex {double re,im;} table[TSIZE];
int i;
struct complex *pc;

// wypełnienie tablicy danymi
for(i=0;i<TSIZE;i++) {
    table[i].re=i;
    table[i].im=i;
}

// dostęp za pośrednictwem symbolu tablicy
for(i=0;i<TSIZE;i++) {
    printf (    "table[i].re=%f table[i].im=%f\n",
               table[i].re, table[i].im);
}
```

Przykład: tablice, struktury, wskaźniki (2)

```
// dostęp za pośrednictwem wskaźnika
for (i=0, pc=table; i<TSIZE; i++) {
    printf ("(pc+i)->re=%f (pc+i)->im=%f\n",
           (pc+i)->re, (pc+i)->im);
}

// wskaźnik może być tak samo traktowany jak
// identyfikator tablicy!
for (i=0, pc=table; i<TSIZE; i++) {
    printf ("pc[i].re=%f pc[i].im=%f\n",
           pc[i].re, pc[i].im);
}
```

Przykład: tablice, struktury, wskaźniki (3)

```
// porównania wartości wskaźników
for (pc=table; pc<table+TSIZE; pc++) {
    printf ("pc->re=%f pc->im=%f\n", pc->re, pc->im);
}

// jawne porównania adresów
for (pc=table;
    (int)pc<(int)table+TSIZE*sizeof(struct complex);
    pc++) {
    printf ("pc->re=%f pc->im=%f\n", pc->re, pc->im);
}
```

Dzięki arytmetyce wskaźników nie trzeba konstruować tak złożonych wyrażeń – generuje je kompilator

Wskaźniki i tablice (9)

Podsumowując:

- Identyfikator tablicy określonego typu i wskaźnik do tego typu w mogą być traktowane wymiennie w wyrażeniach realizujących dostęp do elementów tablicy
- Wyjątkiem jest operator przypisania: identyfikatorowi tablicy nie wolno przypisywać nowej wartości

```
int table1 [10];  
int table2 [10];  
int*p= table1;  // poprawne  
table1=table2 ; // niepoprawne
```

- Argument funkcji `TYPE []` oraz `TYPE*` oznaczają to samo – wskaźnik do tablicy elementów `TYPE`

```
void foo(int a[],int size);  
void foo(int*a,int size);
```

Wskaźniki i tablice wielowymiarowe (1)

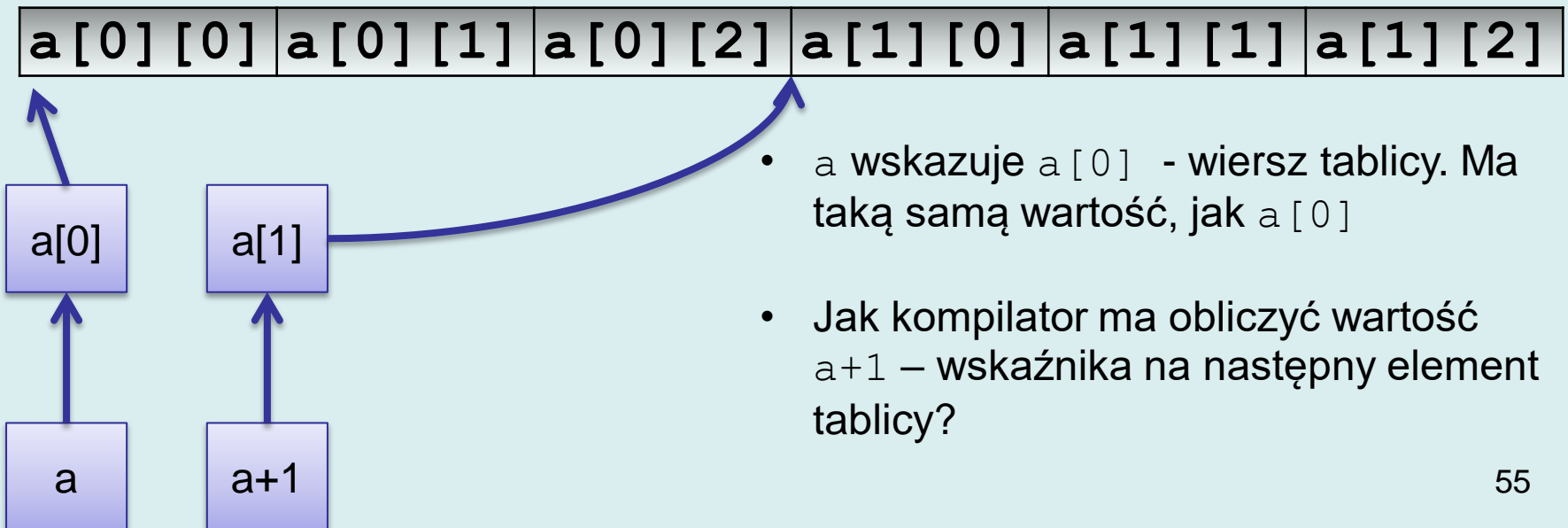
Deklaracja:

