Учебный Центр компьютерного обучения «СПЕЦИАЛИСТ» при МГТУ им. Н.Э.Баумана

Язык программирования C

В.Г.Тетерин – Microsoft Solution Developer (Visual C++)

teterin@specialist.ru

www.specialist.ru

Учебный центр компьютерного обучения «СПЕЦИАЛИСТ» при МГТУ им. Н.Э.Баумана

МОДУЛЬ 10

СТРУКТУРЫ

Модуль 10. Структуры

- Декларация структур.
- Инициализация и доступ к элементам структуры.
- Вложенные структуры и массивы структур.
- Объединения.

Декларация структур

- *Структуры*, называемые также *записями*, подобно массивам являются составными объектами, но, в отличие от последних, могут содержать в себе *разнотипные элементы*.
- Записи удобны для хранения наборов данных произвольных типов,

  например, анкета сотрудника, квитанция и т.п.

Фамилия И.О.	**Name**	Иванов И.И.
Возраст	**Age**	24
Дата приема	**Admit**	1.04.2001
Должность	**Position**	Менеджер
Оклад	**Salary**	25000

- Элементы записи называются полями и им присваиваются уникальные (в пределах записи) имена.
- Имена используются в операциях доступа к значениям полей.
- Записи и их наборы (например, массивы) в языках программирования предоставляют удобный способ работы с реляционными базами данных.

Декларация структур (продолжение)

- Описание структуры
  - начинается ключевым словом **struct**,
  - за ним следует имя типа структуры (тэг структуры),
  - затем в фигурных скобках перечисляются типы и имена элементов структуры,
  - завершается описание точкой с запятой (**;**).

  – Так, например, структура для представления даты могла бы иметь описание:

```
struct date
{
    char day;  // Число     (1-31)
    char mon;  // Месяц (1=январь)
    int year;  // Год
};
```

- Это описание является *объявлением* нового типа данных – "структуры типа **date**", но оно задает лишь "шаблон", схему размещения элементов в памяти: первыми должны располагаться два символьных значения, следом за ними – значение целого типа.
- Для того, чтобы *определить* такую структуру, то есть выделить для нее память, необходимо использовать описание вида

```
struct date today;
```

  которое создает структурную переменную с именем **today**, имеющую тип **date**.

- Все элементы структуры имеют один общий класс памяти - это класс памяти всей структуры (структурной переменной).

Декларация структур (продолжение)

- Для нового типа данных **struct date** можно создать псевдоним

```
typedef struct date date;
```

и использовать его повсюду далее:

```
date birthday;
```

- Описание структуры и ее псевдонима можно совместить.
- Можно опустить имя типа структуры, если нигде в программе далее оно не будет использоваться.
- Если некоторые или все элементы структуры имеют одинаковый тип, то их описание можно совмещать, как это обычно делается при описании простых переменных.
- Например, :

```
typedef struct
{
    char day, mon;     // Число(1-31), Месяц (1=январь)
    int year;          // Год
} date;
```

Инициализация и доступ к элементам структуры

- Структуры можно инициализировать при их определении, заключая список инициирующих значений в фигурные скобки

```
date birthday = {1,4,1980};
```

  - Значения из списка последовательно присваиваются элементам структуры:
    - если в списке значений меньше, чем количество элементов, то оставшиеся элементы структуры получают нулевое значение, если больше – это ошибка.

- Доступ к отдельному элементу структуры осуществляется по *составному имени*: имени структурной переменной и имени элемента, разделенных точкой:

```
int age = today.year – birthday.year;

today.day++;
```

  - Отсюда следует, что элементы, составляющие структуру, не могут иметь совпадающих имен - каждое имя внутри структуры должно быть *уникально*.
  - Это ограничение на выбор имен элементов структуры является единственным, так как в С имена переменных, имена типов структур и имена элементов структур образуют три различных *класса имен* и конфликты между ними невозможны.
    - Четвертый класс имен в языке С образуют *метки* операторов (инструкций).

