Объявление и определение структуры

Перед тем, как использовать структуру, ее надо объявить. Для определения структуры применяется ключевое слово **struct**. Есть два типа объявления структуры. В первом случае после ключевого слова **struct** идет имя структуры:

```
struct имя_структуры;
```

Имя_структуры представляет произвольный идентификатор, к которому применяются те же правила, что и при наименовании переменных.

Однако это **объявление** без **определения**. С такой структурой сложно что-то сделать, и в данном случае она представляет **неполный тип** (incomplete type), поскольку мы не знаем, какой размер занимает эта структура.

Второй тип объявления структуры - с определением выглядит следующим образом:

```
struct имя_структуры
{
    компоненты_структуры
};
```

После имени структуры в фигурных скобках помещаются компоненты структуры - объекты, которые составляют структуру.

Все элементы структуры объявляются как обычные переменные. Но в отличие от переменных при определении элементов структуры для них не выделяется память, и их нельзя инициализировать. По сути мы просто определяем новый тип данных.

Можно объявить структуру с одним и тем же именем несколько раз, но нельзя определить структуру более одного раза.

```
#include <stdio.h>

struct person;
struct person {
    char * name;
    int age;
};
```

```
struct person;
struct person;

int main(void){

    return 0;
}
</stdio.h>
```

В примере выше все объявления структуры person будут относиться к одной и той же структуре.

Использование структуры

```
// определение структуры person
struct person
{
    char * name;
    int age;
};

int main(void)
{
    // определение переменной, которая представляет структуру person
    struct person tom;
}
```

Здесь определена переменная tom, которая представляет структуру `person`. И при каждом определении переменной типа структуры ей будет выделяться память, необходимая для хранения ее элементов.

Инициализация структуры

При определении переменной структуры ее можно сразу инициализировать, присвоив какое-нибудь значение. Инициализация структур аналогична инициализации массивов: в фигурных скобках передаются значения для элементов структуры. Есть два способа инициализации структуры.

- **По позиции**: значения передаются элементам структуры в том порядке, в котором они следуют в структуре:

```
struct person tom = {23, "Tom"};
```

- Так как в структуре person первым определено свойство age, которое представляет тип int - число, то в фигурных скобках вначале идет число,

которое передается элементу age. Вторым идет элемент name, который представляет указатель на тип char или строку, соответственно вторым идет строка. И так далее для всех элементов структуры по порядку.

- **По имени**: значения передаются элементам структуры по имени, независимо от порядка:

```
struct person tom = {.name="Tom", .age=23};
```

- В этом случае перед именем элемента указывается точка, например, .name.

Обращение к элементам структуры

Также после создания переменной структуры можно обращаться к ее элементам - получать их значения или, наоборот, присваивать им новые значения. Для обращения к элементам структуры используется операция "точка":

```
имя_переменной_структуры.имя_элемента
```

Теперь объединим все вместе в рамках программы:

```
#include <stdio.h>

struct person
{
    int age;
    char * name;
};

int main(void)
{
    struct person tom = {23, "Tom"};
    printf("Age: %d \t Name: %s", tom.age, tom.name);
    return 0;
}
```

Консольный вывод программы:

```
Age: 23    Name: Tom
```

Можно инициализировать элементы структуры по отдельности:

```
#include <stdio.h>

```

```
struct person
{
    int age;
    char * name;
};

int main(void)
{
    struct person tom;
    tom.name ="Tom";
    tom.age = 22;
    printf("Name:%s \t Age: %d", tom.name, tom.age);
    return 0;
}
```

Объявление, определение и использование

Стоит отметить, что мы полноценно использовать структуру (обращаться к ее полям) в основном после ее определения. И в данном случае важно зафиксировать, как влияет определение структуры на ее использование. Например:

```
#include <stdio.h>

struct person; // структура объявлена, но НЕ определена, она пока представляет неполный тип

// структура определена, мы ее можем полноценно использовать
struct person {
    char * name;
    int age;
};

int main(void){

    struct person tom = {"Tom", 22};
    printf("Name: %s  Age: %d\n", tom.name, tom.age);

    return 0;
}
</stdio.h>
```