```
int a[2][3]
```

- Logiczne rozmieszczenie elementów tablicy:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

- Fizyczne rozmieszczenie elementów tablicy:



Wskaźniki i tablice wielowymiarowe (2)

Identyfikator `a` jest typu `int (*) [3]` – wskaźnik do 3-elementowej tablicy liczb całkowitych.

```
int main()
{
#define ROWS 2
#define COLS 3
    int a[ROWS][COLS];
    int i,j;
    int (*p)[COLS]=a;
    for(i=0;i< ROWS;i++)
        for(j=0;j<COLS;j++) p[i][j]=COLS*i+j;

    for(i=0;i<ROWS;i++)
        for(j=0;j<COLS;j++)
            printf("a[%d][%d]=%d address = %d\n",
                i,j, p[i][j], &p[i][j]);
    printf("a address = %d\n",a);
    printf("first row address = %d\n",a[0]);
    printf("second row address = %d\n",a[1]);
}
```

```
a[0][0]=0 address = 2665536
a[0][1]=1 address = 2665540
a[0][2]=2 address = 2665544
a[1][0]=3 address = 2665548
a[1][1]=4 address = 2665552
a[1][2]=5 address = 2665556
a address = 2665536
first row address = 2665536
second row address = 2665548
```


Wskaźniki i tablice wielowymiarowe (3)

Analogiczne zasady dotyczą wskaźników do wielowymiarowych tablic o zmiennej wielkości (VLA).

```
void printTab(int n, int m, int tab[n][m])
{
    int (*p)[m] = tab;
    int i,j;
    for(i=0;i<n;i++){
        for(j=0;j<m;j++){
            printf("%d, ",p[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    int x[2][3]={0,1,2},{3,4,5};
    printTab(2,3,x);
    return 0;
}
```

```
0, 1, 2,
3, 4, 5,
```

Tablica wskaźników

Wyobraźmy sobie, że mamy dłuższy tekst i chcemy podzielić go na wiersze. Wskaźniki do wierszy mają być umieszczone w tablicy.

```
char text[]="Beware the Jabberwock, my son!\n"
           "The jaws that bite, the claws that catch!\n"
           "Beware the Jubjub bird, and shun\n"
           "The frumious Bandersnatch!";
```

```
int main(){
    char*lines[100];
    int cnt=0;
    lines[cnt++]=text;
    char*ptr=text;
    while(*ptr){
        if(*ptr=='\n'){
            *ptr=0; //1
            lines[cnt++]=ptr+1; //2
        }
        ptr++;
    }
    lines[cnt]=0; //3
    //...
```

//1 Do oryginalnego tekstu w tablicy wstawiamy znaki 0 w miejsce znaków nowej linii.

//2 Ustawiamy wskaźnik w tablicy lines na następny znak

//3 Nie musimy, ale dodajemy na końcu tablicy wartownika (sentinel) – zerowy wskaźnik

Drukujemy

```
// kontynuacja main...
print_lines(lines,cnt);
printf("~~~\n");
print_lines_upto_sentinel(lines);
}

void print_lines(char*lines[],int cnt){
    for(int i=0;i<cnt;i++){
        printf("%s\n",lines[i]);
    }
}

void print_lines_upto_sentinel(char*lines[]){
    char**ptr_to_lines=lines;
    while(ptr_to_lines){
        printf("%s\n",*ptr_to_lines);
        ptr_to_lines++;
    }
}
```

Wynik:

```
Beware the Jabberwock, my son!
The jaws that bite, the claws
that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!
~~~
Beware the Jabberwock, my son!
The jaws that bite, the claws
that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!
```

Deklaracja typedef (1)

W języku C/C++ możliwe jest zdefiniowanie własnej nazwy dla typu wbudowanego lub typu własnego.

