Wykład 5

- Struktury, dynamiczne struktury danych, pliki projektu i kompilacja programu

Instytut Informatyki

dr inż. Marcin Radom

Struktury

- Teoria, przykłady

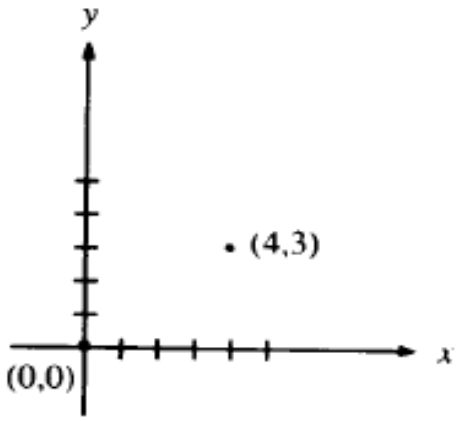
Struktury

- **Struktura** jest obiektem złożonym z jednej lub wielu zmiennych zgromadzonych pod jedną nazwą („rekordy” w Pascalu).
- Ułatwiają zorganizowanie skomplikowanych danych (traktowane są jako jeden obiekt a nie zestaw oddzielnych obiektów).
- Przykład – atrybuty opisujące pracownika w firmie: imię, nazwisko, stanowisko, data zatrudnienia, pensja, inne.
- Struktura jest więc zestawem pól (nazywane składowymi struktury), gdzie pole może być daną lub strukturą danych.
- Co można robić ze strukturami?
  - przypisywać jedną do drugiej
  - kopiować jedną do drugiej
  - zwracać jako wartość funkcyjną
  - przekazywać do funkcji
- **Definicja struktury:**

```
struct typ_struktury { lista_pól }
                lista_identyfikatorów  ;
```

Struktury

- Etykieta struktury (typ struktury) jest opcjonalna.
- Deklaracja struktury, po której nie występuje lista zmiennych (identyfikatorów) nie rezerwuje pamięci (opisuje wzorzec struktury).
- **Przykład** – współrzędne punktu na płaszczyźnie:




```
struct point {          // nie rezerwuje pamięci
           int x;
           int y;
};
struct point pt; // definicja zmiennej pt będącej strukturą typu point
point pt; // w C++ (można używać samej nazwy typu bez słowa struct)
```

Struktury

- Strukturę można zainicjować dopisując na końcu jej definicji listę wartości początkowych (wyrażeń stałych) dla jej składowych, np.:

```
struct point maxpt = {1920, 1200};
```

- W inicjalizacji bezpośredniej (jak wyżej) wartości składowych przypisywane są w kolejności ich występowania w opisie struktury.

- **Przykłady:**

```
struct{
   int szerokosc;
   int dokladnosc;
   char konwersja;
}  wzorzec;
```

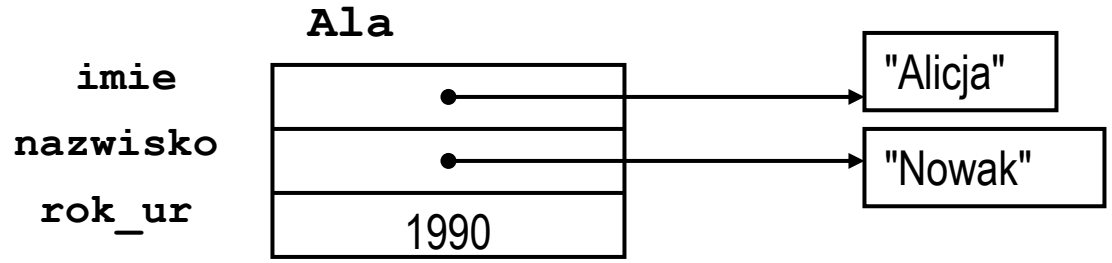
```
struct{
   float re;
   float im;
}  z0, z1, z2;
```

Struktury

- **Przykłady:**

```
struct osoba
{
   char *imie;
   char *nazwisko;
   int rok_urodzenia;
};
struct osoba Ala, Ola, Ela;

struct adres
{
   char miejscowosc[16];
   char ulica[32];
   int dom;
   int mieszkanie;
};
struct adres AAla, AOla, AEla;
```

Struktury


Ala
imie
nazwisko
rok_ur
"Alicja"
"Nowak"
1990


AAla
miejscowosc
"Koło"
16
ulica
"Mała"
32
dom
5
mieszkanie
11

Struktury

- Mając daną strukturę:

```
struct point {
      int x;
      int y;
}pt, pt1;
```

- odwołania do poszczególnych składowych danej struktury realizuje się poprzez:

`nazwa_struktury.nazwa_składowej`

np.

```
pt.x = 320;
pt.y = 200;
printf("%d \t %d", pt.x, pt.y); // wypisanie składowych
```

- Operator **. (kropka)** wiąże nazwę struktury z nazwą składowej.

- Np. obliczanie odległości między dwoma punktami (**`double sqrt(double)`** - funkcja zwracająca pierwiastek kwadratowy ze swojego argumentu wywołania):

```
double wynik = 0.0;
wynik = sqrt(pow((double)pt.x - pt1.x, 2) + pow((double)pt.y - pt1.y, 2));
```

Struktury – zagnieżdżanie struktur

- **Przykłady**:

```
struct silnik
{
   float pojemnosc;
   char *paliwo;
};

struct samochod
{
  struct
  {
     char *producent;
     char *model;
  } marka;
 struct silnik naped;
};
```

Struktury – zagnieżdżanie struktur

- Jak zdefiniować prostokąt? Na przykład podając współrzędne dwóch punktów, które wskazują na przeciwległe wierzchołki:

```
struct rect
{
      struct point p1;
      struct point p2;
}
```

- Odwołanie:

```
struct rect moj_prostokat;
moj_prostokat.p1.x = 10;          // odnosi się do współrzędnej x punktu
                                  // p1 będącego składową zmiennej
                                  // mój_prostokat
moj_prostokat.p1.y = 20;          //itd.
```

Struktury – dozwolone operacje

- **Dozwolone operacje:**
  - Przypisanie w całości jednej struktury do drugiej
  - Skopiowanie w całości struktury na inną
  - Pobranie adresu struktury za pomocą operatora &
  - Odwołanie do poszczególnych składowych
  - Strukturę można zainicjować listą wartości stałych (jej składowe) oraz przez przypisanie.

- **Struktur NIE MOŻNA bezpośrednio porównywać**, tj. należałoby napisać do tego oddzielną funkcję, nie można jednak użyć operatorów **==** oraz **!=** na całych strukturach (co najwyżej na składowych).

