Język ANSI C

część 15

struktury rekurencyjne i ich zastosowania
listy

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- struktury i listy
  - struktury mogą zawierać w sobie definicje "rekurencyjne" czyli wskaźniki do siebie samych
  - dzięki tej właściwości możliwe jest tworzenie takich złożonych struktur danych jak listy lub drzewa
  - listy i drzewa stosuje się w przypadkach gdy z góry nie wiadomo ile będzie trzeba przechowywać danych
  - przykład: policzyć ile razy występuje każde słowo w danym pliku tekstowym. Właściwa struktura danych: drzewo binarne
  - szczegóły budowy konkretnej listy / drzewa zależą od jej zastosowania

```
// lista 1-kierunkowa
typedef struct linked_list {
  struct linked_list *next;   // następna pozycja na liście
  char   name[30];            // dane przechowywane na liście
  int    age;                 // dane przechowywane na liście
} LIST;
```

`typedef` bardzo ułatwi zapisy

```
// lista 2-kierunkowa
typedef struct double_list {
  struct double_list *next;   // w przód
  struct double_list *prev;   // w tył
  int    age;                 // dane przechowywane na liście
} DLIST;
```

- struktury i listy

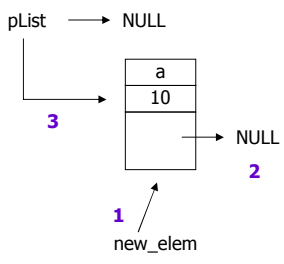
```
// dodawanie elementów do listy 1-kierunkowej
void  add_elem (char *_name, int _age);
LIST *add_elem2 (LIST *, char *_name, int _age);
int   add_elem3 (LIST *, char *_name, int _age);
int   add_elem4 (LIST **, char *_name, int _age);

// różne operacje na liście 1-kierunkowej
int   find (LIST *, char *_name);
int   del_elem (LIST *, int _age);
void  free_list (LIST *);
void  printList (LIST *);
void  printDList (DLIST *);
void  doTask (LIST *, void (*fun) (int*, int) );  // wykonaj fun() na elementach listy
void  elemFun (int *, int);

// sortowanie listy
int compare (const void *, const void *);         // postać wymagana przez qsort()
void  sortList (LIST *);                          // z użyciem qsort()

// dodawanie elementów do listy 2-kierunkowej
void  add_Delem (DLIST *, int _age);
```

– UWAGA: w rzeczywistości projekt powinien składać się np. z 3 plików:
  `lista.h` (prototypy, deklaracje struktur, zmienne globalne)
  `lista.c` (definicje funkcji)
  `main.c` (program główny)
  \+ "sterowanie" całością odpowiednimi dyrektywami preprocesora `(#ifndef, #endif, ...)`

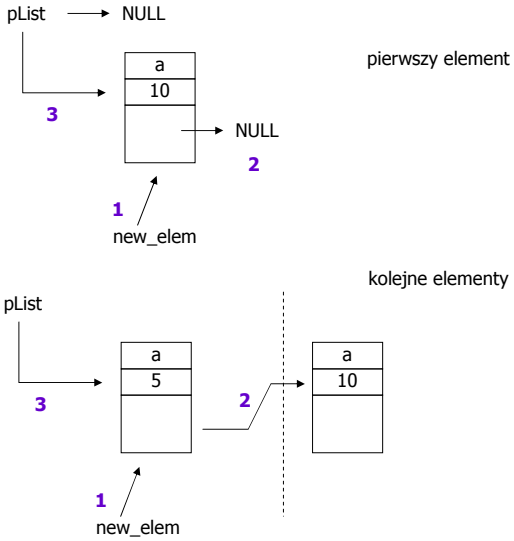
- struktury i listy

```
/* zmienna globalna */
LIST *pList = NULL;

int main (void)
{
  add_elem( "a", 10 );                  // 10
  add_elem( "a", 5 );                   // 5 10
  ...
```

```
// dodawanie elementu na początek listy
// pList zmienna globalna, źle !!!

void add_elem (char *_name, int _age)
{
  LIST *new_elem;
  new_elem = (LIST *) malloc ( sizeof(LIST) );    // 1
  strcpy ( new_elem->name, _name);
  new_elem->age = _age;
  new_elem->next = pList;                         // 2
  pList = new_elem;                               // 3
}
```