Здесь сначала мы объявляем структуру без определения:

```
struct person;
```

И лишь затем определяем:

```
struct person {
    char * name;
    int age;
};
```

После этого в функции main мы сможем создавать ее переменные, обращаться к ее полям

```
struct person tom = {"Tom", 22};
printf("Name: %s  Age: %d\n", tom.name, tom.age);
```

Но рассмотрим другую ситуацию:

```
#include <stdio.h>

struct person; // структура объявлена, но НЕ определена, она пока представляет неполный тип

int main(void){

    // в этой точке структура person объявлена, но НЕ определена. Мы не можем определять ее
переменные, обращаться к полям
    // error: variable ‘tom’ has initializer but incomplete type
    // struct person tom = {"Tom", 22};
    // printf("Name: %s  Age: %d\n", tom.name, tom.age);

    return 0;
}
// структура определена, мы ее можем полноценно использовать
struct person {
    char* name;
    int age;
};
  </stdio.h>
```

Здесь структура person определена уже после функции main, и до своего определения она представляет неполный тип - компилятор не знает его размера, никакие поля в нем, соответственно полноценно использовать мы ее не можем. Единственным сценарием использования неполного типа (в том числе и неопределенной структуры), является определение указателя на этот тип:

```
#include <stdio.h>

struct person; // структура объявлена, но НЕ определена, она пока представляет неполный тип

int main(void){

    struct person* tom;  // указатель на структуру person
```

```
 8
 9    return 0;
10}
11// структура определена, мы ее можем полноценно использовать
12struct person {
13    char* name;
14    int age;
15};
16</stdio.h>
```

Здесь переменная tom представляет именно указатель на структуру person.

Объединение определения структуры и ее переменных.

Мы можем одновременно совмещать определение типа структуры и ее переменных:

```
1#include <stdio.h>
2
3struct person
4{
5    int age;
6    char * name;
7} tom;      // определение структуры и ее переменной
8
9int main(void)
10{
11    tom = {38, "Tom"};
12    printf("Name:%s \t Age: %d", tom.name, tom.age);
13    return 0;
14}
```

После определения структуры, но до точки с запятой мы можем указать переменные этой структуры. А затем присвоить их элементам значения.

Можно тут же инициализировать структуру:

```
1#include <stdio.h>
2
3struct person
4{
5    int age;
6    char * name;
7} tom = {38, "Tom"};
8
9int main(void)
10{
```

```
    printf("Name:%s \t Age: %d", tom.name, tom.age);
    return 0;
}
```

Можно определить сразу несколько переменных:

```
struct person
{
    int age;
    char * name;
} tom, bob, alice;
```

При подобном определении мы можем даже не указывать имя структуры:

```
struct
{
    int age;
    char * name;
} tom;
```

В этом случае компилятор все равно будет знать, что переменная tom представляет структуры с двумя элементами name и age. И соответственно мы также с этими переменными сможем работать. Другое дело, что мы не сможем задать новые переменные этой структуры в других местах программы.

typedef

Еще один способ определения структуры представляет ключевое слово **typedef**:

```
#include <stdio.h>

typedef struct
{
    int age;
    char* name;
} person;

int main(void)
{
    person tom = {23, "Tom"};
    printf("Name:%s \t Age: %d", tom.name, tom.age);
    return 0;
}
```

В конце определения структуры после закрывающей фигурной скобки идет ее обозначение - в данном случае `person`. В дальнейшем мы можем использовать это обозначение для создания переменной структуры. При этом в отличие от

примеров выше здесь при определении переменной не надо использовать слово **struct**.

