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

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

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 następnych instrukcji (żadnego przyspieszania wykonania programu przy pomocy „potokowego” wykonania instrukcji przez zaawansowany procesor, itp.)

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 nietykalności zmiennych podawanych w argumentach wykonania, np.:

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

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

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

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?

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

119

ISO/ANSI C – zm. dynamiczne

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

Zmienne dynamiczne:

- tworzone w trakcie działania programu poleceniem alokacji bloku pamięci,
- rozmiar może być określony dopiero czasie działania programu (nie musi być wcześniej znany programiście),
- kiedy zmienna przestaje być potrzebna, blok pamięci można zwolnić.

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 (*tylko w ANSI C!*).

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) );
```


121

ISO/ANSI C – zm. dynamiczne

Usuwanie zaalokowych zmiennych

Kiedy zmienna przestaje być potrzebna zaalokowane zasoby pamięci należy zwolnić.

```
void free( void *memblock );
```

Argumentem wywołania jest wskaźnik na zmienną.

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

122

ISO/ANSI C – zm. dynamiczne

Usuwanie zaalokowych 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.*

ISO/ANSI C – zm. dynamiczne

Usunięcie wskaźnika z adresem zmiennej dynamicznej nie zwalnia tej zmiennej dynamicznej. Utracenie adresu zmiennej dynamicznej powoduje całkowitą utratę kontroli nad tą zmienną.

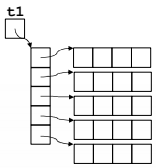
Przykład z tablicami dynamicznymi:

```
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 */
```


ISO/ANSI C – zm. dynamiczne

Dwuwymiarowe tablice dynamiczne



Alokowanie:

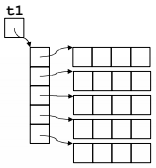
```
int i;
double **t1 = malloc(5*sizeof(double*));
for (i=0; i<5; i++)
  t1[i] = malloc(4*sizeof(double));
```


ISO/ANSI C – zm. dynamiczne

Dwuwymiarowe tablice dynamiczne

Zwalnianie:

```
int i;
for (i=0; i<5; i++)
  free( t1[i] );
free( t1 );
```



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 );
```


ISO/ANSI C – zm. dynamiczne

Zmiana rozmiaru zaalokowanego bloku

void *realloc(void * *memblock*, **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 **memblock!=NULL**, bądź jeżeli nie udało się rozszerzyć zaalokowanego bloku do żądanych rozmiarów.
Jeżeli podano **memblock==NULL**, działa jak zwykły **malloc**.

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

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
  adipisicing 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: */
```


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 przekażemy 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

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" );
```


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
```


ISO/ANSI C

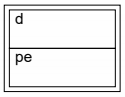
Zmienne dynamiczne:

Rekurencyjne typy danych

ISO/ANSI C – zm. dynamiczne

Rekurencyjne typy danych

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

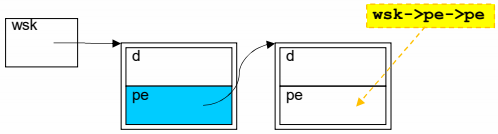


135

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));
```



136

ISO/ANSI C – zm. dynamiczne

```
struct element Jan, Ewa, Iza;
```


```
struct element *Jan = malloc(sizeof(struct element));
wsk->pe = malloc(sizeof(struct element));
```

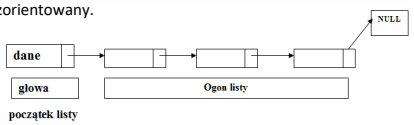
bratowa Jana.. *Mama bratowej Jana..*

137

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;
}
```

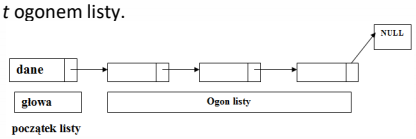


138

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.



139

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.:

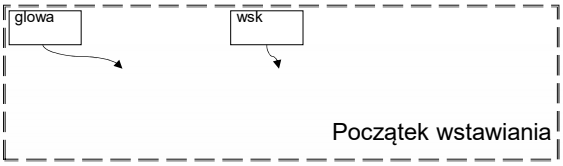
```
struct element g;       // notacja ANSI C
element g;              // notacja C++
```

W drugim przypadku głowa listy jest jednocześnie pierwszym elementem przechowującym dane

140

ISO/ANSI C – zm. dynamiczne

Tworzenie nowej listy



141

ISO/ANSI C – zm. dynamiczne

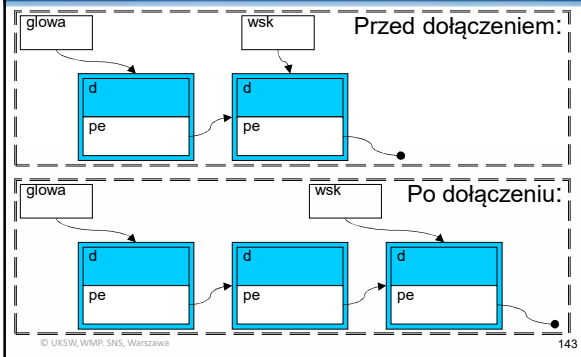
Tworzenie nowej listy

1. utwórz wskaźniki: na głowę listy ”glowa”, oraz na ostatni element listy ”wsk”.
2. while (są dane do wprowadzenia) {
   1. utwórz nowy element listy
   2. jeżeli ”glowa” nie wskazuje na nic, niech ”glowa” i ”wsk” wskazują 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

   }

142

ISO/ANSI C – zm. dynamiczne



143

ISO/ANSI C – zm. dynamiczne

Przykładowy program tworzący listę

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

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

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


144