

ISO/ANSI C

Klasy pamięci

111

ISO/ANSI C – klasy pamięci

Zmienne mogą być definiowane z modyfikatorami: `auto`, `extern`, `static`

- **auto** – zmienna lokalna istniejąca tylko w obrębie bloku: tworzona w momencie gdy sterowanie wchodzi do danego bloku i usuwana gdy opuszcza (*wszystkie zmienne zadeklarowane w obrębie bloku są z definicji lokalne, dlatego tego modyfikatora się nie używa*).
- **extern** – deklaracja zmiennej zdefiniowanej w innym pliku (dla kodu programu, który jest podzielony na pliki)
- **static** – zmienne statyczne, istniejące w jednym egzemplarzu

© UKSW, WMP, SNS, Warszawa

112

ISO/ANSI C – klasy pamięci

Czym różnią się zmienne `static` od globalnych?

Zmienne `static` są widziane tylko w module/pliku, w którym zostały zadeklarowane; nie można się do nich odwołać z innego pliku za pomocą `extern`

Zmienna `static` zadeklarowana w bloku nie jest usuwana po wyjściu z tego bloku. Po powrocie sterowania do tego bloku wartość zmiennej `static` jest zachowana

© UKSW, WMP, SNS, Warszawa

113

ISO/ANSI C – klasy pamięci

zmienne `volatile`

może ulec zmianie „bez wiedzy” programu, np. ponieważ reprezentuje obszar danych będący pod kontrolą interfejsu do urządzenia zewnętrznego, lub gdy programujemy wielowątkowo i zmienna pozwala na wymianę informacji między wątkami

Kompilator traktują ją jako „niepewną”, tj. przed każdym użyciem odczytuje jej wartość z pamięci (a nie np. z cache’u), a każda modyfikacja musi zostać wykonana przed przystąpieniem do wykonania następných instrukcji (żadnego przyspieszania wykonania programu przy pomocy „potokowego” wykonania instrukcji przez zaawansowany procesor, itp.)

© UKSW, WMP, SNS, Warszawa

114

ISO/ANSI C – klasy pamięci

zmienne `const`

wartość zmiennej nie może ulec zmianie po jej utworzeniu – jest inicjalizowana w momencie jej definiowania

Komentarz na marginesie:
modyfikator `const` wykorzystywany jest również przy deklarowaniu argumentów funkcji dla zagwarantowania nietykliwości zmiennych podawanych w argumentach wykonania, np.:

```
char *strcpy( char *strDestination,
              const char *strSource );
```

Uwaga: typ `'char*'` to nie to samo co `'const char*'`

© UKSW, WMP, SNS, Warszawa

115

Pamięć programu

Pamięć jest podzielona na cztery segmenty:
dane, sterta, stos i kod

- zmienne globalne – dane
- zmienne statyczne – dane
- zmienne `const` – kod i/lub dane
- zmienne lokalne (zadeklarowane i zdefiniowane w funkcjach) – stos
- zmienne zadeklarowane i zdefiniowane w `main` – stos
- wskaźniki – dane lub stos, w zależności od kontekstu
- pamięć alokowana dynamicznie – sterta

© UKSW, WMP, SNS, Warszawa

116



ISO/ANSI C

Zmienne dynamiczne

117

ISO/ANSI C – zm. dynamiczne

- Zmienne deklarowane w kodzie programu są tworzone:
 - w momencie uruchomienia programu (zmienne globalne), lub
 - w momencie wejścia sterowania do bloku instrukcji, w którym zostały zadeklarowane (zmienne lokalne)
- Zmienne te są identyfikowane przez nazwy
- Programista musi przewidzieć, ile zmiennych będzie potrzebował dla swojego algorytmu

A co robić, jeżeli zdarzy się, że programista nie jest w stanie tego przewidzieć?

© UKSW, WMP, SNS, Warszawa

118

ISO/ANSI C – zm. dynamiczne

Sytuacje, kiedy nie wiemy nic pewnego o rozmiarze zbioru danych, na którym będziemy operować:

- praca na danych z pliku (rozmiar pliku nie jest znany)
- wczytywanie danych, co do których nie znamy ograniczenia na ilość (wprowadzanie faktur, produktów do cennika)
- odbiór danych z interfejsu zewnętrznego (billingi telefoniczne, w medycynie: pomiary ciągłe stanu pacjenta – temperatura, tętno, itp.)

