Niestandardowe typy danych

Przemysław Gawroński
D-10, p. 234

Wykład 14

25 stycznia 2021

Outline

Niestandardowe typy danych

Sposoby definiowania w języku **C** niestandardowych typów danych:

- **struktury** - grupy zmiennych, do których można odwoływać się za pośrednictwem jednej nazwy,

Niestandardowe typy danych

Sposoby definiowania w języku **C** niestandardowych typów danych:

- **struktury** - grupy zmiennych, do których można odwoływać się za pośrednictwem jednej nazwy,
- **unie** - pozwalają definiować kilka typów dla jednego obszaru pamięci,

Niestandardowe typy danych

Sposoby definiowania w języku **C** niestandardowych typów danych:

- **struktury** - grupy zmiennych, do których można odwoływać się za pośrednictwem jednej nazwy,
- **unie** - pozwalają definiować kilka typów dla jednego obszaru pamięci,
- **pola bitowe** - elementy struktur, pozwalają operować na poszczególnych bitach,

Niestandardowe typy danych

Sposoby definiowania w języku **C** niestandardowych typów danych:

- **struktury** - grupy zmiennych, do których można odwoływać się za pośrednictwem jednej nazwy,
- **unie** - pozwalają definiować kilka typów dla jednego obszaru pamięci,
- **pola bitowe** - elementy struktur, pozwalają operować na poszczególnych bitach,
- **wyliczenia** - listy stałych wartości o określonych nazwach,

Niestandardowe typy danych

Sposoby definiowania w języku **C** niestandardowych typów danych:

- **struktury** - grupy zmiennych, do których można odwoływać się za pośrednictwem jednej nazwy,
- **unie** - pozwalają definiować kilka typów dla jednego obszaru pamięci,
- **pola bitowe** - elementy struktur, pozwalają operować na poszczególnych bitach,
- **wyliczenia** - listy stałych wartości o określonych nazwach,
- **typedef** - definiuje nową nazwę dla istniejącego typu.

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.
- ```
  struct {
      int   id_nmbr;
      char  name[40];
      int   on_shelf;
  } item1, item2;
  ```
  ```

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.
- ```
  struct {
      int   id_nmbr;
      char  name[40];
      int   on_shelf;
  } item1, item2;
  ```
- **Struktura** to zbiór pól (wartości/zmiennych) różnego typu.
  ```

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.
- ```
  struct {
      int   id_nmbr;
      char  name[40];
      int   on_shelf;
  } item1, item2;
  ```
- **Struktura** to zbiór pól (wartości/zmiennych) różnego typu.
- Każde **pole** struktury ma nazwę, której używamy by do niego się odwołać.
  ```

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.
- ```
  struct {
      int   id_nmbr;
      char  name[40];
      int   on_shelf;
  } item1, item2;
  ```
- **Struktura** to zbiór pól (wartości/zmiennych) różnego typu.
- Każde **pole** struktury ma nazwę, której używamy by do niego się odwołać.
- Pola struktury przechowywane są w pamięci w porządku w jakim zostały zadeklarowane.
  ```

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.
- ```
  struct {
      int   id_nmbr;
      char  name[40];
      int   on_shelf;
  } item1, item2;
  ```
- **Struktura** to zbiór pól (wartości/zmiennych) różnego typu.
- Każde **pole** struktury ma nazwę, której używamy by do niego się odwołać.
- Pola struktury przechowywane są w pamięci w porządku w jakim zostały zadeklarowane.
- Deklaracja zmiennej strukturalnej ma taką samą formę jak deklaracje zmiennych prostych w języku **C**:
  ```

Struktury

- **Struktura** to wygodny sposób na przechowywanie pokrewnych informacji, do których można odwołać się za pomocą jednej nazwy.
- ```
  struct {
      int   id_nmbr;
      char  name[40];
      int   on_shelf;
  } item1, item2;
  ```
- **Struktura** to zbiór pól (wartości/zmiennych) różnego typu.
- Każde **pole** struktury ma nazwę, której używamy by do niego się odwołać.
- Pola struktury przechowywane są w pamięci w porządku w jakim zostały zadeklarowane.
- Deklaracja zmiennej strukturalnej ma taką samą formę jak deklaracje zmiennych prostych w języku **C**:
  Wyrażenie **struct** {**...** } określa typ, natomiast **item1** i **item2** są zmiennymi typu strukturalnego.
  ```

Struktury

- Każda struktura reprezentuje nowy zakres widzialności zmiennych.

Struktury

- Każda struktura reprezentuje nowy zakres widzialności zmiennych.
- Każda struktura reprezentuje oddzielną przestrzeń nazw.

Struktury

- Każda struktura reprezentuje nowy zakres widzialności zmiennych.
- Każda struktura reprezentuje oddzielną przestrzeń nazw.