Struktury – przypisanie struktur

```
struct podpis
{
   char *Imie;
   char Inicjal;
   char *Nazwisko;
};
struct podpis AK = {"Anna", 'M', "Kowalska"}, NN, st;

st.Imie  =  "Andrzej" ; // tekst stały - nie można go modyfikować!

NN  =  AK ;        // gdy typy równe
//    NN . Imie  =  AK . Imie ;
//    NN . Inicjal  =  AK . Inicjal ;
//    NN . Nazwisko  =  AK . Nazwisko ;
//  AK.Imie i NN.Imie wskazują ten sam tekst stały
//  AK.Nazwisko i NN.Nazwisko wskazują ten sam tekst stały
```

Struktury – przypisanie struktur

```
char PiekneImie[20] = { "Teofil" };
AK.Imie = PiekneImie;       // AK.Imie wskazuje na początek tablicy PiekneImie
NN = AK;                    // teraz NN.Imie wskazuje tak samo jak AK.Imie
AK.Imie[0] = 'M';           // modyfikacja będzie widoczna w tablicy PiekneImie
                            // oraz AK.Imie i NN.Imie

printf("%s, %s", AK.Imie, NN.Imie);
// Meofil, Meofil
```

Struktury – przypisanie struktur

- **Przykład**:

```
int T1[3], T2[3] = { 1, 2, 3 };
T1 = T2;       // błąd, T1 nie jest zmienną wskaźnikową i nie można
               // modyfikować jej wskazania (adresu). Nie nastąpi też
               // automatyczne przekopiowanie elementów


struct TT{
     int T[3];
};
struct TT T1, T2 = { { 1, 2, 3} };
T1 = T2;     // poprawnie, kopiowanie tablic będących składowymi
               // struktur T1 i T2
// ale
int a = T1.T[1];
```

Tablice struktur

- Teoria, przykłady

Tablice struktur

- Załóżmy, że mamy strukturę zawierającą następujące składowe: ciąg znaków (tekst) oraz zmienną typu **int**:

```
struct key {
    char *word;
    int count;
}
```

- Tablicę takich struktur (tablicę, której elementy są strukturami) można zadeklarować w ten sposób:

```
struct key keytab[50];
```

- czyli składnia jest następująca:

```
struct nazwa_struktury nazwa_tablicy [ rozmiar ]
```

Tablice struktur

- **Przykład:**

```
struct kartoteka
{
   char  nazwa [ 16 ] ;
   int  liczba ;
}  komputery [ 100 ] ;            // rezerwuje pamięć na tablicę

struct kartoteka drukarki [ 100 ] ; // rezerwuje pamięć na tablicę


int  ile_k  =  0,  ile_d  =  0;

for ( int  i  =  0 ; i  < 100 ;  ++i ) // sumuje
{
   ile_k  +=   komputery [ i ] . liczba ;
   ile_d  +=   drukarki [ i ]  . liczba ;
}
```

Tablice struktur: Inicjalizacja

- Jak pamiętamy, tablicę można zainicjować listą wartości początkowych w momencie jej definiowania, np.

```
int rok[] = {31, 28, 31, …, 30, 31};
```

- Inicjowanie tablicy struktur wykonuje się podobnie - po definicji podaje się ujętą w klamry listę wartości początkowych:

```
struct key {
     char *word;
     int count;
} keytab[] = {
     "auto",0,
     "break",0,
     /* … */
     "while",0
};
```

  - Wartości początkowe wymienione są parami odpowiadającymi składowym struktury (czyli zgodnie z kolejnością ich występowania w strukturze).

Tablice struktur: Inicjalizacja

- Precyzyjniej byłoby ująć w nawiasy klamrowe wartości początkowe dla każdego „wiersza” tablicy (każdej struktury), np.:

```
struct key {
    char *word;
    int count;
} keytab[] = {        // liczba elementów tablicy keytab zostanie
                      // wyliczona automatycznie
    {"auto",0},
    {"break",0},
    /* … */
    {"while",0}
};
```

- ale gdy podaje się wszystkie pola struktury bez wyjątku, oraz są to proste napisy lub stałe, to może ich nie być.

Tablice struktur: Przykład

```
const int MAX = 100;

struct Telewizor{
          char Marka[32];
          int Cena;
          int Liczba;
};
// Napisać program, który umożliwia przetwarzanie prostego rejestru
// telewizorów
int main(int argc, char* argv[]){
          struct Telewizor TabTel[MAX];

          int ile = 0, ktory;
          bool jeszcze = true;
          char opcja;

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

Tablice struktur: Przykład cd.

```
switch(opcja & 0x5F){
// dodaj nowy telewizor; zmienna ile przechowuje ilość telewizorów w
// rejestrze i musi być mniejsza od MAX
case 'N': if (ile < MAX){
                  printf("Podaj nazwe : ");
                  scanf("%31s", TabTel[ile].Marka); // marka jest tablicą
                  printf("Podaj cene : ");
                  scanf("%d", &TabTel[ile].Cena);
                  printf("Podaj liczbe sztuk : ");
                  scanf("%d", &TabTel[ile++].Liczba);
          } else
                  printf("Tablica pelna.\n");
          break;

// wyświetl wszystkie telewizory
case 'W': for(int i = 0; i < ile; ++i)
                  printf("%d. Telewizor : %s, Cena : %d, Sztuk : %d\n",
                  i, TabTel[i].Marka, TabTel[i].Cena, TabTel[i].Liczba);
          break;
```

Tablice struktur: Przykład cd.

```
            // usuń telewizor o podanym numerze (indeks w tablicy)
            case 'U':  printf("Podaj numer telewizora : ");
                       scanf("%d", &ktory);
                       if (ktory >= 0 && ktory < ile){
                       // w miejsce usuniętego wstawia element ostatni
                                  TabTel[ktory] = TabTel[--ile];
                                  printf("Usunieto.\n");
                       } else printf("Bledny numer.\n");
                       break;
            // oblicz sumaryczną wartość wszystkich towarów rejestrze
            case 'S':
                       ktory = 0;
                       for (int i = 0; i < ile; ++i)
                                  ktory += TabTel[i].Cena * TabTel[i].Liczba;
                       printf("Wartosc razem : %d\n", ktory);
                       break;
            // koniec programu
            case 'Q':  jeszcze = false;
                       break;
            default:printf("Bledna opcja.\n");
            }
}

return 0;
}
```

Struktury – wskaźniki do struktur

- **Przykład:**

```
struct key {
      char *word;
      int count;
}
struct key *p;            // wskaźnik na strukturę key, chwilowo jeszcze na
                          // nic nie wskazuje
struct key tablica[] = { <wypełnienie tablicy> };
p = &tablica[0];          // wskaźnik wskazuje na pierwszy element
                          // tablicy struktur tablica

//
struct AZ{
   char z1;
   char z2;
   int lz;
} p1 = { 'a', 'b', 37 },
  p2 = { 'x', 'y', 35 };

struct AZ p3 = { 'k', 'l', 36 };
```

Struktury – wskaźniki do struktur

- **Przykład:**