Algorytm musi być przygotowany na to, że dopiero w trakcie działania dowie się, jaki jest rozmiar danych roboczych i musi umieć odpowiednio do tego rozmiaru na bieżąco przygotować struktury danych na jego przyjęcie

© UKSW, WMP, SNS, Warszawa

119

ISO/ANSI C – zm. dynamiczne

Dynamiczne dostosowanie się programu do danych wejściowych jest możliwe z pomocą zmiennych dynamicznych.

Zmienne dynamiczne są tworzone w trakcie działania programu za pomocą polecenia alokacji danych. Ich rozmiar może być określony dopiero czasie działania programu (nie musi być wcześniej znany programiście), a po ich wykorzystaniu, zaalokowany obszar pamięci można zwolnić i z powrotem oddać do dyspozycji systemowi.

Alokowanie zmiennej dynamicznej (<stdlib.h> i <malloc.h>):

```
void* malloc( size_t size );
```

Funkcja zwraca adres do obszaru pamięci o rozmiarze 'size'.

Zwracany typ `void*` pozwala na zapisanie wyniku do zmiennej wskaźnikowej dowolnego typu.

© UKSW, WMP, SNS, Warszawa

120

ISO/ANSI C – zm. dynamiczne

Przykład:

```
char *string1 = malloc( 80 );
if( string1 == NULL )
    printf("Brak wystarczającej ilości pamięci\n");
else
    printf("Zaalokowano zmienną dynamiczną\n");
```

Aby zagwarantować przenośność kodu, zamiast samemu obliczać całkowity rozmiar pamięci, lepiej wykorzystać funkcję `sizeof`:

```
string1 = malloc( 80 * sizeof(char) );
```

© UKSW, WMP, SNS, Warszawa

121

ISO/ANSI C – zm. dynamiczne

Usuwanie zaalokowanych zmiennych

Kiedy zmienna przestaje być potrzebna zaalokowane zasoby pamięci należy zwolnić.

```
void free( void *mемblock );
```

Argumentem wywołania jest wskaźnik na zmienną.

Zapominanie o zwalnianiu zaalokowanych zasobów jest typowym błędem, powoduje, że w trakcie pracy aplikacji zużywa ona coraz więcej i więcej zasobów. Takie zjawisko jest nazywane wyciekaniem pamięci.

© UKSW, WMP, SNS, Warszawa

122

ISO/ANSI C – zm. dynamiczne

Usuwanie zaalokowanych zmiennych

Przykład:

```
char *string1 = malloc( 80 );
if( string1 == NULL )
    printf( „Brak wystarczającej ilości pamięci\n” );
else {
    printf( „Zaalokowano zmienną dynamiczną\n” );
    ... /* kod programu wykorzystujący zaalokowaną zmienną */
    free( string1 );
}
```

Rozmiar zaalokowanej pamięci jest zawarty w dodatkowym, niewielkim nagłówku dołączonym do obszaru zaalokowanego przez `malloc`. Dlatego funkcja `free` nie musi rozpoznawać rozmiaru pamięci do zwolnienia po typie wskaźnika.

© UKSW, WMP, SNS, Warszawa

123

123

ISO/ANSI C – zm. dynamiczne

Zgubienie wskaźnika do zmiennej dynamicznej powoduje całkowitą utratę kontroli nad tą zmienną, ale nie zwalnia zasobów.

Przykład z tablicami dynamicznymi:

```
double * t1 = malloc(100*sizeof(double));
double * t2 = malloc(100*sizeof(double));
/* zamieniamy się tablicami */
t1 = t2; t2 = t1;
/* Bez sensu. Tak się nie robi .. */

/* prawidłowo zamieniamy się tablicami */
double *t3; /* tworzymy zmienną pomocniczą 't3' */
t3 = t1; t1 = t2; t2 = t3; /* zamieniamy */
```

© UKSW, WMP, SNS, Warszawa

124

124

ISO/ANSI C – zm. dynamiczne

Dwuwymiarowe tablice dynamiczne

Alokowanie:

```
int i;
double **t1 = malloc(100*sizeof(double*));
for (i=0; i<100; i++)
    t1[i] = malloc(100*sizeof(double));
```

Zwalnianie:

```
int i;
for (i=0; i<100; i++)
    free( t1[i] );
free( t1 );
```

© UKSW, WMP, SNS, Warszawa

125

125

ISO/ANSI C – zm. dynamiczne

Alokowanie i inicjalizacja zerami

```
void *calloc( size_t num, size_t size );
```

```
long *buffer;
buffer = calloc( 40, sizeof( long ) );
if (buffer == NULL) {
    printf( "Can't allocate memory\n" );
    exit(1);
}
... /* działania na zmiennej 'buffer' */
free( buffer );
```