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item1, item2;
```

```
struct {
  int  id_number;
  char name[40];
  int  age;
} person1, person2;
```

Struktury

- Każda struktura reprezentuje nowy zakres widzialności zmiennych.
- Każda struktura reprezentuje oddzielną przestrzeń nazw.

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item1, item2;
```

```
struct {
  int  id_number;
  char name[40];
  int  age;
} person1, person2;
```

- Pola **id_number** i **name** w **item1** i **item2** nie powodują konfliktu nazw ze polami **id_number** i **name** w **person1** i **person2**.

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:
- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100},
     item2 = {914, "Printer cable", 57};
  ```
  ```

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:
- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100},
     item2 = {914, "Printer cable", 57};
  ```
- Wartości w liście inicjacyjnej muszą pojawiać się w tej samej kolejności w jakiej występują pola w strukturze.
  ```

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:
- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100},
     item2 = {914, "Printer cable", 57};
  ```
- Wartości w liście inicjacyjnej muszą pojawiać się w tej samej kolejności w jakiej występują pola w strukturze.
- Lista inicjacyjna może mieć mniej elementów niż inicjowana struktura, każdy „pominięty” element otrzymuje **0** jako wartość początkową.
  ```

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:
- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100},
     item2 = {914, "Printer cable", 57};
  ```
- Wartości w liście inicjacyjnej muszą pojawiać się w tej samej kolejności w jakiej występują pola w strukturze.
- Lista inicjacyjna może mieć mniej elementów niż inicjowana struktura, każdy „pominięty” element otrzymuje **0** jako wartość początkową.
- W standardzie **C99** można użyć inicjatorów desygnowanych:
  ```

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:
- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100},
     item2 = {914, "Printer cable", 57};
  ```
- Wartości w liście inicjacyjnej muszą pojawiać się w tej samej kolejności w jakiej występują pola w strukturze.
- Lista inicjacyjna może mieć mniej elementów niż inicjowana struktura, każdy „pominięty” element otrzymuje **0** jako wartość początkową.
- W standardzie **C99** można użyć inicjatorów desygnowanych:
  ```
  {.id = 528, .name = "Hard drive", .os = 100}
  ```
  ```

Inicjalizacja zmiennych strukturalnych

- Zmienna strukturalna może być zainicjalizowana w momencie deklaracji.
- Aby zainicjować zmienną strukturalną, przygotowujemy listę wartości, które mają być przechowywane w strukturze i zamykamy je w nawiasy klamrowe:
- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100},
     item2 = {914, "Printer cable", 57};
  ```
- Wartości w liście inicjacyjnej muszą pojawiać się w tej samej kolejności w jakiej występują pola w strukturze.
- Lista inicjacyjna może mieć mniej elementów niż inicjowana struktura, każdy „pominięty” element otrzymuje **0** jako wartość początkową.
- W standardzie **C99** można użyć inicjatorów desygnowanych:
  ```
  {.id = 528, .name = "Hard drive", .os = 100}
  ```
- Inicjatory desygnowane mogą być umieszczone w dowolnej kolejności.
  ```

Operacje na strukturach

```
struct {
   int id; char name[40]; int os;
} item1 = {528, "Hard drive", 100};
```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100};
  ```
- Aby uzyskać dostęp do pola struktury, najpierw piszemy nazwę struktury, potem kropkę, a następnie nazwę pola.
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100};
  ```
- Aby uzyskać dostęp do pola struktury, najpierw piszemy nazwę struktury, potem kropkę, a następnie nazwę pola.
- Na przykład:
  ```
  printf("Item id: %d\n", item1.id);
  printf("Item name: %s\n", item1.name);
  printf("Item quantity: %d\n", item1.os);
  ```
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100};
  ```
- Aby uzyskać dostęp do pola struktury, najpierw piszemy nazwę struktury, potem kropkę, a następnie nazwę pola.
- Na przykład:
  ```
  printf("Item id: %d\n", item1.id);
  printf("Item name: %s\n", item1.name);
  printf("Item quantity: %d\n", item1.os);
  ```
- Pola struktury są **l-wartościami**, więc mogą pojawiać się po lewej stronie wyrażeń lub jako operand w wyrażeniu inkrementacji lub dekrementacji:
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100};
  ```
- Aby uzyskać dostęp do pola struktury, najpierw piszemy nazwę struktury, potem kropkę, a następnie nazwę pola.
- Na przykład:
  ```
  printf("Item id: %d\n", item1.id);
  printf("Item name: %s\n", item1.name);
  printf("Item quantity: %d\n", item1.os);
  ```
- Pola struktury są **l-wartościami**, więc mogą pojawiać się po lewej stronie wyrażeń lub jako operand w wyrażeniu inkrementacji lub dekrementacji:
- Na przykład:
  ```
  item1.id = 767;
  item1.os++;
  scanf("%d", &item1.id);
  ```
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100}, item2;
  ```
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100}, item2;
  ```
- Operacja przypisania - przykład:
  ```
  item2 = item1;
  ```
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100}, item2;
  ```
- Operacja przypisania - przykład:
  ```
  item2 = item1;
  ```
- W wyniku tej instrukcji przypisania pole **item1.id** zostanie skopiowane do **item2.id**, pole **item1.name** zostanie skopiowane do **item2.name** i tak dalej.
  ```

