Типы данных, определяемые пользователем

Структуры

Чтобы программы была более ясной, можно задать типу новое имя с помощью ключевого имя typedef.
Эта директива позволяет задать синоним для встроенного либо пользовательского типа данных

```
typedef тип имя [размерность];
```

//Размерность может отсутствовать.

```
typedef double wages;

typedef vector<int> vec_int;

typedef vec_int test_scores;

typedef bool in_attendance;

typedef int *Pint;
```

Кроме задания типам с длинными описаниями более коротких псевдонимов, `typedef` используют для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью оператора `typedef`, при переносе программы потребуется внести изменения только в эти операторы.

```
typedef struct {
float due;
int over_due;
char name[40];
} client; /* здесь client - это имя нового
типа */

client clist[NUM_CLIENTS]; /* определение
массива структур типа client */
```

Структуры представляют собой группы связанных между собой, как правило, разнотипных переменных, объединенных в единый объект.

Для работы с некоторой структурой в программе необходимо:

1. описать тип данных этой структуры;
2. определить переменные этого типа для хранения соответствующих данных в памяти.

В языке C++ структура является видом класса и обладает всеми его свойствами. Чаще всего ограничиваются тем, как структуры представлены в языке C:

```
struct [имя_типа] {
   тип_1 элемент_1;
   тип _2 элемент_2;
   …
   тип_k элемент_k;
} [ список_описателей ];
```

При описании структуры память для размещения данных не выделяется. Работать с описанной структурой можно только после того, как будет определена переменная (переменные) этого типа данных, только при этом компилятор выделит необходимую память.

Рассмотрим пример: сведения о студенте содержат следующие данные:

- фамилия - **Fam**;
- имя - **Name**;
- год рождения - **Year**;
- пол - **Sex**;
- средний балл - **Grade**.

Представим все эти данные в виде единой структуры. Введем новый тип данных (назовем его **t_Student**) для описания этой структуры:

```
struct  t_Student {
 char  Fam [20],
                Name [16];
 short  Year;
 bool  Sex;
 float  Grade;
};
```

t_Student St1, St2; // Определены две переменные типа t_Student

Определение переменных можно осуществить одновременно с описанием типа данных структуры:

```
struct t_Student {
 char Fam [20],
      Name [16];
 short Year;
 bool Sex;
 float Grade;
} St1, St2;
```

Все поля структурных переменных располагаются в непрерывной области памяти одно за другим. Общий объем памяти, занимаемый структурой, равен сумме размеров всех полей структуры. Для определения размера структуры следует использовать инструкцию **sizeof ()**: **sizeof (t_Student)** или **sizeof (St2)**.

Для того чтобы записать данные в структурную переменную, необходимо каждому полю структуры присвоить определенное значение. Для этого необходимо научиться получать доступ к полям. Для этого используется оператор “точка”. Например:

strcpy (St1.Fam, “Иванов”);
strcpy (St1.Name, “Владимир”);
St1.Year = 1995;
St1.Sex = true; // Надо договориться какое значение соответствует значению пола
St1.Grade = 4.67;

Копирование данных из одной структурной переменной в другую осуществляется простой операцией присваивания не зависимо от количества полей и размера структуры (это можно делать только в том случае, когда обе переменные одного и того же типа):

St2 = St1;

Теперь переменная **St2** содержит те же данные, что и переменная **St1**.

В программировании очень часто используются такие конструкции, как массивы структур. Например, сведения о студентах некоторой учебной группы можно хранить в массиве студентов:

t_Student Gruppa [25];

Здесь мы определили 25-ти элементный массив, каждый элемент которого предназначен для хранения данных одного студента.