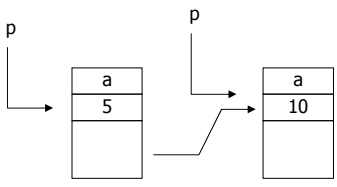
- struktury i listy



- struktury i listy

```
/* zmienna globalna */
LIST *pList = NULL;

int main (void)
{
  add_elem( "a", 10 );
  add_elem( "a", 5 );
  printList(pList);
```

```
void printList ( LIST *p )
{
  while( p )
  {
   printf("(%s %d)", p->name, p->age));
   p = p->next;                // działamy na kopii
  }                            // wskaźnika
   printf("\n");
}
```



- struktury i listy

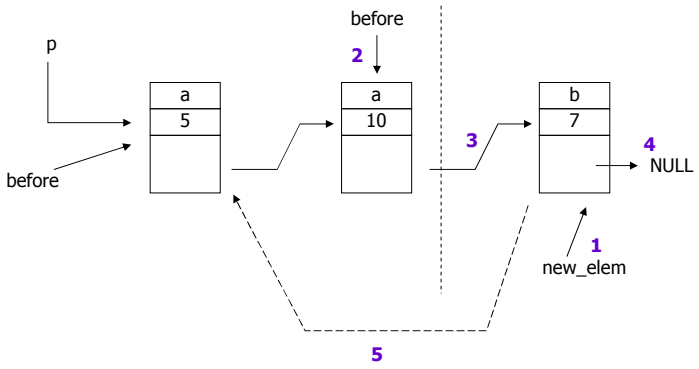
```
/* zmienna globalna */
LIST *pList = NULL;

int main (void)
{
  add_elem( "a", 10 );
  add_elem( "a", 5 );
  printList(pList);

  // pierwsze wywołanie musi być z podstawieniem pLista = ...
  // gdy startujemy od pustej listy (w przykładzie poniżej nie jest to prawdą)
  // gdy lista już istnieje, to podstawienie pLista = ... jest zbędne

  pList = add_elem2( pList, "b", 7);              // 5 10 7
  add_elem2( pList, "b", 20 );                    // 5 10 15 20
  add_elem2( pList, "b", 17 );                    // 5 10 15 20 17
  printList( pList);
```


- struktury i listy

```
// dodawanie elementu na koniec listy

LIST *add_elem2 (LIST *p, char *_name, int _age)
{
  LIST *new_elem;
  LIST *before = p;

  if (p == NULL) {                                // gdy pierwszy element
     p = (LIST *) malloc ( sizeof(LIST) );
     strcpy ( p->name, _name);
     p->age = _age;
     p->next = NULL;
  }
  else {
     new_elem = (LIST *) malloc ( sizeof(LIST) ); // 1
     strcpy ( new_elem->name, _name);
     new_elem->age = _age;
     while( before->next )                        // znajdz ostatni element
       before = before->next;                     // 2

     before->next = new_elem;                     // 3
     new_elem->next = NULL;                       // 4 ( przyda się w del_elem() ! )
     // new_elem->next = p;                       // 5, lista cykliczna
  }
  return p;
}
```

bez `pLista = ...` (patrz poprzedni slajd) dla każdego wstawianego elementu wejdziemy tutaj. Powód: funkcja NIE modyfikuje wartości wskaźnika `p`

- struktury i listy



- struktury i listy

```
/* zmienna globalna */
LIST *pList = NULL;

int main (void)
{
  add_elem( "a", 10 );
  add_elem( "a", 5 );
  printList(pList);
  // pierwsze wywołanie musi być takie gdy startujemy od pustej listy
  // inaczej lista (z chociaż jednym elementem) musi już istnieć
  // gdy lista już istnieje, to podstawienie pLista = ... jest zbędne

  pList = add_elem2( pList, "b", 7);              // 5 10 7
  add_elem2( pList, "b", 20 );                    // 5 10 15 20
  add_elem2( pList, "b", 17 );                    // 5 10 15 20 17
  printList( pList);
  del_elem( pList, 20 );
```

