Struktury i unie

Struktury

Struktury w języku C są złożonymi typami danych, które pozwalają zgromadzić w obrębie jednej zmiennej wartości różnych typów. Są one odpowiednikami rekordów używanych w innych językach programowania. Aby użyć w programie struktury najpierw musimy zdefiniować jej typ. Wzorzec takiej definicji jest następujący:

```
struct nazwa_typu_struktury
{
        typ_pola nazwa_pola_1;
        typ_pola nazwa_pola_2;
        ...
        typ_pola nazwa_pola_n;
};
```

Pola wewnątrz struktury są deklarowane tak jak zwykłe zmienne i mogą mieć dowolny typ, w tym mogą być tablicą, a nawet strukturą lub unią. Język C pozwala także na tworzenie struktur, które nie zawierają żadnych pól.

Aby zadeklarować zmienną, która jest strukturą musimy w jej deklaracji nie tylko użyć nazwy typu struktury, ale także umieścić przed nią słowo kluczowe *struct*. Wzorzec deklaracji takiej zmiennej jest następujący:

```
struct nazwa_typu_struktury nazwa_zmiennej;
```

Można także zadeklarować zmienną strukturalną w miejscu definicji jej typu, według następującego wzorca:

```
struct nazwa_typu_struktury
{
    typ_pola nazwa_pola_1;
    ...
    typ_pola nazwa_pola_n;
} nazwa_zmiennej_1, nazwa_zmiennej_2;
```

W powyższym wzorcu zdefiniowano dwie zmienne typu strukturalnego. Jeśli potrzebowalibyśmy tylko jednej, to jej nazwę wystarczy umieścić między zamykającym nawiasem klamrowym, a średnikiem.

Zastosowanie struktur

Struktury mają wiele zastosowań. Używa się ich do gromadzenia danych różnych typów, ale logicznie ze sobą powiązanych (zazwyczaj). Mogą one np. posłużyć do przechowywania informacji o osobie:

```
struct personal_data
{
    char name[30], surname[30];
    unsigned char age, height, weight;
};
```

Zmienne typu *struct personal_data* mogą przechowywać takie informacje jak imię, nazwisko, wiek, waga i wzrost określonej osoby. Należy zwrócić uwagę, że tak jak w przypadku zwykłych zmiennych możemy zadeklarować kilka pól tego samego typu podając najpierw identyfikator typu, a potem kolejne ich nazwy rozdzielając je przecinkami i kończąc całość średnikiem.

Struktury mogą być także użyte do przechowywania pewnej liczby wartości tego samego typu, ale mających szczególne znaczenie w rozwiązywanym problemie. Przykładowo, struktura może posłużyć do przechowywania współrzędnych punktu w trójwymiarowej przestrzeni:

```
struct coordinates
{
    double x,y,z;
} point;
```

Zagnieżdżanie struktur i tablice struktur

Język C pozwala na zagnieżdżanie struktur. Przykładowo, można uzupełnić strukturę *struct personal_data* o pole przechowujące adres osoby (*personal_address*) określając typ tego pola jako strukturę:

```
struct personal_data
{
    char name[30], surname[30];
    unsigned char age, height, weight;
    struct address
    {
        char street_name[30], postal_code[10];
        unsigned short int house_number;
    } personal_address;
};
```

Język C pozwala tworzyć także tablice struktur. Przykładowo, można utworzyć tablicę zdefiniowanych wcześniej struktur typu *struct personal_data* w następujący sposób:

```
struct personal_data people[NUMBER_OF_ELEMENTS];
```

Stała NUMBER_OF_ELEMENTS określa liczbę elementów takiej tablicy i powinna zostać zdefiniowana przed deklaracją tej tablicy. Można również zadeklarować tablice struktur w miejscu definiowania typu struktury, podobnie jak zwykłą zmienną.

