Język ANSI C

część 14

struktury, unie, pola bitowe

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- struktury
  - struktura (słowo kluczowe `struct`) umożliwia zgrupowanie w pojedynczym rekordzie kilku zmiennych różnych typów (składowych - pól struktury)
  - składowymi struktury mogą być zmienne proste dowolnego typu, tablice, inne struktury, zmienne wskaźnikowe i wskaźniki struktur (w tym wskaźnik do struktury definiowanej) oraz funkcje (C++). Każde pole ma zarezerwowane osobne miejsce w pamięci.
  - pola są umieszczane w pamięci szeregowo zgodnie z kolejnością występowania w strukturze.
  - struktura przechowuje w danym momencie wartości wszystkich swoich składowych.

dobry zwyczaj to umieszczanie definicji struktur w plikach nagłówkowych *.h

ważne, w uniach będzie inaczej

```
struct osoba {
  char naz[19];           // średnik !
  unsigned rok, mies, dzien;
  long id;
};                        // średnik. jego pominięcie może być trudnym
                          // do znalezienia błędem

//lub zamiast };
} o1, o2;        // deklaracja struktury + deklaracja dwóch zm. strukt.
```

  - deklaracja struktury (unii) jest wzorcem, który opisuję budowę struktury
  - sama deklaracja typu struktury nie powoduje rezerwacji pamięci, a jedynie określa matrycę według, której będą tworzone zmienne strukturalne

- struktury

```
typedef struct osoba {    // "nazwany" własny typ danych
  char naz[19];
  unsigned rok, mies, dzien;
  long id;
}  DANA;                  //DANA – alternatywna nazwa typu osoba

struct osoba w1;          // zmienna typu osoba
DANA w2;                  // zmienna typu osoba
```


- struktury

```
struct osoba {
  char naz[19];
  unsigned rok, mies, dzien;
  long id;
} o1 = { "Kowal", 1985, 11, 9, 123 },
  o2 = { "Kostek",1981, 10, 9, 234 };

struct osoba o3 = { "Kowalski", 1984, 6, 15, 345 };
struct osoba o4 = { "Nowak" };     // zm. zewn. – pozostałe składowe są zerami
static struct osoba o5 = { "Nowaczek", 1990 };// pozostałe pola są zerami

o1 = o2;          // przypisanie takich samych struktur jest możliwe
if (o1 == o2)     // błąd. struktur nie wolno porównywać
```

struct osoba o3	ANSI C
osoba o3	C++

– dostęp do danych struktury

```
printf("%s\n", o1.naz);   //  "Kowal"
printf("%u\n%u\n%u\n", o1.rok, o1.mies, o1.dzien);
printf("%ld\n", o1.id);
printf("\n");
```

- wskaźniki do struktur

```
struct osoba {
  char naz[19];
  unsigned rok, mies, dzien;
  long id;
};

struct osoba o;           // zmienna strukturalna
struct osoba *po;         // wskaźnik do zmiennej strukturalnej (do struktury)

po = &o;                  // pobranie adresu struktury
po -> id = 1234; // nowy operator -> dostępu do pól s. poprzez wskaźnik
strcpy(po -> naz, "Gramacki");

printf("%ld\n", po->id);  // tak wygodniej
printf("%ld\n", (*po).id);// nawiasy konieczne
printf("%ld\n", *po.id);  // z reguły błąd składniowy.
                          // interpretacja kompilatora: *(po.id)
                          // ale gdy id będzie wskaźnikiem, to błędu
składniowego
                  // nie będzie. Błąd logiczny będzie jednak trudno znaleźć
printf("%s\n", po->naz);
```

przypomnij sobie składnię korzystania ze wskaźników do funkcji

  - sytuacje podatne na błędy

```
++po->id;                 // ++(po->id), zwiększenie liczby zamiast wskaźnika
(++po)->id;               // zwiększenie po przed odwołaniem do id
(po++)->id;               // zwiększenie po po odwołaniu do id
```


