Programowanie Niskopoziomowe

Wykład 6

- Funkcje

Instytut Informatyki

dr inż. Marcin Radom

Funkcje

- Pozwalają podzielić dowolny skomplikowany problem na mniejsze jednostki, które łatwiej napisać niż całość.
- Zamykają w sobie pewien fragment rozwiązania, który może być później wykorzystywany do innych problemów (np. funkcja obliczająca maksimum z kilku liczb w tablicy).
- Dzięki podzieleniu zadania na mniejsze jednostki, łatwiej stwierdzić w przypadku błędów co dokładnie nie działa.
- Funkcje w C dzielimy na obliczające wartości oraz nie obliczające wartości.
- Niedozwolone jest:
  - Wywoływanie niezadeklarowanych funkcji.
  - Tworzenie funkcji zagnieżdżonych (funkcji wewnątrz innych funkcji).
- **Deklaracja funkcji:**

```
typ  identyfikator_funkcji
     (lista_argumentów_formalnych );
```

  - ***typ***: **liczbowy, wskaźnikowy, referencyjny, strukturowy**
  - ***argument_formalny:*** `typ identyfikator`

Funkcje

- **Sygnatura funkcji** to nazwa funkcji, liczba argumentów formalnych oraz uporządkowany zbiór typów argumentów formalnych funkcji.

- **Przykłady:**

```
// deklaracja i definicja funkcji muszą być zgodne!!!
int f1( int x, int y );

float wartosc( float cena, int liczba );

char* odpowiedz ( char *pytanie );

// błąd, każdy argument funkcji powinien mieć postać: typ
// identyfikator, nie wolno stosować list zmiennych należących do
// tego samego typu
double alfa( double x1, x2 );

max( int k, int l );      // typ domyślny int - przestarzałe

void druk( char *napis );  // bez wyniku, nie zwraca żadnej wartości
```

Funkcje

- **Definicja funkcji:**

```
typ  identyfikator_funkcji (lista_argumentów_formalnych )
{  blok instrukcji }
```

  - ***blok*** **: ciąg deklaracji, definicji i instrukcji wykonywany :**
    - **do końca bloku**
    - **do instrukcji** `return`
  - ***argument_formalny*** **:** `typ identyfikator`

- Argumenty formalne są zmiennymi lokalnymi, stanowiącymi prywatną własność funkcji (podobnie jak zmienne zadeklarowane wewnątrz funkcji).

- Najprostszą i najmniejszą formą (prawidłową!) funkcji byłoby:

```
funkcja () { }  // nic nie przyjmuje, nic nie robi
```

- Jeżeli nie podamy jaki jest typ wartości funkcji (typ zmiennej zwracany przez funkcję), przyjmuje się, że funkcja zwraca wartość typu `int`.

Funkcje – instrukcja *return*

- Dzięki instrukcji ***return*** wywołana funkcja przekazuje do miejsca wywołania wartość pewnego wyrażenia (może to być dowolne wyrażenie):

```
return wyrazenie;
```

- Jeśli jest taka potrzeba to *wyrazenie* zostanie przekształcone do typu wartości zwracanej przez funkcję.

- Instrukcja:

```
return;
```

  powoduje, że do miejsca wywołania nie jest przekazywana żadna wartość (niczego nie zwraca, po prostu przerywa wykonywanie funkcji).

- Możliwa jest więcej niż jedna instrukcja ***return*** wewnątrz funkcji.

- Nie jest błędem jeśli funkcja z jednego miejsca zwraca wartość instrukcją ***return***, a z innego nie (ale może to oznaczać kłopoty!).

- Jeśli funkcji nie udało się zwrócić poprawnej wartości to jej zwracaną wartością są śmieci.

Funkcje

- **Przykłady:**

```
float sqr( float x )
{
   return x * x ;
}

void max( int *m, int a, int b )
{
   *m = a > b ? a : b ;
}
// wykonywanie funkcji kończy się po osiągnięciu
// zamykającego nawiasu klamrowego - ten sam efekt co:   return;

void DrukujWynik  ( float x )
{
     printf ( "\nWynik : %10.2f" , x);
}
```

Funkcje

- **Przykład:**

```
// dzieli tekst w tablicy S, w ten sposób, że znaki znajdujące się w tablicy S
// pod nieparzystymi indeksami wpisuje do tablicy D2 a te znajdujące się pod
// parzystymi indeksami do D1 */
int split( char *S, char *D1, char *D2 )
{
                        // zlicza znaki w tekście S do wystąpienia znaku '\0'
   int licz_S = 0;      // zmienna lokalna
   while ( *S )
   {                    // znaki o nieparzystym indeksie w tablicy znaków S
      if ( licz_S & 0x01 )
         *D2++ = *S++;
      else
         *D1++ = *S++;
      licz_S++;
   }
                        // wstawia znak końca tekstu '\0' w tablicach D1 i D2
   *D1 = *D2 = 0;
   return licz_S;
}
```

Wywołanie funkcji

- **Składnia:**

  identyfikator_funkcji (lista_argumentów_aktualnych)

  - Gdzie **argument aktualny** (wywołania funkcji) może być wyrażeniem.

- **Przykład:**

```
a = 1 + sqr( 1.25 );                    // wywołanie w wyrażeniu
max( &lepsza, cena_1, cena_2 );         // wywołanie jako instrukcja
sqr(3 + b * d );                        // argument aktualny jest wyrażeniem
DrukujWynik ((sqrt(delta) - 4 * a * c ) / ( 2 * a ) ) ;
printf( "\n%d\t%d\n", a + 2, a - 1 );
```

- Argumenty formalne funkcji uzyskują swoje wartości dzięki użyciu w wywołaniu funkcji *argumentów aktualnych (faktycznych)*.

- Mówiąc w skrócie: argument formalny jest zmienną w wywoływanej funkcji, a argument aktualny jest konkretną wartością przypisywaną zmiennej – argumentowi formalnemu.

- Argumenty formalne są kopiami danych z funkcji wywołującej, więc wszelkie modyfikacje, jakich dokonuje na nich funkcja wywoływana, nie pociągają za sobą zmiany danych wyjściowych.

Funkcje – powiązanie argumentów aktualnych i formalnych

- **Przykład:**

```
int Funa ( int a, int b, int c )
{
     if ( a > 3 * b – 1)
          return c + 2;
     else
          return c * c + 1;
}
// .........
int k, r;
// .........
int x = Funa (12, k + 4, r - 3); // wywołanie funkcji
     // a = 12,  b = k + 4,  c = r - 3
```

Funkcje

- Jeśli deklaracja i definicja funkcji nie są zgodne, a sama wywoływana funkcja zdefiniowana jest w innym pliku, kompilator może nie wykryć błędu. Jeśli zdefiniowana jest w tym samym pliku to kompilator zgłosi błąd.

