Тема 1.8 Записи (структуры)

Структуры - это составной объект, в который входят элементы любых типов, за исключением функций.

В отличие от массива, который является однородным объектом, структура может быть неоднородной.

Тип структуры определяется записью вида:

struct { список определений }

В структуре обязательно должен быть указан хотя бы один компонент.

Определение структур имеет следующий вид:

тип-данных описатель;

где тип-данных указывает тип структуры для объектов, определяемых в описателях. В простейшей форме описатели представляют собой идентификаторы или массивы.

Синтаксис:

```
struct Имя_структуры
{
    Тип_1 Имя_переменной_члена_1;
    Тип_2 Имя_переменной_члена_2;
            . . .
    Тип_N Имя_переменной_члена_N;

};  // Не забывайте о точке с запятой
```

```
struct Automobile
{
      int year;
     int doors;
    doube horse_power;
    char model;
};
```

```
year                2байта |
doors               2байта  |  Automobile(13 байт)
horse_power         8байт   |
model               1байт   |
```

Переменные структурного типа определяются так же, как и переменные других типов.

Automobile my_car, your_car;

Объём памяти структурные переменные *my_car, your_car* занимают: 2b +2b + 8b + 1b = 13b.

Для доступа к переменным-членам структуры используется оператор точки.

my_car.year, my_car.doors, my_car.horse_power, my_car.model.

Ключевое слово struct начинает определение структуры. Идентификатор Automobile – тег (обозначение, имя-этикетка) структуры. Тег структуры используется при объявлении переменных структур данного типа. В этом примере *имя нового типа* – Automobile. Имена, объявленные в фигурных скобках описания структуры – это *элементы структуры*. Элементы структуры могут быть любого типа, и одна структура может содержать элементы многих разных типов. **Структура не может включать саму себя**. Например, в описании структуры Automobile не может быть объявлена переменная типа структуры Automobile. Тем не менее, может быть включен указатель на структуру типа Automobile.

```
struct Time {
        int hour;
        int minute;
        int second;
};
```

Это **ОПРЕДЕЛЕНИЕ структуры данных не резервирует никакого пространства в памяти**; определение только создает новый тип данных, который используется для объявления переменных.

ОБЪЯВЛЕНИЕ структуры данных резервирует пространство в памяти:

Time timeObject, timeArray[10], *timePtr;

объявляет переменную *timeObject* типа Time, *timeArray* - массив с десятью элементами типа *Time, timePtr* - указатель на объект типа Time.

Для доступа к элементам структуры используются операции доступа к элементам – **операция точка (.) и операция стрелка (->).** Операция точка обращается к элементу структуры по имени переменной или по ссылке на объект. Например, чтобы напечатать элемент hour структуры *timeObject* используется оператор:

```
cout << timeObject.hour;
```

Операция стрелка обеспечивает доступ к элементу структуры через указатель на объект. Допустим, что указатель *timePtr* был уже объявлен как указывающий на объект типа Time и что адрес структуры *timeObject* был уже присвоен *timePtr*. Тогда, чтобы напечатать элемент hour структуры *timeObject,* можно использовать оператор

```
cout << timePtr->hour;
```

Выражение *timePtr->hour* эквивалентно *(*timePtr).hour*, которое разыменовывает указатель и делает доступным элемент hour через операцию точка. Скобки нужны здесь потому, что операция точка имеет более высокий приоритет, чем операция разыменования указателя (*). Операция стрелка и точка наряду с круглыми и квадратными скобками имеют второй наивысший приоритет и ассоциативность слева направо.

Для каждого объявленного в определении структуры имени члена существует одно значение члена. Например, значение типа *Automobile* является набором из четырёх значений членов: два из них типа *int,* одно типа *double* и одно типа *char.* Значения членов, совместно образующих значение структуры, хранятся в рассматриваемых переменных-членах.

Каждый определяемый структурой тип устанавливает перечень имён членов. В структуре *Automobile* имеется четыре члена с именами: *year, doors, horse_power, model.* Эти имена можно использовать в составляющих структуру переменных, которые получили название переменных-членов. Обращение к переменным-членам выполняется посредством имени структурной переменной, после которого следует точка и имя члена. Переменные-члены могут использоваться подобно любым другим переменным этих типов. **Одни и те же имена могут использоваться несколькими структурами.**

```
sruct FertilizeStock
{
double quantity;
double nitrogen_content;
};

struct CropYield
{
int quantity;
double size;
};

FertilizeStock  super_grow;
CropYield apples;
super_grow.quantity;  // количество удобрения
apples.quantity;        // количество собранных яблок
```

Оператор точки и структурная переменная точно определяют, какой именно структуре принадлежит переменная *quantity.*