- struktury i pola wskaźnikowe
  - struktury mogą zawierać pola typu wskaźnikowego
  - reguły inicjowania i modyfikowania składowych wskaźnikowych są takie same jak w przypadku innych zmiennych wskaźnikowych
  - w szczególności należy pamiętać, aby nie wstawiać danych do obszaru pamięci, który nie został przydzielony

```
struct tosoba {
  char naz[19];
  char *imie;     // wskaznik do char, patrz inicjacja niżej
  int  *ptr;      // wskaźnik do int
  unsigned r, m, d;
  long id;
} z1 = { "Kowal", "Adam", (int *)0xB8000000, 1981, 11, 8, 123},
  z2 = { "Nowak", NULL,  NULL, 1981, 10, 9, 234 }
;

gets(z2.imie);    // Błąd. Nie wolno wczytywać danych do pamięci wskazywanej
                  // przez niezainicjowane pola wskaźnikowe

z2.imie = (char*) malloc (50);
gets(z2.imie);            // OK
strcpy(z2.imie, "Jan");   // OK
```

- wskaźniki do struktur i pola wskaźnikowe
  - sytuacje podatne na błędy

```
struct {          // struktura nienazwana (samo struct) jest poprawna gdy
  int len;        // zaraz po niej pojawia deklaracja zmiennej (tu: *p)
  char *str;
} *p;
```

```
++p->len;  // zwiększenie len zamiast zwiększenia wskaźnika gdyż: ++(p->len)
*p->str;          // udostępnia to na co wskazuje str
*p->str++;        // zwiększa str po udostępnieniu obiektu wskazywanego przez
str
                  // (tak jak *str++)
(*p->str)++;      // zwiększa to na co wskazuje str
*p++->str;  // zwiększa p po udostępnieniu obiektu wskazywanego przez str
```

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left

```
// właściwość struktur: wyrównywanie pól struktury
struct test {
  char c;
  int i; };

sizeof(struct test);      // = 8,  !!!, ???
                          // gdyż wyrównywanie pól struktury
                          // (zależne od implementacji !)
// dlatego pliki z zapisanymi strukturami są (prawie zawsze)  nieprzenośne
```


- struktury i funkcje
  - do funkcji można przekazać:
    + składniki struktury oddzielnie
    + całą strukturę
    + wskaźnik do struktury

```
struct point * makepoint2(int x, int y) {
  struct point *ptr =
  (struct point*) malloc(sizeof(struct point);

  ptr->x = x;
  ptr->y = y;
  return ptr;
 }                  // 2 x makepoint(), Błąd
                    // gdyż nie można rozróżnić
                    // funkcji po liście argumentów
```

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

struct point makepoint(int x, int y)
{
  struct point temp; // zmienna tymczasowa

  temp.x = x;     // x - nazwa pola i nazwa argumentu
  temp.y = y;     // nie ma tu żadnego konfliktu
  return temp; }

struct point addpoint (struct point p1, struct point p2)
{
                  // brak zmiennej tymczasowej, bo i po co ona ...
  p1.x += p2.x;   // ... p1, p2 - lokalne kopie
  p1.y += p2.y;
  return p1; }

void addpoint (struct point *ptrp1, struct point *ptrp2) // przez wskaźniki
{
  ptrp1->x += ptrp2->x;
  ptrp1->y += ptrp2->y; }
```

przeciążanie nazw funkcji: mechanizm C++

2 x addpoint(), OK
gdyż różne argumenty funkcji

- struktury i funkcje

```
// przykładowe wywołania

struct point p1 = makepoint(1, 4);          // utwórz dwa punkty
struct point p2 = makepoint(10, 40);        // deklaracja zm. strukt plus ich
                                            // inicjalizacja funkcją
printf("(%d %d)\n", p1.x, p1.y);
printf("(%d %d)\n", p2.x, p2.y);

struct point *ptr1 = makepoint2(2, 5);      // utwórz dwa kolejne punkty
struct point *ptr2 = makepoint2(20, 50);
printf("(%d %d)\n", ptr1->x, ptr1->y);
printf("(%d %d)\n", ptr2->x, ptr2->y);

struct point p3 = addpoint(p1, p2);         // struktura przez wartość.
                        // wewnątrz funkcji powstaje ich lokalna kopia
printf("(%d %d)\n", p3.x, p3.y);

addpoint(ptr1, ptr2);
printf("(%d %d)\n", ptr1->x, ptr1->y);      // struktura przez wskaźnik