- Oznacza to, że np. jeśli funkcja z innego pliku niż ten, w którym jest wywoływana, zwraca wartość typu **double**, a programista próbuje jej (funkcji) wynik przypisać do zmiennej typu **int**, kompilator nie zaprotestuje. W efekcie program zacznie od tego momentu operować na bezsensownych wynikach – błąd logiczny, który trudno czasem wykryć (tj. program działa, ale generuje bzdury).

- Jeśli w deklaracji funkcji nie podano parametrów np.:

```
double atof();
```

to przyjmuje się, że o parametrach funkcji **atof** nie należy robić żadnych założeń – wyłącza się więc całą kontrolę poprawności.

Funkcje - Przykład

```
/* atof: konwersja napisu do liczby rzeczywistej */
double atof(char s[]){
     double val, power;
     int i, sign;
     for (i = 0; isspace(s[i]); i++)
          ; /* pomiń białe znaki */

     sign = (s[i] == '-') ? -1 : 1;
     if (s[i] == '+' || s[i] == '-')
          i++;
     for (val = 0.0; isdigit(s[i]); i++)
          val = 10.0 * val + (s[i] - '0');
     if (s[i] == '.')
          i++;
     for (power = 1.0; isdigit(s[i]); i++) {
          val = 10.0 * val + (s[i] - '0');
          power *= 10;
     }
     return sign * val / power;
}

// Przykładowo dla łańcucha znaków 12.34 wartość val
//po odczytaniu całego łańcucha będzie wynosić 1234.0,
// natomiast zmiennej power 100.0.
// Wynik równy jest 1234.0 / 100.0 = 12.34 */
```

```
12.34
Czesc liczby przed przecinkiem:
s[0] = 1, val = 1
s[1] = 2, val = 12
Czesc liczby po przecinku:
s[3] = 3, val = 123, power = 10
s[4] = 4, val = 1234, power = 100
Wynik = 12.34
```

Funkcje – Przykład cd.

```
/* atoi: konwersja napisu do liczby całkowitej z użyciem atof */
int atoi(char s[])
{
    double atof(char s[]);// prototyp atof
    return (int) atof(s);  // rzutowanie na int
}
```

- W takim przypadku funkcja atof() najpierw konwertuje łańcuch *s* na liczbę typu **double**, która zostaje automatycznie przekształcona do **int** przed wykonaniem instrukcji **return (**zanim funkcja zewnętrzna *atoi* zwróci wynik).
- Może nastąpić konwersja stratna (utrata dokładności) w wyniku takiego przekształcenia – odcięcie części ułamkowej.
- Użycie operatora rzutowania **(int)** stanowi jednak informację dla kompilatora, że programista wie co robi (kompilator nie wygeneruje ostrzeżeń).

Przekazywanie argumentów do funkcji

- Zgodny typ argumentu formalnego i aktualnego :
  - Typy identyczne
  - Możliwa konwersja

Przekazywanie argumentów do funkcji

- Problemy, przykłady

Przekazywanie argumentów do funkcji

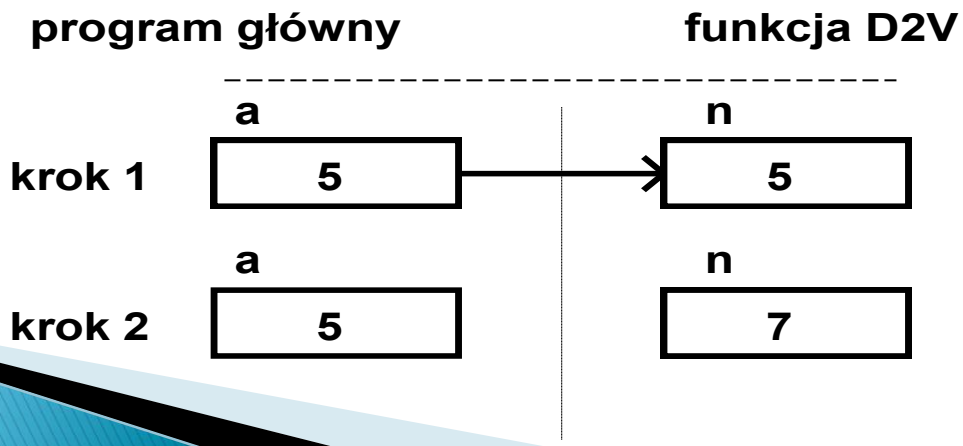
- **Przykład:**

```
long suma( long x )
{
      return x + 1000;
}

suma(3);          // poprawnie int -> long
suma(1543443L);   // poprawnie bez konwersji
suma(17.8);       // ryzykownie double -> long, możliwa utrata dokładności
suma('A');        // poprawnie char -> long
suma("ALA");      // błąd, konwersja niemożliwa

int oblicz( float *wf );

int *wi;
oblicz( wi );     // błąd, nieodpowiedni wskaźnik
oblicz (( float*) wi);  // poprawnie, ryzykownie
```

Funkcje – sposoby przekazywania argumentów

- Sposoby przekazywania argumentów:
  - **przekazywanie wartości**

```
void D2V ( int n )  // przekazywanie wartości
{
   n = n + 2;                  // krok 2
}
int a = 5;
D2V ( a );                     // krok 1
/* wartość służy do inicjalizacji parametru formalnego, czyli zmiennej
lokalnej tworzonej przez funkcję na stosie */
```



Funkcje – sposoby przekazywania argumentów

- W języku C argumenty przekazywane są przez wartość.
- Nie ma bezpośredniego sposobu na to, aby wywołana funkcja mogła zmienić wartość zmiennej należącej do funkcji wywołującej.
- Np. gdybyśmy mieli funkcję zamieniająca 'miejscami' dwa elementy za pomocą funkcji *swap* i wywołali ją:

```
swap(a, b);
```

Jeśli funkcja *swap* jest zdefiniowana jak poniżej:

```
void swap(int x, int y) { // ŹLE
     int tmp = x;
     x = y;
     y = tmp;
}
```

- Funkcja *swap* nie ma dostępu do argumentów *a* i *b* (ponieważ przesyłamy je przez wartość) i zamienia jedynie miejscami wartości kopii tych argumentów.

Funkcje – sposoby przekazywania argumentów

- Innymi słowy: NIC, co taka funkcja zrobi, nie będzie widoczne na zewnątrz tej funkcji, po zakończeniu jej działania.

- Jedyny sposób na osiągnięcie celu polega na przekazaniu do funkcji wskaźników do obiektów, których wartości należy zamienić miejscami:

```
swap(&a, &b);
```

- Operator **&** daje adres zmiennej, więc **&a** jest wskaźnikiem do *a*.