Инициализация и доступ к элементам структуры (продолжение)

- Подобно массивам, доступ к элементам структуры можно получить не только через ее имя, то также и с использованием указателя на структуру.
  - Определение указателя на структуру заданного типа:

```
date *p = &today;
```

Теперь выражение

```
p->day
```

(это сокращение для `(*p).day`)

имеет тот же смысл, что и

```
today.day
```

- Указатели на структуры наиболее часто используются при передаче структур в функции:

```
void read_date(date *);
void print_date(const date *);
...
read_date(&today);
print_date(&today);
```

Вложенные структуры и массивы структур

- Структура может иметь в качестве своих элементов не только переменные простых типов, но и массивы.
- Структура может содержать другие структуры в качестве своих элементов.
  - Например, для хранения сведений о сотруднике (фамилия, возраст, дата приема на работу, должность, оклад) можно определить структуру следующего вида:

```
struct employee
{
    char name[21];
    int age;
    date admit;
    char position[21];
    float salary;
} emp, *pt=&emp;
```

Здесь использована описанная ранее структура типа date.

  - Для определения года поступления на работу в данном случае следует применить выражение

```
emp.admit.year
```

аналогичного эффекта можно достичь с помощью выражения

```
pt->admit.year
```

Вложенные структуры и массивы структур (продолжение)

- Широкое применение в С находят *массивы структур* для обработки наборов сложно организованных данных.
  - Пример. Массив – картотека сведений о сотрудниках:

```
#define MEMBERS 50
struct employee staff[MEMBERS];
```

Здесь **staff** – массив с числом элементов **MEMBERS**, каждый элемент которого является структурой типа **employee**.

Тогда значением выражения

```
staff[0].salary
```

будет оклад сотрудника, записанного в этой картотеке первым, а значением выражения

```
staff[3].name[0]
```

- первая буква фамилии сотрудника, записанного в картотеке четвертым по порядку.

Из приведенных примеров следует, что индекс всегда записывается после имени соответствующего массива.

Вложенные структуры и массивы структур (продолжение)

- Структуры предоставляют большую гибкость для отображения сложно организованных данных.
- Никакая структура не может содержать *саму себя* в качестве своего элемента, но она может содержать *ссылки на себя*. Это позволяет поддерживать данные, организованные в виде списков, деревьев и так далее.
  - Например, с помощью объявления-"шаблона"

```
struct tnode
{
    int contents;
    struct tnode *left;
    struct tnode *right;
};
```

обычно организуют данные в виде так называемого двоичного дерева: "шаблон" описывает одну вершину дерева:

- **contents** - содержимое этой вершины,
- **left** - указатель на левое поддерево,
- **right** - указатель на правое поддерево.

Для хранения дерева либо можно определить соответствующий массив, например,

```
#define  NELEM    100
struct   tnode tree[NELEM];
```

либо использовать библиотечные функции С для динамического выделения памяти.

Битовые поля

- В отличие от большинства языков программирования, язык С имеет встроенный метод доступа к отдельным битам внутри байта или машинного слова.
  - Этот метод может оказаться полезным по ряду причин:
    - Во-первых, при дефиците памяти может оказаться необходимым упаковывать несколько объектов, особенно таких как одноразрядные признаки - флаги, в одно машинное слово.
    - Во-вторых, интерфейсы работы с внешними устройствами часто требуют возможности выделения части слова.
    - В-третьих, некоторые программы кодирования информации также нуждаются в доступе к отдельным битам.
  - Хотя эти действия могут быть выполнены с помощью битовых операторов, использование *битовых полей* может улучшить структуру и эффективность программы.
  - Метод, используемый в С для доступа к отдельным битам, базируется на структуре.
- **Битовое поле** - это специальный тип структуры, который определяет, какой будет длина в битах каждого элемента структуры.
  - В С битовое поле может рассматриваться:
    - как целое со знаком - **`signed int`** (по умолчанию просто **`int`**),
    - или без знака - **`unsigned int`**.
  - Битовое поле располагается от младших битов машинного слова к старшим.
  - Поле не может пересекать границу машинного слова, если такое происходит, то оно целиком переносится в следующее слово, а неиспользованные биты предыдущего слова будут недоступными.
  - Для принудительного выравнивания на границу следующего слова используется специальный размер поля 0.
  - Если некоторому полю не присвоено имени, то его биты будут также недоступными.