// Wynik:
(1 4)
(10 40)
(2 5)
(20 50)
(11 44)
(22 55)
```


- tablice struktur – analogicznie do "zwykłych" tablic
  tablice wskaźników do struktur

```
struct osoba {
  char naz[19];
  unsigned rok, mies, dzien;
  long id; };

struct osoba T1osoba[100];                  // tablica struktur

struct osoba T2osoba[ ] =                   // tablica struktur
{{"Wojciech", 1999, 12, 5, 177},
  {"Piotr",   1997, 7,  8, 178}};

struct osoba *T3osoba[2];          // tablica (dwóch) wskaźników do struktur

gets( T1osoba[5].naz );
T1osoba[5].id = 187;

printf("%s\n", T1osoba[5].naz);
printf("%s\n", T2osoba[0].naz);
printf("%c\n", T2osoba[0].naz[3]);          // uwaga na 2 x []

for (int i=0; i<2; i++)
  T3osoba[i] = &T2osoba[i];                 // zapełniamy tablicę wskaźników

printf("%s\n", (*T3osoba[0]).naz);          // uwaga na nawiasy
printf("%s\n", (*T3osoba[1]).naz);
```

- zapis / odczyt struktur na / z pliku

na plik

```
#include <stdio.h> #include <stdlib.h>
int main(void)  {

struct personel{
   char name[40];
   int id;
   double height;
} agent;                                          // agent's data are here

   char buf[81]; FILE *fptr;  long int file_size;
   struct personel *agent_ptr;                    // pointer to memory for agents

   if( !(fptr = fopen("agents.rec", "wb")))       // tryb binarny konieczny
      { printf("\nCan't open file agents.rec"); exit(1); }
   do {
      getchar();
      printf("\nEnter name: ");
      gets(agent.name);
      printf("Enter number: ");
      gets(buf);
      agent.id = atoi(buf);
      printf("Enter height: ");
      gets(buf);
      agent.height = atof(buf);
                                                  // w pętli po jednym agencie do pliku
      fwrite ( &agent, sizeof(agent), 1, fptr );  // adres struktury: &agent
      printf("Add another agent (y/n)? ");
      }
   while(getchar() == 'y');     fclose(fptr);                          c.d.n
```


- zapis / odczyt struktur na / z pliku

z pliku

```
 if( !(fptr = fopen("agents.rec", "rb")))
      { printf("\nCan't open file agents.rec\n"); exit(1); }

   fseek(fptr, 0, SEEK_END);    // put file ptr at end of file
   file_size = ftell(fptr);     // file size is file pointer
   fseek(fptr, 0, SEEK_SET);    // return file ptr to start

   // allocate memory for entire file
   if( !(agent_ptr = malloc((size_t)file_size)))
      { printf("\nAllocation error"); fclose(fptr); exit(1); }

   // calculate n from file size
   n = (int)file_size / sizeof(struct personel);

   // odczyt całej bazy (n agentów) z pliku do pamięci
   if( fread ( agent_ptr, sizeof(struct personel), n, fptr ) != n)
      { printf("\nCan't read file"); fclose(fptr); exit(1); }

 printf("\nFile read. Total agents is now %d.\n", n);

 for(int i=0; i<n; i++){
   printf("%s %d %f: \n", agptr->name, agptr->id, agptr->height);
   agptr++;
   }

  fclose(fptr);
return 0;
}
```

- odczyt pliku do pamięci

Na plik / z pliku; inna wersja

```
// wczytaj plik do pamięci (do struktury)
#include <stdio.h>  #include <stdlib.h>
structF = wczytajPlik ( FILE * );

typedef struct
 {
    char *bufor               // dane pliku (ciąg bajtów)
    long int wielkosc         // wielkość pliku
 } structF;

int main(void)
{
  structF dane;
  FILE *fp;
  fp = fopen ( "plik.abc", "rb");
  dane = wczytajPlik ( fp );                              // funkcja wczytajPlik()
  fclose( fp);