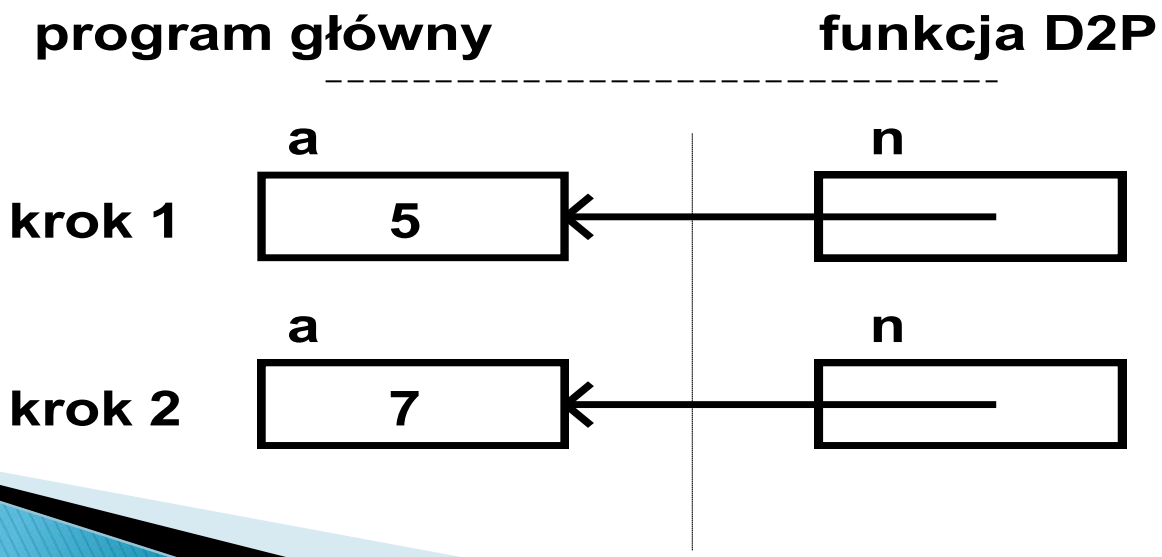
- Poprawna definicja funkcji:

```
void swap(int *x, int *y) // zamień miejscami *x i *y
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Funkcje – sposoby przekazywania argumentów

- **przekazywanie wskaźnika**

```
void D2P( int *n )        // przekazywanie wskaźnika
{
   *n = *n + 2;                  // krok 2
}
int a = 5;
D2P( &a );                       // krok 1
```



Wskaźnik do stałej a stała wskaźnikowa - Przykłady

- **Wskaźnik do stałej** to taki wskaźnik, który pokazywany obiekt uznaje za stały. Nie może go więc modyfikować

```
const long *ws_st;              // wskaźnik do stałej
const long dystans = 5786;

ws_st = & dystans;              // ustaw wskaźnik na zmienną dystans

long odleglosc = *ws_st;        // poprawnie, odległość == dystans

// błąd, nie można modyfikować obiektu pokazywanego przez ws_st:
*ws_st = 1298;

long war = 10, ukrop;
ws_st = & war;
war = 150;                      // poprawnie
ukrop = *ws_st;                 // poprawnie, ukrop = 150

// błąd, za pomocą tego wskaźnika nie może modyfikować wskazywanego obiektu:
*ws_st = 150;
```

Wskaźnik do stałej a stała wskaźnikowa - Przykłady

- **Stała wskaźnikowa**
  - To taki wskaźnik, który zawsze pokazuje na to samo. Nie można nim poruszyć.

```
float cena = 12.5, netto;
float *const swx = & cena;
   // stała wskaźnikowa, wartość początkowa konieczna
cena = 15.8;                  // poprawnie
*swx = 15.8;                  // poprawnie

// błąd, nie można pokazywać tym wskaźnikiem na obiekt inny niż cena:
swx = &netto;
```

Funkcje – sposoby przekazywania argumentów

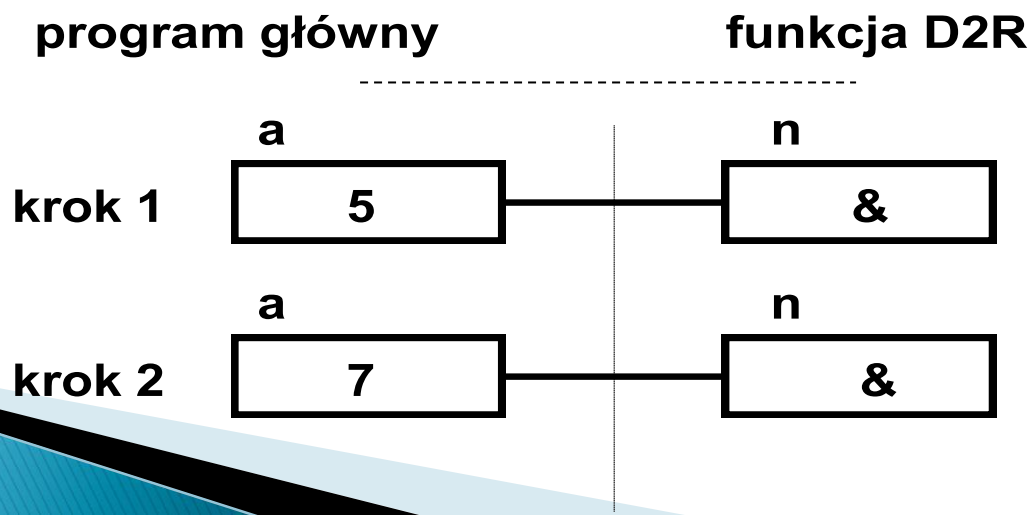
- **przekazywanie wskaźnika cd.**
  - Dla argumentu stałego:

```
// lpi to wskaźnik do stałej
float PoleKola(const float *lpi, float *pr)
{
// błąd, stała nie może stać po lewej stronie przypisania
      *lpi = 2.5;
      /* ... */
}
```

Funkcje – sposoby przekazywania argumentów

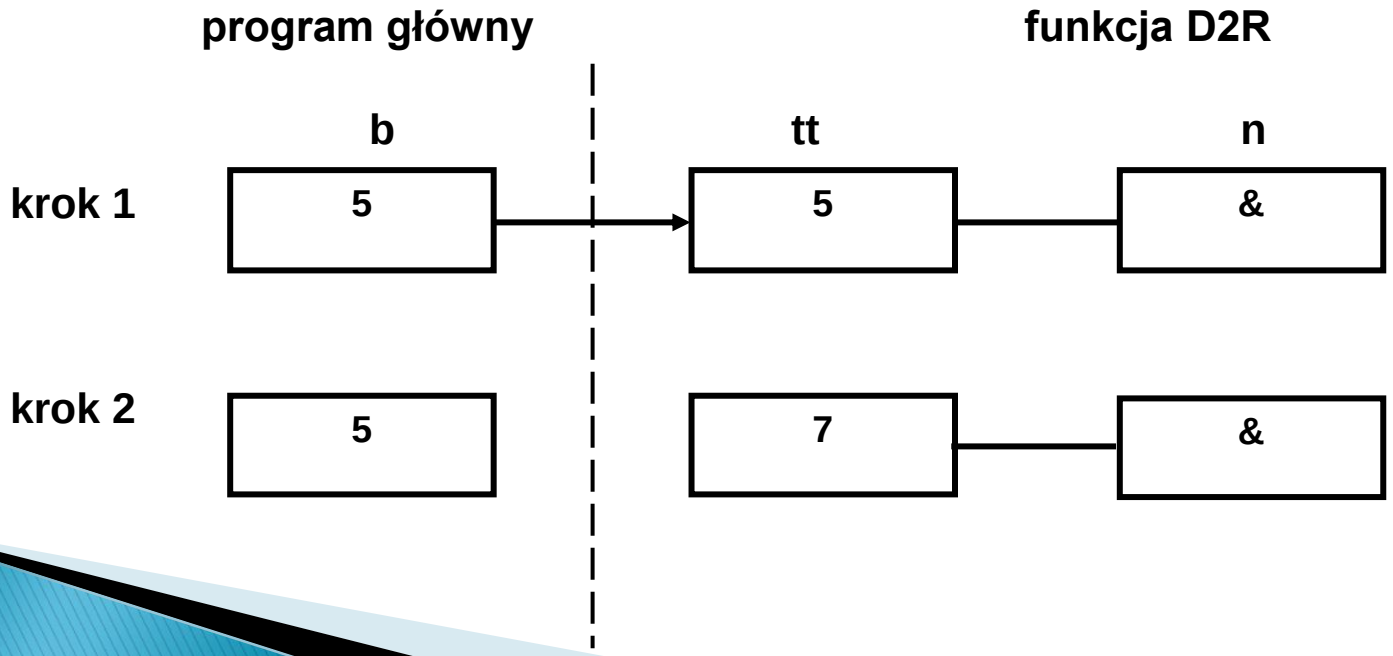
- **przekazywanie referencji**