```
struct S1{
   int i;
   float f;
} es1;

struct S2{
   int i;
   long l;
} es2;

struct S1 *wst1;
struct S2 *wst2;

wst1 = &es1;
wst2 = &es2;
wst1 = &es2;  // błąd, nieodpowiedni typ struktury
wst2 = wst1;  // błąd, nieodpowiedni typ wskaźnika
```

Struktury – dostęp do składowej

- Dostęp do pól (składowych) struktury realizowany jest następująco:
  - **Dla identyfikatora .**
    `nazwa_struktury.nazwa_składowej`
  - **Dla referencji .**
    `referencja.nazwa_składowej`
  - **Dla wskaźnika ->**
    `wskaznik->nazwa_składowej`
- **Przykład:**

```
struct podpis
{
   char *Imie;
   char Inicjal;
   char *Nazwisko;
};

struct podpis st, *wst = &st ;
st . Imie   =  "Andrzej" ;          // st jest identyfikatorem
wst -> Inicjal  =  'K' ;            // wst jest wskaźnikiem do struktury
```

Struktury dynamiczne

- Wskaźniki, listy...

Struktury – wskaźniki do struktur

- Wskaźniki do struktur są takimi samymi wskaźnikami jak do zwykłych zmiennych:

```
struct point {
      int x;
      int y;
}pt;

struct point *pp = &pt;
```

  - ***pp*** jest wskaźnikiem do struktury typu **`struct point`**
  - jeśli ***pp*** wskazuje na strukturę typu *point*, to ****pp*** jest taką strukturą, a ***(*pp).x*** oraz ***(*pp).y*** są jej składowymi.
  - Nawiasy okrągłe są konieczne, ponieważ operator składowej struktury **.** ma wyższy priorytet niż operator adresowania pośredniego ***.**
  - Wyrażenie ****pp.x*** oznacza to samo co ****(pp.x)***, więc jest błędne ponieważ *x* nie jest wskaźnikiem.

Struktury – wskaźniki do struktur

```
struct rect{     // definicja prostokąta
      struct point pt1;
      struct point pt2;
}
struct rect r, *rp = &r;
```

- Operatory **.** oraz **->** są lewostronnie łączne, więc następujące wyrażenia są równoważne:

```
r.pt1.x
rp->pt1.x  // pt1.x, bo pt1 nie jest wskaźnikiem
(r.pt1).x
(rp->pt1).x
```

- Cztery operatory:
  - odwołanie do elementów struktury: . (kropka)
  - operator: ->
  - operator: () - wywołania funkcji
  - operator: [] - indeksowanie tablicy

  mają NAJWYŻSZY priorytet, czyli zawsze wykonują się przed innymi (najsilniej wiążą swoje argumenty).

Struktury – wskaźniki do struktur

- **Przykład:**

```
struct {
      int len;
      char *str;
} *p;

++p->len       // zwiększy zmienną len, ponieważ ++ ma niższy
               // priorytet niż ->
```

- Na tej samej zasadzie wyrażenia:

  - **`(++p)->len`** zwiększa *p* przed odwołaniem do *len*
  - **`p++->len`** zwiększa *p* po odwołaniu do *len*
  - **`*p->str`** udostępnia to coś, na co wskazuje *str*
  - **`*p->str++`** z większa *str*, po udostępnieniu tego, na co *str* wskazuje (tak jak konstrukcja **`*s++`**)
  - **`(*p->str)++`** zwiększa to, na co wskazuje *str*
  - **`*p++->str`** zwiększa *p*, po udostępnieniu tego co wskazuje *str*

- Wniosek? Lepiej jest stosować nawiasy. Minimalizujemy błędy wynikające ze złego zrozumienia priorytetów operatorów (a nawiasy mają najwyższy).

Struktury – dynamiczny przydział pamięci

- **Przykład:**

```
struct pies{
   char *smycz;
   char *obroza;
   char *pies_wlasciwy;
};

struct pies *Morus;
Morus = new struct pies;    // alokuje pamięć dla struktury; wskaźniki
                            // będące składowymi struktury na razie na
                            // nic nie wskazują

     // należy dla każdej składowej będącej wskaźnikiem przydzielić pamięć, np.:
Morus->smycz = new char[20];
delete [] Morus->smycz;     // pamięć przydzieloną składowym trzeba
                            // zwolnić osobno, przed usunięciem z pamięci samej struktury
delete Morus;    // zwalnia pamięć zaalokowaną dla struktury
```

Struktury – dynamiczny przydział pamięci, Przykład

```
// Napisać program, który umożliwia przetwarzanie prostego rejestru
// rowerów: dodaj, usuń, wyświetl wszystko, zsumuj wartość wszystkich
const int MAX = 100;

struct Rower{
          char *Marka;
          int Cena;
          int Liczba;
};

int main(int argc, char* argv[]){
     struct Rower* TabRow[MAX]; // tablica wskaźników do struktur
     int ile = 0, ktory;
     bool jeszcze = true;
     char opcja;
     char Bufor[64];

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

Struktury – dynamiczny przydział pamięci, Przykład cd.

```
switch(opcja & 0x5F){
// dodaj nowy rower; zmienna ile przechowuje ilość rowerów w
// rejestrze i musi być mniejsza od MAX
case 'N': if (ile < MAX){
          // przydziel pamięć na nową strukturę w tablicy TabRow
          TabRow[ile] = new struct Rower;

          printf("Podaj nazwe : ");
          scanf("%63s", Bufor);
          // przydziel pamięć na składową Marka nowej struktury
          // na podstawie długości wczytanego z wejścia tekstu
          TabRow[ile]->Marka = new char[strlen(Bufor) + 1];
          // skopiuj opis ze zmiennej bufor do składowej Marka
          strcpy(TabRow[ile]->Marka, Bufor);

          printf("Podaj cene : ");
          scanf("%d", &TabRow[ile]->Cena);
          printf("Podaj liczbe sztuk : ");
          scanf("%d", &TabRow[ile++]->Liczba);
} else
          printf("Tablica pelna.\n");
break;
```

Struktury – dynamiczny przydział pamięci, Przykład cd.

