1 Занятие №6

1.1 Структуры

Ранее мы рассмотрели методы определения и работы с массивами и перечислимыми типами. Теперь рассмотрим определение и использование *структур*. Структура в языке Си — это объединение нескольких компонент произвольных типов, то есть это аналог понятия «запись» в языке Паскаль.

Определение структуры имеет вид

```
struct <тег структуры>
{
  <список определений полей>
};
```

Список определений полей имеет точно такой же вид, как и в случае определения локальных или глобальных переменных (однако инициализация не допускается). Например,

```
struct circle
{
  double x, y, r;
  int    c;
};
```

Для того, чтобы определить переменную данного типа, как и в случае перечислимых типов, нужно использовать тег структуры вместе с ключевым словом **struct**. Например,

```
struct circle c1;
```

Обратите внимание, что в отличие от языка Си++ использование ключевого слова **struct** с тегом структуры *обязательно*.

Все теги (структур, объединений, перечислений) вместе образуют отдельное пространство имён, то есть теги не пересекаются с идентификаторами и метками.

Имена полей структуры должны быть уникальны в пределах этой структуры. В разных структурах допускаются поля, имеющие одинаковое имя.

Структура может содержать определения других структур или перечислимых типов, но в этом случае вложенное определение будет видно на том же лексическом уровне, на котором определена сама структура. Например,

```
struct foo
{
  struct bar
  {
    double x;
  };
  int        y;
  struct bar z;
};
struct bar t;
```

Является допустимым фрагментом на Си.

Для структурных типов допускается неполное определение типа. Вводится только тег структуры, но не определяется, какие поля содержит данная структура. Например,

```
struct tree;
```

Размер такой структуры неопределён, и попытка определения объекта такого типа (например переменной) вызовет ошибку до тех пор, пока структура не будет определена полностью, как показано выше. Тем не менее, можно определять переменные типа указателя на эту структуру.

Переменные одного и того же структурного типа можно присваивать друг другу. Два переменные считаются одного и того же типа, если при их определении был использован один и тот же тег структуры (эквивалентность типов по имени). Два типа структуры с разными тегами, даже содержащие одни и те же поля не считаются эквивалентными. Структуры можно передавать в качестве параметров функций (передаются по значению) и возвращать в качестве результата функции.

Доступ к полю переменной структурного типа производится указанием имени переменной, затем оператор «точка» и имя поля. Например,

```
a.b, a[10].b, a.foo.bar;
```

Размер структуры, определяемый операцией **sizeof**, может быть больше, чем сумма размеров всех составляющих её полей. Архитектура, на которой компилируется программа, может, например, требовать, чтобы целые числа были выравнены по границе слова. Поэтому компилятор может оставлять неиспользуемое пространство между полями для обеспечения требований архитектуры.

1.1.1 Рекурсивные типы данных

Тип структуры не может иметь поля типа самой этой структуры и неполных типов, однако допускаются указатели на тип самой структуры и на неполные типы. Например, тип элемента списка может быть объявлен следующим образом:

```
struct list
{
  struct list *next;
  int data;
};
```

А если требуется взаимно рекурсивный тип данных, можно использовать неполное объявление:

```
struct node2;
struct node1
{
  struct node2 *ptr;
  int data;
};
struct node2
{
  struct node1 *ptr;
  double data;
};
```

Память под элементы рекурсивных типов выделяется, как правило, в куче, и работа с ними ведётся через указатели. Пусть p объявлен как

```
struct list *p;
```

тогда память под один элемент списка выделяется с помощью вызова `calloc` следующим образом:

```
p = (struct list*) calloc(1, sizeof(*p));
```

предпочтительнее использовать `calloc`, так как выделяемая память будет в этом случае обнулена.

Для работы с полями через указатель можно использовать комбинацию `.` и `*`: `(*p).next = 0;`, но здесь требуются скобки из-за приоритета операций. Для сокращенной записи в **Си** используется операция `->`, то есть приведённый выше фрагмент будет записан следующим образом: `p->next = 0;`.