Битовые поля (продолжение)

- Пример определения битовых полей:

```
struct fields
{
        int i           :       2;
        unsigned j      :       5;
        int             :       4;
        int k           :       1;
        unsigned m      :       4;
 } a;
```

  - В этом примере в машинном слове выделяются 5 полей.
- Доступ к полям осуществляется так же как к элементам обычной структуры.
- Для полей существуют специфические ограничения:
  - они не могут быть массивами
  - и не имеют адресов, поэтому к ним нельзя применять операцию **&** получения адреса поля.
- Разрешается смешивать обычные элементы структуры с битовыми полями:

```
struct tx
{
        char       x;
        unsigned ready    : 1;
        int               : 6;
        unsigned error    : 1;
};
```

Перечисление (нумератор)

- *Перечисления*, введенные в стандарте ANSI, предназначены для описания переменных, которые могут принимать конечное (и обычно небольшое) множество значений, имеющих символические наименования.
- Переменные такого типа и их возможные значения вводятся с помощью ключевого слова **enum**:

```
enum days{sun,mon,tues,wed,thur,fri,sat};
```

  - Это описание объявляет имя нового типа - перечисления **days** и определяет набор значений **sun**, **...** ,**sat**, которые могут принимать переменные типа **days**.
- После этого объявления имя типа **days** может использоваться в программе наряду с именами основных типов, таких как **int**, **char** и др.
  - Для того, чтобы определить переменную с именем **today**, имеющую перечислимый тип **days**, следует использовать описание:

```
enum days today;
```

- Объявление перечислимого типа и определение переменных можно совместить в одном описании:

```
enum days{sun,mon,tues,wed,thur,fri,sat} today, yesterday;
```

и даже можно опустить имя типа, если им не придется пользоваться в программе:

```
enum{sun,mon,tues,wed,thur,fri,sat} today, yesterday;
```

Перечисление (нумератор) (продолжение)

- Сами символические значения во внутреннем представлении кодируются целыми константами:
  - первое имя кодируется значением 0,
  - каждое следующее получает код на 1 больший предыдущего.

- Последовательный способ кодирования символических значений перечисления можно изменить, используя в перечислении явную инициализацию.
  - Если не все элементы явно инициализируются, то не имеющие инициатора элементы получают код на 1 больше, чем код непосредственно предшествующего ему элемента:

```
enum text_modes{LASTMODE=-1,BW40=0,C40,BW80,C80,MONO=7};
```

  - При таком описании **C40** имеет код 1, **C80** - код 3, а **MONO** - код 7.

- Хотя элементы перечисления (символические имена) и элементы перечислимого типа могут использоваться в операциях присваивания, сравнения и в других конструкциях языка, где разрешено использование переменных типа **int**, но наиболее часто перечисления рассматривают как удобный способ введения логически связанных между собой целых констант, имеющих символические имена.
  - Так описанное выше перечисление **text_modes** используется в C для кодирования текстовых режимов работы дисплея.

Объединения

- **Объединение**, называемое также *смесью*, - это участок памяти, который используется несколькими разными переменными или другими объектами программы, причем типы этих переменных или объектов могут быть различными.
  - Практически смесь - это специальный тип структуры, в котором
    - все элементы имеют нулевое смещение относительно ее начала,
    - правильное выравнивание в памяти (на начало байта или слова), чтобы можно было работать со всеми смешиваемыми типами.
- Описание смеси выполняется аналогично описанию обычной структуры с заменой ключевого слова **`struct`** на слово **`union`**, например:

```
union mix
{
        int number;
        double var;
        char symbol;
        char *string;
} umix;
```

  - В каждый момент времени эта смесь типа mix может содержать одно из четырех значений, а что именно - это должен помнить сам программист.
- Все операции манипулирования со смесями полностью аналогичны операциям со структурами.
- Смеси могут быть элементами записей и массивов и наоборот.
  - Одно из интересных применений смеси состоит в том, что с ее помощью можно создать массив, состоящий из элементов одинакового размера, каждый из которых может содержать различные типы данных, - *гетерогенный* массив.

Итоги

- В этом модуле Вы изучили:
  - Понятие структур как агрегатов различных типов данных, синтаксис их определения и семантику использования
  - Битовые поля как разновидность структур
  - Перечислимый тип данных
  - Объединения (смеси) и возможности их применения

Вопросы?

- В.Г.Тетерин – Microsoft Solution Developer (Visual C++)
- teterin@specialist.ru

ПРИЛОЖЕНИЕ

ЗАДАЧИ

Задачи

1. Примените структуры для представления данных в реализации задач модуля 9
2. Разработайте связный список с размещением его узлов в динамической памяти и примените его для представления данных в реализации задач модуля 9