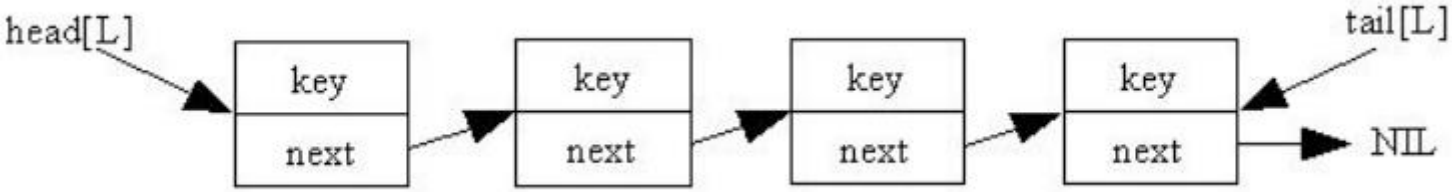
```
// wyświetl wszystkie rowery
case 'W': for(int i = 0; i < ile; ++i)
          printf("%d. Rower : %s, Cena : %d, Sztuk : %d\n",
          i, TabRow[i]->Marka, TabRow[i]->Cena, TabRow[i]->Liczba);
          break;

// usuń rower o podanym numerze (indeks w tablicy)
case 'U': printf("Podaj numer roweru : ");
          scanf("%d", &ktory);
          if (ktory >= 0 && ktory < ile){
                    // najpierw usuń składowe struktury, którym przydzielono
                    // pamięć dynamicznie
                    delete [ ] TabRow[ktory]->Marka;
                    delete TabRow[ktory]; // usuń strukturę
                    // przepisz ostatni element na miejsce usuniętego
                    TabRow[ktory] = TabRow[--ile]; // skopiowanie adresu
                    printf("Usunieto.\n");
          } else
                    printf("Bledny numer.\n");
          break;
```

Struktury – dynamiczny przydział pamięci, Przykład cd.

```
        // oblicz sumaryczną wartość wszystkich towarów rejestrze
        case 'S': ktory = 0;
                for (int i = 0; i < ile; ++i)
                        // suma cena * ilość dla wszystkich rowerów
                        ktory += TabRow[i]->Cena * TabRow[i]->Liczba;
                printf("Wartosc razem : %d\n", ktory);
                break;

        // koniec programu
        case 'Q': jeszcze = false;
                break;

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

Listy

- **Lista jednokierunkowa** (każda struktura ma wskaźnik na jedną następną).



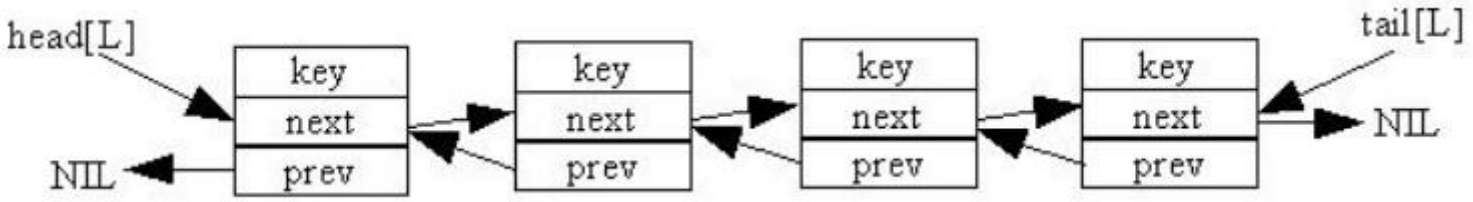
- Deklaracja pojedynczego węzła (rekurencyjna):

```
struct lnode{
        int number;
        struct lnode *next;
}
```

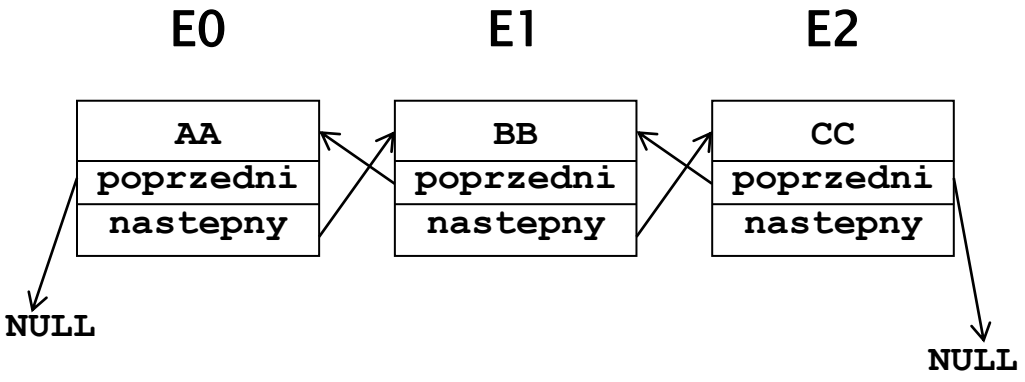
  - Wskaźnik *next* jest to wskaźnik do STRUKTURY *lnode*, ale nie na nią samą, tj. struktura nie wskazuje na samą siebie w pamięci, ale ma wskaźnik, który musi wskazywać na strukturę tego samego typu (czyli *lnode*), ale oczywiście z INNĄ ZAWARTOŚCIĄ, w innym miejscu pamięci komputera.

Listy

- **Lista dwukierunkowa** - Każdy struktura ma wskaźnik na dwie inne, odpowiednio (umownie!) na 'poprzednika' i 'następnika'.



- **Przykład:**