1.2 Объединения

Объединение — это структура данных, очень похожая на структуру. Определение объединения выглядит следующим образом:

```
union <тег объединения>
{
  <список определений полей>
};
```

Все прочие аспекты работы с объединениями, такие как правила перекрытия, доступ к полям — точно такие же, как и для структур. Единственное отличие структур от объединений состоит в том, что все элементы, составляющие объединение, располагаются по одному адресу, перекрывая друг друга. Например,

```
struct s { short  low, high; };
union  i
{
  int      val_i;
  struct s val_s;
};
```

Это способ, как можно разбить целое значение на младшую и старшую половины **short**, при условии, что тип **int** в два раза длиннее **short**. Обращаться можно без ограничений к любым полям, например

```
union i val_i;
val_i.val_i = x;
printf("%d,%d\n", val_i.val_s.low, val_i.val_s.high);
```

Весь контроль за правильностью таких манипуляций возлагается на программиста.

Типы объединения в основном используются для моделирования «вариантных записей», какие существуют в языке Паскаль, и для моделирования механизма наследования объектно-ориентированных языков.

Рассмотрим такой пример. Часто в структурах данных типа «дерево» в узлах дерева необходимо хранить другую информацию, нежели в листьях. Так, в деревьях синтаксического разбора в узлах хранятся код операции и ссылки на поддеревья, а в листьях — значения констант, имена переменных и т. д. В этом случае поступают следующим образом: создают перечислимый тип, в котором определяют константы для всех возможных типов узла дерева,

определяют структуры для хранения данных в узлах каждого типа, а затем объединяют все структуры в одно объединение.

```
enum node_type {NODE_ADD, NODE_MUL, NODE_SUB, NODE_CONST };
union tree_node;
struct node_binop { int tag; union tree_node *left, *right; };
struct node_const { int tag; int val; };
union tree_node
{
  int tag;
  struct node_binop b;
  struct node_const c;
};
```

Работа с таким типом узла дерева может происходить, например, так.

```
union tree_node *p = (union tree_node*) calloc(1,sizeof(*p));
p->tag = NODE_CONST;
p->c.val = 1;
```

1.3 Многомерные массивы

Двухмерные массивы (матрицы) определяются следующим образом

```
<тип> <имя>[разм. 1][разм. 2];
```

Обратите, что каждое из измерений массива записывается в отдельной паре квадратных скобок. Массивы большей размерности определяются аналогично.

В памяти массивы размещаются по строкам, то есть быстрее всего изменяется последний индекс. Например для матрицы `int x[2][3]` элементы в памяти будут размещены следующим образом

```
x[0][0],x[0][1],x[0][2],x[1][0],x[1][1],x[1][2]
```

Поскольку матрица располагается по строкам, выражение `x[i]` имеет тип `int [3]`, то есть каждую строку матрицы можно рассматривать как целое (например, передавать параметром в функции). Аналогично и для массивов большей размерности.

Обращение к элементу массива выглядит так `x[i][j]`. Каждый индекс всегда записывается в своей паре квадратных скобок.

Правила передачи многомерных массивов в качестве параметров в функции отличаются в C90 и C99. В старом стандарте языка требовалось, чтобы при передачи многомерных массивов в функции у них должны быть указаны все измерения, *кроме первого*. Это связано с тем, что для вычисления адреса элемента массива компилятор должен знать все измерения, кроме первого (напишите формулу вычисления адреса и проверьте!).

Стандарт C99 допускает многомерные массивы переменного размера. Поэтому, например, функция матричного умножения может иметь следующий заголовок:

```
void mmult(int n, double a[n][n], double b[n][n], double c[n][n])
{
  // ...
}
```

В прототипах функций допускается указание * вместо переменного размера:

```
void mmult(int, double [*][*], double [*][*], double [*][*]);
```

1.4 Выражения

1.4.1 Перечисление операций

1. Наивысший приоритет имеют постфиксные операции `++`, `--`, `[]`, `()`, `.`, `->`. Читаются слева направо.

1	`++`	постинкремент
	`--`	постдекремент
	`[]`	индексация массива, например `a[i]`
	`()`	вызов функции, например `sin(M_PI)`
	`.`	доступ к полю
	`->`	доступ к полю через указатель

2. Следующая группа операций — префиксные операции. Читаются справа налево.