Operacje na strukturach

- ```
  struct {
      int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100}, item2;
  ```
- Operacja przypisania - przykład:
  ```
  item2 = item1;
  ```
- W wyniku tej instrukcji przypisania pole **item1.id** zostanie skopiowane do **item2.id**, pole **item1.name** zostanie skopiowane do **item2.name** i tak dalej.
- Tablic nie można bezpośrednio kopiować za pomocą operatora $=$.
  ```

Operacje na strukturach

- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100}, item2;
  ```
- Operacja przypisania - przykład:
  ```
  item2 = item1;
  ```
- W wyniku tej instrukcji przypisania pole **item1.id** zostanie skopiowane do **item2.id**, pole **item1.name** zostanie skopiowane do **item2.name** i tak dalej.
- Tablic nie można bezpośrednio kopiować za pomocą operatora $=$.
- Można stworzyć strukturę, która opakuje tablicę, aby użyć $=$ do skopiowania tablicy. :
  ```
  struct { int arr[10]; } t1, t2;
  ...
  t1 = t2;
  ```
  ```

Operacje na strukturach

- ```
  struct {
     int id; char name[40]; int os;
  } item1 = {528, "Hard drive", 100}, item2;
  ```
- Operacja przypisania - przykład:
  ```
  item2 = item1;
  ```
- W wyniku tej instrukcji przypisania pole **item1.id** zostanie skopiowane do **item2.id**, pole **item1.name** zostanie skopiowane do **item2.name** i tak dalej.
- Tablic nie można bezpośrednio kopiować za pomocą operatora $=$.
- Można stworzyć strukturę, która opakuje tablicę, aby użyć $=$ do skopiowania tablicy. :
  ```
  struct { int arr[10]; } t1, t2;
  ...
  t1 = t2;
  ```
- Operator $=$ może być używany tylko pomiędzy strukturami kompatybilnych typów.
  ```

Typ strukturalny

- Stwórzmy dwie zmienne strukturalne:

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item1;
```

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item2;
```

Typ strukturalny

- Stwórzmy dwie zmienne strukturalne:

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item1;
```

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item2;
```

- Zgodnie z regułami **C** zmienne **item1** oraz **item2** nie mają kompatybilnych typów.

Typ strukturalny

- Stwórzmy dwie zmienne strukturalne:

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item1;
```

```
struct {
  int  id_number;
  char name[40];
  int  on_shelf;
} item2;
```

- Zgodnie z regułami **C** zmienne **item1** oraz **item2** nie mają kompatybilnych typów.
- Zmienna **item1** nie może być przypisana zmiennej **item2** i vice versa.

Typ strukturalny

- W języku **C** możemy zdefiniować nazwę, która reprezentuje **typ** strukturalny, a nie tylko konkretną **zmienną** strukturalną.

Typ strukturalny

- W języku **C** możemy zdefiniować nazwę, która reprezentuje **typ** strukturalny, a nie tylko konkretną **zmienną** strukturalną.
- **Etykieta** struktury to nazwa używana do identyfikacji określonego rodzaju struktury.

Typ strukturalny

- W języku **C** możemy zdefiniować nazwę, która reprezentuje **typ** strukturalny, a nie tylko konkretną **zmienną** strukturalną.
- **Etykieta** struktury to nazwa używana do identyfikacji określonego rodzaju struktury.
- Deklaracja struktury o etykiecie **item**:

```
struct item {
   int   id_number;
   char  name[40];
   int   on_shelf;
}; //prosze nie zapominac o sredniku
```

Typ strukturalny

- W języku **C** możemy zdefiniować nazwę, która reprezentuje **typ** strukturalny, a nie tylko konkretną **zmienną** strukturalną.
- **Etykieta** struktury to nazwa używana do identyfikacji określonego rodzaju struktury.
- Deklaracja struktury o etykiecie **item**:

```
struct item {
  int  id_number;
  char name[40];
  int  on_shelf;
}; //prosze nie zapominac o sredniku
```

- Deklaracja struktury z etykietą **item** może posłużyć do deklaracji zmiennych strukturalnych:

```
struct item it1, it2;
```

Typ strukturalny

- Deklaracja typu strukturalnego może być łączona z deklaracją zmiennych strukturalnych:

```
struct item {
   int  id_number;
   char name[40];
   int  on_shelf;
} it1, it2;
...
struct item it3 = {528, "Hard drive", 100};
it2 = it3; // ten sam typ
```

Typ strukturalny

- Deklaracja typu struktur to szablon, którego można używać do tworzenia kolejnych instancji struktury.

Typ strukturalny

- Deklaracja typu struktur to szablon, którego można używać do tworzenia kolejnych instancji struktury.
- Ogólna postać deklaracji typu strukturalnego :