```
void D2R(int& n)  // przekazywanie referencji
{
   n = n + 2;                    // krok 2
}
int a = 5;
D2R( a );                        // krok 1
/* przesyłany jest adres zmiennej a, ale w sposób niezauważalny z poziomu
wywołania funkcji – może być źródłem trudnych do wykrycia błędów. Przydane
w C++ np. w przypadku dużych obiektów, których przekazanie przez wartość
powodowałoby spowolnienie wywoływania funkcji. W innych przypadkach należy
unikać. */
```



Funkcje – sposoby przekazywania argumentów

```
long b = 5;
D2R( b );                       // krok 1
/* błąd kompilacji – nie można przekonwertować parametru z long do int&.
error: invalid initialization of reference of type 'int&' from expression of
type 'long int'
Gdyby parametr przekazywany był przez wartość konwersja nastąpiłaby
automatycznie. */
```



Przekazywanie struktur do funkcji

- **Strukturę można przekazać do funkcji w następujący sposób:**
  - **Przekazanie składowych osobno**
    - Np. funkcja, która z podanych dwóch wartości tworzy strukturę typu *point*:

```
struct point makepoint(int x, int y)
{
      struct point tmp;

      tmp.x = x;
      tmp.y = y;
      return tmp;
}
```

    - Jak widać nie ma konfliktu pomiędzy nazwami zmiennych lokalnych funkcji *x*, *y* a między nazwami składowych *x*, *y* struktury typu *point*.
    - Funkcję *makepoint* można stosować do inicjowania dowolnej struktury.

Przekazywanie struktur do funkcji

- Przykład wykorzystania w programie:

```
struct rect screen;                     // rezerwuj miejsce na strukturę typu rect
struct point middle;                    // rezerwuj miejsce na strukturę typu point
struct point makepoint(int, int);       // prototyp funkcji
screen.pt1 = makepoint(0,0);            // inicjowanie struktury poprzez przypisanie
screen.pt2 = makepoint(XMAX, YMAX);     // j.w.

middle = makepoint((screen.pt1.x + screen.pt2.x)/2, (screen.pt1.y + screen.pt2.y)/2);
// współrzędne punktu środkowego wewnątrz prostokąta
// opisanego współrzędnymi w zmiennej screen
```

- **Przekazanie całej struktury**
  - Struktury są tak samo przekazywane do funkcji przez wartość, jak wszystkie inne argumenty. Np. dodawanie dwóch punktów:

```
struct point addpoint(struct point p1, struct point p2){
      p1.x += p2.x;             // operacja na kopii p1 wewnątrz funkcji!
      p1.y += p2.y;             // jak wyżej!
      return p1;                // zwraca WYNIK
}       // oba argumenty i zwracana wartość są strukturami
```

Przekazywanie struktur do funkcji

- W poniższym przykładzie funkcja *Wplata* nie ma dostępu do argumentu *AB* (ponieważ przesyłany jest przez wartość) i modyfikuje jedynie kopię tego argumentu:

```
struct Konto{
      long numer;
      long suma;
};
void Wplata (struct Konto kk, long kwota){
      kk.suma += kwota;
}
struct Konto AB = { 84404, 0 };
Wplata ( AB, 3285 );   //  AB.suma  == 0, bez zmiany
```

- Funkcja jako wynik swojego działania zwraca strukturę ze zmodyfikowaną wartością składowej

```
struct Konto Wplata (struct Konto kk, long kwota){
      kk.suma += kwota;
      return kk;
}
AB = Wplata ( AB, 3285 );  //  AB.suma == 3285
```

Przekazywanie struktur do funkcji

- **Przekazanie wskaźnika do struktury**
  - Jeśli do funkcji trzeba przekazać dużą strukturę to przekazywanie wskaźnika jest zwykle bardziej skuteczne niż kopiowanie zawartości całej struktury.
  - Wskaźniki do struktur są takimi samymi wskaźnikami jak do zwykłych zmiennych.
  - **Przykład:**

```
struct informacja {
      int x, y, z;
};

void odejmij (struct informacja *info, int px, int py, int pz)
{
     info->x -= px;
     info->y -= py;
     info->z -= pz;
}
```

Funkcje – Przykład 1

```
/* Opracować program, który oblicza podatek należny od darowizny zgodnie z następującymi
zasadami:
I   stopień pokrewieństwa : do 6000 zł -> podatek 0 zł
                            ponad 6000 -> podatek 10% kwoty
II stopień pokrewieństwa : do 5000 zł -> podatek 25 zł
                         ponad 5000 zł -> podatek 25 zł + 15% z nadwyżki
III stopień pokrewieństwa : podatek wynosi 20% kwoty
*/

          int St1 (int Kwota){ // I stopień pokrewieństwa
               if (Kwota <= 6000)
                    return 0;
               else
                    return int(Kwota * 0.1); // 10% kwoty
          }
          int St2 (int Kwota){ // II stopień pokrewieństwa
               if (Kwota <= 5000)
                    return 25;
              else
                    return int(25 + (Kwota - 5000) * 0.15);
          }
          int St3(int Kwota){ // III stopień pokrewieństwa
              return int(Kwota * 0.2); // 20% kwoty
          }
```

Funkcje – Przykład 1 cd.

