Podstawy programowania.

Wykład 14

Co jeszcze...

Przypomnienia, uzupełnienia

Arytmetyka wskaźników

- Na wskaźnikach można dokonywać następujących operacji arytmetycznych (obok podstawienia: NULL, adresu zmiennej lub wartości innego wskaźnika tego samego typu):
  - zakładając definicje:
    - int a[]; int* pi; int* qi; int n;
  - dodanie (odjęcie) liczby całkowitej do (od) wskaźnika:
    - np. pi+n – efekt: jeśli pi == a (&a[0]) to pi+n == &a[n];
    - uwaga: arytmetycznie wynik operacji zależy od typu wskaźnika
  - dodanie (odjęcie) dwóch wskaźników (wynik typu *ptrdiff _t* lub *size_t*)
    - np. qi-pi – efekt: jeśli pi == &a[0] i qi = &a[n] to qi-pi == n
    - wynik nie zależy od typu, ale typ a, pi i qi musi być taki sam
  - porównanie wskaźnika do zera (NULL) lub innego wskaźnika
    - np. if(pi<qi){.....} // jeśli pi wskazuje na wcześniejszy wyraz niż qi
    - pi i qi muszą być wskaźnikami do wyrazów tej samej tablicy

Arytmetyka wskaźników

- Argument przesłany jako adres (niekoniecznie wartość zmiennej) zostaje skopiowany na stos i wewnątrz wywołanej funkcji traktowany jest jak zmienna wskaźnikowa, na której można wykonywać dopuszczalne operacje:

```
void main ( void)
{
   int a[10];
   // a++; // niedozwolone – miejsce a jest określone
   funkcja( a );
}

void funkcja( int *wsk_a ){
   printf("%d\n", *wsk_a); // *wsk_a== a[0], wsk_a = a
   wsk_a++; // dozwolone – wsk_a jest zmienną na stosie
   printf("%d\n", *wsk_a); // *wsk_a== a[1], wsk_a = a+1
}
```

Arytmetyka wskaźników

- Arytmetyka wskaźników prowadzi do zwięzłego kodu
  - kod może być trudny do zrozumienia
  - teoretyczny zysk czasowy wynikający z unikania arytmetyki indeksów (i stosowania arytmetyki adresów) zazwyczaj jest pozorny
    - optymalizujący kompilator sam dokona odpowiednich modyfikacji przy tłumaczeniu na asembler
      - kompilatorowi może być łatwiej optymalizować kod używający notacji tablicowej
    - kod notacji tablicowej i arytmetyki indeksów może być łatwiejszy w rozumieniu i utrzymaniu niż kod arytmetyki wskaźników
- Przykład arytmetyki wskaźników
  - funkcja kopiowania dwóch tablic o długości *n*

```
void tabncpy(int n, int * restrict p, int * restrict q)
{
   while (n-- > 0) *p++ = *q++;
 }
```

Arytmetyka wskaźników

- Przykłady arytmetyki wskaźników
  - dwie wersje funkcji obliczania długości tablicy znaków:

```
int strlen_1(char *s)
{
   int n;
   for (n = 0; *s != '\0'; s++) n++;
   return n;
}
```

```
int strlen_2(char *s)
{
   char *p = s;
   while (*p != '\0') p++;
   return p - s;
}
```

  - funkcja kopiowania tablicy znaków

```
void strcpy(char *s, char *t)
{
   while (*s++ = *t++);
}
```

Typy danych - przypomnienie

- Podstawowe wbudowane typy danych języka C:
  - _Bool – 0 i 1 (C99)
  - znaki (*char*) – 7 bitów dla znaków ASCII
  - liczby całkowite (*int*), od INT_MIN do INT_MAX (z <limits.h>)
  - liczby rzeczywiste (zmiennoprzecinkowe)
    - pojedynczej (*float*), podwójnej (*double*) i rozszerzonej (*long double*) precyzji (standardy IEEE)
    - liczby zespolone: *float _Complex*, *double _Complex* i *long double _Complex* (C99 - opcjonalne)
- określenia (specyfikatory): *short, long, signed, unsigned*
  - sugerują jak wiele miejsca w pamięci przeznaczyć na zmienną i czy przeznaczyć jeden z bitów na przechowywanie znaku zmiennej
    - *signed char, short int, int, long int, long long int* – zmienne całkowite ze znakiem
    - dla każdego typu ze znakiem istnieje odpowiednik bez znaku - *unsigned*

