

Основы алгоритмизации и программирования

Лекция 13
Структуры, объединения,
перечисления

Введение

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе базовых типов данных, массивов и указателей. Языки высокого уровня позволяют программисту определять свои типы данных и правила работы с ними, т.е. типы, определяемые пользователем. В языке Си к ним относятся структуры, объединения и перечисления.

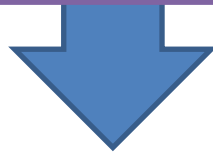
Структуры

Структура – это **составной объект** языка Си, представляющий собой совокупность логически связанных данных различных типов, объединенных в группу под одним идентификатором. Данные, входящие в эту группу, называют **полями**.

Термин «*структура*» в языке Си соответствует двум разным по смыслу понятиям

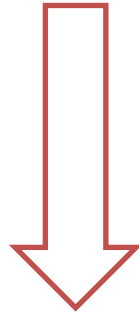
Структура – это обозначение участка оперативной памяти, где располагаются конкретные значения данных; в дальнейшем – это структурная переменная, поля которой располагаются в смежных областях ОП;

Структура – это правила формирования структурной переменной, которыми руководствуется компилятор при выделении ей места в ОП и организации доступа к ее полям



Структуры

Определение объектов типа
структуры
производится за два шага



декларация структурного типа данных, не приводящая к выделению участка памяти

определение структурных переменных объявленного структурного типа с выделением для них памяти

Структурный тип данных задается
в виде шаблона:

```
struct ID структурного типа {  
    описание полей  
};
```

Атрибут «***ID структурного типа***» является **необязательным** и может отсутствовать. Описание полей производится обычным способом: указываются типы переменных и их идентификаторы

Структуры

Пример

Необходимо создать шаблон, описывающий информацию о студенте: номер группы, Ф.И.О. и средний балл. Один из возможных вариантов:

```
struct Stud_type {  
    char Number[10];  
  
    char Fio[40];  
    double S_b;  
};
```

Размещение данного объекта типа *Stud_type* в ОП схематически будет выглядеть следующим образом:

Number	Fio	S_b
10 байт	40 байт	8 байт

Поля одного типа при описании можно объединять в одну группу:

```
struct Stud_type {  
    char Number[10], Fio[40];  
  
    double S_b;  
};
```

Структуры


Структурный тип данных удобно применять для групповой обработки логически связанных объектов. Параметрами таких операций являются адрес и размер структуры.

Примеры групповых операций



захват и освобождение памяти для объекта

запись и чтение данных, хранящихся на внешних носителях как физические и/или логические записи с известной структурой (при работе с файлами)



```
struct Stud_type {  
    char *Number, *fio  
    double S_b;  
};
```

Так как одним из параметров групповой обработки структурных объектов является размер, **не рекомендуется** декларировать поле структуры указателем на объект **переменной размерности**, т.к. в данном случае многие операции со структурными данными будут некорректны

Создание структурных переменных

Описание структуры не приводит к выделению под нее места в ОП. Для работы со структурами необходимо **создать нужное количество переменных** приведенного структурного типа, сделать это можно двумя способами.

Способ 1. В любом месте программы для декларации структурных переменных, массивов, функций и т.д. используется объявленный в шаблоне структурный тип, например:

struct Stud_type student; – структурная переменная;

Stud_type Stud[100]; –
массив структур

Stud_type *p_stud; –
указатель на структуру

Stud_type* Fun(Stud_type); – прототип функции с параметром структурного типа, возвращающей указатель на объект структурного типа.

Способ 2. В шаблоне структуры между закрывающейся фигурной скобкой и символом «;» указывают через запятые идентификаторы структурных данных.

Для нашего примера можно записать:

```
struct Stud_type {  
    char Number[10], Fio[40];  
    double S_b;  
} student, Stud[100], *p_stud;
```

Если дальше в программе не понадобится вводить новые данные объявленного структурного типа, идентификатор **Stud_type** можно не указывать.

Создание структурных переменных

При декларации структурных переменных возможна их одновременная инициализация, например:

```
struct Stud_type {  
    char Number[10], Fio[40];  
    double S_b;  
}  
  
} student = {"123456", "Иванов И.И.", 6.53 };
```

или:

```
Stud_Type stud1 = {"123456", "Иванов И.И." };
```

Если список инициализаций будет короче, то оставшиеся поля структурной переменной заполняются нулями.

Создание структурных переменных

Особенности

поля не могут иметь атрибут, указывающий «класс памяти», данный атрибут можно определить только для всей структуры;

идентификаторы полей могут совпадать с идентификаторами других объектов программы, т.к. шаблон структуры обладает собственным пространством имен;

при наличии в программе функций пользователя шаблон структуры рекомендуется поместить глобально перед определениями всех функций и в этом случае он будет доступен всем функциям.

Обращение к полям структур

Обращение к полям структур производится путем создания составных имен, которые образуются **двумя способами**

при помощи операции принадлежности
(.) общий вид которой:

ID_структуры . ID_поля

или

(*указатель_структуры). ID_поля

при помощи операции косвенной адресации (->) в виде:

указатель_структуры -> ID_поля

Или

(&ID_структуры) -> ID_поля

Если в программе созданы объекты объявленного ранее шаблона:

Stud_Type s1, *s2;

то к полям объекта **s1** можно обратиться следующим образом:

s1. Number, **s1. Fio,** **s1. S b;**

или

(&s1) -> Number, (&s1) -> Fio, (&s1) -> S b;

а к полям объекта, адрес которого **s2**:

s2 → Number, **s2 → Fio,** **s2 → S_b;**

или

(*s2) . Number, (*s2) . Fio, (*s2) . S_b;

Вложенные структуры

Структуры могут быть вложенными, т.е. поле структуры может быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основному шаблону.