```
int main() {
    int Darowizna, Podatek, Stopien = -1;

    printf("Podaj kwote darowizny : ");
    scanf("%d", &Darowizna);

    while (Stopien < 1 || Stopien > 3){
        printf("\nPodaj stopien pokrewienstwa [1, 2, 3] : ");
        scanf("%d", &Stopien);
    }

    switch (Stopien){
        case 1 : Podatek = St1(Darowizna); break;
        case 2 : Podatek = St2(Darowizna); break;
        case 3 : Podatek = St3(Darowizna); break;
    }

    printf("\nPodatek wynosi %d zl.\n\n", Podatek);
    return 0;
}
```

Funkcje – Przykład 2

Opracować program, który oblicza wartość sumy iloczynów:

$$R = \sum_{i=1}^{n} \prod_{j=1}^{m} \frac{F1(i,j) - F2(i,j)}{F1^2(j+2, i+3) + 1}$$

gdzie

$$F1(i,j) = \frac{\sin^2 i + \cos^2 j}{i+j} \quad \text{gdy} \quad i \geq j$$

$$F1(i,j) = \frac{\cos^3 i + \sin^3 j}{2i+j} \quad \text{gdy} \quad i < j$$

$$F2(i,j) = \frac{3i + 2j + 1}{2i + 5i + 4}$$

Funkcje – Przykład 2 cd.

```
/* Suma Iloczynów */

double F1(int i, int j ){
      if (i >= j)
          return (pow(sin((double)i), 2) + pow(cos((double)j), 2)) / (i + j);
      else
          return (pow(cos((double)i), 3) + pow(sin((double)j), 3)) / (2 * i + j);
}

double F2(int i, int j){
      return (3 * i + 2 * j + 1) / (2 * i + 5 * j + 4.0);
}

double Iloczyn(int i, int m){
      double ilo = 1.0;

      for (int j = 1; j <= m; ++j)
          ilo *= (F1(i,j) - F2(i,j))/ (pow(F1(j + 2, i + 3), 2) + 1);
      return ilo;
}
```

Funkcje – Przykład 2 cd.

```
int main(){
      int n, m;
      double R = 0;

      printf("Podaj n, m : ");
      scanf("%d%d", &n, &m);

      // obliczenie sumy iloczynów od i = 1 do n
      for (int i = 1; i <= n; ++i)
          R += Iloczyn(i, m);

      printf("R = %.3lf", R);

      printf("\n\n");

      return 0;
}
```

Tablice jako argumenty funkcji

- Jeśli argumentem funkcji jest nazwa tablicy, to funkcja w rzeczywistości otrzymuje ADRES pierwszego elementu tablicy (a przy okazji reszty) – dlatego funkcja ma pełne prawo do TRWAŁEGO zmieniania wartości w tablicach.
- Wewnątrz wywoływanej funkcji temu argumentowi odpowiada zwykła zmienna lokalna.
- Parametr będący nazwą tablicy jest wskaźnikiem, czyli zmienną zawierającą adres.
- **Przykład:** funkcja obliczająca długość tekstu:

```
#include <stdio.h>
int dlugosc(char *);//prototyp

int main(){
	int n = dlugosc("Ahoj!");
	printf("%d\n", n);
		return 0;
}
```

Tablice jako argumenty funkcji

```
int dlugosc(char *s){
      int n;
      for(n=0; *s != '\0'; s++) n++;
      return n;
}
```

- Zwiększanie zmiennej *s* jest poprawne, ponieważ jest ona zmienną wskaźnikową.
- Operacja *s++* nie ma wpływu na ciąg znaków w miejscu wywołania funkcji *dlugosc*, ponieważ **zwiększa jedynie jej prywatną kopię wskaźnika**.

- Poniższe wywołania zadziałają poprawnie:

```
dlugosc("witaj swiecie"); // stała napisowa
dlugosc(tab); // char tab[100];
dlugosc(wsk); // char *wsk;
```

Tablice jako argumenty funkcji

- Do funkcji wolno przekazać część tablicy, podając wskaźnik do początku podtablicy, np. ***f(&a[3])***

- Poniższy przykład na niektórych kompilatorach zadziała (pamiętajmy, że tablica jest zwartym blokiem w pamięci, więc pomimo ograniczania „widoczności”, wcześniejsze komórki wciąż istnieją na swoich miejscach).
  - Nie zmienia to faktu, że poniższy przykład jest też absurdalny i mało bezpieczny!

```
int fun(int tab[]) {
   printf("Wartosc: %d\n", tab[-1]); // !!!
}
...
int tablica[] = {0,1,2,3,4,5}
fun(&tablica[2]); // wyświetli: „1”
```

Tablice jako argumenty funkcji i rozkaz sizeof

- Uwaga! Rozmiar tablicy można poprawnie określić rozkazem sizeof **zanim wyślemy tablicę do funkcji**.
- W momencie wysłania tablicy jako argumentu, wewnątrz funkcji nazwa argumentu formalnego (tablicy) redukuje się do wskaźnika, który w połączeniu z sizeof raczej nie da prawidłowego wyniku!

```
int fun(int tab[]) {
   printf("Rozmiar: %d\n", sizeof(tab) / sizeof(tab[0]) ); // !!!
}
...
int tablica[] = {0,1,2,3,4,5}
fun(tablica); // wyświetli... nie wiadomo co albo:
//  sizeof on array function parameter will return size of 'int *'
// instead of 'int []' [-Wsizeof-array-argument]
```

- Rozwiązaniem jest określenie rozmiaru tablicy **przed** wysłaniem jej do funkcji (w miejscu kodu gdzie nazwa tablicy jest wciąż... tablicą ☺), jako kolejnego argumentu tejże funkcji!

Tablice jako argumenty funkcji

- **Tablice jednowymiarowe – różne sposoby deklaracji:**

```
// Poniższe deklaracje są równoważne
// informacja dla kompilatora, że przekazujemy wskaźnik na zmienną
// preferowana – wyraźniej mówi, że parametr jest wskaźnikiem
int FT(int *);                              // deklaracje
// informujemy kompilator, że przekazujemy tablicę do funkcji.
// Przekazywany jest adres pierwszego elementu tablicy
int FT(int[ ]);
// efekt taki jak dla powyższej deklaracji, informacja o ilości
// elementów w tablicy jest zbędna
int FT(int[10]);

int FT(int *T) { ... }        // definicje
int FT(int T[ ]) { ... }
int FT(int T[10]) { ... }
//  T [ 3 ]
//  T  +   3   *   sizeof (  int  )
```

Tablice jako argumenty funkcji

- **Tablice jednowymiarowe, Przykład:**

```
void ZAP1( int T[100] ){
   for ( int i = 0; i < 100; ++i )
       T[ i ] = i;
}

int ALFA[15];
ZAP1 ( ALFA ) ;                // zapis + wyjście poza zakres tablicy
int BETA[100];
ZAP1 ( &BETA [0] );                        // zapis całej tablicy
int GAMMA[150];
ZAP1 ( GAMMA );                // zapis 2/3 tablicy
```

- Prawidłowa definicja powyższej funkcji:

```
void ZAP2(int T[ ], int rozmiar){ // podajemy ilość elementów i
                                  // adres pierwszego elementu tablicy
   for ( int i = 0; i < rozmiar; ++i )
       T[i] = i;
}
```

Tablice jako argumenty funkcji – Przykład 1

```
void F1(int T[50]){
          for(int i = 0; i < 50; ++i) T[i] = i;
}

int main(){

     int x;

     int T1[70] = {0};
     int T2[10];
     F1(T2);

     x = 0;

     return 0;
}
// program poprawnie się skompiluje, ale wystąpi błąd w trakcie wykonywania
// (wykonywanie programu zostanie przerwane przez system operacyjny)
```

Tablice jako argumenty funkcji

- **Tablice wielowymiarowe:**
  - Jeśli przekazujemy taką tablicę (np. `int daytab[2][13]`) do funkcji to trzeba podać w deklaracji liczbę kolumn.
  - Liczba wierszy nie jest ważna, ponieważ funkcji zostanie przekazany wskaźnik do tablicy wierszy.
  - Jeśli chcemy przekazać tablicę wielowymiarową do funkcji to deklaracja takiej funkcji może mieć postać:
    - `fun(int daytab[2][13]) { … }` -
    - `fun(int daytab[][13]) { … }` – ponieważ liczba wierszy nie jest ważna.

Tablice jako argumenty funkcji

- **Tablice wielowymiarowe, Przykład:**

```
void ZAP3( float TT[ ][10], int wierszy )
{
   for (int i = 0; i < wierszy; ++i)
       for (int j = 0; j < 10; ++j)
            TT[i][j] = i + j;
}

//  odwołanie do elementu tablicy TT [ 3 ] [ 5 ]
//  TT + 3 * 10 * sizeof ( float ) +
         5  * sizeof ( float )
```

Tablice jako argumenty funkcji – Przykład 2

- **Napisać program, który umożliwia przetwarzanie prostego rejestru telewizorów**

```
const int MAX = 100;
struct Telewizor{
     char Marka[32];
     int Cena;
     int Liczba;
};
// dodaj nowy telewizor; ilość telewizorów w rejestrze musi być < MAX
void Op_N(struct Telewizor Tab[ ], int &gdzie){
    if (gdzie < MAX){
         printf("Podaj nazwe : ");
         scanf("%31s", Tab[gdzie].Marka);
         printf("Podaj cene : ");
         scanf("%d", &Tab[gdzie].Cena);
         printf("Podaj liczbe sztuk : ");
         scanf("%d", &Tab[gdzie++].Liczba);
    } else
          printf("Tablica pelna.\n");
}
```

Tablice jako argumenty funkcji – Przykład 2 cd.

```
// wyświetl informacje o wszystkich telewizorach w rejestrze
void Op_W(struct Telewizor Tab[ ], int koniec) {
    for(int i = 0; i < koniec; ++i)
        printf("%d. Telewizor : %s, Cena : %d, Sztuk : %d\n", i, Tab[i].Marka,
                  Tab[i].Cena, Tab[i].Liczba);
}

// usuń telewizor o podanym numerze (indeks w tablicy)
void Op_U(struct Telewizor Tab[ ], int &koniec) {
    int ktory;

    printf("Podaj numer telewizora : ");
    scanf("%d", &ktory);
    if (ktory >= 0 && ktory < koniec) {
        Tab[ktory] = Tab[--koniec];
        printf("Usunieto.\n");
    } else printf("Bledny numer.\n");
}
```

Tablice jako argumenty funkcji – Przykład 2 cd.

```
// oblicz sumaryczną wartość wszystkich towarów rejestrze
void Op_S(struct Telewizor Tab[ ], int koniec) {
    int Suma = 0;

    for (int i = 0; i < koniec; ++i)
         Suma += Tab[i].Cena * Tab[i].Liczba;
    printf("Wartosc razem : %d\n", Suma);
}

int main() {
    struct Telewizor TabTel[MAX];
    int ile = 0;
    bool jeszcze = true;
    char opcja;

    while(jeszcze) {
        printf("Wybierz opcje [N, W, U, S, Q] : ");
        fflush(stdin);
        scanf("%c", &opcja);
```

Tablice jako argumenty funkcji – Przykład 2 cd.

```
    switch(opcja & 0x5F){
          case 'N': Op_N(TabTel, ile); break;

          case 'W': Op_W(TabTel, ile); break;

          case 'U': Op_U(TabTel, ile); break;

          case 'S': Op_S(TabTel, ile); break;

          case 'Q': jeszcze = false; break;

          default:printf("Bledna opcja.\n");
          }
    }

    return 0;
}
```

Wartości domyślne argumentów funkcji

- W **C++** istnieje możliwość definiowania funkcji o domyślnych wartościach argumentów. Oznacza to, że wywołując funkcję nie musimy podawać wszystkich argumentów: dla tych argumentów, których wartość nie została podana, przyjęta zostanie wartość domyślna.

```
float SUM3(float x, float y = 2.5, float z = 3.5)
{   return x + y + z;  }

float a, b, c;

SUM3( a, b, c ); // a + b + c
SUM3( a, b );    // a + b + 3.5
SUM3( a );       // a + 2.5 + 3.5
SUM3( );                // błąd, pierwszy argument nie jest domniemany
SUM3( a, ,c );          // błąd, w języku C++ nie może wystąpić taka
                        // sytuacja, że dwa przecinki stoją koło siebie
```

Rekurencja

- Teoria, przykłady

Funkcje - rekurencja

- O rekurencji mówimy wówczas, gdy definicja jakiegoś pojęcia odwołuje się do niego samego.
- Używanie rekurencji w programach jest możliwe dzięki podprogramom (procedurom, funkcjom).
- Funkcje języka C mogą być wywoływane **rekurencyjnie**, co oznacza, że funkcja może wywołać samą siebie zarówno bezpośrednio jak i pośrednio.
- **Rekurencja bezpośrednia:**
  - W ciele podprogramu *P* wywoływany jest podprogram *P*:

```
void P(){
     /* ... */
     P();
     /* ... */
}
```

Funkcje - rekurencja

- **Rekurencja pośrednia:**
  - W ciągu wywołań podprogramów jest więcej niż jedno wystąpienie (instancja) *P*.