  // tu praca z danymi
  // (UWAGA: zakładamy, że wielkość pliku nie zmieni się)

  fp = fopen ( "plik.abc", "wb");
  fwrite ( dane.bufor, sizeof(char), dane.wielkosc, fp);    // dane na plik
  fclose( fp);

  return(0);
}
```

c.d.n

- odczyt pliku do pamięci

Na plik / z pliku; inna wersja

```
// wczytaj plik do pamięci (do struktury)

structF wczytajPlik ( FILE *fp)
{
  long int dlugoscPliku;
  structF dane;

  fseek ( fp, 0 , SEEK_END );           // jaki duży plik
  dlugoscPliku = ftell ( fp );
  fseek ( fp, 0 , SEEK_SET );

  dane.bufor = ( char * ) malloc ( sizeof(char) * dlugoscPliku);
  if ( !dane.bufor ) exit(1);

  fread ( dane.bufor, sizeof(char), dlugoscPliku, fp);      // cały plik do pamięci
  dane.wielkosc = dlugoscPliku;

  return dane;
}
```

W strukturach było szeregowo

- unie
  - unia (słowo kluczowe `union`) jest strukturą, której składowe są umieszczane równolegle w tym samym obszarze pamięci.
  - w danym momencie unia przechowuje wartość tylko jednej składowej - tej, która została zmodyfikowana jako ostatnia.
  - zmienna "unijna" jest wystarczająco obszerna aby pomieścić w sobie wartość największego z typów składowych (zależy to oczywiście od implementacji)
  - unie są podobne do rekordów z wariantami w Pascal-u

```
union dana {
  char z[5];
  int  k;
  long p;
  float w;
};
union dana u1, u2;  // oba słowa: union i dana konieczne w C (w C++ już nie)


union {                   // unia bez nazwy typu
  char z[5];
  int  k;
  long p;
  float w;
} u1;                     // nazwa zmiennej teraz konieczna
```


- unie
  - w przypadku unii wszystkie składowe są pamiętane we wspólnym obszarze pamięci
  - dlatego unię inicjuje się jedną daną odpowiadającą typowi pierwszej składowej
  - jeśli dana jest innego typu to następuje automatyczna konwersja (o ile jest to możliwe) do typu pierwszego pola unii.

```
union dana
{
  char z[5];
  int  k;
  long p;
  float w;
} u1 = { 'A', 'B', 'C'}, u2 = {"Ala"};     // unie u1, u2

union dana u3 = { 65 };   // z[0] = 'A';  pozostałe pola tablicy - zera
union dana u4 = { (char)66.42 };// z[0] = 'B';  rzutowanie (char) 66.42 = 66

printf("%s\n", u1.z);     //   u1  "ABC"
printf("%s\n", u2.z);     //   u2  "Ala"
printf("%c\n", u3.z[0]);  //   u3  A (kod 65)
printf("%c\n", u3.z[1]);  //   u3  ???
printf("%d\n", u4.k);     //   u4  66
```

- pola bitowe
  - pozwalają oszczędzać pamięć poprzez "upakowanie" informacji do bitów (zamiast np. do całych słów)
  - w przypadku pól typu całkowitego można w deklaracji struktury (unii) określić ile bitów będą one zajmowały
  - zakres wartości, które można przechowywać w danym polu zależy od liczby bitów przydzielonych polu oraz od tego czy jest to pole ze znakiem, czy bez znaku
  - najmniejsza ilość pamięci przeznaczona na pojedyncze pole wynosi 1 bit. Największa wynosi sizeof(int) = 32
  - pole bitowe definiuje się podając typ danej, nazwę pola i po dwukropku liczbę bitów wykorzystywanych przez pole

```
struct  atrybut1 {
  int bit1 : 1;                  // pole jednobitowe int
} A;

struct  atrybut2 {
  unsigned bit1 : 1;             // pole jednobitowe unsigned
} B;
```

Jeśli pole 1-bitowe jest typu signed int (char), to może ono przyjmować wartości 0 lub –1.

Jeśli pole jest typu unsigned int (char), to może ono przyjmować wartości 0 lub 1

- pola bitowe
  - pola bitowe w pewnym sensie są kontrpropozycją dla pewnych zastosowań dyrektywy `#define`