```
struct lista{
       char etykieta[8];
       struct lista *poprzedni;
       struct lista *nastepny;
} E0 = { "AA", NULL, NULL };
struct lista E1 = { "BB" },
      E2 = { "CC", NULL, NULL };
E0.nastepny = &E1; // E0 pamięta wskaźnik na następny element E1
E1.nastepny = &E2; // E1 pamięta wskaźnik na następny element E2
E2.poprzedni = &E1; // E2 pamięta wskaźnik na poprzedni element E1
E1.poprzedni = &E0; // E1 pamięta wskaźnik na poprzedni element E0
```

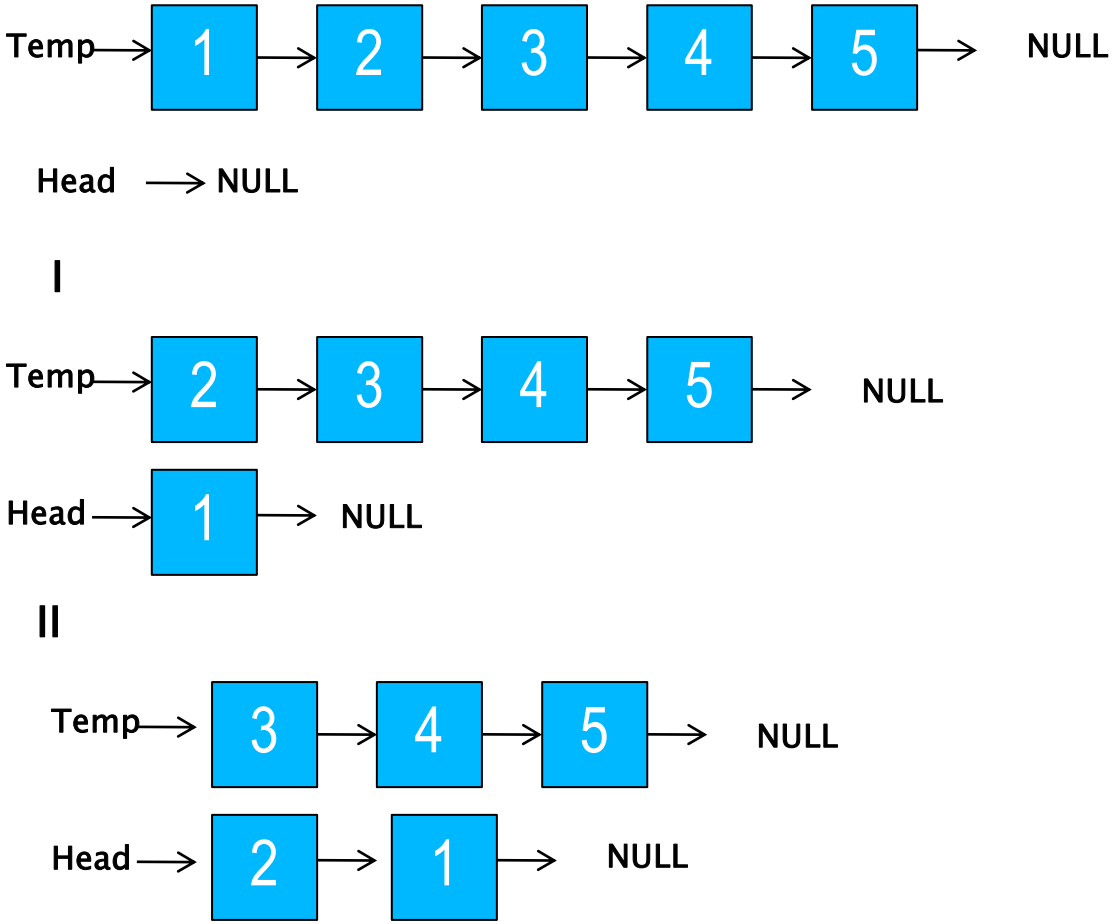
Lista jednokierunkowa - Przykład

```
struct Elem {                   // deklaracja pojedynczego węzła
          Elem *Next;           // wskaźnik na następny węzeł listy
          int Value;
};
struct Elem *Head;              // wskaźnik do struktury typu Elem

int main(){
          //------------- Utworzenie listy --------------
          // przypisz wartości poszczególnym węzłom listy
     Elem e1 = {NULL,  10};
     Elem e2 = {NULL,  20};
     Elem e3 = {NULL,  30};
     Elem e4 = {NULL,  40};
     Elem e5 = {NULL,  50};

     e1.Next = &e2; // e1 pamięta wskaźnik na następny element e2
     e2.Next = &e3; // e2 pamięta wskaźnik na następny element e3
     e3.Next = &e4; // e3 pamięta wskaźnik na następny element e4
     e4.Next = &e5; // e4 pamięta wskaźnik na następny element e5
     Head = &e1; // wskaźnik na pierwszy element listy e1 (tzw. Głowa listy)
```

Lista jednokierunkowa – Przykład cd.

```
//------------ Zmienne pomocnicze ---------------
Elem *Temp, *Aux;

//------------ Wyświetlenie całej listy -------------
printf("\nPrzed odwroceniem :\n\n");

Temp = Head; // przypisz wskaźnik na pierwszy element listy

while (Temp != NULL){ // do ostatniego elementu listy
     printf("%3d", Temp->Value); // wypisz wartość w danym węźle
     Temp = Temp->Next; // przejdź do kolejnego węzła listy
}

//---------------- Odwrócenie listy -------------------
Temp = Head; // bieżący
Head = NULL; // głowa odwróconej listy
while (Temp != NULL){
     Aux = Temp->Next; // ogon listy wejściowej (odwracanej)
     Temp->Next = Head; //nowa głowa dla częściowo odwróconej listy
     // pierwszy element ogona listy wejściowej staje się głową listy
     Head = Temp;
     Temp = Aux; // skrócona lista wejściowa (o głowę)
}
```

Lista jednokierunkowa – Przykład cd.



Lista jednokierunkowa – Przykład cd.

```
    //---------------- Wyświetlenie całej listy --------------------

    printf("\n\nPo odwroceniu :\n\n");

    Temp = Head; // przypisz wskaźnik na pierwszy element listy

    while (Temp != NULL){
        printf("%3d", Temp->Value);
        Temp = Temp->Next;
    }

    printf("\n\n");
    return 0;
}
```

Unia

- Unia jest zmienną i jej składnia wzorowana jest na strukturach, z tym, że w danej chwili unia może przechowywać tylko jedną zmienną (ale dowolnego typu zawartego w unii).
- Unia w różnych momentach może zawierać obiekty różnych typów i rozmiarów, przy czym kompilator dba o spełnienie wymagań dotyczących rozmiaru i położenia w pamięci.
- **Wszystkie pola unii znajdują się w tym samym obszarze pamięci.**

- **Przykład:**

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

  - Zmienna *u* będzie na tyle obszerna, żeby pomieścić w sobie wartość największego z tych trzech typów.
  - Zmiennej *u* można przypisać wartość każdego z tych typów, a następnie używać jej w wyrażeniu.
  - W danej chwili zmienna *u* może przechowywać wartość *int*, *float* lub wskaźnik na ciąg znaków. Ale tylko jedną z nich w danej chwili!

Unia

- Do programisty należy kontrola typu obiektu aktualnie znajdującego się w unii (jeśli zapamiętamy coś z jednym typem a pobierzemy z innym, to wynik jest zależny od implementacji).

- **Postać odwołania do składowych unii:**

  `nazwa-unii.składowa`

  lub

  `wskaźnik-do-unii->składowa`

  czyli identyczna, jak dla struktur.

- Załóżmy, że mamy zmienną *utype*, która służy do kontroli tego, co ma w sobie unia *u*:

```
enum typy {INT, FLOAT, STRING};

typy utype;
/* … */
if (utype == INT) printf("%d\n", u.ival);
else if (utype == FLOAT) printf("%f\n", u.fval);
else if (utype == STRING) printf("%s\n", u.sval);
else printf("bad type %d in utype\n", utype);
```

Unia

- **Przykłady:**

```
union
{
   char p1 ;            //  1 bajt
   int  p2 ;            //  4 bajty
   long long p3 ;       //  8 bajtów
} u1;                   //  8 bajtów

u1.p1 = 'A';         // pola p2, p3 niezdefiniowane
u1.p2 = 1357;        // pola p1, p3 niezdefiniowane
u1.p3 = 15432678LL; // pola p1, p2 niezdefiniowane


union umaszczenie
{
     char kolor_pior [ 64 ] ;
     char kolor_siersci [ 64 ] ;
}
```

Unia wewnątrz struktury

- Unie mogą występować w tablicach, strukturach i vice versa.
- Notacja dla odwołania do składowej unii w strukturze (lub odwrotnie) jest identyczna jak dla zagnieżdżonych struktur. Np. dla tablicy struktur:

```
struct {
      char *name; // nazwa symbolu
      int flags; // znacznik stanu
      int utype; // typ wartości
      union { // wartość
            int ival;
            float fval;
            char *sval;
      } u;
} symtab[NSYM]; // tablica symboli
```

  - do składowej *ival* odwołamy się poprzez:

```
symtab[i].u.ival
```

  - do pierwszego znaku tekstu wskazywanego przez *sval* na dwa równoważne sposoby:

```
// nie: symtab[i].u.*sval !!!! – błąd, nie ma zmiennej sval
*symtab[i].u.sval // kropka ma wyższy priorytet niż *
symtab[i].u.sval[0]
```

Unia – zagnieżdżona definicja

- **Przykład:**

```
struct artykul{
   char autor[64];
   char tytul[64];
   int rok;
   union {
      struct {
         char tytul[64];
         int numer;
         int strona;
      } czasopismo;
      struct {
         char nazwa[64];
      } konferencja;
   } miejsce;
} a1;
// string.h
strcpy ( a1.miejsce.konferencja.nazwa, "Polman" );
```

Projekt na wiele plików źródłowych

- Teoria, zastosowanie, zasady organizacji zmiennych

Łączenie wielu plików programu

- **Podział programu na moduły pozwala na:**
  - Stworzenie programu, który jest bardziej czytelny. Duży program pisany jako jedna całość jest nieczytelny (brak pośrednich form organizacji poza podziałem na funkcje), łatwiej w nim także popełnić błędy (zmodyfikowanie zmiennej globalnej, zależność działania jednej funkcji od ubocznych efektów działania drugiej).
  - Łatwiejsze wykorzystanie poszczególnych modułów w innych programach**.**
  - Kompilowanie poszczególnych modułów niezależnie. Jeśli dokonamy zmian w jednym z modułów to nie będziemy musieli kompilować ponownie całości, tylko ten jeden plik.
- **Modułem** będziemy nazywać fragment programu kompilowany jako jeden plik.
- **Każdy moduł składa się z:**
  - Interfejsu (tego co dany plik udostępnia) czyli pliku nagłówkowego (*.h lub .hpp)
  - Implementacji (realizacji) czyli pliku implementacyjnego (*c. lub *.cpp). Na początku takiego pliku należy włączyć odpowiadający implementacji plik nagłówkowy (Np. dla pliku *module.c* będzie to plik *module.h*).

Łączenie wielu plików programu

- **Typowy plik nagłówkowy może zawierać następujące elementy:**
  - Stałe jawne (np. **EOF**, **NULL** w pliku *stdio.h*)
  - Funkcje-makra np. *getchar()* jest z reguły definiowana jako zamiennik *getc(stdin)* czy funkcje z rodziny *ctype.h*
  - Deklaracje funkcji, np. plik *string.h* zawiera deklaracje (prototypy) funkcji operujących na łańcuchach.
  - Definicje szablonów, struktur, np. standardowe funkcje we/wy korzystają ze struktury zawierającej informacje o pliku i związanym z nim buforze (szablon tej struktury znajduje się w pliku *stdio.h*).
  - Definicje typów, np. typ **FILE** jest wskaźnikiem do struktury zdefiniowanym w pliku *stdio.h* (za pomocą *typedef* lub dyrektywy *#define*). W plikach nagłówkowych zdefiniowane są również typy *size_t* oraz *time_t*.

- Każdy plik nagłówkowy powinien być napisany tak, aby można go było kilkakrotnie włączać do dowolnego modułu i w dowolnym miejscu.

Łączenie wielu plików programu

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Łączenie wielu plików programu

- Wielu programistów opracowuje własne pliki nagłówkowe i wykorzystuje je w swoich programach. Niektóre z nich są przeznaczone do specjalnych zastosowań, inne można dołączać do niemal każdego programu.

- Dołączane pliki mogą same zawierać dyrektywy *#include*.

- Pliki nagłówkowe mogą zawierać deklaracje zmiennych zewnętrznych współużytkowanych przez kilka plików. Uważane jest to jednak za rozwiązanie wadliwe stylistycznie, ponieważ sprawdzenie, czy dana zmienna została zadeklarowana wymaga w tym przypadku otwarcia dodatkowego pliku.

- Mimo to pliki nagłówkowe nadają się dobrze do przechowywania danych typu ***const static***.
  - Kwalifikator ***const*** chroni przed przypadkowymi modyfikacjami, a słowo ***static***, że każdy z plików dołączających plik nagłówkowy otrzyma własną kopię stałej.
  - Nie trzeba więc pamiętać o umieszczeniu w jednym pliku deklaracji definiującej, a we wszystkich pozostałych deklaracji nawiązujących (za pomocą słowa kluczowego ***extern***).

- Warto umieszczać w pliku nagłówkowym komentarz zawierający nazwę pliku nagłówkowego.

Łączenie wielu plików programu

- W jednym pliku programu można umieścić polecenie włączenia innego pliku z kodem źródłowym.
- Program zapisany jest fizycznie w kilku plikach, ale to co zobaczy kompilator po przetworzeniu przez preprocesor to tekst programu w całości. Dla kompilatora będzie to zatem jeden **moduł**, choć fizycznie zapisany jest w dwóch lub więcej plikach. Taki moduł zwany jest **jednostką translacji**.
- Oznacza to, że zmiana wprowadzona w jednym pliku będzie wymagała ponownej kompilacji całego programu a nie pojedynczej jednostki translacji.
- W ogólności program może składać się z wielu jednostek translacji, z których każda może być kompilowana osobno. Upraszcza to pisanie, analizowanie i pielęgnację kodu.
- Jednostki translacji mogą, i powinny jednocześnie stanowić podstawę podziału programu na jednostki logiczne.
- Każda jednostka translacji kompilowana jest niezależnie.

Łączenie wielu plików programu

- W C sprawdzane są typy zmiennych i poprawność wywołań funkcji. Każda funkcja, która jest wywoływana w danej jednostce translacji musi być w tej jednostce zadeklarowana.
- Definicja funkcji powinna być tylko jedna, umieszczona w jednej tylko jednostce translacji. Wszystkie deklaracje i definicja funkcji muszą być zgodne.

- Po połączeniu przez **linker** (konsolidator), wszystkie funkcje z różnych jednostek translacji „widzą” się nawzajem bez dodatkowych zabiegów.

- Nazwy funkcji globalnych należą do „wspólnego” zakresu złożonego z zakresów globalnych wszystkich modułów (mówimy, że są ***eksportowane***).
- Zmienne globalne nie podlegają uwspólnianiu. Zmienna globalna ***x*** z jednej jednostki translacji jest widoczna tylko wewnątrz tej jednostki; inny moduł może bezkonfliktowo zdefiniować zmienną globalną o tej samej nazwie i będą to dwie oddzielne zmienne, każda widoczna tylko w swoim module.
- Jeśli zmienną globalną chcemy eksportować, to należy ją zdefiniować w jednym tylko module, a w pozostałych jednostkach translacji, w których będziemy z niej korzystać, zadeklarować ją jako zmienną zewnętrzną za pomocą specyfikatora **extern**, np.:

```
extern double x;
```

Łączenie wielu plików programu – Przykład 1

- **Przykład:** Program zapisany jest fizycznie w kilku plikach, ale dla kompilatora jest to jeden moduł (jednostka translacji).