© UKSW, WMP, SNS, Warszawa

126

126

ISO/ANSI C – zm. dynamiczne

Zmiana rozmiaru zaalokowanego bloku

```
void *realloc( void *mемblock, size_t size );
```

```
long *buffer;
buffer = malloc( 1000 * sizeof( long ) );
if (buffer == NULL) exit( 1 );
/* Realokacja */
buffer = realloc( buffer, 2000 * sizeof( long ) );
if (buffer == NULL) exit( 1 );
```

Zwraca adres do tego samego bloku pamięci, jeżeli udało się go rozszerzyć. Zwraca `NULL` jeżeli podano `size=0` a `mемblock!=NULL`, bądź jeżeli nie udało się rozszerzyć zaalokowanego bloku do żądanych rozmiarów. Jeżeli podano `mемblock=NULL`, działa jak zwykły `malloc`.

© UKSW, WMP, SNS, Warszawa

127

127

ISO/ANSI C – zm. dynamiczne

Funkcje operujące na pamięci <string.h>

```
void *memcpy( void *dest, const void *src,
              size_t count );
void *memmove( void *dest, const void *src,
               size_t count );
```

Kopiuje z obszaru pamięci wskazanego przez `src` liczbę `count` bajtów do obszaru pamięci wskazanego przez `dest`

- `memcpy` – szybsze (różnica widoczna przy wielkich rozmiarach danych kopiowanych po wielokroć)
- `memmove` – zabezpiecza poprawne działanie w przypadku, kiedy wskazane obszary pamięci mogą się częściowo pokrywać

© UKSW, WMP, SNS, Warszawa

128

128

ISO/ANSI C – zm. dynamiczne

Funkcje operujące na pamięci <string.h>

```
void *memcpy( void *dest, const void *src,
              size_t count );
void *memmove( void *dest, const void *src,
              size_t count );
```

Przykład:

```
char tab[] = "Lorem ipsum dolor sit amet, consectetur
adipiscing elit";
char*ptab = calloc(100,sizeof(char));
int i = sizeof(tab); /* zwróci rozmiar całej tablicy */

memcpy(ptab,tab, i); /* tak jest szybciej: */
memmove(tab+5,tab, i); /* tak bezpieczniej: */
```

© UKSW, WMP, SNS, Warszawa

129

129

ISO/ANSI C – zm. dynamiczne

Dygresja: działanie funkcji `sizeof ()` dla tablic i wskaźników

- Zwraca rozmiar całej tablicy (w bajtach) tylko jeżeli kompilator jednoznacznie identyfikuje argument jako tablicę:

```
char tab[] = "Zakopane na pokaz";
int i = sizeof(tab); // i = 18 (tablica rozpoznana!)
long double *ptab = calloc(100, sizeof(long double));
ptab[0] = sizeof(ptab); // ptab[0] = 4
ptab[1] = sizeof(*ptab); // ptab[1] = 8
```

- Jeżeli tablicę prześlemy jako argument wywołania funkcji np.: `void fun(char *w)` tj. zostanie przekazana za pomocą argumentu typu wskaźnik, to `sizeof(w)` wywołany w kodzie funkcji zwróci rozmiar zmiennej wskaźnikowej

© UKSW, WMP, SNS, Warszawa

130

130

ISO/ANSI C – zm. dynamiczne

Funkcje operujące na pamięci <string.h>

Porównanie znaków w dwóch buforach:

```
int memcmp( const void *buf1, const void *buf2,
            size_t count );

char first[] = "12345678901234567890";
char second[] = "12345678901234567891";
int result = memcmp( first, second, 19 );
if( result < 0 )
    printf( „Pierwszy jest mniejszy niż drugi.\n” );
else if( result == 0 )
    printf( „Pierwszy i drugi są równe.\n” );
else
    printf( „Pierwszy jest większy niż drugi.\n” );
```

© UKSW, WMP, SNS, Warszawa

131

131

ISO/ANSI C – zm. dynamiczne

Funkcje operujące na pamięci <string.h>

Wypełnienie bufora znakami:

```
void *memset( void *dest, int c, size_t count );

char buffer[] = „To tylko taki mały test memset”;
printf( „Przed: %s\n”, buffer );
memset( buffer, '*', 14 );
printf( „Po: %s\n”, buffer );

Przed: To tylko taki mały test memset
Po: *****mały test memset
```