Dostęp do pól struktury

Dostęp do pola struktury uzyskujemy za pomocą jego nazwy poprzedzonej kropką i nazwą zmiennej strukturalnej, w której jest ono osadzone, czyli według następującego wzorca:

```
nazwa_zmiennej.nazwa_pola
```

Przykład dla zadeklarowanej wcześniej zmiennej point:

```
point.x = 3;
```

Odwołanie do pola struktury zagnieżdżonej w innej strukturze wymaga podwójnego użycia kropki, np.:

```
person.personal_address.house_number = 127;
```

Jeśli struktura jest elementem tablicy, to do pola tej struktury można uzyskać dostęp zastępując nazwę zmiennej we wzorcu odwołaniem do konkretnego elementu w tablicy, np.:

```
people[0].age = 37;
```

Inicjacja zmiennych strukturalnych

Sposób 1

Inicjacji zmiennej typu strukturalnego możemy dokonać w miejscu jej deklaracji, w podobny sposób, jak inicjacji tablicy - umieszczając wartości dla pól w nawiasach klamrowych i rozdzielając je przecinkami:

```c
# include <stdio.h>
struct coordinates
{
    double x, y, z;
} point = {1.0, 2.0, 3.0};

int main(void)
{
    struct coordinates another_point = {4.0, 5.0, 6.0};
    printf("x: %f ",another_point.x);
    printf("y: %f ", another_point.y);
    printf("z: %f\n", another_point.z);
    return 0;
}
```

Sposób 2

Drugi sposób jest podobny do pierwszego, ale występuje w nim dodatkowy element w postaci nazw pól, którym przypisujemy wartości. Są one umieszczane w nawiasie klamrowym i poprzedzone znakiem kropki, np.:

```c
# include <stdio.h>
struct coordinates
{
    double x, y, z;
} point = {.x=1.0, .z=2.0, .y=3.0};
int main(void)
{
    struct coordinates another_point = point;
    printf("x: %f ",another_point.x);
    printf("y: %f ", another_point.y);
    printf("z: %f\n", another_point.z);
    return 0;
}
```

Sposób 3

Ostatni sposób inicjacji zmiennej strukturalnej polega na przypisaniu wartości jej polom poza miejscem jej deklaracji:

```c
# include <stdio.h>
struct coordinates
{
    double x, y, z;
};
int main(void)
{
    struct coordinates point;
    point.x = 1.09;
    point.y = 2.95;
    point.z = 1.22;
    printf("x: %f ", point.x);
    printf("y: %f ", point.y);
    printf("z: %f\n", point.z);
    return 0;
}
```

Unie

Unia jest podobną konstrukcją do struktury w języku C. Różnica między nimi, oprócz sposobu definicji typu i deklaracji zmiennej, polega na tym, że pola zdefiniowane wewnątrz unii są umieszczone w tym samym obszarze pamięci, czyli nakładają się na siebie. W związku z tym unia zajmuje mniej miejsca w pamięci operacyjnej niż struktura o takich samych polach, ale modyfikacja jednego z jej pól wpływa najczęściej również na wartości pozostałych pól.

Typ unii jest definiowany analogicznie do typu struktury. Również zmienne takiego typu są deklarowane tak jak zmienne typów strukturalnych.

```c
union union_type_example
{
    char character;
    int integer;
    char array[8];
} union_example;
```

Powyższy przykład zawiera definicję typu unii i deklarację zmiennej, która nią jest. Jej rozmiar (liczbę zajmowanych bajtów) można określić za pomocą operatora *sizeof*. Uzyskane rezultaty mogą być różne w zależności od konfiguracji komputera i kompilatora, ale zawsze unia zajmuje mniej miejsca w pamięci niż odpowiadająca jej struktura.

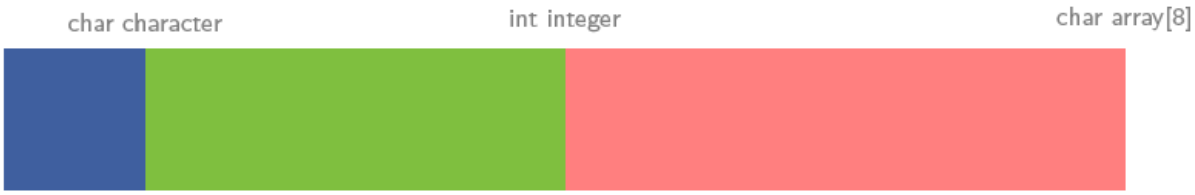
```c
# include <stdio.h>
struct sCoordinates
{
    double x, y, z;
};
union uCoordinates
{
    double x, y, z;
};
int main(void)
{
    struct sCoordinates sPoint = {4.0, 5.0, 6.0};
    printf("ROZMIAR STRUKTURY: %d\n\n", sizeof(sPoint));

    union uCoordinates uPoint = {4.0, 5.0, 6.0};
    printf("ROZMIAR UNII: %d\n\n", sizeof(uPoint));

    return 0;
}
```