Значение структуры может рассматриваться и как набор отдельных значений членов, и как единое значение. Структурные переменные и значения структур могут использоваться точно так же, как простые значения базовых типов. Значения структур могут назначаться обычным оператором присваивания.

```
CropYield apples, oranges;
apples = oranges;
```

это эквивалентно двум следующим присваиваниям.

```
apples.quantity = oranges.quantity;
apples.size = oranges.size;
```

Использование иерархических структур

Иногда требуется наличие структур, члены которых, в свою очередь являются структурами.

```
struct Date
{
     int month;
    int day;
   int year;
};

 struct PersonInfo
{
   double height;
   int      weight;
  Date    birthday;
};
```

Структурная переменная типа *PersonInfo* объявляется как обычно:
PersonInfo person1;

Значение структурной переменной *person1* хранит среди прочей информации дату рождения, и год рождения может быть выведен на экран с помощью следующей инструкции:
cout << person1. birthday.year;

Подобные выражения с применением точки в записи считываются слева направо.

Инициализация структур

Инициализация структурной переменной может быть выполнена непосредственно при её объявлении. Чтобы присвоить структурной переменной значение, следует поставить после её имени знак равенства и список значений членов, заключённый в фигурные скобки.

```
struct Date
{
     int month;
    int day;
   int year;
};

Date due_date = {12, 31, 1999};
```

Если количество инициализирующих значений превышает количество членов структуры, это является ошибкой. Если же их меньше, чем членов структуры, то имеющиеся значения используются для инициализации переменных-членов в порядке их объявления. Тем переменным-членам, которым не хватило инициализирующих значений, присваиваются нулевые значения соответствующего переменной-члену типа.

Свойства структуры

```
# include <iostream.h>
struct inflatable
	{
		char name[20];
		float volume;
		double price;
	};
int main ()
{
	inflatable guest =
		{
			“Glorious Gloria”,
			1.88,
			29.99
		};
	inflatable pal =
		{
			“Audacious Arthur”,
			3.12,
			32.99
		};
	cout << “Expand your guest list with ”
		<< guest.name;
	cout << “ and ” << pal.name << “!\n”;
	cout << “You can have both for $”;
	cout << guest.price + pal.price << “!\n”;
	return 0;
}
```

Результат выполнения программы:
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!

Массивы структур

Структура inflatable содержит массив name. Можно также создавать массивы, элементы которых будут строками. Например, чтобы создать массив из 100 структур inflatable, выполнить следующее:

```
inflatable gifts[100]; // массив из 100 структур inflatable
```

В результате будет создан массив gifts типа inflatable. Следовательно, каждый элемент массива – это объект типа inflatable, и его можно использовать с оператором принадлежности:

```
cin >> gifts[0].volume; // используется элемент volume первой структуры
cout << gifts[99].price << endl; // показан элемент price последней структуры
```

Имеется ввиду, что сам gifts – это массив, а не структура, так что конструкции типа gifts. Price не являются правильными.

Т.к. каждый элемент м массива - это структура, ее значение представлено инициализацией структуры. Таким образом, получится заключенный в фигурные скобки, разделенный запятыми

список значений, каждое из которых непосредственно заключено в фигурные скобки, разделенным запятыми списком значений:

```
inflatable guests[2] = // инициализация массива структур
{
    {“Bambi”, 0.5, 21.99}, // первая структура в массиве
    {“Godzilla”, 2000, 565.99} // следующая структура в массиве
};
```

Битовые поля

Язык C++ предоставляет возможность задавать количество битов, в которых хранятся элементы типов unsigned или int структуры. Такие элементы называют битовыми полями. Битовые поля позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов. Элементы битовые поля должны быть объявлены как тип unsigned или int.

Рассмотрим следующее описание структуры:

```
struct BitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};
```

Это описание включает три битовых поля типа unsigned: face, suit, color. Количество битов определяется ожидаемым диапазоном значений для каждого элемента структуры.

Можно задавать неименованное битовое поле; в этом случае поле используется в структуре как заполнение. Например, описание структуры

```
struct Example {
    unsigned a : 13;
    unsigned : 3;
    unsigned b : 4;
};
```

использует неименованное 3-х битовое поле как заполнение: ничто не может храниться в этих трех битах. Элемент b хранится в другом элементе памяти.

Неименованное битовое поле нулевой ширины используется для выравнивания следующего битового поля по границе нового элемента памяти. Например, описание структуры

```
struct Example {
    unsigned a : 13;
    unsigned : 0;
    unsigned b : 4;
};
```

использует неименованное поле нулевой ширины, чтобы пропустить оставшиеся биты в том элементы памяти, в котором хранится элемент a, и выровнять элемент b по границе следующего элемента памяти.