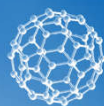
© UKSW, WMP, SNS, Warszawa

132

132

ISO/ANSI C

Zmienne dynamiczne:
Rekurencyjne typy danych

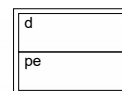


133

ISO/ANSI C – zm. dynamiczne

Rekurencyjne typy danych

```
struct element {
    double d;
    struct element * pe;
}
```



© UKSW, WMP, SNS, Warszawa

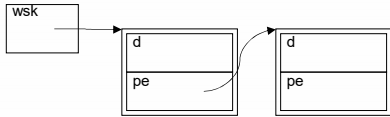
134

134

ISO/ANSI C – zm. dynamiczne

Rekurencyjne typy danych

```
struct element {
    double d;
    struct element * pe;
};
struct element *wsk = malloc(sizeof(struct element));
wsk->pe = malloc(sizeof(struct element));
```



© UKSW, WMP, SNS, Warszawa

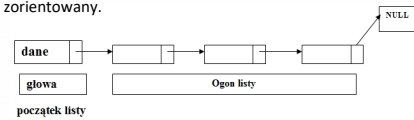
135

135

ISO/ANSI C – zm. dynamiczne

- Wskaźników używa się np. do budowy grafów skończonych, drzew, list (jedno- i dwukierunkowych) i do manipulacji na nich.
- Poniżej pokazany jest przykład listy interpretowanej jako graf liniowy zorientowany.

```
struct element {
    struct Dane_t dane;
    struct element * pe;
}
```



© UKSW, WMP, SNS, Warszawa

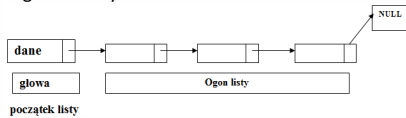
136

136

ISO/ANSI C – zm. dynamiczne

Lista jest szczególnym przypadkiem rekurencyjnego typu danych, w którym przyjmuje się, że:

- zbiór pusty wierzchołków jest listą,
- jeśli h jest wierzchołkiem listy, a t jest listą, to para uporządkowana (h,t) jest listą, w której h jest nazywana głową listy, a t ogonem listy.



© UKSW, WMP, SNS, Warszawa

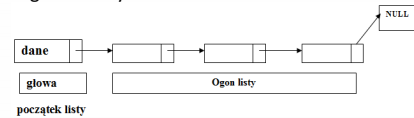
137

137

ISO/ANSI C – zm. dynamiczne

Lista jest szczególnym przypadkiem rekurencyjnego typu danych, w którym przyjmuje się, że:

- zbiór pusty wierzchołków jest listą,
- jeśli h jest wierzchołkiem listy, a t jest listą, to para uporządkowana (h,t) jest listą, w której h jest nazywana głową listy, a t ogonem listy.



© UKSW, WMP, SNS, Warszawa

138

138

ISO/ANSI C – zm. dynamiczne

Głową listy może być zmienna wskaźnikowa, przechowująca adres pierwszego elementu listy, np.:

```
struct element* gp; // notacja ANSI C
element *gp; // notacja C++
```

lub zmienna typu „struct ...”, np.:

```
struct element g; // notacja ANSI C
element g; // notacja C++
```

W drugim przypadku głowa listy jest jednocześnie pierwszym elementem przechowującym dane