Stałe (literały)

- Stałe typu całkowitego (oprócz standardowych):
  - long – np. `2L`
  - unsigned – np. `12u`
  - unsigned long – np. `25UL`
- Zapis stałych typu całkowitego
  - standardowy – jak wyżej
  - ósemkowy (*octal*) – `037 == 31`
  - szesnastkowy (*hexadecimal*) – `0x1F == 31`
  - `0` na początku zawsze oznacza notację ósemkową, a `0x` notację szesnastkową (możliwe warianty *long*, *unsigned*) – `0xFUL == 15UL`
- Stałe typu znakowego
  - małe liczby całkowite, zapisane w specyficznej notacji – `'x'`
    - `'0' == 48`
    - *escape sequences* – np. `'\b'`, `'\n'` `'\t'`, `'\\'`, `'\?'`, `'\"'`, `'\''`
      - zapis ósemkowy: `'\170'`, zapis szesnastkowy: `'\x5D'`

Stałe (literały)

- Stałe napisowe
  - "Hello world\n"
  - tablice znaków zakończone znakiem NULL - '\0'
  - długość napisu (np. zwracana przez funkcję *strlen*) zazwyczaj nie obejmuje '\0' – alokacja pamięci do przechowywania napisu musi to uwzględniać
  - "x" != 'x'
- Stałe typu zmiennoprzecinkowego
  - float – 3.14f
  - double – 3.14
  - long double – 3.14L
- Wyrażenia stałe
  - wyrażenia zbudowane wyłącznie ze stałych – 3.14+1
  - wyrażenia stałe zazwyczaj obliczane są w trakcie kompilacji

Sposoby przechowywania zmiennych

- Poza standardowo definiowanymi zmiennymi: lokalnymi (definiowanymi w konkretnym bloku programu) i globalnymi, istnieją jeszcze zmienne definiowane z określeniem static
  - takie zmienne definiowane poza funkcjami są widoczne we wszystkich funkcjach definiowanych w danym pliku
    - czas życia jest równy czasowi wykonania programu
  - jeśli zmienna definiowana jest z określeniem static wewnątrz funkcji (bloku), jej widzialność pozostaje określona przez miejsce definicji, ale zmienia się czas życia na równy czasowi wykonania programu
    - jej wartość zostaje zachowana pomiędzy wywołaniami funkcji

```
static int zm_statyczna_glob = 0; // jednokrotne inicjowanie
void funkcja(void){
    static int zm_statyczna_lok = 0; // jednokrotne inicjowanie
    zm_statyczna_glob++;
    zm_statyczna_lok++;
}
```

Sposób linkowania (linkage)

- Sposób linkowania funkcji i obiektów (zmiennych) określa czy dane dwie deklaracje odnoszą się do tego samego obiektu
  - **sposób linkowania statyczny** (*static linkage*) **– jeden obiekt w pliku**
    - nielokalne obiekty posiadające sposób przechowywania *static*
  - **sposób linkowania zewnętrzny** (*external linkage*) **– jeden obiekt w programie**
    - obiekty posiadające sposób przechowywania *extern* (z pewnymi wyjątkami)
    - np. wszystkie nie-statyczne funkcje
  - **sposób linkowania nieokreślony** (*none*) **- każda definicja odnosi się do odrębnego obiektu**
    - takie obiekty w kodzie posiadają wyłącznie definicje – deklaracje (nie będące definicjami) są niepotrzebne
    - np. obiekty na stosie (argumenty i standardowe zmienne lokalne procedur (bez określeń static i extern))

Czas życia obiektów

- Obiekty (m.in. zmienne) mają jeden z czterech określonych czasów życia (okresów przechowywania, ***storage duration***)
  - statyczny (***static***) - czas życia równy czasowi wykonywania programu
    - jednokrotne inicjowanie na początku wykonania programu
      - jawne lub niejawne za pomocą zer
    - zmienne statyczne (*static*) i globalne (*extern*)
  - automatyczny (***automatic***) - czas życia od momentu definicji do końca bloku (funkcja jest także blokiem), w którym obiekt jest zdefiniowany
    - każdorazowe wejście do bloku rozpoczyna "nowe życie" obiektu
      - bez domyślnego inicjowania - jawne inicjowanie jest przeprowadzane przy każdym napotkaniu definicji z inicjowaniem
  - "dynamiczny", zaalokowany (***allocated***) - czas życia wynikający z użycia funkcji dynamicznego alokowania i zwalniania pamięci
  - ***thread*** - czas życia związany z istnieniem odrębnego wątku wykonującego określony fragment programu

Zmienne stałe(?)