```
void P(){
     /* ... */
     Q(); // funkcja Q wywołuje funkcję P
     /* ... */
}

void Q(){
     /* ... */
     P();
     /* ... */
}
```

Funkcje - rekurencja

- **Rozważając wywołania rekurencyjne należy pamiętać o następujących faktach**:
  - Wywołanie funkcji z wnętrza samej siebie to nie skok do początku funkcji. Tworzony jest nowy kontekst wywołania, a więc nowo wywołana funkcja ma swoje własne zmienne lokalne, argumenty (otrzymuje nowy komplet wszystkich zmiennych automatycznych, niezależny od poprzedniego) oraz wartość zwracaną, jakkolwiek realizuje tę samą listę instrukcji, co funkcja wywołująca.
  - Do czasu zakończenia wykonania funkcji wywoływanej, zatrzymane jest działanie funkcji wywołującej.

- Umieszczając w programie wywołanie funkcji oczekujemy, że po wykonaniu instrukcji, realizowanych przez wywołaną funkcję, nastąpi realizacja dalszych zaprogramowanych czynności.

- Aby było to możliwe, w momencie wywołania funkcji należy zapamiętać adres, pod którym znajduje się instrukcja, od której należy kontynuować wykonanie programu (tzw. **adres powrotu**) po zakończeniu działania wywołanej funkcji.

Funkcje - rekurencja

- Ponieważ wywoływana funkcja może zawierać kolejne wywołania pewnych funkcji, koniecznym jest sekwencyjne zapamiętywanie adresów powrotu.

- W miarę kończenia działania poszczególnych wywołań funkcji, należy przywracać działanie programu w kolejnych zapamiętanych adresach, przy czym adresy te wykorzystywane są w *odwrotnej kolejności*.

- Zapisywanie sekwencji adresów powrotu wykonywane jest na tzw. ***stosie*** – strukturze danych, która cechuje się tym, że dane zapamiętane w niej jako ostatnie, zostaną odczytane jako pierwsze (ang. **LIFO** – *Last In-First Out*).

- Takiego zachowania oczekuje się od struktury danych, która przechowuje adresy powrotu z funkcji, stąd realizacja procesu wywoływania funkcji w oparciu o stos jest powszechna.

- Rola stosu w procesie wywoływania funkcji nie ogranicza się tylko do przechowywania adresów powrotu z funkcji.

- Na stosie przechowywane są także: zmienne lokalne funkcji, jej argumenty, a w zależności od sytuacji, może być również przechowywana wartość zwracana przez funkcję.

Funkcje - rekurencja

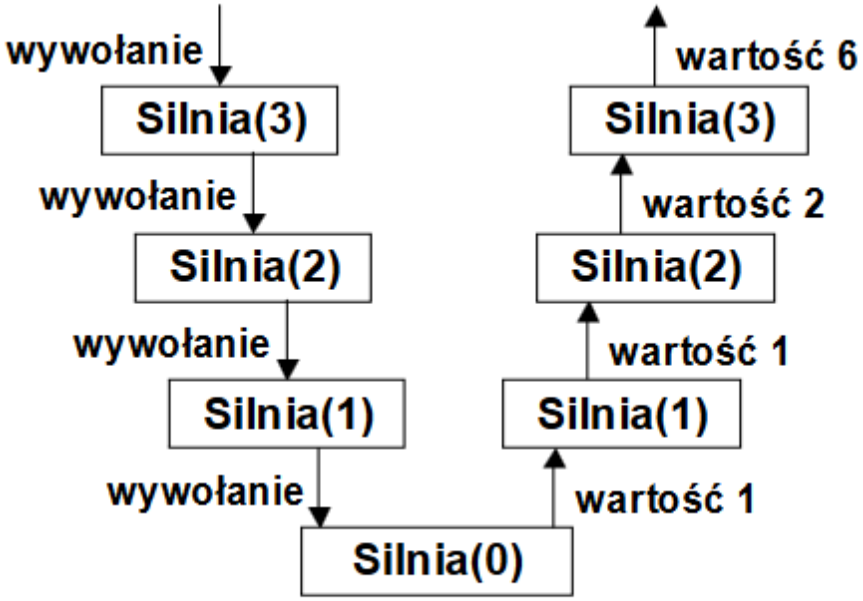
Silnia

Dekompozycja problemu

$$S(0) = 1 \quad \Rightarrow \quad S(n)=1 \qquad \text{dla } n=0$$
$$S(n) = 1*2*3* \ldots *(n-1)*n$$
$$= S(n-1) * n \quad \Rightarrow \quad S(n)=S(n-1)*n \qquad \text{dla } n>0$$

```
int Silnia(int n){
    if (n == 0) return 1;
    else return Silnia(n - 1) * n;
}
```

Ciąg instancji procedur



Funkcje - rekurencja

- **Przykład – wypisywanie liczby w postaci ciągu znaków:**

```
#include <stdio.h>
void printd(int n) {
      if (n < 0) {
          putchar('-');
          n = -n;
      }
      // cyfry generowane są w odwrotnej kolejności; Funkcja printd najpierw
      // wywołuje samą siebie dla uzyskania cyfr początkowych, a następnie wypisuje cyfrę
końcową
      if (n / 10)
          printd(n / 10);

      putchar(n % 10 + '0'); // % - operator modulo, reszta z dzielenia
}
```

- Przy wywołaniu *printd(123)* pierwsze wywołanie dostaje wartość 123, drugiemu zanim cokolwiek wypisze przesyła 12, drugie przed wypisaniem czegokolwiek wysyła trzeciemu wartość 1.

- Trzecie rekurencyjne wywołanie wypisuje '1', sterowanie wraca do drugiego, które wypisuje '2' (12 % 10 = 2), następnie sterowanie powraca do pierwszego wywołania funkcji *printd*, która wypisuje '3' (123 % 10 = 3).

Funkcje - rekurencja

- Rekurencja nie musi przynosić oszczędności pamięci, ponieważ trzeba przechowywać stos używanych wartości.
- Nie przyspiesza też działania funkcji.
- Postać rekurencyjna jest jednak bardziej zwarta i często łatwiejsza do napisania i zrozumienia (bardziej przejrzysta) niż odpowiadająca jej wersja iteracyjna.
- Funkcje rekurencyjne nadają się zwłaszcza do obsługi rekurencyjnie zdefiniowanych struktur danych takich jak np. drzewa.
- Trzeba **bezwzględnie** pamiętać o zaprojektowaniu "wyjścia" z rekurencji Bez tego program wszedłby w nieskończoną pętlę wywołań (w praktyce program zakończy się błędem).

Funkcje - rekurencja

- **Przykład – zapisz rekurencyjnie następujące wyrażenie:**

```
// mnożenie wykonywane jest tak długo dopóki n > 0
n  *  ( n - 2 )  *  ( n - 4 )  * ... //  > 0
```

- Wersja rekurencyjna:

```
int IR ( int  n ){
   return n <= 0 ? 1 : n * IR( n – 2 );
}
```

- Wersja iteracyjna:

```
int IP ( int  n ){
      int s = 1;
      while ( n > 0 )   {
      s *= n;
      n -= 2;
      }
      return s;
}
```

Funkcje inline (podprogramy wstawiane)

- To funkcje deklarowane ze słowem kluczowym *inline* umieszczonym przed nagłówkiem funkcji.
- Kod wynikowy takiej funkcji może być wpisany przez kompilator w każdorazowym miejscu jej wywołania (czas jej wykonania jest krótszy niż czas wywołania i powrotu).

- Słowo **inline** jest ignorowane jeśli funkcja zawiera pętle iteracyjne.

- **Przykład:**

```
inline double WB(double x, double y, double z)
{
      return x > y ? z : x;
}

inline int mnoz(int x, int y)
{
      return x*y;
}
```

Funkcje – Przykład

- Napisz program przechowujący nazwiska oraz numery Pesel podanych osób. Dane osób powinny być przechowywane w tablicy. Kluczem jest nazwisko osoby, a wartością numer Pesel. Wykorzystaj funkcję mieszającą.

- **Funkcja mieszająca (haszująca)**
  - Zadaniem funkcji mieszającej jest transformacja kluczy w indeksy tablicy.
  - Funkcja taka przyporządkowuje każdej liczbie wartość określaną jako **hasz** lub inaczej **skrót**.
  - Mając dany klucz K w pierwszym kroku należy obliczyć związany z nim indeks, w drugim natomiast sprawdzić, czy miejsce to jest dostępne (nie znajduje się tam już inny obiekt).
  - Przypadek znalezienia pod danym indeksem innego obiektu nazywamy **kolizją**.
  - Natomiast algorytm wyznaczający alternatywne indeksy nazywamy algorytmem usuwającym kolizje.

Funkcje – Przykład cd.