```
struct etykieta {
  typ pole_struktury;
} ;
```

Typ strukturalny

- Deklaracja typu struktur to szablon, którego można używać do tworzenia kolejnych instancji struktury.
- Ogólna postać deklaracji typu strukturalnego :

```
struct etykieta {
  typ pole_struktury;
} ;
```

- Po deklaracji typu strukturalnego nie powstaje jeszcze żadna zmienna. Zadeklarowany został jedynie nowy złożony typ danych, a nie zmienna.

Typ strukturalny

- Deklaracja typu struktur to szablon, którego można używać do tworzenia kolejnych instancji struktury.
- Ogólna postać deklaracji typu strukturalnego :

```
struct etykieta {
   typ pole_struktury;
} ;
```

- Po deklaracji typu strukturalnego nie powstaje jeszcze żadna zmienna. Zadeklarowany został jedynie nowy złożony typ danych, a nie zmienna.
- W trakcie deklaracji zmiennej strukturalnej kompilator automatycznie rezerwuje odpowiednią ilość pamięci na wszystkie pola składowe struktury.

Przekazywanie pól struktury do funkcji

- Przekazując pole struktury, przekazujemy wartość zmiennej i nie ma znaczenia, że wartość pochodzi ze składowej struktury.

Przekazywanie pól struktury do funkcji

- Przekazując pole struktury, przekazujemy wartość zmiennej i nie ma znaczenia, że wartość pochodzi ze składowej struktury.
- Przykład.

  Deklaracja funkcji, do której przekazujemy wartość całkowitą: **int is_on_shelf(int);**

  Wywołanie:**int qi = is_on_shelf(item1.on_shelf);**

  Pole **item1.on_shelf** jest typu **int**.

Przekazywanie pól struktury do funkcji

- Przekazując pole struktury, przekazujemy wartość zmiennej i nie ma znaczenia, że wartość pochodzi ze składowej struktury.
- Przykład.

  Deklaracja funkcji, do której przekazujemy wartość całkowitą: **int is_on_shelf(int);**

  Wywołanie:**int qi = is_on_shelf(item1.on_shelf);**

  Pole **item1.on_shelf** jest typu **int**.
- Aby przekazać adres składowej struktury należy skorzystać z operatora **&**.

Przekazywanie pól struktury do funkcji

- Przekazując pole struktury, przekazujemy wartość zmiennej i nie ma znaczenia, że wartość pochodzi ze składowej struktury.
- Przykład.

  Deklaracja funkcji, do której przekazujemy wartość całkowitą: **int is_on_shelf(int);**

  Wywołanie:**int qi = is_on_shelf(item1.on_shelf);**

  Pole **item1.on_shelf** jest typu **int**.
- Aby przekazać adres składowej struktury należy skorzystać z operatora **&**.
- Przykład.

  Deklaracja:**int put_on_shelf(int*);**

  Wywołanie:**int iq = put_on_shelf(&item1.on_shelf);**

Przekazywanie pól struktury do funkcji

- Przekazując pole struktury, przekazujemy wartość zmiennej i nie ma znaczenia, że wartość pochodzi ze składowej struktury.
- Przykład.

  Deklaracja funkcji, do której przekazujemy wartość całkowitą: **int is_on_shelf(int);**

  Wywołanie:**int qi = is_on_shelf(item1.on_shelf);**

  Pole **item1.on_shelf** jest typu **int**.
- Aby przekazać adres składowej struktury należy skorzystać z operatora **&**.
- Przykład.

  Deklaracja:**int put_on_shelf(int*);**

  Wywołanie:**int iq = put_on_shelf(&item1.on_shelf);**
- Przykład.

  Wywołanie:**scanf("%d",&item1.on_shelf);**

Przekazywanie struktur do funkcji

- Funkcje mogą mieć struktury jako argumenty.

Przekazywanie struktur do funkcji

- Funkcje mogą mieć struktury jako argumenty.
- Przykład.

```
struct item{
   int id; char name[40]; int os;
}; //deklaracja struktury

void print_it (struct item t){
   printf("Item nmbr: %d\n", t.id);
   printf("Item name: %s\n", t.name);
   printf("On shelf: %d\n", t.os);
} //definicja funkcji
...
struct item it = {914, "Printer cable", 57};
print_it(it); //wywolanie
```

Przekazywanie struktur do funkcji

- Funkcje mogą mieć struktury jako argumenty.
- Przykład.