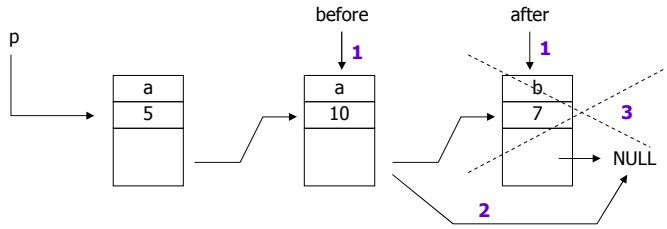
- struktury i listy

```
// kasuje dana z listy uporządkowanej
// nie mozna usunać pierwszego elementu
// usuwany element musi znajdować się na liście ! źle

int del_elem(LIST *p, int _age)
{
  LIST *before = p;
  LIST *after = p->next;

   // przegladanie listy
   while( after != NULL && after->age != _age)
   {
     before = before->next;             // 1
     after = after->next;               // 1

   }

   before->next = after->next;          // 2 tu błąd gdy kasowanego elem. nie ma na liście
   free(after);                         // 3

   return (after != NULL);              // czy skasowano cos?
}
```

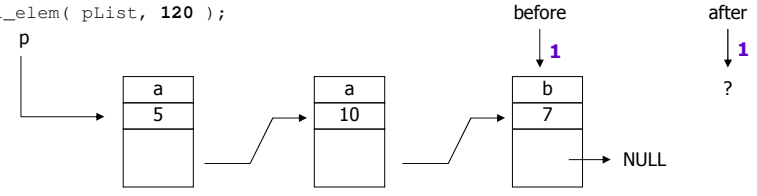
za chwilę taką listę będziemy budowali

- struktury i listy

```
del_elem( pList, 10 );
```




```
del_elem( pList, 120 );
```



2 błąd pojawi się w tej linii:

```
before->next = after->next;
```

- struktury i listy

```
// dodawanie elementu w sposób uporządkowany (wg. pola age)
// lista musi już istnieć
// ma sens tylko dla listy uporządkowanej
// nie wstawi elementu mniejszego niż najmniejszy na liście
// (gdyż należałoby zmodyfikować korzeń, tu: p)
// dopuszcza wstawianie duplikatów
```

```
/* zmienna globalna */
LIST *pList = NULL;

int main (void)
{
  add_elem( "a", 10 );
  add_elem( "a", 5 );
  printList(pList);
  // pierwsze wywołanie musi być takie gdy startujemy od pustej listy
  // inaczej lista (z chociaż jednym elementem) musi już istnieć
  // gdy lista już istnieje, to podstawienie pLista = ... jest zbędne

  pList = add_elem2( pList, "b", 7);              // 5 10 7
  add_elem2( pList, "b", 20 );                    // 5 10 7 20
  add_elem2( pList, "b", 17 );                    // 5 10 7 20 17
  printList( pList);
  // del_elem( pList, 20 );

  add_elem3( pList, "c", 25 );                    // 5 10 7 20 17 25
  add_elem3( pList, "c", 7 );                     // 5 7 10 7 20 17 25
  add_elem3( pList, "c", 15 );                    // 5 7 10 7 15 20 17 25
  printList( pList);
```