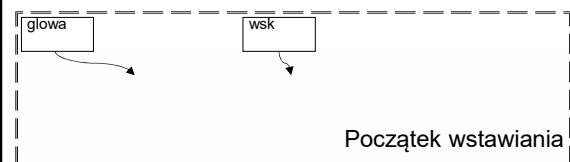
© UKSW, WMP, SNS, Warszawa

139

139

ISO/ANSI C – zm. dynamiczne

Tworzenie nowej listy



© UKSW, WMP, SNS, Warszawa

140

140

ISO/ANSI C – zm. dynamiczne

Tworzenie nowej listy

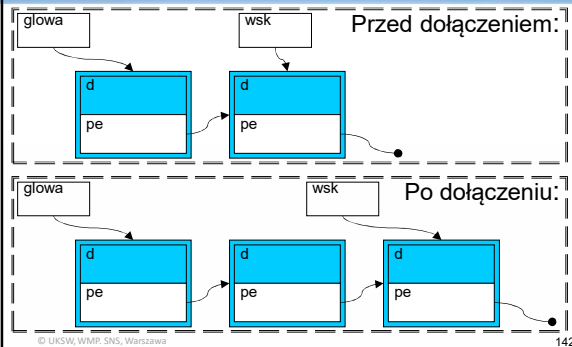
1. utwórz wskaźniki: na głowę listy "głowa", oraz na ostatni element listy "wsk".
2. while (są dane do wprowadzenia) {
 1. utwórz nowy element listy
 2. jeżeli "głowa" nie wskazuje na nic, niech "głowa" i "wsk" wskazywać na nowy element. W przeciwnym przypadku – niech pole wskaźnikowe elementu listy wskazywanego przez "wsk" wskazuje na nowy. Następnie zmień "wsk" tak, aby również wskazywał na nowy
 3. wypełnij danymi zawartość nowego elementu
 4. zapisz NULL w polu wskazującym na następny element listy w nowym elemencie

© UKSW, WMP, SNS, Warszawa

141

141

ISO/ANSI C – zm. dynamiczne



© UKSW, WMP, SNS, Warszawa

142

142

ISO/ANSI C – zm. dynamiczne

Przykładowy program tworzący listę

```
struct film_t {
    char tytuł[80];
    int rok;
    struct film_t *nast;
};

struct film_t *wsk, *głowa = NULL;
char tytuł[80];
int rok;
FILE* stream;
if( (stream = fopen("filmy.txt", "r")) == NULL )
    exit(1);
```

© UKSW, WMP, SNS, Warszawa

```
/* zakładamy, że w pliku „filmy.txt” wiersze zawierają
   tytuł i rok produkcji. Tytuł jest jednym słowem */
fscanf(stream, "%s%i", tytuł, &rok);
while (!feof( stream )) {
    if (głowa == NULL)
        głowa = malloc(sizeof(struct film_t));
    else {
        wsk->nast = malloc(sizeof(struct film_t));
        wsk = wsk->nast;
    }
    strcpy(wsk->tytuł, tytuł);
    wsk->rok = rok;
    wsk->nast = NULL;
    fscanf(stream, "%s%i", tytuł, &rok);
}
fclose(stream);
```

143

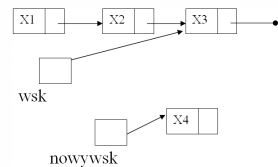
143

ISO/ANSI C – zm. dynamiczne

Dodanie elementu na końcu listy:

```
nowywsk = malloc(sizeof(struct film_t));
strcpy(nowywsk->tytuł, "Kingsajz");
nowywsk->rok = 1987;
nowywsk->nast = NULL;

wsk->nast = nowywsk;
wsk = wsk->nast;
```



© UKSW, WMP, SNS, Warszawa

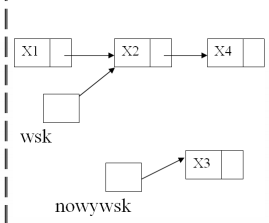
144

144

ISO/ANSI C – zm. dynamiczne

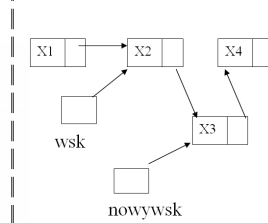
Dodanie do listy uporządkowanej (np. wg roku)

Przed dodaniem:



© UKSW, WMP, SNS, Warszawa

Po dodaniu:



145

145

ISO/ANSI C – zm. dynamiczne

Dodanie do listy uporządkowanej malejąco (np. wg roku)

```
nowywsk = malloc(sizeof(struct film_t));
strcpy(nowywsk->tytuł, "Bogowie");
nowywsk->rok = 2014;
nowywsk->nast = NULL;

if(głowa==NULL)
    głowa = wsk = nowywsk;
else
    if (głowa->rok < nowywsk->rok) {
        nowywsk->nast = głowa;
        głowa = nowywsk;
    }
```

© UKSW, WMP, SNS, Warszawa

```
else
    if (głowa->nast == NULL)
        głowa->nast = nowywsk;
    else {
        wsk = głowa;
        while (wsk->nast->rok > nowywsk->rok) {
            wsk = wsk->nast;
            if (wsk->nast == NULL)
                break;
        };
        nowywsk->nast = wsk->nast;
        wsk->nast = nowywsk;
    }
```

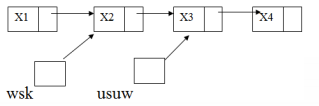
146

146

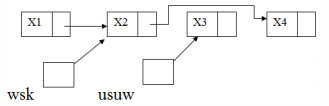
ISO/ANSI C – zm. dynamiczne

Usuwanie wskazanego elementu z listy

Przed usunięciem:



Po usunięciu:



© UKSW, WMP, SNS, Warszawa

147

147

ISO/ANSI C – zm. dynamiczne

Usuwanie wskazanego elementu z listy (*filmu z roku 2010*)

```
rok = 2010; // wskazuje na rok
           // filmu do usunięcia
```

```
if (glowa->rok == rok) {
    usuw = glowa;
    glowa = usuw->nast;
}
```

```
else {
    wsk = glowa;
    while ((wsk->nast != NULL)) {
        if (wsk->nast->rok == rok)
            break;
        wsk = wsk->nast;
    };
    if (wsk->nast != NULL) { // znaleziono!
        usuw = wsk->nast;
        wsk->nast = wsk->nast->nast;
        free(usun);
    };
}
```

© UKSW, WMP, SNS, Warszawa

148

148

ISO/ANSI C – zm. dynamiczne

Zalety list jednokierunkowych:

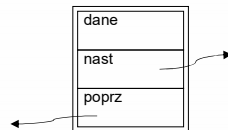
- zużywają pamięć proporcjonalnie do potrzeb

Wady list jednokierunkowych:

- wskaźniki mogą przesuwac się jedynie do przodu

Listy dwukierunkowe:

```
struct film_t {
    char tytul[80];
    int rok;
    struct film_t *nast;
    struct film_t *poprz;
};
```



© UKSW, WMP, SNS, Warszawa

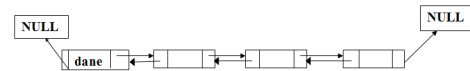
149

149

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe:

- możliwość poruszania się wskaźnika w obydwu kierunkach
- zużywają minimalnie więcej pamięci, są za to szybsze w dostępie do danych



© UKSW, WMP, SNS, Warszawa

150

150

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe:

```
struct film_t *wsk;
...
wsk = wsk->nast; /* Przeszczanie się w przód: */
...
wsk = wsk->poprz; /* Przeszczanie się w tył: */
```

© UKSW, WMP, SNS, Warszawa

151

151

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – dodawanie:

```
struct film_t *wsk, *nowy;
nowy – wskazuje na nowy element do umieszczenia
wsk – wskazanie na element po którym mamy umieścić 'nowy'

nowy->nast = wsk->nast;
nowy->poprz = wsk;
wsk->nast->poprz = nowy;
wsk->nast = nowy;
```

© UKSW, WMP, SNS, Warszawa

152

152

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – dodawanie:

```
struct film_t *wsk, *nowy;  
nowy – wskazuje na nowy element do umieszczenia  
wsk – wskazanie na element przed którym mamy umieścić 'nowy'  
  
nowy->nast = wsk;  
nowy->poprz = wsk->poprz;  
wsk->poprz->nast = nowy;  
wsk->poprz = nowy;
```

© UKSW, WMP, SNS, Warszawa

153

153

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – usuwanie:

```
struct film_t *wsk, *nowy;  
wsk – wskazanie na element, po którym znajduje się element do  
usunięcia  
usun – wskaźnik pomocniczy  
  
usun = wsk->nast;  
wsk->nast = wsk->nast->nast;  
wsk->nast->poprz = wsk;  
free (usun);
```

© UKSW, WMP, SNS, Warszawa

154

154

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – usuwanie:

```
struct film_t *wsk, *nowy;  
wsk – wskazanie na element, przed którym znajduje się element  
do usunięcia  
usun – wskaźnik pomocniczy  
  
usun = wsk->poprz;  
wsk->poprz = wsk->poprz->poprz;  
wsk->poprz->nast = wsk;  
free (usun);
```

© UKSW, WMP, SNS, Warszawa

155

155

ISO/ANSI C – zm. dynamiczne

Wnioski:

- Listy są przydatne szczególnie wtedy, gdy:
 - Dodawane są lub usuwane dowolne elementy ciągu, oraz
 - nie jest znana ich liczba.
- Tablice i pliki również umożliwiają wykonanie tych operacji, ale..
.. dla tablicy musimy przepisywać elementy leżące za wstawianym lub
usuwanym elementem, a dla plików musimy przepisywać całe pliki.
- Tablice są wygodniejsze do realizacji operacji wyszukiwania, przeglądania i
dostępu do wybranego elementu. (dla tablicy posortowanej oszacowanie
z góry kosztu tzw. wyszukiwania binarnego to $\log(n)$, natomiast dla list
stosowane jest wyszukiwanie liniowe – n).

© UKSW, WMP, SNS, Warszawa

156

156