```
/*
Kodowanie mieszające

W - wpisanie nowej osoby (Nazwisko , Pesel)
P - poszukaj osoby (wg Nazwiska) i wyświetl Pesel jeśli znaleziono osobę
Q - koniec programu
*/

#include <stdio.h>
#include <string.h>

const int MAX = 100;

struct Osoba
{
    char *Nazwisko;
    char Pesel[12];
};
```

Funkcje – Przykład cd.

```
struct Pozycja{
     struct Osoba *Kto;
     int Zajete;   // -1    : wolne,
                   // -2    : zajęte i koniec listy,
                   //  >= 0 : następny (kolejna osoba)
};
int Mieszaj (char *Nazwa); // funkcja mieszająca
// znajduje wolne miejsce w tablicy
int Wolny (int poz, struct Pozycja TAB[]);
// wyszukiwanie osoby po nazwisku
int Szukaj (int poz, struct Pozycja TAB[ ], char *Nazwa);


int main( ){
     char Nazwa[64];
     struct Osoba *Nowy;
     int indeks, liczba = 0;
     struct Pozycja TAB[MAX]; // tablica osób
     char Pol = 'X';

     printf("\nWprowadz polecenie [W, P, Q]\n");
```

Funkcje – Przykład cd.

```
for (int i=0; i<MAX; ++i) TAB[i].Zajete = -1; // inicjalizacja
do {  // główna pętla programu
     printf("\n>"); // oczekiwanie na podanie opcji
     fflush(stdin);
     scanf("%c", &Pol);
     switch(Pol & 0x5F){
          case 'W' :
                if (liczba < MAX){ // dodanie nowej osoby
                    liczba++; // liczba wszystkich osób w tablicy
                    Nowy = new struct Osoba;
                    //wczytanie danych od użytkownika
                    printf("\nPodaj Nazwisko i Pesel : \n");
                    scanf("%62s", Nazwa);
                    scanf("%11s", Nowy->Pesel);
                    // przydział pamięci na nazwisko i skopiowanie
                    Nowy->Nazwisko = new char[strlen(Nazwa) + 1];
                    strncpy(Nowy->Nazwisko, Nazwa, strlen(Nazwa) + 1 );
                    // wstawienie nowej osoby do tablicy
                    TAB[Wolny(Mieszaj(Nazwa), TAB)].Kto = Nowy;
                } else
                    printf("\nTablica zapełniona\n");
                break;
```

Funkcje – Przykład cd.

```
        // wyszukiwanie osoby po nazwisku
        case 'P' :
             printf("\nPodaj Nazwisko : \n");
             scanf("%63s", Nazwa);
             // funkcja szukaj zwróci numer indeksu w tablicy lub -1
             // jeśli takiej osoby nie ma w tablicy
             indeks = Szukaj(Mieszaj(Nazwa), TAB, Nazwa);

             if (indeks == -1)
                   printf("\nNie ma.\n");
             else
                   printf("\n%s\n", TAB[indeks].Kto->Pesel); // odczytaj Pesel
             break;
        case 'Q' :
              printf("\nKoniec\n\n");
              break;

        default  :
              printf("\nZle polecenie [W, P, Q]\n");
        }
  }
  while ((Pol & 0x5F) != 'Q')
        ; // dopóki użytkownik nie poda Q lub q

  return 0;
}
```

Funkcje – Przykład cd.

```
// Najprostszy sposób odwzorowania słowa na liczbę to dodanie kodów
// wszystkich liter. Tak więc H("ala")=97+108+97=302
int Mieszaj (char *Nazwa){ // funkcja mieszająca
    int poz = 0;

    while (*Nazwa) // dopóki nie dojdzie do końca tekstu
         poz += *Nazwa++; // sumuj kody ascii wszystkich znaków tekstu
    // proponowany indeks w tablicy, pod który należy wpisać nową osobę
    return poz % MAX;
}

// znajduje wolne miejsce w tablicy, pod którym można wpisać nową osobę
int Wolny (int poz, struct Pozycja TAB[  ]){
    int koniec = poz, nast;
    // Pole Zajete służy do przechowywania informacji o kolejnych
    // elementach na liście elementów o tej samej wartości skrótu (hash).
    if (TAB[poz].Zajete == -1) {// czy wolne
         TAB[poz].Zajete = -2; // ustaw znacznik końca listy
         return poz;
    }
```

Funkcje – Przykład cd.

```
        else {
            // przejdź na koniec listy obiektów o tym samym hashu (skrócie)
            // na koniec tej listy wpiszemy indeks w tablicy dodawanej osoby

            while (TAB[koniec].Zajete != -2)
                  koniec = TAB[koniec].Zajete;

            // ustalenie nowego indeksu w tablicy dla dodawanej osoby
            nast = (koniec + 1) % MAX;
            // jeśli wybrane miejsce jest zajęte to szukaj nowego miejsca z
            // krokiem (nast + 1) % MAX
            while (TAB[nast].Zajete != -1)
                  nast = (nast + 1) % MAX;

            // ustaw znacznik końca listy
            TAB[nast].Zajete = -2;
            TAB[koniec].Zajete = nast; // wstaw nowy element na koniec listy
            return nast;
        }
    }
```

Funkcje – Przykład cd.

```
// wyszukiwanie osoby po nazwisku
int Szukaj (int poz, Pozycja TAB[ ], char *Nazwa){
    int nast = poz;

    // osoba o podanym nazwisku nie znajduje się w tablicy
    if (TAB[nast].Zajete == -1) return -1;

    // osoba o podanym nazwisku została znaleziona w tablicy
    if (strcmp(TAB[nast].Kto->Nazwisko, Nazwa) == 0)
         return nast;
    else
     // doszliśmy do końca listy (znacznik -2) - osoba o podanym
     // nazwisku nie znajduje się w tablicy
         if (TAB[nast].Zajete == -2) return -1;
    // kontynuuj przeglądanie listy rekurencyjnie
         else return Szukaj(TAB[nast].Zajete, TAB, Nazwa);
}
```

Pytania?