```
struct item{
   int id; char name[40]; int os;
}; //deklaracja struktury

void print_it (struct item t){
   printf("Item nmbr: %d\n", t.id);
   printf("Item name: %s\n", t.name);
   printf("On shelf: %d\n", t.os);
} //definicja funkcji
...
struct item it = {914, "Printer cable", 57};
print_it(it); //wywolanie
```

- W C99 można tworzyć struktury „w locie”:

Przekazywanie struktur do funkcji

- Funkcje mogą mieć struktury jako argumenty.
- Przykład.

```
struct item{
   int id; char name[40]; int os;
}; //deklaracja struktury

void print_it (struct item t){
   printf("Item␣nmbr:␣%d\n", t.id);
   printf("Item␣name:␣%s\n", t.name);
   printf("On␣shelf:␣%d\n", t.os);
} //definicja funkcji
...
struct item it = {914, "Printer␣cable", 57};
print_it(it); //wywolanie
```

- W C99 można tworzyć struktury „w locie”:

```
print_it((struct item) {914,"Prntr␣cbl",57});
```

Funkcja zwracająca strukturę

- Funkcje mogą zwracać struktury za pomocą instrukcji **return**.

```
struct item{
   int id; char name[40]; int os;
};// deklaracja struktury
struct item build_it(int n, const char *name,
    int ns){
   struct item temp; //zmienna tymczasowa

   temp.id = n; //blok przypisan
   strcpy(temp.name, name);
   temp.os = ns;
   return temp; //zwracamy strukture przez
    wartosc
}
...
struct item it1 = build_it(914, "Printer cable
    ", 57); //wywolanie i przypisanie
```

Struktury i funkcje

- Przekazywanie struktury do funkcji i zwracanie struktury z funkcji wymaga wykonania przez program kopii wszystkich pól struktury.

Struktury i funkcje

- Przekazywanie struktury do funkcji i zwracanie struktury z funkcji wymaga wykonania przez program kopii wszystkich pól struktury.
- Przekazywanie i zwracanie struktury przez wartość wiąże się ze znacznym spadkiem wydajności programu, szczególnie jeśli struktura jest duża.

Struktury i funkcje

- Przekazywanie struktury do funkcji i zwracanie struktury z funkcji wymaga wykonania przez program kopii wszystkich pól struktury.
- Przekazywanie i zwracanie struktury przez wartość wiąże się ze znacznym spadkiem wydajności programu, szczególnie jeśli struktura jest duża.
- Często wskazane jest operowanie na wskaźnikach do struktur niż bezpośrednio na strukturach, zarówno gdy przekazujemy strukturę do funkcji, jak i w momencie, gdy ją zwracamy.

Struktury i funkcje

- Przekazywanie struktury do funkcji i zwracanie struktury z funkcji wymaga wykonania przez program kopii wszystkich pól struktury.
- Przekazywanie i zwracanie struktury przez wartość wiąże się ze znacznym spadkiem wydajności programu, szczególnie jeśli struktura jest duża.
- Często wskazane jest operowanie na wskaźnikach do struktur niż bezpośrednio na strukturach, zarówno gdy przekazujemy strukturę do funkcji, jak i w momencie, gdy ją zwracamy.
- Dla przykładu, w bibliotece standardowej języka **C** operacje na plikach wykonywane są z użyciem wskaźnika do typu **FILE**.

Zagnieżdżanie struktur

- Struktury można zagnieżdżać.

Zagnieżdżanie struktur

- Struktury można zagnieżdżać.
- Zadeklarujmy strukturę:

```
struct person_name{
   char first[20+1];
   char last[20+1];   };
```

Zagnieżdżanie struktur

- Struktury można zagnieżdżać.
- Zadeklarujmy strukturę:

```
struct person_name{
   char first[20+1];
   char last[20+1];   };
```

- Możemy teraz użyć struktury **person_name** jako pola wewnątrz innej struktury:

Zagnieżdżanie struktur

- Struktury można zagnieżdżać.
- Zadeklarujmy strukturę:

```
struct person_name{
   char first[20+1];
   char last[20+1];   };
```

- Możemy teraz użyć struktury **person_name** jako pola wewnątrz innej struktury:

```
struct student{
   struct person_name name;
   int id, age;
} s1, s2;
```

Zagnieżdżanie struktur

- Struktury można zagnieżdżać.
- Zadeklarujmy strukturę:

```
struct person_name{
   char first[20+1];
   char last[20+1];   };
```

- Możemy teraz użyć struktury **person_name** jako pola wewnątrz innej struktury:

```
struct student{
   struct person_name name;
   int id, age;
} s1, s2;
```

- Przykład odwołania się do pola zagnieżdzonej struktury:

```
strcpy(s1.name.first, "Fred");
```

Zagnieżdżanie struktur

- Struktury można zagnieżdżać.
- Zadeklarujmy strukturę:

```
struct person_name{
   char first[20+1];
   char last[20+1];   };
```

- Możemy teraz użyć struktury **person_name** jako pola wewnątrz innej struktury:

```
struct student{
   struct person_name name;
   int id, age;
} s1, s2;
```

- Przykład odwołania się do pola zagnieżdzonej struktury:

```
strcpy(s1.name.first, "Fred");
```

- **s1** to zmienna typu **struct student** posiada pole **name** typu **struct person_name**, a wewnątrz znajduje się pole **first**, które jest łańcuchem.

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.
- Tablicę struktur można potraktować jako prostą bazę danych:

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.
- Tablicę struktur można potraktować jako prostą bazę danych:

```
struct item{
  int id; char name[40]; int os;
};
struct item inventory[100];
```

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.
- Tablicę struktur można potraktować jako prostą bazę danych:

```
struct item{
  int id; char name[40]; int os;
};
struct item inventory[100];
```

- Aby uzyskać dostęp elementów tablicy używamy operatora nawiasowego:

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.
- Tablicę struktur można potraktować jako prostą bazę danych:

```
struct item{
  int id; char name[40]; int os;
};
struct item inventory[100];
```

- Aby uzyskać dostęp elementów tablicy używamy operatora nawiasowego:

```
for(int i=0;i<100;i++)
  print_it(inventory[i]);
```

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.
- Tablicę struktur można potraktować jako prostą bazę danych:

```
struct item{
  int id; char name[40]; int os;
};
struct item inventory[100];
```

- Aby uzyskać dostęp elementów tablicy używamy operatora nawiasowego:

```
for(int i=0;i<100;i++)
  print_it(inventory[i]);
```

- Dostęp do pola struktury **item** wymaga kombinacji operatora nawiasowego i operatora dostępu do pola struktury.

Tablice struktur

- Struktury i tablice można łączyć w zasadzie bez ograniczeń.
- Jedną z najczęstszych kombinacji tablic i struktur jest tablica, której elementami są struktury.
- Tablicę struktur można potraktować jako prostą bazę danych:

```
struct item{
  int id; char name[40]; int os;
};
struct item inventory[100];
```

- Aby uzyskać dostęp elementów tablicy używamy operatora nawiasowego:

```
for(int i=0;i<100;i++)
  print_it(inventory[i]);
```

- Dostęp do pola struktury **item** wymaga kombinacji operatora nawiasowego i operatora dostępu do pola struktury.

```
inventory[7].id = 883;
inventory[20].name[0] = '\0';
```

Tablice struktur

- Inicjalizacja tablicy struktur odbywa się w taki sam sposób, jak inicjalizacja tablicy wielowymiarowej.

Tablice struktur

- Inicjalizacja tablicy struktur odbywa się w taki sam sposób, jak inicjalizacja tablicy wielowymiarowej.
- Każda struktura ma własną listę inicjacyjną zamkniętą nawiasami klamrowymi.

Tablice struktur

- Inicjalizacja tablicy struktur odbywa się w taki sam sposób, jak inicjalizacja tablicy wielowymiarowej.
- Każda struktura ma własną listę inicjacyjną zamkniętą nawiasami klamrowymi.
- Zadeklarujmy typ strukturalny:

```
struct critter{
   const char *name;
   const char *species;
};
```

Tablice struktur

- Inicjalizacja tablicy struktur odbywa się w taki sam sposób, jak inicjalizacja tablicy wielowymiarowej.
- Każda struktura ma własną listę inicjacyjną zamkniętą nawiasami klamrowymi.
- Zadeklarujmy typ strukturalny:

```
struct critter{
   const char *name;
   const char *species;
};
```

- Inicjalizacja będzie wyglądać następująco:

```
struct critter muppets[] = { {"Kermit", "frog"
   }, {"Piggy", "pig"} };
```

Tablice struktur

- Inicjalizacja tablicy struktur odbywa się w taki sam sposób, jak inicjalizacja tablicy wielowymiarowej.
- Każda struktura ma własną listę inicjacyjną zamkniętą nawiasami klamrowymi.
- Zadeklarujmy typ strukturalny:

```
struct critter{
   const char *name;
   const char *species;
};
```

- Inicjalizacja będzie wyglądać następująco:

```
struct critter muppets[] = { {"Kermit", "frog"
   }, {"Piggy", "pig"} };
```

- Wewnętrzne nawiasy klamrowe są opcjonalne.

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;
- Wskaźniki do struktur służą do:

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;
- Wskaźniki do struktur służą do:
  - przekazywania struktur do funkcji przez referencję,

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;
- Wskaźniki do struktur służą do:
  - przekazywania struktur do funkcji przez referencję,
  - tworzenia dynamicznych struktur danych.

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;
- Wskaźniki do struktur służą do:
  - przekazywania struktur do funkcji przez referencję,
  - tworzenia dynamicznych struktur danych.
- Przekazując strukturę do funkcji tworzona jest jej kopia na stosie.

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;
- Wskaźniki do struktur służą do:
  - przekazywania struktur do funkcji przez referencję,
  - tworzenia dynamicznych struktur danych.
- Przekazując strukturę do funkcji tworzona jest jej kopia na stosie.
- Gdy funkcja otrzymuje wskaźnik do struktury tylko adres struktury umieszczany jest na stosie.

Wskaźniki do struktur

- Ogólna postać deklaracji wskaźnika do struktury:
  struct etykieta *nazwa-zmiennej;
- Wskaźniki do struktur służą do:
  - przekazywania struktur do funkcji przez referencję,
  - tworzenia dynamicznych struktur danych.
- Przekazując strukturę do funkcji tworzona jest jej kopia na stosie.
- Gdy funkcja otrzymuje wskaźnik do struktury tylko adres struktury umieszczany jest na stosie.
- Funkcja może efektywniej operować strukturze, za pomocą wskaźnika do struktury, np. **FILE ***.

Wskaźniki do struktur

- Przykład.

```
struct person{
   char first[SIZE+1];
   char last[SIZE+1];
};

...

struct person p1 = { "Fred", "Smith" };
struct person *ptp1 = &p1;

printf("%s, %s\n", ptp1->first, (*ptp1).last);
```

Tablice struktur

```
struct critter{
   const char *name; const char *species;
};

struct critter muppets[] = { {"Kermit", "frog"
   }, {"Piggy", "pig"} };
```

Tablice struktur

- ```
  struct critter{
    const char *name; const char *species;
  };

  struct critter muppets[] = { {"Kermit", "frog"
     }, {"Piggy", "pig"} };
  ```
- ```
  void print_critter(const struct critter *c){
    printf("%s, the %s\n", c->name, c->species);
  }
  ```

Tablice struktur

- ```
  struct critter{
    const char *name; const char *species;
  };

  struct critter muppets[] = { {"Kermit", "frog"
     }, {"Piggy", "pig"} };
  ```
- ```
  void print_critter(const struct critter *c){
    printf("%s, the %s\n", c->name, c->species);
  }
  ```
- ```
  int critter_cmp(const void *c1, const void *c2
     ){
    return strcmp( ( (struct critter *)c1 )->
     name, ( (struct critter *)c2 )->name );
  }
  ```
  ```

Tablice struktur - qsort, bsearch

- ```
  qsort(muppets, sizeof(muppets) / sizeof(struct
      critter), sizeof(struct critter),
    critter_cmp);
  ```
  ```

Tablice struktur - qsort, bsearch

- ```
  qsort(muppets, sizeof(muppets) / sizeof(struct
      critter), sizeof(struct critter),
     critter_cmp);
  ```
- ```
  struct critter *find_critter (const char *name
     , struct critter *muppets, int cnt){

    struct critter target;
    target.name = name;

    return bsearch(&target, muppets, cnt, sizeof
     (struct critter), critter_cmp);
  }
  ```

Tablice struktur - qsort, bsearch

```
struct critter *result = find_critter("Kermit"
    , muppets, sizeof(muppets) / sizeof(struct
    critter));

if(result)
  print_critter (result);
```

Wyliczenia

- Wyliczenie to zbiór stałych całkowitych o określonych nazwach.

Wyliczenia

- Wyliczenie to zbiór stałych całkowitych o określonych nazwach.
- Ogólna postać wyliczenia:

```
enum [etykieta] {lista wartosci} [zmienne];
```

Wyliczenia

- Wyliczenie to zbiór stałych całkowitych o określonych nazwach.
- Ogólna postać wyliczenia:

```
enum [etykieta] {lista wartosci} [zmienne];
```

- Przykład:

```
enum moneta {penny, nickel, dime, quarter,
   half_dolar, dolar};

enum moneta x;
...
x = dime;

if (x == quarter) ....
```

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.
- Wyliczeń można używać tam gdzie można użyć liczby całkowitej.

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.
- Wyliczeń można używać tam gdzie można użyć liczby całkowitej.
- Każdy symbol otrzymuje wartość o jeden większą od wartości poprzedniego symbolu. Wartość pierwszego symbolu to zero.

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.
- Wyliczeń można używać tam gdzie można użyć liczby całkowitej.
- Każdy symbol otrzymuje wartość o jeden większą od wartości poprzedniego symbolu. Wartość pierwszego symbolu to zero.
- Można również określić wartości jednego lub kilku symboli.

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.
- Wyliczeń można używać tam gdzie można użyć liczby całkowitej.
- Każdy symbol otrzymuje wartość o jeden większą od wartości poprzedniego symbolu. Wartość pierwszego symbolu to zero.
- Można również określić wartości jednego lub kilku symboli.
- Przykłady:

```
enum mntA {penny = 1, nickel = 2, dime = 3,
   quarter = 4, half_dolar = 5, dolar = 6};
```

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.
- Wyliczeń można używać tam gdzie można użyć liczby całkowitej.
- Każdy symbol otrzymuje wartość o jeden większą od wartości poprzedniego symbolu. Wartość pierwszego symbolu to zero.
- Można również określić wartości jednego lub kilku symboli.
- Przykłady:

```
enum mntA {penny = 1, nickel = 2, dime = 3,
   quarter = 4, half_dolar = 5, dolar = 6};

enum mntB {penny = 11, nickel = 22, dime = 13,
    quarter = 4, half_dolar = 15, dolar = 66};
```

Wyliczenia

- Każdy symbol wyliczenia reprezentuje stałą całkowitą.
- Wyliczeń można używać tam gdzie można użyć liczby całkowitej.
- Każdy symbol otrzymuje wartość o jeden większą od wartości poprzedniego symbolu. Wartość pierwszego symbolu to zero.
- Można również określić wartości jednego lub kilku symboli.
- Przykłady:

```
enum mntA {penny = 1, nickel = 2, dime = 3,
   quarter = 4, half_dolar = 5, dolar = 6};

enum mntB {penny = 11, nickel = 22, dime = 13,
    quarter = 4, half_dolar = 15, dolar = 66};

enum mntC {penny, nickel = 7, dime, quarter,
   half_dolar, dolar = 15};
```

Wyliczenia

Przykład:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS; // i = 1
s = 0;        // s = 0 CLUBS
s++;          // s = 1 DIAMONDS
i = s + 2;    // i = 3
```

typedef

- Używając słowa kluczowego **typedef** można zdefiniować nowe nazwy typów.

typedef

- Używając słowa kluczowego **typedef** można zdefiniować nowe nazwy typów.
- Ogólna postać instrukcji:

```
typedef typ nowa_nazwa_typu;
```

typedef

- Używając słowa kluczowego **typedef** można zdefiniować nowe nazwy typów.
- Ogólna postać instrukcji:

```
typedef typ nowa_nazwa_typu;
```

- Przykład:

```
typedef float saldo;
saldo wrzesien;

typedef struct telement{
   int wiek;
   struct telement *nast;
} element;

element *root; //struct telement *root;
```

typedef

- Rozważmy poniższą deklarację:

```
int *(*x[10])(void);
```

typedef

- Rozważmy poniższą deklarację:

```
int *(*x[10])(void);
```

- Deklarator, który zaczyna się od $*$ oznacza wskaźnik. Deklarator, który kończy się $[]$ oznacza tablicę. Deklarator, który kończy się $()$ oznacza funkcję.

typedef

- Rozważmy poniższą deklarację:

```
int *(*x[10])(void);
```

- Deklarator, który zaczyna się od $*$ oznacza wskaźnik. Deklarator, który kończy się $[]$ oznacza tablicę. Deklarator, który kończy się $()$ oznacza funkcję.
- Deklaracje odczytujemy od środka.

typedef

- Rozważmy poniższą deklarację:

```
int *(*x[10])(void);
```

- Deklarator, który zaczyna się od $*$ oznacza wskaźnik. Deklarator, który kończy się $[]$ oznacza tablicę. Deklarator, który kończy się $()$ oznacza funkcję.
- Deklaracje odczytujemy od środka.
- Jeżeli $*$ poprzedza identyfikator a $[]$ następuje po nim, to identyfikator reprezentuje tablicę. Jeżeli $*$ poprzedza identyfikator a $()$ następuje po nim, to identyfikator reprezentuje funkcję.

typedef

- Rozważmy poniższą deklarację:

```
int *(*x[10])(void);
```

- Deklarator, który zaczyna się od $*$ oznacza wskaźnik. Deklarator, który kończy się $[]$ oznacza tablicę. Deklarator, który kończy się $()$ oznacza funkcję.
- Deklaracje odczytujemy od środka.
- Jeżeli $*$ poprzedza identyfikator a $[]$ następuje po nim, to identyfikator reprezentuje tablicę. Jeżeli $*$ poprzedza identyfikator a $()$ następuje po nim, to identyfikator reprezentuje funkcję.
- Możemy użyć nawiasów by zmienić priorytet $[]$ oraz $()$ nad $*$.

typedef

- Rozważmy poniższą deklarację:

```
int *(*x[10])(void);
```

- Deklarator, który zaczyna się od $*$ oznacza wskaźnik. Deklarator, który kończy się $[]$ oznacza tablicę. Deklarator, który kończy się $()$ oznacza funkcję.
- Deklaracje odczytujemy od środka.
- Jeżeli $*$ poprzedza identyfikator a $[]$ następuje po nim, to identyfikator reprezentuje tablicę. Jeżeli $*$ poprzedza identyfikator a $()$ następuje po nim, to identyfikator reprezentuje funkcję.
- Możemy użyć nawiasów by zmienić priorytet $[]$ oraz $()$ nad $*$.
- Deklaracja z wykorzystaniem **typedef**

```
typedef int * Fcn(void);
typedef Fcn * Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```