Składnia:

```
typedef type-specification declarator
```

- Składnia jest podobna do deklaracji zmiennych lub funkcji. Poprzedzenie słowem kluczowym `typedef` powoduje, że identyfikator zamiast zmiennej lub funkcji staje się synonimem nazwy typu.
- Nazwę typu wprowadzoną za pomocą deklaracji `typedef` można używać zamiennie z nazwą podstawową.

Deklaracja typedef (2)

Przykłady

```
typedef struct tagComplex
{
    double re,im;
}Complex;

typedef Complex* PComplex;
typedef int INT ,*PINT;

typedef unsigned long UINT;
typedef char HANDLE[8];
```

Deklaracja typedef (3)

- Zastosowania
- W przypadku struktur pozwala na pominięcie słowa kluczowego `struct`.
- Pozwala na ukrycie implementacji typów danych (np.: `INT` może być implementowane jako `short` lub `long`).
- Pozwala na zmniejszenie złożoności deklaracji (deklarację kilkuetapową.)

```
char handleTable[100][8];  
HANDLE handleTable[100];
```

```
struct tagComplex*pa,*pb,*pc  
PComplex pa,pb,pc;
```

Wskaźniki do funkcji (1)

- Po skompilowaniu każdej funkcji przydziela się pewien obszar w pamięci. Podczas wywołania funkcji – po przeprowadzeniu niezbędnych inicjalizacji – program dokonuje skoku do instrukcji mieszczącej się pod adresem początkowym bloku kodu.
- Adres tego obszaru może zostać pobrany i wykorzystany do wywołania funkcji.
- Wskaźnik do funkcji jest zmienną, która zawiera adres funkcji. Posługując się wskaźnikiem można tę funkcję wywołać.
- Typową praktyką przy projektowaniu bibliotek w C/C++ jest możliwość przekazania wskaźnika do funkcji, która, na przykład, będzie odpowiedzialna za: wyświetlanie pewnych informacji, porównywanie elementów, zapis i odczyt danych.

Wskaźniki do funkcji (2)

- Kompilator języków C/C++ zwraca uwagę na zgodność typów. W przypadku wskaźników do funkcji typ określony jest przez typ **zwracanej wartości** i typy **argumentów**.
- Deklaracje wskaźników do funkcji jest kłopotliwa. Najlepiej posłużyć się prostym przepisem:

jeżeli funkcja jest zadeklarowana jako

```
return-type function(arg-list)
```

wówczas

```
return-type (*function-pointer) (arg-list)
```

deklaruje wskaźnik o nazwie **function-pointer** do funkcji zwracającej `return-type` i biorącej za argumenty `arg-list`.

- W przypadku bardziej złożonych definicji najlepiej przeprowadzić deklarację dwuetapową wykorzystując `typedef`:

```
typedef return-type (*fp-type) (arg-list);
```

```
fp-type function-pointer;
```


Wskaźniki do funkcji (3)

Przykład

```
void foo(int a)
{
    printf("%d", a);
}

typedef void (*VOID_INT_FP) (int);

int main()
{
    VOID_INT_FP myptr = foo;
    if(myptr) myptr (7);
    return 0;
}
```

Analogicznie, jak dla tablic, identyfikator funkcji jest niemodyfikowalnym wskaźnikiem do funkcji!