Директива define

Еще один способ определить структуру представляет применение препроцессорной директивы **#define**:

```
#include <stdio.h>

#define PERSON struct {int age; char name[20];}

int main(void)
{
    PERSON tom = {23, "Tom"};
    printf("Name:%s \t Age: %d", tom.name, tom.age);
    return 0;
}
```

В данном случае директива define определяет константу PERSON, вместо которой при обработке исходного кода препроцессором будет вставляться код структуры `struct {int age; char name[20];}`

Область действия структуры

Как и функции, и переменные, структуры имеют область видимость и видны только в той области, в которой они объявлены. Если структура объявлена вне функций на уровне файла, то она видена с точки, где она объявлена, до конца файла./p>

```
#include <stdio.h>

struct person
{
    char * name;
    int age;
};

void print_person(){
    struct person bob = {"Bob", 33};
    printf("Name: %s   Age: %d\n", bob.name, bob.age);
}
int main(void)
{
```

```
    struct person tom = {"Tom", 22};
    printf("Name: %s  Age: %d\n", tom.name, tom.age);
    print_person();
    return 0;
}
```

В данном случае структура person объявлена вне какой-либо функции и видна в любом месте файла после объявления. И для демонстрации в примере выше мы можем ее использовать в функции main и функции print_person (и вообще любой другой функции, которая идет после объявления структуры).

Если структура объявлена в блоке кода, то соответственно она видна до конца этого блока.

```
#include <stdio.h>

// здесь нельзя использовать структуру person, так как она определена в функции main
void print_person(){
    //struct person bob = {"Bob", 33};
    // printf("Name: %s   Age: %d\n", bob.name, bob.age);
}
// структура person видна только в функции main с момента своего определения
int main(void)
{
    struct person {
        char * name;
        int age;
    };
    struct person tom = {"Tom", 22};
    printf("Name: %s  Age: %d\n", tom.name, tom.age);
    return 0;
}
</stdio.h>
```

Причем в качестве области видимости может выступать любой блок кода, необязательно функция. Например, анонимный блок кода:

```
#include <stdio.h>

int main(void){

    // вне этого блока кода структура person не видна
    {
        struct person {
            char * name;
            int age;
```

```
    };
    struct person tom = {"Tom", 22};
    printf("Name: %s  Age: %d\n", tom.name, tom.age);
}
// здесь структура person уже не видна
//struct person bob = {"Bob", 33};
//printf("Name: %s   Age: %d\n", bob.name, bob.age);

return 0;
}
</stdio.h>
```

Здесь структура person объявлена внутри блока кода, соответственно, вне этого блока мы ее использовать не сможем.

Если одна и та же структура объявлена в двух разных областях, то одна структура скрывает ранее объявленную:

```
#include <stdio.h>

// структура уровня файла
struct number {
    int x;
};

int main(void){

    // структура уровня функции - она скрывает структуру number уровня файла
    struct number {
        long y;
    };
    struct number n1 = {22};
    printf("n1: %ld\n", n1.y);

    return 0;
}
</stdio.h>
```

Здесь структура `struct number { long y;}` скрывает структуру `struct number { int x;}`. Поэтому внутри функции main обращение к структуре number будет представлять именно структуру `struct number { long y;}`.

Но даже если внешняя структуру из окружающей области видимости скрывается, ее поля по прежнему видны:

```
#include <stdio.h>
```

```
2
3// структура уровня файла
4struct number {
5    int x;
6};
7
8struct number external = {5};
9
10int main(void){
11
12    // структура уровня функции - она скрывает структуру number уровня файла
13    struct number {
14        long y;
15    };
16    struct number internal = {22};
17
18    printf("external: %d\n", external.x);  // external: 5
19    printf("internal: %ld\n", internal.y);    // internal: 22
20
21    return 0;
22}
23</stdio.h>
```

Здесь мы сначала объявляем структуру number уровня файла. Затем мы определяем переменную external этого типа. В функции main мы объявляем другой тип структуры с тем же именем, который скрывает внешнее объявление. Затем мы объявляем переменную internal с этим новым типом. В функции main мы по-прежнему можем получить доступ к полям обеих переменных. Даже внутри функции main компилятор знает о внешней структуре number, и он также знает, что переменная external принадлежит этому типу.