Эквивалентность типов

Два структурных типа считаются различными даже тогда, когда они имеют одни и те же члены. Например, ниже определены различные типы:

```
struct s1 { int a; };
struct s2 { int a; };
```

В результате имеем:

```
s1 x;
s2 y = x; // ошибка: несоответствие типов
```

Кроме того, структурные типы отличаются от основных типов, поэтому получим:

```
s1 x;
int i = x; // ошибка: несоответствие типов
```

Структуры в качестве аргументов функции

Параметры структурного типа могут передаваться функции, как по значению, так и по ссылке. Возвращаемое функцией значение также может иметь структурный тип.

```
CDAccount shrink_wrap(double the_balance, double the_rate, int the_term)
{
  CDAccount temp;
temp.balance = the_balance;
temp.interest_rate =the_rate;
temp.term = the_term;
return temp;
}
int main()
{
CDAccount new_account;
new_account = shrink_wrap(10000.00, 5.1, 11);
   . . .
return 0;
}

// Создание структуры, задание и печать ее элементов
#include "stdafx.h"
#include "iostream.h"
#include "conio.h"

struct Time {
                int hour;
                int minute;
                int second;
        };
void printMilitary (const Time &);
void printStandard (const Time &);
main()
{
        Time dinnerTime;
        dinnerTime.hour = 18;
        dinnerTime.minute =30;
        dinnerTime.second = 0;

        cout<< "Обед состоится в ";
printMilitary (dinnerTime);
cout << "по военному времени, "<<endl << "что соответствует";
printStandard (dinnerTime);
cout<< "по стандартному времени."<< endl;

//задание элементам неправильных значений
dinnerTime.hour = 29;
        dinnerTime.minute =73;
        dinnerTime.second =103;
        cout << endl << "Время с неправильными значениями: ";
        printMilitary (dinnerTime);
        cout << endl;
        return 0;
}
//Печать времени в военном формате
```

```
void printMilitary (const Time &t)
{
        cout << (t.hour < 10 ? "0" : "")<< t.hour
             << ":"<< (t.minute < 10 ? "0" : "") << t.minute
             << ":"<< (t.second < 10 ? "0" : "") << t.second;
}
//Печать времени в стандартном формате
void printStandard (const Time &t)
{
        cout << ((t.hour == 0 || t.hour == 12) ? 12 : t.hour %12)
             << ":"<< (t.minute < 10 ? "0" : "") << t.minute
             << ":"<< (t.second < 10 ? "0" : "") << t.second
             << (t.hour < 12 ? "AM" : "PM");
}
```

Обед состоится в 18:30:00 по военному времени,
Что соответствует 6:30:00 PM по стандартному времени
Время с неправильными значениями: 29:73:103

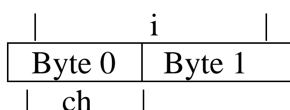
Существует два способа передачи в функции информации о структурах. Вы можете либо передать всю структуру в целом, либо передать отдельные элементы структуры. По умолчанию данные (за исключением отдельных элементов – массивов) передаются вызовом по значению. Структуры и их элементы можно также передавать вызовом по ссылке, используя передачу ссылок или указателей. Массивы структуры (аналогично всем другим массивам) автоматически передаются вызовом по ссылке. Массив может быть передан вызовом по значению с помощью структуры. Для этого нужно создать структуру с массивом в качестве элемента.

Объединения (смеси)

Объединения – это объект, позволяющий нескольким переменным различных типов занимать один участок памяти (или: это формат данных, который может содержать различные типы данных, но только один тип одновременно). В любой момент времени объединение может содержать максимум один объект, потому что элементы объединения совместно используют одну и ту же область памяти. На программиста возлагается обязанность следить за тем, чтобы к данным в объединении обращались по имени элемента соответствующего типа данных. Объявление объединения похоже на объявление структуры. Объединения объявляются при помощи ключевого слова *union.*

```
union union_type{
            int i;
           char ch;
      };
    union_type cnvt;
```

В данном объявлении вводится тип *union_type* с элементами *int i, char ch.*
В *cnvt* как целое число *i,* так и символ *ch* занимают один участок памяти.



Когда объявлено объединение, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении. Объединения обычно определяются в программе до функции *main()* для того, чтобы в любой функции программы можно было объявить переменные типа этого объединения.

cnvt.i = 10;

Использование объединений помогает создавать машинно-независимый код. Поскольку компилятор отслеживает настоящие размеры переменных, образующих объединение, уменьшается зависимость от компьютера. Также целью объединений является – экономия памяти, когда элемент данных может использовать два или больше формата, но не одновременно.

Объединения часто используются при необходимости преобразования типов, поскольку можно обратиться к данным, хранящимся в объединении, совершенно различными способами.