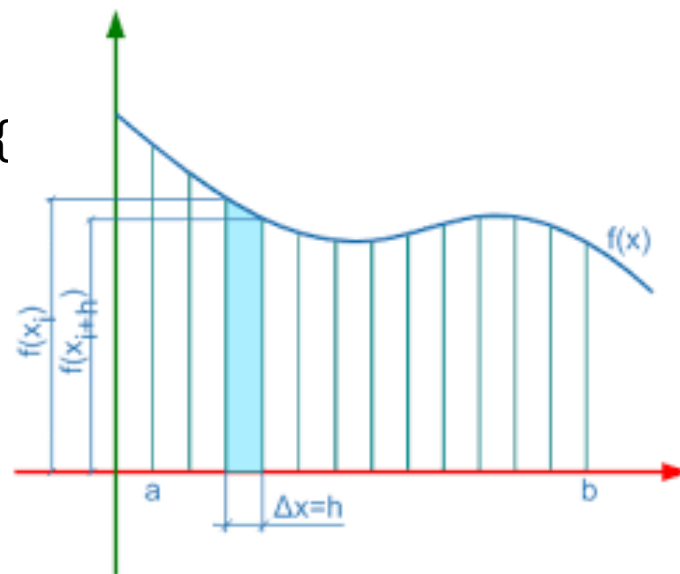
Z użyciem wskaźników do funkcji wiążą się analogiczne problemy, jak ze wskaźnikami do danych:

- Mogą mieć wartość nieokreśloną (wywołanie spowoduje zapewne błąd wykonania)
- Można testować, czy nie mają wartości zerowej i wywoływać funkcję opcjonalnie

Przykład

Funkcja ogólnego zastosowania implementująca algorytm całkowania numerycznego metodą trapezów.

```
double integrate(double (*f)(double), double a, double b,  
                int steps) {  
    double delta = (b - a) / steps;  
    double sum = 0;  
    double fs = f(a);  
    for (int i = 1; i <= steps; i++) {  
        double fe = f(a + i * delta);  
        sum += (fs + fe) / 2 * delta;  
        fs = fe;  
    }  
    return sum;  
}
```



[<https://www.obliczeniowo.com.pl/704>]

f – wskaźnik do funkcji podcałkowej
(a, b) – dolny i górny zakres całkowania
steps – liczba kroków algorytmu

Funkcje podcałkowe

```
double polynomial_value(double*a,int n,double x){
    double r=0;
    double pow=1;
    for(int i=n-1;i>=0;i--){
        r+=a[i]*pow;
        pow*=x;
    }
    return r;
}

double f1(double x){
    double a[]={1,2,3,4,5};
    return polynomial_value(a,5,x);
}

double f2(double x){
    double a[]={1,-2,3,4};
    return polynomial_value(a,4,x);
}
```

Funkcja `polynomial_value` oblicza wartość wielomianu dla współczynników przekazanych jako tablica. Funkcje `f1()` i `f2()` przechowują lokalnie tablice współczynników.

$$f_1(x) = x^4 + 2x^3 + 3x^2 + 4x + 5$$
$$f_2(x) = x^3 - 2x^2 + 3x + 4$$

Wynik

```
int main(){
    int steps = 100;
    printf("steps=%d\n", steps);
    printf("%f\n", integrate(f1, 0, 10, steps));
    printf("%f\n", integrate(f2, 0, 10, steps));
    printf("%f\n", integrate(sin, 0, M_PI, steps));
}
```

Dokładność zależy od liczby kroków steps:

```
steps=100
26253.883300
2023.550000
1.999836

steps=1000
26250.038833
2023.335500
1.999998

steps=10000
26250.000388
2023.333355
2.000000
```

Całkujemy także $\sin()$ w zakresie 0 do π



SymPy

[<https://live.sympy.org/>]

```
>>> integrate(1*x**4+2*x**3+3*x**2+4*x+5, (x, 0, 10))
```

26250

```
>>> integrate(1.0*x**3-2.0*x**2+3.0*x+4.0, (x, 0, 10))
```

2023.333333333333

```
>>> integrate(sin(x), (x, 0, pi))
```

Sortowanie i wyszukiwanie (1)

Typowe zastosowania

Biblioteczne implementacje funkcji do sortowania i wyszukiwania elementów tablicy: `qsort()` i `bsearch()`.

`qsort()` implementuje algorytm *quick sort*

`bsearch()` implementuje algorytm binarnego przeszukiwania posortowanej tablicy

Sortowanie i wyszukiwanie (2)

Funkcja qsort

Sortuje tablicę elementów przekazaną jako parametr za pomocą algorytmu *quick sort*. Użytkownik musi zaimplementować własną funkcję do porównywania elementów.