Копирование структур

Одну структуру можно присваивать другой структуре того же типа. При копировании элементы структуры получают копии значений:

```
1#include <stdio.h>
2
3struct person
4{
5    int age;
6    char * name;
7};
8
```

```
int main(void)
{
    struct person tom = {38, "Tom"};
    // копируем значения из структуры tom в структуру bob
    struct person bob = tom;
    bob.name = "Bob";
    printf("Name: %s \t Age: %d \n", bob.name, bob.age);
    printf("Name: %s \t Age: %d \n", tom.name, tom.age);
    return 0;
}
```

Здесь в переменную `bob` копируются данные из структуры `tom`. Далее мы для структуры `bob` меняется значение поля `name`. В итоге мы получим следующий консольный вывод:

```
Name: Bob          Age: 38

Name: Tom          Age: 38
```

Ввод с консоли данных для структуры

С элементами структуры можно производить все те же операции, что и с переменными тех же типов. Например, добавим ввод с консоли:

```
#include <stdio.h>

struct person
{
    int age;
    char name[20];
};
int main(void)
{
    struct person tom = {23, "Tom"};
    printf("Enter name: ");
    scanf("%s", tom.name);
    printf("Enter age: ");
    scanf("%d", &tom.age);
    printf("Name:%s \t Age: %d", tom.name, tom.age);
    return 0;
}
```

Typedef — **это** ключевое слово, которое используется для предоставления существующим типам данных нового имени. Ключевое слово typedef в языке C используется для переопределения имени уже существующих типов данных. Когда имена типов данных становятся сложными для использования в программах, typedef используется с определяемыми пользователем типами данных, которые ведут себя аналогично определению псевдонима для команд.

typedef против #define

Препроцессор #define также можно использовать для создания псевдонима, но между typedef и #define в C есть некоторые основные различия :

1.#define также может определять псевдонимы для значений, например, вы можете определить 1 как ONE, 3.14 как PI и т. д.

2.Typedef ограничен предоставлением символических имен только типам.

3.Препроцессоры интерпретируют операторы #define, в то время как компилятор интерпретирует операторы typedef.

4.В конце #define не должно быть точки с запятой, но она должна быть в конце typedef.

5.В отличие от #define, typedef фактически определяет новый тип путем копирования и вставки значений определения.

Определить псевдоним для типа указателя

```
#include <stdio.h>

// Creating alias for pointer

typedef int* ip;

int main() {

    int a = 10;

    ip ptr = &a;


    printf("%d", *ptr);

    return 0; }
```

Определить псевдоним для массива

```
#include <stdio.h>

// Here 'arr' is an alias

typedef int arr[4];

int main() {

    arr a = { 10, 20, 30, 40 };

    for (int i = 0; i < 4; i++)

        printf("%d ", a[i]);

    return 0;

}
```

Определить псевдоним для структуры

```
#include <stdio.h>

#include <string.h>

// Using typedef to define an alias for structure

typedef struct Students {

    char name[50];

    char branch[50];

    int ID_no;

} stu;

int main() {

      // Using alias to define structure

    stu s;

    strcpy(s.name, "Geeks");

    strcpy(s.branch, "CSE");

    s.ID_no = 108;

    printf("%s\n", s.name);

    printf("%s\n", s.branch);

    printf("%d", s.ID_no);

    return 0;

}
```

Определить псевдоним для встроенного типа данных

```
typedef long long ll;

int main() {

    // Using typedef alias name to declare variable

    ll a = 20;

    printf("%lld", a);

    return 0;

}
```

Кратко

typedef используется для создания псевдонимов других типов данных.
Приведенный пример лучше скорректировать, например:

```
typedef struct LINE {
...
} t_line;
```

typedef может использоваться не только для структур, но и для любых других типов:

```
typedef int t_message_id;
typedef enum e_Colour {...} t_colour;
```

Основные причины использования typedef-объявлений
Сокращение имен типов данных для улучшения читабельности и простоты набора кода. В приведенном примере без использования typedef придется писать struct LINE