- Zmienne mogą posiadać określenie *const* informujące, że ich wartości nie mogą być zmieniane w trakcie wykonania
  - zmienne stałe muszą zostać odpowiednio zainicjowane
    - `const long double PI_L = 3.14159265358979323846L;`
  - można jako stałe określać całe tablice
    - `const char napis[] = "Hello world";`
  - sens użycia określenia *const* polega na umożliwieniu dokonania odpowiednich optymalizacji przez kompilator
    - kompilator może nawet usunąć zmienną stałą z programu
    - w wersji niezoptymalizowanej zmienna taka zawsze pozostanie, np. w celu umożliwienia debugowania
  - określenia *const* można używać dla argumentów funkcji
    - kompilator sprawdza, czy zmienne określone jako *const* (np. wartości w tablicach przekazanych za pomocą wskaźnika) nie są zmieniane wewnątrz funkcji
  - używanie określenia *const* zwiększa czytelność kodu i odporność na błędy

Stałe

- Określenie *const* stanowi trzeci (obok *#define* i *enum*) sposób określania stałych nazwanych w kodzie źródłowym
- Zaletą użycia *const* jest widoczność zmiennych w debugerach, wadą niemożność wymiarowania tablic o stałych rozmiarach
- Stałe nazwane najczęściej umieszczane są w plikach nagłówkowych
  - pliki nagłówkowe włączane są do wielu plików źródłowych, przez co ich treść jest powielana w wielu jednostkach kompilacji
    - można użyć definicji w jednym z plików źródłowych kodu i deklaracji *extern* w pliku nagłówkowym
      - wtedy w pliku nagłówkowym nie ma wartości przypisanej stałej
    - lepszym rozwiązaniem jest statyczna zmienna widoczna w pliku
      - umieszczenie statycznej zmiennej w pliku nagłówkowym włączanym do wielu plików, oznacza, że dla każdego pliku źródłowego istnieje odrębna zmienna stała o tej samej wartości

Zmienne stałe(?)

- Stała z użyciem *const*
  - static const double PI = 3.14159265358979;
  - kompilator dopuszcza istnienie wielu zmiennych statycznych o tej samej nazwie
- Przykład złożonego inicjowania stałej
  - static const char * const dni_tygodnia_tab[] = { "Poniedziałek", \ "Wtorek", "Środa", "Czwartek", "Piątek", "Sobota", "Niedziela" };
  - dni_tygodnia_tab jest tablicą wskaźników
    - długość jest obliczana i ustalana na podstawie instrukcji inicjowania
    - każdy element tablicy jest wskaźnikiem do tablicy
      - wskaźnik zostaje utożsamiony z tablicą znaków (wartość każdego wskaźnika jest ustalana jako adres pewnego napisu - stałej tablicy znaków w pamięci)
      - taki sposób działania jest możliwy tylko dla stałych napisów (literałów łańcuchowych)
  - wykorzystanie
    - np. printf("Trzeci dzień tygodnia: %s\n", dni_tygodnia_tab[2]);

Zmienne stałe(?)

- Składnia użycia *const* w przypadku wskaźników decyduje o tym czy niezmienny ma pozostawać wskaźnik czy wskazywana wartość
  - int * const cpi = &i; // stały wskaźnik
    - pokazuje cały czas w to samo miejsce pamięci, które może zmieniać swoją zawartość:

      cpi++ - BŁĄD, (*cpi)++ - OK
  - const int * pci = &i; // wskaźnik do stałej wartości
    - może wskazywać na różne miejsca pamięci, ale nie można dokonywać zmian wskazywanych zmiennych poprzez ten wskaźnik

      pci++ - OK, (*pci)++ - BŁĄD
  - const int * const cpci = &i; // stały wskaźnik do stałej wartości

    cpci++ - BŁĄD, (*cpci)++ - BŁĄD
    - pokazuje cały czas w to samo miejsce pamięci, nie można dokonywać zmian wskazywanych zmiennych poprzez ten wskaźnik