Deklaracja:

```
void qsort( void *base, size_t num, size_t width,  
int ( __cdecl *compare ) (const void *elem1, const void  
*elem2 ) );
```

- **base** – adres początku tablicy elementów
- **num** – liczba elementów tablicy
- **width** – rozmiar elementu w bajtach
- **compare** – wskaźnik do funkcji do porównywania elementów

Sortowanie i wyszukiwanie (3)

Funkcja do porównywania elementów powinna być zdefiniowana jako:

```
int compare(const void *elem1, const void*elem2)
```

i zwracać wartość:

< 0	jeżeli elem1 jest mniejszy niż elem2 (powinien być umieszczony wcześniej)
0	jeżeli elementy są równe (ich kolejność jest nieistotna)
> 0	jeżeli elem1 jest większy niż elem2 (powinien być umieszczony później)

- Wskaźniki elem1 i elem2 są typu `const void*`. Funkcja do porównywania:
 - powinna rzutować je na wskaźniki odpowiedniego typu, np.: `const int*`
 - zastosować dereferencję
 - porównać elementy
- Modyfikator `const` wskazuje, że podczas porównywania, elementy nie powinny być zmieniane.
- Aby zmienić kolejność sortowania elementów wystarczy odwrócić znak rezultatu zwracanego przez funkcję.

Sortowanie i wyszukiwanie (4)

Przykład 1 – sortowanie tablicy liczb całkowitych

```
#include <stdio.h>
#include <stdlib.h>

int compInt(const void*e1,const void*e2)
{
    return *(const int*)e1 - *(const int*)e2;
}

int main()
{
    int table[]={2,1,5,6,2,8,9,0,3,4};
    int i;
    qsort(table,sizeof(table)/sizeof(table[0]),
           sizeof(int),compInt);
    for(i=0;i<sizeof(table)/sizeof(table[0]);i++)
        printf("%d ",table[i]);
    return 0;
}
```

0 1 2 2 3 4 5 6 8 9

Sortowanie i wyszukiwanie (5)

Przykład 2 – sortowanie tablicy wskaźników do tekstów

```
int compString(const void*e1,const void*e2)
{
    return strcmp (*(const char**)e1,*(const char**)e2);
}

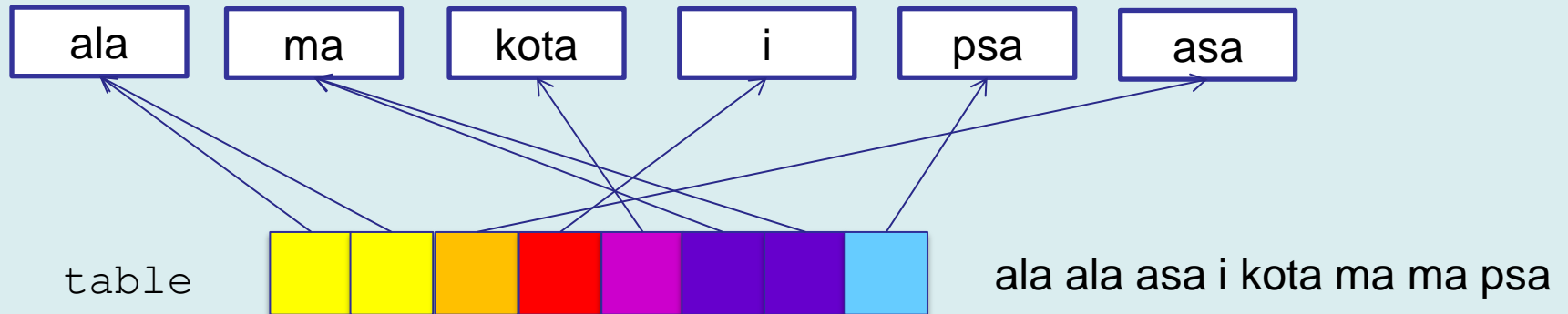
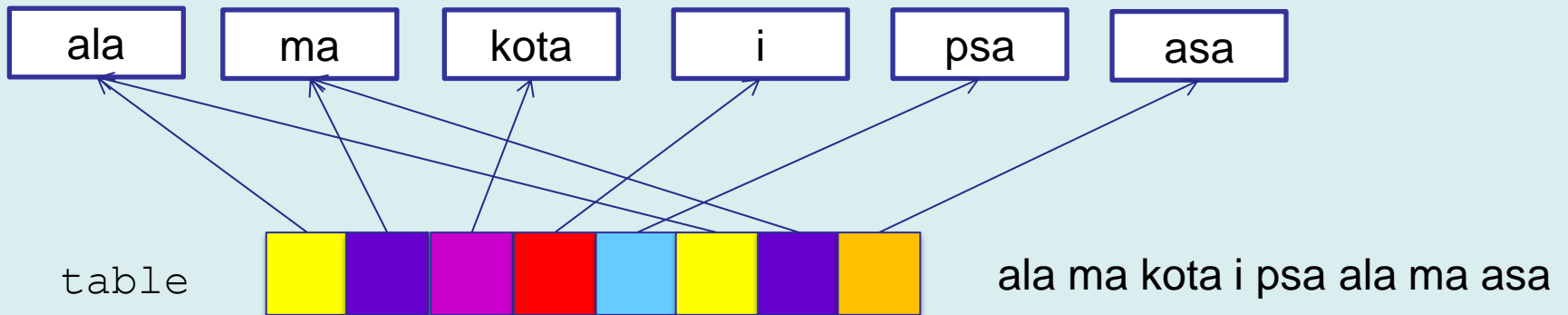
void test2()
{
    char*table[]={"ala","ma","kota","i","psa","ala","ma","asa"};
    int i;
    qsort(table,sizeof(table)/sizeof(table[0]),sizeof(char*),
        compString);
    for(i=0;i<sizeof(table)/sizeof(table[0]);i++)
        printf("%s ",table[i]);
}
```