Например, в структуре **Person**, содержащей сведения – ФИО, дата рождения, сделать дату рождения внутренней структурой *date* по отношению к структуре *Person*. Тогда шаблон такой конструкции будет выглядеть так:

```
struct date {  
    int day, month, year;  
};  
struct Person {  
    char fio[40];  
    struct date f1;  
};
```

Объявляем переменную и указатель на переменные такой структуры:

```
struct Person a, *p;
```

Инициализируем указатель *p* адресом переменной *a*:

```
p = &a;
```

Вложенные структуры

Обращение к полям структурной переменной ***a*** будет выглядеть следующим образом:

***a* . fio** ***a* . f1 . day** ***a* . f1 . month** ***a* . f1 . Year**

или

***p*→fio** ***p*→f1.day** ***p*→f1.month** ***p*→f1.year**

Можно в качестве связи с вложенной структурой использовать указатель на нее:

```
struct date {  
    int day, month, year;  
};  
struct Person {  
    char fio[40];  
    struct date *f1;  
};
```

Тогда обращение к полям будет следующим:

***a* . fio** ***a* . f1→day** ***a* . f1→month** ***a* . f1→year**

или

***p*→fio** ***p*→f1→day** ***p*→f1→month** ***p*→f1→year**

Массивы структур

Структурный тип «***struct ID_структуры***», как правило, используют для декларации массивов, элементами которых являются структурные переменные. Это позволяет создавать программы, оперирующие с простейшими базами данных.

Например, массив структур, объявленного ранее типа:

```
struct Person spisok[100];
```

причем ключевое слово *struct* можно не писать. Декларацию массива можно выполнить и в описании шаблона следующим образом:

```
struct Person {  
    char fio[40];  
    int day, month, year;  
} spisok[100];
```

В данном случае обращение к полю, например, *day* элемента массива с индексом *i* может быть выполнено одним из следующих способов:

```
spisok[i].day=22;    *(spisok+i).day=22;    (spisok+i)->day=22;
```

Объединения

Объединение – поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента.

Объединенный тип данных декларируется следующим образом:

```
union ID_объединения {  
    описание полей  
};
```

Пример

```
union word {  
    int nom;  
    char str[20];  
};
```

Объединения применяют для **экономии памяти** в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

Декларация данных типа ***union***, создание переменных этого типа и обращение к полям объединений производится **аналогично структурам**.

Перечисления

Перечисления – средство создания типа данных посредством задания ограниченного множества значений.

Определение перечисляемого типа данных имеет вид

```
enum ID_перечисляемого_типа {  
  
    список_значений  
  
};
```

Значения данных перечисляемого типа указываются идентификаторами, например:

```
enum marks {  
    zero, one, two, three, four, five  
  
};
```

Компилятор последовательно присваивает идентификаторам списка значений целочисленные величины 0, 1, 2,

Перечисления

При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения. Например:

```
enum level {  
    low=100, medium=500, high=1000, limit  
};
```

Константа *limit* по умолчанию получит значение, равное **1001**.

Примеры объявления переменных перечисляемого типа:

```
enum marks Est;  
enum level state;
```

Переменная типа *marks* может принимать только значения из множества {zero, one, two, three, four, five}.

Перечисления

Основные операции с данными
перечисляемого типа

```
graph TD; A[Основные операции с данными перечисляемого типа] --> B[присваивание переменных и констант одного типа]; A --> C[сравнение для выявления равенства либо неравенства];
```

присваивание переменных и
констант одного типа

сравнение для выявления
равенства либо неравенства

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

Перечисления

Пример

```
...
typedef enum {
    mo=1, tu, we, th, fr, sa, su
} days;
```

```
void main(void)
{
```

```
days w_day; // Переменная перечисляемого типа
int t_day, end, start;
```

// Текущий день недели, начало и конец недели соответственно

```
puts(" Введите день недели (от 1 до 7) : ");
scanf("%d", &t_day);
w_day = su;
start = mo;
end = w_day - t_day;
printf("\n Понедельник – %d день недели, \n
сейчас %d – й день и \n\
до конца недели %d дн. ", start, t_day, end );
```

Результат

Введите день недели (от 1 до 7) : 5
Понедельник – 1 день недели, сейчас 5-й день и до конца недели 2 дн.

Битовые поля

Битовые поля – это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти – 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа), не превышающая разрядности поля типа *int*



```
struct fields {  
    unsigned int flag: 1;  
    unsigned int mask: 10;  
    unsigned int code: 5;  
};
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом – **по имени**. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры

Битовые поля

Следует учитывать, что **операции с отдельными битами** реализуются гораздо **менее эффективно**, чем с байтами и словами, так как компилятор должен генерировать специальные коды и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры. В основном битовые поля размещаются последовательно в поле типа ***int***, а при нехватке места для очередного битового поля происходит переход на следующее поле типа ***int***. Возможно объявление безымянных битовых полей, а длина поля 0 означает необходимость перехода на очередное поле ***int***:

```
struct areas {  
    // безымянное поле длиной 2 бита;  
    unsigned f1: 1;  
        : 2;  
    //признак перехода на следующее поле int;  
    unsigned f2: 5;  
        : 0  
    //структура может содержать элементы любых  
    типов данных;  
    unsigned f3:5;  
        double data;  
    char buffs[100];  
};
```

Битовые поля могут использоваться в выражениях как целые числа соответствующей длины поля разрядности в двоичной системе исчисления. Единственное отличие этих полей от обычных объектов – запрет операции определения адреса (&). Следует учитывать, что использование битовых полей снижает быстродействие программы по сравнению с представлением данных в полных полях из-за необходимости выделения битового поля.