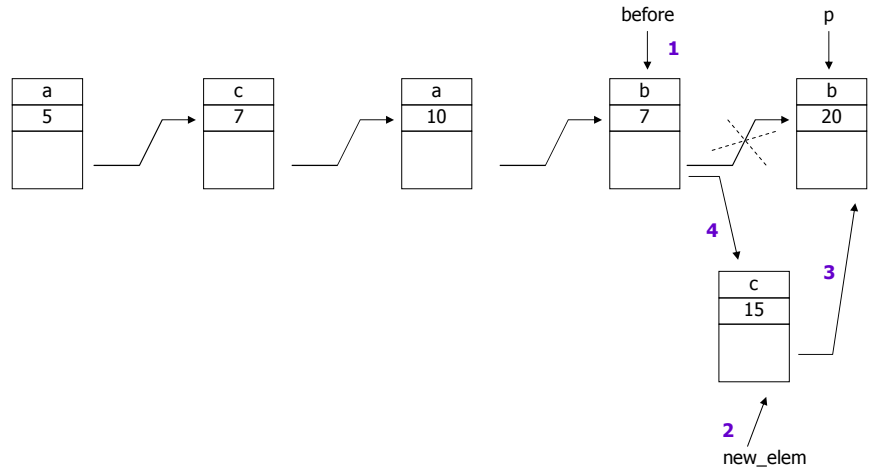
efekt wstępnego nieuporządkowania listy

- struktury i listy

```
int add_elem3 (LIST *p, char *_name, int _age)
{
  LIST *new_elem;
  LIST *before = p;

   // przeglądanie listy
   while( p != NULL && (p->age < _age)) {
     before = p;
     p = p->next; // modyfikacja wskaźnika p OK, działamy na kopii wskaźnika
   }                                              // 1, 5

   //  nowy węzeł
   new_elem = (LIST *) malloc ( sizeof(LIST) );   // 2, 6
   if ( new_elem == NULL ) return FALSE;
   strcpy ( new_elem->name, _name);
   new_elem->age = _age;

   // wstawienie nowego węzła do listy
   new_elem->next = p;                            // 3, 7
   before->next = new_elem;                       // 4, 8 ???

   return TRUE;
}
```

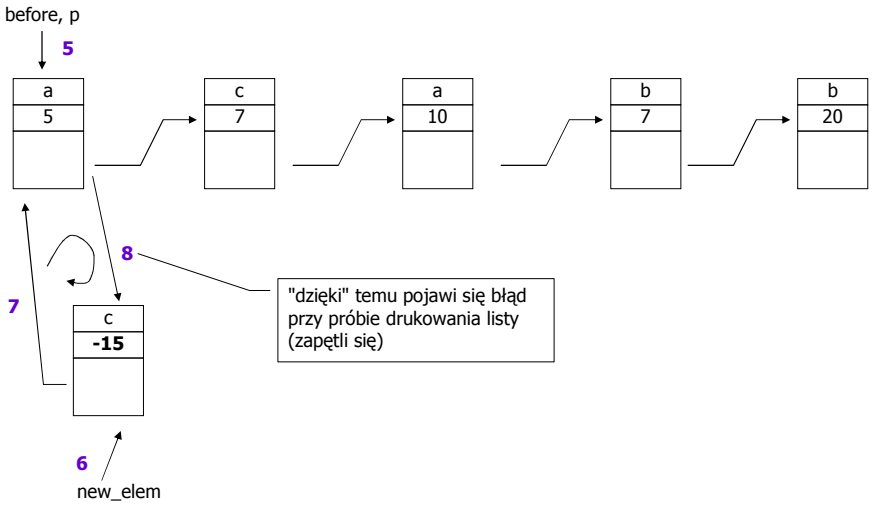
- struktury i listy

```
add_elem3( pList, "c", 15 );
```



- struktury i listy

```
add_elem3( pList, "c", -15 );
```



- struktury i listy

```
...

add_elem3( pList, "c", 25 );                    // 5 10 7 20 17 25
add_elem3( pList, "c", 7 );                     // 5 7 10 7 20 17 25
add_elem3( pList, "c", 15 );                    // 5 7 10 7 15 20 17 25
printList( pList);

del_elem( pList, 15 );
del_elem( pList, 25 );       // kasowany element musi być na liście
printList( pList );

add_elem4( &pList, "d", -1 );
add_elem4( &pList, "d", 1 );
add_elem4( &pList, "d", -7 );
add_elem4( &pList, "d", 20 );    // duplikat
printList( pList );
```

```
// dodawanie elementu w sposób uporządkowany (wg. pola age)
// lista musi już istnieć
// ma sens tylko dla listy uporządkowanej
// dopuszcza wstawianie duplikatów
// gdy wstawiamy mniejszy element niż już jakikolwiek na liście
// to modyfikuje korzeń (**p)
```

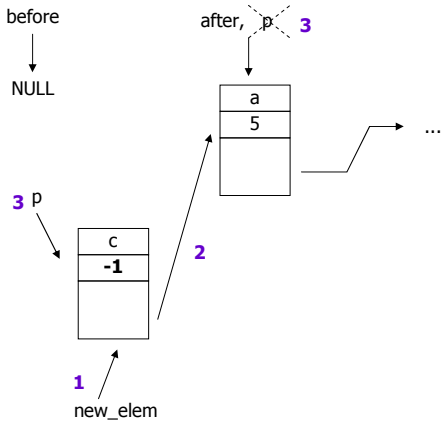
inny sposób to zwrócenie (zmodyfikowanego) wskaźnika z funkcji instrukcją `return`.
Porównaj z funkcją `add_elem2()`