2	`++`	преинкремент
	`--`	предекремент
	`&`	взятие адреса
	`*`	разыменование
	`-`	унарная смена знака
	`+`	
	`!`	логическое отрицание
	`~`	побитовое отрицание
	`sizeof`	вычисление размера типа
	`()`	приведение типа

3. Остальные операции по мере убывания их приоритета. Читаются слева направо.

3	`*`	умножение
	`/`	деление
	`%`	взятие остатка
4	`+`	сложение
	`-`	вычитание
5	`<<`	сдвиг влево
	`>>`	сдвиг вправо
6	`<`, `<=`, `>`, `>=`	операции сравнения
7	`==`, `!=`	операции сравнения
8	`&`	побитовая операция AND
9	`^`	побитовая операция XOR
10	`|`	побитовая операция OR
11	`&&`	логическое «и»
12	`||`	логическое «или»
13	`?:`	условная операция, читается справа налево
14	`=`	присваивание, читается справа налево
	`*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	присваивания с операцией, читается справа налево
15	`,`	последовательное вычисление

1.5 Операции ++ и --

Основной эффект операции — это выработка некоторого значения, которое может быть далее использовано в выражении. Побочный эффект — это другое воздействие на среду выполнения (например, изменение значения переменной).

В языке Си операции инкремента (увеличения на 1) переменной ++, и декремента (уменьшения на 1) переменной -- существуют в двух формах: префиксной и постфиксной, которые отличаются основным эффектом операции. Преинкремент `++a` — значение переменной вначале увеличивается на 1, и это уже увеличенное значение является значением операции, и далее в выражении может быть использовано. Постинкремент `a++` — значением этой операции является значение переменной до увеличения на 1. Например,

```
int a = 5, b, c;
b = 5 + ++a;
c = 5 + a++;
```

значением переменной `b` будет 11, а переменной `c` — 10.

1.5.1 Операции << и >>

Бинарные операции << и >> выполняют сдвиг влево и вправо своего первого аргумента на количество бит, задаваемое вторым аргументом. Операции применимы только к аргументам целых типов. Тип результата совпадает с типом первого аргумента. Если количество бит сдвига оказывается отрицательным или большим количества бит в первом аргументе, результат операции неопределён.

Операция >> может выполнять как знаковый (арифметический), так и беззнаковый (логический) сдвиг вправо. Если тип первого аргумента — беззнаковый, то выполняется логический сдвиг влево, то есть на место старшего бита записывается 0. Если тип первого аргумента — знаковый, то выполняется арифметический сдвиг влево, то есть старший бит сохраняется.

1.5.2 Операции && и ||

Операции `&&` и `||` вычисляются по «короткой» схеме, то есть если значение всего выражения известно после вычисления первого операнда, второе выражение не вычисляется. Операцию `a && b` можно понимать так:

```
if (a) { if (b) 1; else 0; } else 0;
```

а операцию `a || b` так:

```
if (a) 1; else if (b) 1; else 0;
```

1.5.3 Операция ?:

Операция `?:` тернарная, то есть имеет три операнда, например `a?b:c`. Сначала вычисляется условие `a`. Если оно истинно, тогда вычисляется `b`, а `c` не вычисляется, а если `a` ложно, тогда вычисляется `c`, а `b` не вычисляется. Эта операция читается справа налево, то есть выражение `a?b:c?d:e` эквивалентно `a?b:(c?d:e)`, выражение `a?b?c:d:e` эквивалентно `a?(b?c:d):e`.

1.5.4 Операция ,

Операция `,` («запятая») последовательно вычисляет оба аргумента. Значением операции является значение второго аргумента, значение первого теряется.

1.6 Упражнения

1. Дан указатель на начало односвязного списка из элементов типа **`int`**. Напишите фукнцию `free_list`, которая освобождает всю память, занимаемую элементами списка.
2. Напишите функцию `merge`, которая сливает два односвязных списка с элементами типа `int` в один и возвращает указатель на его начало.
3. Напишите функцию `add_tree`, добавляющую новый элемент в бинарное дерево поиска.
4. Напишите функцию `madd`, которая складывает две матрицы произвольного размера.
5. Напишите функцию `getbit`, которая возвращает по одному биту из входного потока.