Nakładanie sie pól

Poniższy rysunek ilustruje nakładanie się pól unii (odniesienie do unii union_type_example prezentowanej wyżej).



Ilustracja ma charakter poglądowy. Sposób nakładania pól zależny jest od typu i konfiguracji komputera, niemniej zawsze modyfikacja jednego z nich może oznaczać również modyfikację pozostałych lub przynajmniej części z nich.

Zadanie

Sprawdź działanie poniższego kodu prezentującego strukturę z 3 polami typu double i odpowiadającą jej unię. Jakie dostrzegasz różnice między strukturą i unią w kontekście możliwości nakładania się pól?

```c
# include <stdio.h>
struct sCoordinates
{
    double x, y, z;
};
union uCoordinates
{
    double x, y, z;
};
int main(void)
{
    struct sCoordinates sPoint;
    sPoint.x=2.0;
    sPoint.y=4.99;
    sPoint.z=3.33;
    printf("struktura:\tx: %f\ty: %f\tz: %f\n", sPoint.x, sPoint.y, sPoint.z);

    union uCoordinates uPoint;
    uPoint.x=2.0;
    uPoint.y=4.99;
    uPoint.z=3.33;
    printf("unia:\tx: %f\ty: %f\tz: %f\n", uPoint.x, uPoint.y, uPoint.z);

    return 0;
}
```

Podobieństwa między strukturami i uniami

Nie tylko definicje typów i deklaracje zmiennych struktur i unii są podobne. Unie mogą być inicjowane podobnie jak struktury, ale jeśli zastosujemy pierwszy z opisanych w tych materiałach sposobów do inicjacji więcej niż jednego pola unii, to kompilator wystosuje ostrzeżenie, że ostateczna wartość pól może być inna od tej, której się spodziewamy.

Zarówno w przypadku unii, jak i struktur możemy skrócić zapis typu zmiennej, który składa się ze słowa *stuct* lub *union* i nazwy typu, nadając tym typom inną nazwę przy pomocy słowa kluczowego *typedef*. To rozwiązanie zmniejsza jednak czytelność kodu i nie zawsze zalecane jest jego stosowanie.

Przykład

```
# include <stdio.h>
typedef struct
{
    double x, y, z;
}sCoordinates;
typedef union uCoordinates
{
    double x, y, z;
}uCoordinates;
int main(void)
{
    sCoordinates sPoint = {4.44, 5.55, 6.66};
    printf("struktura:\tx: %f\ty: %f\tz: %f\n", sPoint.x, sPoint.y, sPoint.z);

    uCoordinates uPoint;
    uPoint.y=4.99;
    printf("unia:\tx: %f\ty: %f\tz: %f\n", uPoint.x, uPoint.y, uPoint.z);

    return 0;
}
```

Zastosowanie unii

Bardzo często fakt nakładania się pól unii jest wykorzystywany do konwersji typów różnych wartości, np. adresów IP z zapisu dziesiętnego na binarny, czy liczby z zapisu dziesiętnego na kod BCD. Taka konwersja polega na zapisaniu wartości do określonych pól unii i odczytaniu wartości innych jej pól. Nie jest to jednak zalecane przez standard języka C zastosowanie unii, ponieważ wynik takiej konwersji może być różny dla różnych komputerów i nie zawsze poprawny. Standard zaleca, aby odczytywać zawsze to pole unii, które ostatnio było w programie zapisane. Lepiej więc zastosować unie jako pola struktury, celem zaoszczędzenia miejsca w pamięci. Przy takim sposobie korzystania z unii konieczne jest zadeklarowanie w strukturze pola, które będzie pełniło rolę selektora dla pól unii.