- struktury i listy

```
int add_elem4 (LIST **p, char *_name, int _age)
{
  LIST *new_elem;   LIST *before;   LIST *after;

  after = *p;
  before = NULL;                                  // ważny NULL, patrz niżej

   // przeglądanie listy
   while( after !=  NULL && (after->age < _age)) {
     before = after;
     after = after->next;
   }
   //  nowy wezel
   new_elem = (LIST *) malloc ( sizeof(LIST) );   // 1
   if ( new_elem == NULL ) return FALSE;
   strcpy ( new_elem->name, _name);
   new_elem->age = _age;

   // wstwaienie nowego wezla do listy
   new_elem->next = after;                        // 2
   if ( before == NULL )                          // czy dołączyć na początek ?
     *p = new_elem;                               // jeśli tak, to modyfikuj korzeń, // 3
   else
     before->next = new_elem;                     // dołączanie nie na początek listy
   return TRUE;
}
```

- struktury i listy

```
add_elem4( &pList, "d", -1 );
```



- struktury i listy

```
// szuka napisu na liście

int find(LIST *p, char *_name)
{
  while ( p != NULL ) {
    if ( strcmp( p->name, _name ) == 0 )
      break;
      p = p->next;
  }
   return ( p != NULL );
}
```

```
void free_list(LIST *p)
{
  LIST *before;     // konieczna zmienna pomocnicza. Nie można wykonać free() elementu
                    // z którego danych jeszcze chcemy skorzystać !
   while( p ) {
     before = p;
     p = p->next;
     free ( before );
   }
}
```

- struktury i listy

```
/* zmienna globalna */
LIST *pList = NULL;

int main (void)
{
  add_elem( "a", 10 );
  add_elem( "a", 5 );
  printList(pList);
  // pierwsze wywołanie musi być takie gdy startujemy od pustej listy
  // inaczej lista (z chociaż jednym elementem) musi już istnieć
  // gdy lista już istnieje, to podstawienie pLista = ... jest zbędne

  pList = add_elem2( pList, "b", 7);              // 5 10 7
  add_elem2( pList, "b", 20 );                    // 5 10 7 20
  add_elem2( pList, "b", 17 );                    // 5 10 7 20 17
  printList( pList);
  // del_elem( pList, 20 );

  add_elem3( pList, "c", 25 );                    // 5 10 7 20 17 25
  add_elem3( pList, "c", 7 );                     // 5 7 10 7 20 17 25
  add_elem3( pList, "c", 15 );                    // 5 7 10 7 15 20 17 25
  printList( pList);

  doTask(pList, elemFun);                         // elemFun - wskaźnik do funkcji
  printList( pList );
```


- struktury i listy

```
// do każdego elementu listy dodaj 5 + długość pola name

void doTask (LIST *p, void (*fun)(int*, int) )
{
  for (p; p; p=p->next )
    fun ( &(p->age), strlen(p->name) );           // modyfikacja każdego elementu listy
    // (*fun)( &(p->age), strlen(p->name) );      // lub tak


void elemFun (int *a, int b)
{
  *a = *a + b + 5;
}
```

- struktury i listy

```
...

add_elem3( pList, "c", 25 );                    // 5 10 7 20 17 25
add_elem3( pList, "c", 7 );                     // 5 7 10 7 20 17 25
add_elem3( pList, "c", 15 );                    // 5 7 10 7 15 20 17 25
printList( pList);

del_elem( pList, 15 );
del_elem( pList, 25 );       // kasowany element musi być na liście
printList( pList );

add_elem4( &pList, "d", -1 );
add_elem4( &pList, "d", 1 );
add_elem4( &pList, "d", -7 );
add_elem4( &pList, "d", 20 );    // duplikat
printList( pList );

printf("compare: %d and %d: %d\n", pList->age, pList->next->age,
                                   compare( pList, pList->next) );
printf("compare: %d and %d: %d\n", pList->next->next->age, pList->age,
                                   compare( pList->next->next, pList) );
```