Получение доступа к данным некоторого студента из группы осуществляется обычной индексацией переменной массива:

St1 = Gruppa [10]; // Переменная **St1** содержит сведения об 11-ом студенте

Доступ к некоторому полю студента внутри массива делается так:

double grade = Gruppa[10].Grade; // Переменная **grade** содержит среднюю оценку 11-ого студента

Если некоторое поле структуры представляет собой массив (например, поле **Fam** – это массив символов), доступ к отдельному элементу этого массива можно выполнить так:

St1.Fam[5] = ‘ш’;

Или так:

Gruppa[10].Fam[5] = ‘ш’;

Определяем новый тип данных и переменные для двух групп:

```
struct  t_Gruppa {
  short  NumGr;                    //  Номер группы
  short  Count;                    //  Количество студентов в группе
  t_Student  Students[25];         //  Массив студентов группы
}  Gr1372,  St1373;
```

Получим данные о некотором студенте из группы **Gr1372**:

```
St1 = Gr1372.Students[10];        //  Переменная St1 содержит сведения об 11-ом студенте
```

А вот его средний балл:

```
grade = Gr1372.Students[10].Grade;
```

В структуре в качестве поля нельзя использовать элемент, тип которого совпадает с типом самой структуры (рекурсивное использование структур запрещено).

Указатели на структурные переменные определяются точно так же, как и для обычных переменных:

t_Student * p_Stud; // Переменная **p_Stud** указатель на тип данных **t_Student**
p_Stud = & St1; // Переменной **p_Stud** присвоен адрес переменной **St1**

Разыменование указателя (обращение к данным по адресу, хранящемуся в указателе) осуществляется обычным образом:

St2 = * p_Stud; // Данные по адресу **p_Stud** скопированы в переменную **St2**

Через указатели можно работать с отдельными полями структур. **Для доступа к полю структуры через указатель используется оператор “стрелка”, а не “точка”**:

grade = St1.Grade; // Доступ к полю **Grade** через обычную структурную переменную с помощью оператора “точка”

grade = p_Stud -> Grade; // Доступ к полю **Grade** через указатель на структурную переменную с помощью оператора “ стрелка”

p_Stud -> Grade = 3.75; // Полю **Grade** через указатель на структурную переменную присвоено новое значение

Передача данных по значению:

```
void WriteStudent ( t_Student  S )
{
  cout << setw(14) << "Фамилия: " << S.Fam << endl;
  cout << setw(14) << "Имя: " << S.Name << endl;
  cout << setw(14) << "Год рождения: " << S.Year << endl;
  if ( S.Sex )
       cout << setw(14) << "Пол: " << "М\n";
  else
       cout << setw(14) << "Пол: " << "Ж\n";
  cout << setw(14) << "Средний балл: " << S.Grade << endl;
}
```

Передача данных через указатель:

```
void WriteStudent ( t_Student *S )
{
  cout << setw(14) << "Фамилия: " << S -> Fam << endl;
  cout << setw(14) << "Имя: " << S -> Name << endl;
  cout << setw(14) << "Год рождения: " << S -> Year << endl;
  if ( S -> Sex )
      cout << setw(14) << "Пол: " << "М\n";
  else
      cout << setw(14) << "Пол: " << "Ж\n";
  cout << setw(14) << "Средний балл: " << S -> Grade << endl;
}
```

Передача данных по ссылке:

```
void WriteStudent ( t_Student &S )
{
  cout << setw(14) << "Фамилия: " << S.Fam << endl;
  cout << setw(14) << "Имя: " << S.Name << endl;
  cout << setw(14) << "Год рождения: " << S.Year << endl;
  if ( S.Sex )
      cout << setw(14) << "Пол: " << "М\n";
  else
      cout << setw(14) << "Пол: " << "Ж\n";
  cout << setw(14) << "Средний балл: " << S.Grade << endl;
}
```

Если необходимо предотвратить изменения переданных по адресу аргументов, можно при определении соответствующего параметра объявить его константой (использовать спецификатор **const**). Например, так:

```
void WriteStudent ( const t_Student &S )
{
  ……….
}
```