Typy danych

- Określenia zmiennych (poza *const*):
  - określenia: *register*, *restrict* i *volatile* dotyczą sposobu przechowywania zmiennych
    - określenia są sugestiami dla kompilatora i mogą zostać zignorowane
    - *register* – sugeruje, aby zmienna była przechowana w szybkiej pamięci, np. w rejestrach (co oznacza, że może zniknąć z kodu po kompilacji)
      - konsekwencją jest niemożność posługiwania się w kodzie adresem takiej zmiennej
    - *restrict* – informuje, że wskaźniki których dotyczy operują na wartościach, na których nie operują inne wskaźniki
      - inaczej: obszary tablic w pamięci nie pokrywają się
    - *volatile* – sugeruje, aby dostępy do zmiennej nie podlegały agresywnej optymalizacji przez kompilator
      - może to mieć znaczenie przy korzystaniu z zasobów systemów lub w obliczeniach współbieżnych i równoległych

Wyliczenia

- Standard C wprowadza wyliczeniowy typ zmiennych *enum*
  - typ *enum* definiuje zestaw symboli, które mogą być traktowane jak wyliczenie możliwych wartości, np.
    - enum dni_tygodnia{ Poniedzialek, Wtorek, Sroda, ...}
  - każda zmienna zdefiniowana za pomocą *enum* jest nazwaną stałą, która zamieniana jest na liczbę całkowitą
    - domyślnie jest to kolejna liczba – pozycja symbolu w wyliczeniu:
      - Poniedziałek == 0, Wtorek == 1, itd.
    - można jawnie nadać wartości podstawianych liczb
      - enum dni_tygodnia{ Poniedzialek=1, Wtorek=2, Sroda=3, ...}
      - Poniedziałek == 1, Wtorek == 2, itd.
    - można także podać konkretną wartość dla wybranych stałych, i wtedy kolejne otrzymają kolejne liczby całkowite
      - enum dni_tygodnia{ Poniedzialek=1, Wtorek, Sroda, ...}
      - Poniedziałek == 1, Wtorek == 2, itd.

Wyliczenia

- Użycie typu wyliczeniowego jest alternatywną do *#define* metodą definiowania nazwanych stałych w programach C
  - zaletą stałych *enum* jest to, że ich symboliczne nazwy pozostają w programie po kompilacji
- Zmienne *enum* często stosuje się w instrukcji wyboru *switch*:

```
int dzien_tygodnia = 2;  // konstrukcja switch używa typu całkowitego!
 // enum dni_tygodnia dzien_tygodnia = Sroda; // = 3;
 switch(dzien_tygodnia){
   case Poniedzialek:
       printf("Poniedzialek\n"); harmonogram_poniedzialkowy();
       break;
   case Wtorek:
       printf("Wtorek\n"); harmonogram_wtorkowy();
       break;
    // itd.
 }
```

Pola bitowe

- C umożliwia definiowanie zmiennych o rozmiarach mniejszych niż typy całkowite, aż do rozmiaru pojedynczego bitu
  - zmienne takie nazywane są polami bitowymi
- Zmienne takie pakowane są w struktury, które mogą być przechowywane jako liczby typów całkowitych, z poszczególnymi polami bitowymi jako kolejnymi bitami liczby

```
struct { // struktura zawiera własności pojedynczej liczby
  unsigned int is_positive    :1; // po dwukropku rozmiar – liczba bitów
  unsigned int is_even        :1;
  unsigned int is_power_10    :1;
} flags; // pojedyncza zmienna zawierająca pola bitowe
```

- Dostęp do poszczególnych pól bitowych jest realizowany za pomocą standardowej składni struktur

```
flags.is_positive = 1;
```

Operatory bitowe

- Operatory bitowe
  - dostępne dla argumentów typów: *char*, *short*, *int*, oraz *long*
    - tak ze znakiem, jak i bez znaku
  - dokonują operacji na reprezentacji bitowej liczb
    - a & b – dla każdej pozycji bitowej operacja koniunkcji (AND)
    - a | b – dla każdej pozycji bitowej operacja alternatywy (OR)
    - a ^ b – dla każdej pozycji operacja różnicy symetrycznej (XOR)
      - uzyskanie zera: a ^ a
    - a << b – przesunięcie wartości bitów w a o b pozycji w lewo
      - a << 2 – mnożenie a przez 4 (zwalniane bity są zerowane)
    - a >> b – przesunięcie wartości bitów w a o b pozycji w prawo
      - a >> 1 – dzielenie przez 2 (może zależeć od traktowania znaku)
    - ~a – negacja bitowa, zamiana 0 na 1 i odwrotnie
      - a & ~077 – zerowanie wybranych bitów a

Priorytety operatorów

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right