- struktury i listy

```
// wersja odpowiednia dla qsort()

int compare ( const void *p1, const void *p2 )
{
  LIST *t1 = (LIST*)p1;       // OK
  LIST *t2 = (LIST*)p2;       // OK

  // dla:
  // LIST *t1 = p1;
  // LIST *t2 = p2;
  //"[Warning] initialization discards qualifiers from pointer target type"

  return ( (int*)(t1->age) - (int*)(t2->age) );
}
```

```
// źle (błąd kompilatora)
// "request for member `age' in something not a structure or union"

int compare2(const void *p1, const void *p2)
{
  return ( (int*)(LIST*)(p1->age) - (int*)(LIST*)(p2->age) );
}
```

- struktury i listy

```
DLIST firstDList = { NULL, NULL, -1 };          // pierwszy element
DLIST *pDList = &firstDList;
...


add_elem4( &pList, "d", -1 );
add_elem4( &pList, "d", 1 );
add_elem4( &pList, "d", -7 );
add_elem4( &pList, "d", 20 );    // duplikat
printList( pList );

printf("compare: %d and %d: %d\n", pList->age, pList->next->age,
                                   compare( pList, pList->next) );
printf("compare: %d and %d: %d\n", pList->next->next->age, pList->age,
                                   compare( pList->next->next, pList) );


add_Delem( pDList, 5 );
add_Delem( pDList, 2 );
printDList( pDList );
```

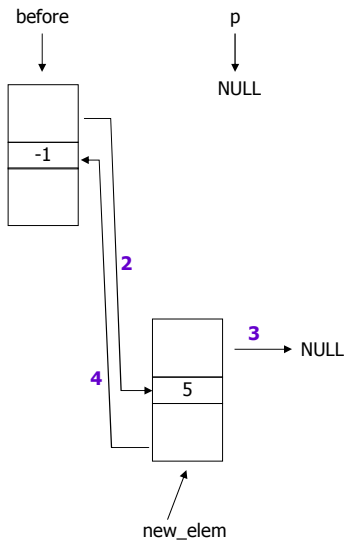
```
// lista 2-kierunkowa
typedef struct double_list {
  struct double_list *next;   // w przód
  struct double_list *prev;   // w tył
  int    age;                 // dane przechowywane na liście
} DLIST;
```


- struktury i listy

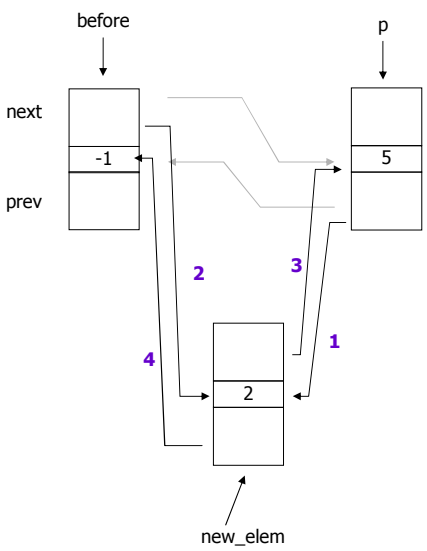
```
// wstawianie do listy 2-kierunkowej w sposób uporządkowany
// lista nie może być pusta
// nie wstawia duplikatów

void add_Delem (DLIST *p, int _age)
{
  DLIST *new_elem;
  DLIST *before;

   // przeglądanie listy
   while( p != NULL && (p->age < _age)) {
     before = p;
     p = p->next;
   }

  new_elem = (DLIST *) malloc ( sizeof(DLIST) );
  new_elem->age = _age;

  // ustaw wskaźniki
  if ( p )                    // obsługa przypadku, gdy jeden element na liście
    p->prev = new_elem;       // 1
  before->next = new_elem;    // 2
  new_elem->next = p;         // 3
  new_elem->prev = before;    // 4
}
```

- struktury i listy

```
add_Delem( pDList, 5 );
```




```
add_Delem( pDList, 2 );
```



- struktury i listy

```
void printDList ( DLIST *p )
{
    DLIST *tmp;
    int count = 0;

    // od przodu, korzystając z pola next
    while( p ) {
      tmp = p;                // tmp podąża za p
      printf("%d ", p->age);
      p = p->next;
      count++;
    }
    printf("\n");

    // od tyłu, korzystając z pola prev
    while ( count-- ) {
      printf("%d ", tmp->age);
      tmp = tmp->prev;
    }
    printf("\n\n");
}
```