```
// kod zawarty w pliku Part1.cpp
double Recip(double x){
    if(x != 0)
        return 1.0/x;
    else
        return 0;
}

// kod zawarty w pliku Part2.cpp
int Power(int n) {
    if (n < 0 || n > 12)
        return 0;
    if (n == 0)
        return 1;
    else
        return n * Power(n-1);
}
```

Łączenie wielu plików programu – Przykład 1

```
// kod zawarty w pliku main.cpp

#include <stdio.h>
#include "Part1.cpp"
#include "Part2.cpp"

int main(int argc, char* argv[])
{
      int k;

      printf("Liczba calkowita : ");
      scanf("%d",&k);
      printf("\n%lf\t%d\n\n", Recip(k), Power(k));

      return 0;
}
```

Łączenie wielu plików programu – Przykład 2

- **Przykład:** Program zapisany jest fizycznie w kilku plikach i są to dla kompilatora osobne moduły (jednostki translacji), które będą kompilowane niezależnie.

```
//kod zawarty w pliku nagłówkowym Part1.h
double Recip(double x);

// kod zawarty w pliku Part1.cpp
#include "Part1.h"

double Recip(double x)
{
      if(x != 0)
            return 1.0/x;
      else
            return 0;
}
```

Łączenie wielu plików programu – Przykład 2

```
// kod zawarty w pliku nagłówkowym Part2.h
int Power(int n);

// kod zawarty w pliku Part2.cpp
#include "Part2.h"

int Power(int n)
{
      if (n < 0 || n > 12)
          return 0;

      if (n == 0)
          return 1;
      else
          return n * Power(n-1);
}
```

Łączenie wielu plików programu – Przykład 2

```
// kod zawarty w pliku main.cpp
#include <stdio.h>
#include "Part1.h"
#include "Part2.h"

int main(int argc, char* argv[])
{
	int k;

	printf("Liczba calkowita : ");
	scanf("%d", &k);
	printf("\n%lf\t%d\n\n", Recip(k), Power(k));

	return 0;
}
```

Zmienne - zasięg

- Lokalność, globalność, statyczność

Zakres i zasięg deklaracji

- **Zasięg zmiennych** jest bardzo istotnym zagadnieniem, ponieważ możemy czasem próbować odwołać się do zmiennej, która w rzeczywistości w danym miejscu nie istnieje.

- Przez **zasięg deklaracji** rozumiane są te fragmenty programu, w których obowiązuje deklaracja (jest ona znana, aktywna).

- Przez **zakres widoczności zadeklarowanej zmiennej** rozumiany jest obszar programu, w którym identyfikator może być użyty (nazwa identyfikatora związana jest z tą deklaracją). Zakres może być węższy od zasięgu deklaracji na skutek **przesłonięcia**.

- **Wyróżniamy:**
  - zakres globalny : cały program
  - zakres lokalny : definicja pojedynczej funkcji

Zakres i zasięg deklaracji

- Zmienne **lokalne** (prywatne, automatyczne) – są zadeklarowane wewnątrz funkcji.
  - Każda zmienna lokalna funkcji zaczyna istnieć dopiero w chwili wywołania funkcji i przestaje istnieć w momencie zakończenia przez funkcję działania.
  - Takie zmienne nie zachowują swoich wartości z jednego wywołania do następnego i muszą być jawnie określane od nowa przy każdym wejściu do funkcji.

- Zmienne deklarowane wewnątrz bloku (ograniczonego nawiasami klamrowymi) powinny być widoczne od miejsca wystąpienia deklaracji do końca bloku, w którym deklaracja wystąpiła (np. zmienne deklarowane w części inicjalizacyjnej pętli ***for***). Nie zawsze tak jest i lepiej takich deklaracji unikać.

- Zmienne **globalne** (zewnętrzne) – widoczne (i dostępne) dla dowolnej funkcji (miejsca) programu.
  - Zmienna **globalna** musi być zdefiniowana raz, na zewnątrz wszystkich funkcji.
  - Zmienna globalna widoczna jest od miejsca deklaracji do końca pliku.
  - Zadeklarowana musi być w każdej funkcji, która chce mieć do niej dostęp (poprzez specyfikator ***extern*** jeśli ma być widoczna w innych modułach (plikach) programu).

Zakres i zasięg deklaracji - Przykład

- **Przykład**

```
int i, j, k;
float X, Y;

int F1(int a, int b){
      char c1, c2;
      float B;
}

int F2(float Z1, float Z2, char cp) {
      int A1[15];
      long A2[15][15];
      float B1, B2, B3;
}

void main(void){
      int m, n, p, q;
      float V1, V2, V3;
      long T1[15][15], T2[15][15];
}
```

Zakres i zasięg deklaracji – Przykład cd.

- **Zakres i zasięg deklaracji w przykładzie:**
  - zakres globalny :

    `i, j, k, X, Y, F1, F2 (F1,F2 – funkcje)`

  - zakres lokalny w funkcji F1 :

    `a, b, c1, c2, B`

  - zakres lokalny w funkcji F2 :

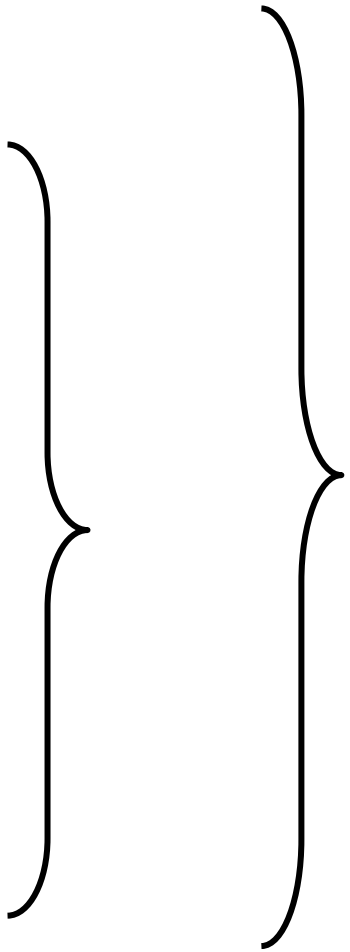
    `Z1, Z2, cp, A1, A2, B1, B2, B3`

  - zakres lokalny w funkcji main :

    `m, n, p, q, V1, V2, V3, T1, T2`

Zakres i zasięg deklaracji