Funkcja **strcmp** porównuje teksty ignorując wielkość znaków.

ala ala asa i kota ma ma psa

Sortowanie i wyszukiwanie (6)

- Analiza



Sortowanie i wyszukiwanie (7)

Przykład 3 - porównywanie struktur według kilku kluczy

```
struct osoba
{
    char imie[32];
    char nazwisko[32];
    char pesel[12];
    /*inne dane*/
};

int compOsoby(const void*e1,const void*e2)
{
    int result;
    const struct osoba*o1=(const struct osoba*)e1;
    const struct osoba*o2=(const struct osoba*)e2;
    result = strcmp(o1->nazwisko,o2->nazwisko);
    if(result!=0)return result;

    result = strcmp(o1->imie,o2->imie);
    if(result!=0)return result;

    return !strcmp(o1->pesel,o2->pesel);
}
```

Sortowanie i wyszukiwanie (8)

Funkcja bsearch

Funkcja implementuje algorytm binarnego przeszukiwania posortowanej tablicy elementów.

Deklaracja:

```
void *bsearch(const void*key, const void *base,  
size_t num, size_t width,  
int ( __cdecl *compare ) (const void *elem1, const void *elem2 ) );
```

- **key** – wskaźnik do zmiennej zawierającej szukany element
- **base** – adres początku posortowanej tablicy elementów
- **num** – liczba elementów tablicy
- **width** – rozmiar elementu w bajtach
- **compare** – wskaźnik do funkcji do porównywania elementów

Funkcja zwraca wskaźnik do elementu tablicy, którego wartość odpowiada szukanemu elementowi `key`. Tablica powinna zawierać unikalne elementy i być posortowana w porządku rosnącym. Funkcja do porównywania elementów `compare` powinna być zdefiniowana analogicznie, jak dla `qsort`.

Sortowanie i wyszukiwanie (9)

Przykład 4 – wywołanie bsearch

```
void test3()
{
    int table[]={2,1,5,6,8,9,0,3,4};
    int key = 2;
    int*found;
    const int size= sizeof(table)/sizeof(table[0]);

    qsort(table,size,sizeof(char*),compInt);
    found = (int*)bsearch(&key,table,size,sizeof(int),compInt);
    if(found)printf("Found key: %d at %p",*found,found);
}
```

Found key: 2 at 0012FF10

Co należy zapamiętać

- Wskaźniki są zmiennymi, których wartościami są adresy innych obiektów (zmiennych, funkcji)
- Operatory adresu i dereferencji
- Przekazywanie wskaźników do funkcji – modyfikacja zewnętrznych obiektów
- Wskaźniki jako implementacja powiązań pomiędzy obiektami
- Wskaźniki i tablice (arytmetyka wskaźników)
- Deklaracja typedef
- Wskaźniki do funkcji