Struktury i unie a funkcje

Zarówno unie jak i struktury mogą być zwracane przez funkcje. Poniżej zamieszczony jest kod źródłowy przykładowej funkcji zwracającej strukturę. Dla unii byłaby ona zdefiniowana analogicznie.

```
struct coordinates get_point(double x, double y, double z)
{
    struct coordinates point;
    point.x = x;
    point.y = y;
    point.z = z;
    return point;
}
```

Struktury i unie mogą pełnić też rolę parametrów w funkcjach. Domyślnie, tak jak w przypadku każdej zmiennej innej niż tablica, są one przekazywane przez wartość. Jeśli chcielibyśmy, aby modyfikacje dokonane w ich polach nie uległy zniszczeniu po zakończeniu działania funkcji, to możemy je przekazać przez wskaźnik. Dostęp do pól w tak przekazanej strukturze można uzyskać na dwa sposoby. Pierwszy jest mniej czytelny i polega na zastosowaniu następującego wzorca odwołania:

```
(*nazwa_struktury).nazwa_pola
```

Drugi jest bardziej czytelny, dzięki zastosowaniu specjalnego operatora zapisywanego jako -> i przez to częściej stosowany:

```
nazwa_struktury->nazwa_pola
```

Przykład

```
void move(struct coordinates *point, struct coordinates vector)
{
    point->x += vector.x;
    point->y += vector.y;
    point->z += vector.z;
}
```

Możliwe jest także tworzenie tablic struktur (tablica z typem strukturalnym), np.:

```
# include <stdio.h>
struct personal_data
{
    char name[30], surname[30];
    int age, height, weight;
};
int main(void)
{
    struct personal_data people[4];
    struct personal_data p1 = {"Adam", "Abacki", 22, 188, 87};
    struct personal_data p2 = {"Basia", "Babacka", 32, 168, 57};
    struct personal_data p3 = {"Czeslaw", "Cabacki", 33, 198, 107};
    struct personal_data p4 = {"Damian", "Dabacki", 66, 173, 64};
    people[0] = p1;
    people[1] = p2;
    people[2] = p3;
    people[3] = p4;

    int i;
    for(i=0; i<4; i++)
        printf("imie: %s nazwisko: %s wiek: %d wzrost: %d waga: %d\n",
               people[i].name, people[i].surname, people[i].age,
               people[i].height, people[i].weight);

    return 0;
}
```

Powyżej zaprezentowany został program, w którym zastosowano tablicę struktur do przechowywania danych osobowych ludzi, takich jak imię, nazwisko, wiek, wzrost i waga.

Zadania

1. Przepisz i przeanalizuj działanie wszystkich przykładów (kodów) zawartych w tej instrukcji.
2. Napisz program zawierający strukturę ***trojkat*** przechowującą długości boków trójkąta. Utwórz funkcję, która otrzymuje jako argument zmienną typu ***struct trojkat*** i zwraca jako wartość obwód trójkąta przekazanego jako parametr funkcji.
3. Zdefiniuj strukturę ***punkt*** służącą do przechowywania współrzędnych punktów w trójwymiarowej przestrzeni kartezjańskiej. Napisz funkcję, która otrzymuje jako argumenty

tablicę ***tab*** o argumentach typu ***struct punkt*** oraz jej rozmiar i wypisuje odległości pomiędzy sąsiednimi punktami przechowywanymi w tablicy ***tab***. Zakładamy, że podana w argumencie tablica zawiera co najmniej dwa punkty.

4. Zdefiniuj strukturę ***zespolone*** służącą do przechowywania liczb zespolonych. Zdefiniowana struktura powinna zawierać pola ***im*** i ***re*** typu ***double*** służące do przechowywania odpowiednio części urojonej i rzeczywistej liczby zespolonej. Napisz funkcję ***dodaj***, która przyjmuje dwa argumenty typu ***zespolone*** i zwraca jako wartość ich sumę.