```
{       int    i;
        ...
        {
               ...
               int    j;
               ...
               {
                      int n;
                      ...
               }
               for(int k=0;... )
               {  ... }
        }
}
```



Przesłanianie identyfikatorów

- Zmienne globalne można **przesłonić** jeżeli wewnątrz funkcji (lub bloku) zadeklarujemy inną zmienną (lokalną), o tej samej nazwie (nie musi być tego samego typu).

- W takiej sytuacji nazwa tej zmiennej w ciele funkcji odnosi się do zmiennej *lokalnej*. Zmienna globalna natomiast istnieje, ale jest w zakresie funkcji (bloku) niewidoczna.

- **Przykład:** deklaracja lokalna przesłania deklarację globalną

```
int i = 5;                          // i == 5
int F1 (int n){
      int i = 7;                    // i == 7, przesłanianie
      return i + n;
}
int F2 (int m)
{ return i + m; }                   // i == 5
int main(void){
      int k, z, i = 0;              // i == 0
      k = F1(0);                    // k == 7
      z = F2(0);                    // z == 5
      return 0;
}
```

Redefinicja identyfikatora w tym samym zakresie: błąd

- **Przykład:**

```
float eps = 0.001;
...
double eps = 0.05;      // błąd – redefinicja w zakresie globalnym
...
void main(void)
{
   long k1;
   ...
   int k1;              // błąd - redefinicja w  zakresie lokalnym
   ...
}
```

Dostęp do obiektów globalnych

- Do przesłoniętej zmiennej globalnej możemy się odwołać używając **operatora zasięgu** ' :: ' (,,czterokropek'').
- Jeśli przykładowo przesłonięta została zmienna o nazwie ***x***, to do globalnej zmiennej o tej nazwie można odwołać się poprzez nazwę kwalifikowaną ***::x***.

- **Przykład:**

```
const int max = 15750;    // zakres globalny
int MAX(int TAB[ ], int rozmiar)
{
   int max = TAB[0];      // zakres lokalny
   for (int i = 1 ; i < rozmiar ; ++i)
      if (::max > TAB[i] && max < TAB[i] ) //zakres globalny( :: ) & lokalny
          max = TAB[i];   // zakres lokalny
   return max;            // zakres lokalny
}
```

Zmienne statyczne – zakres globalny

- **Wyróżniamy zmienne statyczne:**
  - O zakresie globalnym
  - Lokalne poprzedzone słowem **static**

- Załóżmy, że mamy program podzielony na moduły i np. w pliku źródłowym *stack.h* zdefiniowano zmienne *sp* oraz *val*:

```
int sp;
double val[MAXVAL];
```

- Do tak zdefiniowanych zmiennych można odwołać się z innego modułu poprzez **extern** (daje dostęp do zmiennych globalnych innych plików).

- Jeśli z jakichś powodów jednak programista nie chce dopuścić do takiej sytuacji (tj. chce, aby pewne zmienne były globalne TYLKO w obrębie jednego pliku) musi przy definicji zmiennej użyć słowa **static**.

- Deklaracja **static** zastosowana do zmiennych globalnych i funkcji ogranicza więc ich zasięg od miejsca wystąpienia do końca tłumaczonego pliku źródłowego.

- Deklarowanie zewnętrznych obiektów jako **static** jest sposobem na ukrycie ich nazw.

Zmienne statyczne – zakres globalny

- Zewnętrzną deklarację **static** najczęściej stosuje się do zmiennych, ale można ją także stosować do funkcji.
- Nazwy funkcji są zwykle globalne, widoczne dla wszystkich części całego programu.
- Jeśli zadeklarujemy jednak funkcję jako **static**, to jej nazwa staje się niewidzialna poza plikiem zawierającym jej deklarację.
- Jeśli poniższe zmienne oraz funkcje zostaną umieszczone w jednym pliku źródłowym:

```
static char buff[SIZE];
static int indeks;
static void obliczaj(double a):
int inna(int a, int b);
```

  - To w takim wypadku zmienne *indeks* i *buff*, także funkcja *obliczaj* są statyczne – tj. nie są dostępne z innych plików programu.
  - Innymi słowy funkcję statyczną mogą wywołać tylko inne funkcje należące do tego samego pliku.
  - Nazwy zmiennych i funkcji statycznych jednego pliku mogą się powtarzać w innym pliku – wtedy są to (w tym innym pliku) zupełnie inne, niezależne elementy programu, które mogą robić coś zupełnie innego niż ich statyczne (i niewidoczne) odpowiedniki.

Zmienne statyczne – zakres lokalny

- Deklarację **static** można stosować do zmiennych wewnętrznych (lokalnych).
- Wewnętrzne zmienne statyczne są lokalne dla poszczególnych funkcji tak samo jak zmienne automatyczne. W przeciwieństwie do zmiennych automatycznych nie pojawiają się i nie znikają razem z wywołaniem funkcji, lecz istnieją między jej wywołaniami.
- Dzięki temu stanowią prywatną, stałą 'pamięć' wewnętrzną funkcji.
- **Przykład:**

```
int Licznik(void){
   static int ct = 0;
   return ++ct;
}
int numer;
numer = Licznik();            // numer == 1
...................
numer = Licznik();            // numer == 2
...................
numer = Licznik();            // numer == 3
```

Zmienne rejestrowe

- Deklaracja **register** mówi kompilatorowi, że zmienna będzie intensywnie wykorzystywana.
- W celu zmniejszenia i przyspieszenia czasu wykonywania programu taką zmienną chcielibyśmy umieścić w rejestrach maszyny.

- **Przykładowa deklaracja:**

```
register int i;
register char c;
```

- W praktyce kompilator może zignorować taką wskazówkę i potraktować taką zmienną jak każdą inną.
- Także, jeśli będzie zbyt wiele takich zmiennych, skończą się wolne rejestry procesora i następne zmienne ze słowem register nie staną się zmiennymi rejestrowymi. Które i kiedy – nie wiadomo, więc z praktycznego punktu widzenia polecenie register nie ma współcześnie większego zastosowania.
- Nie ma możliwości uzyskania adresu zmiennej rejestrowej niezależnie od tego, czy zmienną rzeczywiście umieszczono w rejestrze, czy też nie.
- Szczegółowe ograniczenia dotyczące liczby i typów zmiennych rejestrowych są zależne od maszyny.

Pytania?