```
#define JEST     01
#define NIE_MA   02
#define POMIEDZY 04 // wartości muszą być potęgami dwójki
                    // gdyż będziemy potem odpowiednio stosowali op. bitowe

// lub
enum { JEST = 01, NIE_MA = 02, POMIEDZY = 04 };

// alternatywa z polami nitowymi
struct {
unsigned int is          : 1; // usigned (tu) konieczne. patrz poprz. strona
unsigned int is_not      : 1;
unsigned int between     : 1;
} flags;                 // trzy jednobitowe pola

// pola zachowują się jak "małe" zmienne całkowite
flags.is = flags.is_not  = 1;             // włączamy bity
flags.is_ between = flags.is_not = 0;     // wyłączamy bity
if (flags.is_not == 0 && flags.is == 0)   // czy oba skasowane ?
```

- pola bitowe

```
int main(void)
{
 A.bit1 = 1;
 printf("%d\n", A.bit1);      // wartość = -1

 A.bit1 = 0;
 printf("%d\n", A.bit1);      // wartość = 0

 B.bit1 = 1;
 printf("%u\n", A.bit1);      // wartość = 1

 B.bit1 = 0;
 printf("%u\n", A.bit1);      // wartość = 0

 return 0;
}
```

```
struct  atrybut {
 int bit1         : 1;
} A;

struct  atrybut1 {
  unsigned bit1   : 1;
} B;
```

```
struct  atrybut {
   int bit1       : 1;
   int bity2_10   : 9;
   int bit11      : 1;
   int bity12_23  : 12;
} A;                          // rozmiar struktury 23 bity

int main() {
A.bit1 = 1;    A.bity2_10  = 32;    A.bit11 = 1;    A.bity12_23 = 1024;

printf("%d\n", A.bit1);               // -1
printf("%d\n", A.bity2_10);           // 32
printf("%d\n", A.bit11);              // -1
printf("%d\n", A.bity12_23);          // 1024
```


- wykorzystanie struktur, liczby zespolone

```
/*
 *  Kod z książki
 *  Programowanie w języku C. FAQ
 *  Addison-Wesley, 1996, ISBN 0-201-84519-9
 *  (pytanie 14.11)
 *
 *  Ten kod może być dowolnie używany i modyfikowany,
 *  bez żadnych ograniczeń.
 */

// complex.h

typedef struct
{
  double real;
  double imag;
} complex;

extern complex cpx_make (double, double);
extern complex cpx_add (complex, complex);
extern complex cpx_subtract (complex, complex);
extern complex cpx_multiply (complex, complex);

extern char *cpx_print(char *, complex);


#define Real(c) (c).real      // asekuracyjnie (c) zamiast c
#define Imag(c) (c).imag
                              c.d.n.
```

- wykorzystanie struktur, liczby zespolone

```
// cfun.c
#include <stdio.h>  #include "complex.h"

complex cpx_make (double real, double imag) {     // zwracamy strukturę
          complex ret;
          ret.real = real;
          ret.imag = imag;
          return ret; }

complex cpx_add (complex a, complex b) {          // pobieramy i zwracamy struktury
          return cpx_make(Real(a) + Real(b), Imag(a) + Imag(b)); }

complex cpx_subtract (complex a, complex b) {
          return cpx_make(Real(a) - Real(b), Imag(a) - Imag(b)); }

complex cpx_multiply (complex a, complex b) {
          return cpx_make(Real(a) * Real(b) - Imag(a) * Imag(b),
                    Real(a) * Imag(b) + Imag(a) * Real(b)); }

char *cpx_print (char *fmt, complex a) {
          static char retbuf[30];
          char *p;
          sprintf (retbuf, fmt, Real(a));
          for(p = retbuf; *p != '\0'; p++)
            ;
          *p++ = '+';
          sprintf (p, fmt, Imag(a));
          for(; *p != '\0'; p++)
            ;
          *p++ = 'i';
          *p = '\0';
          return retbuf; }
```

zobacz wywołanie tej funkcji na następnej stronie

- wykorzystanie struktur, liczby zespolone