Единственными допустимыми встроенными операциями, которые могут выполняться над объединениями, являются: операция присваивания значения одного объединения другому объединения того же типа, операция вычисления адреса объединения и доступ к элементу объединения при помощи операций доступа к элементу структуры (. и ->). Над объединениями не могут выполняться операции сравнения.

```
#include <iostream.h>
union Number
     {
          int x;
          double y;
     };
int main()
{
  Number value;
   value.x = 100;
   cout << "Поместить значение в integer-xkty объединения\n"
           <<"и печатать оба члена объединения.\n int:      "
           << value.x << "\n double:  " << value.y << endl;
   return 0;
}
```

Анонимное объединение – это объединение, которое не имеет имени типа и при определении которого перед завершающей точкой с запятой не задается имя объекта или указателя. При объявлении такого объединения создается не тип, а объект, не имеющий имени. К элементам анонимного объединения можно обращаться непосредственно по их именам в той области действия, в которой объединение объявлено, как к любым локальным переменным; при этом не нужно использовать операцию (.) или стрелка (->). Анонимные объединения имеют некоторые ограничения. Они могу иметь только данные-члены. Все элементы объединения должны иметь открытый уровень доступа.

```
#include <iostream.h>
int main()
{
   union
     {
         int b;
         double c;
         char *fPtr;
     };
   int a = 1;
   double c = 3.3;
   char *ePtr = "Anonymous";
   cout << a << ' ';
   b = 2;
   cout << b << endl;

   cout << c << ' ';
   d = 4.4;
   cout << d << endl;
   cout << ePtr << ' ';
   fPtr = "union";
   cout << fPtr << endl;

  return 0;
}
```

Перечисления

Перечисления – это набор именованных целочисленных констант, определяющий все допустимые значения, которые может принимать переменная. Перечисления определяются с помощью ключевого слова enum, которое указывает на начало перечисляемого типа.

enum ярлык{список перечислений}список переменных;

Как имя перечисления – ярлык, так и список переменных необязательны, но один из них должен присутствовать. Список перечислений – это разделенный запятыми список идентификаторов.

Например,

enum coin{penny, nickel, dime, quarter, half_dollar, dollar};

coin money;
money=dime;
if (money==quarter) cout<< "is a quarter\n";

Этот оператор выполняет две функции:

1. делает *coin* названием нового типа данных, *coin* называется перечислением.
2. определяет *penny, nickel, dime, quarter, half_dollar, dollar* как символические константы для целых чисел от 0 до 7. Эти константы называются перечислителями.

В перечислениях каждому символу ставится в соответствие целочисленное значение и поэтому перечисления могут использоваться в любых целочисленных выражениях.

cout << "The value of quarter is " << quarter; совершенно корректно

Если явно не проводить инициализацию, значение первого символа перечисления будет – 0, второго – 1 и так далее.

cout<< penny <<" " << dime; *(0 и 2)*

Можно определить значения одного или нескольких символов, используя инициализатор. При использовании инициализатора, символы, следующие за инициализационным значением, получают значение большее, чем указанное перед этим.

enum coin{penny, nickel, dime, quarter=100, half_dollar, dollar};
penny 0
nickel 1
dime 2
quarter 100
half_dollar 101
dollar 102

money=dollar;
cout << money; Мы получим число 102, а не текст dollar.

Символ *dollar* – это просто имя для целого числа, а не строка. Следовательно, невозможно с помощью cout вывести строку «*dollar*», используя значение в money.

Money="dollar"; // не работает

Надо:

switch (money) {
case penny: cout << "penny";

```
            break;
        ...
    case dollar: cout << "dollar";
    }
```

Можно использовать название перечислителя, чтобы объявить переменную этого типа:
coin band;

Единственными действительными значениями, которые можно назначить переменной типа перечисления без преобразования типа, являются значения перечислителя, использованные в определении типа.

```
money = nickel; // правильно, nickel – это перечислитель
money = 2000; // неправильно, 2000 – это не перечислитель
```

Для перечислений определен только оператор присваивания. В частности арифметические действия не определены:

```
money = nickel; // правильно
++ money; // неправильно
money = nickel + dollar; // неправильно
```

Перечислители имеют целочисленный тип и могут быть преобразованы в тип int, но элементы типа int не преобразовываются автоматически в тип перечислителя:

```
int col = nickel; // правильно, тип coin преобразуется в int
money = 3; // неправильно, int не преобразуется в coin
col = 3 + nickel; // правильно, nickel преобразуется в int
```

Иногда возможно объявлять массив строк и использовать значения перечислений как индекс для их перевода в соответствующие строки.

```
Char name [][12] ={
        "penny",
        "nickel",
        "dime",
        "quarter",
         "half_dollar",
         "dollar"
}
cout << name[money];
```

Это работает только в том случае, если не используется инициализатор, поскольку массив строк должен инициализироваться с 0.