```
// main.c
#include <stdio.h>
#include "complex.h"

int main(void)
{
  complex a = cpx_make (1, 2);  complex b = cpx_make (3, 4);  complex c = cpx_add (a, b);

  printf("a: %s\n", cpx_print ("%g", a));
  printf("b: %s\n", cpx_print ("%g", b));
  printf("c=a+b: %s\n", cpx_print ("%g", c));

  c = cpx_subtract (a, b);   printf("c=a-b: %s\n", cpx_print("%g", c));

  c = cpx_multiply (a, b);   printf("c=a*b: %s\n", cpx_print("%g", c));

  printf("a+b: %s\n", cpx_print("%g", cpx_add(a, b)));
  printf("1+2i - 3+4i: %s\n", cpx_print("%g",

  cpx_subtract (cpx_make(1, 2), cpx_make(3, 4))));
  {
    complex c = cpx_add(cpx_make(1, 2), cpx_make(3, 4));
    printf("c=1+2i + 3+4i: %s\n", cpx_print("%g", c));
  }

  c = cpx_multiply(cpx_make(1, 2), cpx_make(3, 4));
  printf("c=1+2i * 3+4i: %s\n", cpx_print("%g", c));
  return 0;
}
```

- wykorzystanie unii, float2bin
  - dzięki unii, możliwe jest "oszukanie" kompilatora, który nie pozwala na stosowanie operatorów bitowych do typów nie-całkowitych

```
#include <stdio.h> #include <ctype.h> #include <stdlib.h> #include <math.h>
void printToFile(int size, double *digit);

int main(int argc, char *argv[])
{
  union   {
    float f;
    double d;
    unsigned char c[8];
  } u;

  int x, i, j, bit;
  unsigned char mask;

  if (*argv[1] == 'f') {
    u.f =  (float) atof(argv[2]);
    x = 4;
    printf("You entered: \n%s\n\n", argv[2]);
    printf("After converted to float (displayed using printf): \n%.40f\n\n", u.f);
    printf("Displayed as hexadecimal (internal representation):
            \n%02x %02x %02x %02x\n\n", u.c[0], u.c[1], u.c[2], u.c[3]);   }
  else  {
    u.d =  (double) atof(argv[2]);
    x = 8;
    printf("You entered: \n%s\n\n", argv[2]);
    printf("After converted to double (displayed using printf): \n%.40f\n\n", u.d);
    printf("Displayed as hexadecimal (internal representation): \n");
    printf("%02x %02x %02x %02x  ",   u.c[0], u.c[1], u.c[2], u.c[3]);
    printf("%02x %02x %02x %02x\n\n", u.c[4], u.c[5], u.c[6], u.c[7]);     }
```

```
You entered:
0.1

After converted to float (displayed using printf):
0.10000000149011611900000000000000000000

Displayed as hexadecimal (internal representation):
3d cc cc cd

Displayed as binary (internal representation):
00111101 11001100 11001100 11001101
```

gdyż kod formatujący `%x` nie zadziała dla argumentu `float` !

- wykorzystanie unii, float2bin

```
  // convert float to binary
  printf("Displayed as binary (internal representation): \n");
  for (i=0; i<x; i++)
  {
    mask=0x80;
    if(i>0) printf(" ");
    for (j=0; j<8; j++)
    {
      bit = (mask & u.c[i]) ? 1 : 0;    // to już "jest" typ całkowity
      printf("%d", bit);                // można stosować więc operatory bitowe
      mask >>= 1;
    }
  }

  if (*argv[1] == 'f')
    printToFile(4, (double*)&u.f);
  else
    printToFile(8, &u.d);

  return 0;
}


void printToFile(int size, double *digit)
{
  FILE *fptr;
  fptr = fopen("digit.bin", "wb");
  fwrite(digit, size, 1, fptr);
  fclose(fptr);
}
```

gdyż operator bitowy `&` nie zadziała dla argumentu `float` !