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.

1.2 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. Например, если мы хотим определить тип структуры, который может содержать информацию о геометрических фигурах, мы можем поступить, например, так

```
enum fig_type {square, rect, circle, ellipse};
struct fig_square  { int type; double a; };
struct fig_rect    { int type; double a, b; };
struct fig_circle  { int type; double r; };
struct fig_ellipse { int type; double a, b; };
union fig
{
  int                type;
  struct fig_square  square;
  struct fig_rect    rect;
  struct fig_circle  circle;
  struct fig_ellipse ellipse;
};
```

Работа с таким объединением может происходить, например так.

```
union fig figure;
figure.type = square;
figure.square.a = 1.4;
```

1.3 Указатели

Указатель — это переменная, которая хранит адрес другой объекта, например, переменной или функции. Аналогом являются ссылочные типы в языке Паскаль. Простейшее определение переменной указательного типа имеет вид:

```
<тип> *<переменная>;
```

Знак `'*'` («звёздочка») относится к имени переменной, а не к типу, поэтому определение

```
int *p, c;
```

определяет указатель на **int** и переменную типа **int**. Функция, возвращающая указатель, определяется похожим образом

```
char *strdup(char *ptr);
```

Аналогично массив указателей

```
int *aptr[20];
```

Допускаются двойные указатели, тройные указатели и т. д. Полные правила чтения и записи деклараторов мы рассмотрим позже.

Указатели могут хранить адрес любого объекта в программе, при условии, что типы объекта и указателя согласованы. На самом деле, поскольку адрес объекта, как правило, имеет размер, не зависящий от размера объекта, возможно практически неограниченное явное приведение одних указательных типов к другим.

Для взятия адреса объекта используется операция `&`. Например, выражение

```
p = &c;
```

присваивает переменной указательного типа p адрес переменной c. Точно так же могут браться адреса элементов массива, полей структур, локальных и глобальных переменных и пр.

Для разыменования указателя (взятия значения по адресу, хранящемуся в указателе) применяется унарная операция *. Например, `printf("%d\n", *p);`.

В языке Си массивы и указатели тесно связаны друг с другом. Во-первых, имя массива является константным указателем на его первый элемент, соответственно любой указатель можно рассматривать как массив из некоторого числа элементов, в простейшем случае, как массив из одного элемента. Например, если есть определение переменной

```
int arr[32], *p;
```

то выражения `&arr[0]` и `arr` полностью эквивалентны друг другу. Например, после выполнения инструкции

```
p = arr;
```

указатель p будет содержать адрес нулевого элемента массива, и имя указателя p может везде использоваться для обращения к массиву arr. Например,

```
p[0] = 2;
c = p[3] + p[4];
```

Существует два отличия между переменной, объявленной как массив и переменной, объявленной как указатель: **во-первых**, имени массива нельзя присвоить никакого значения, то есть выражение вида

```
arr = &p[3];      /* неправильно! */
```

во-вторых, при определении переменной типа массива под эту переменную отводится память, достаточная для хранения заявленного в определении переменной числа элементов массива, а при определении указателя отводится память, достаточная для хранения указателя. Так, для массива arr будет отведено 128 байт (в предположении, что **int** занимает 4 байта), а для указателя p — только 4 байта (если указатель занимает 4 байта памяти).

Указатели одного типа можно сравнивать друг с другом на равенство и неравенство. Любой указательный тип может сравниваться на равенство или неравенство с целой константой 0. Любому указателю может присваиваться целая константа 0, что означает, что данный указатель не указывает ни на какую область памяти. Стандартная библиотека определяет константу NULL, которую можно использовать для символической (возможно более наглядной) записи нулевого указателя. Попытки записи или чтения по нулевому указателю обычно вызывают исключение операционной системы. Если же указатель не инициализирован, то он указывает на некоторую случайную область памяти. Выражение указательного типа может использоваться в условиях циклов, оператора **if** и т. д. Это, как всегда, обозначает неявное сравнение с нулём. Например:

```
for (p = head; p; p = p->next)
```

цикл с таким заголовком будет повторяться до тех пор, пока p не обратиться в нуль.

Указатель можно складывать с целой величиной. Результатом такой операции является указатель того же типа. Этот указатель будет указывать на элемент массива, отстоящий от исходного элемента массива на указанное число элементов. Например, если `int arr[10], *p = &arr[5];`, тогда

```
p + 3 == &arr[8];
p - 4 == &arr[1];
```

В языке Си справедливо тождество (на самом деле, это определение операции индексирования), связывающее операцию индексирования и арифметические операции с указателями

```
arr[ind] == *(arr + ind)
```

Соответственно, для указателей определены операции -, +=, -=, ++, --.

Два указателя можно вычитать друг из друга. Эти два указателя должны указывать на элементы одного и того же массива (должны, естественно, иметь один и тот же тип). Результатом вычитания является целое число — разность между указателями, выраженная в числе элементов массива данного типа. Если p — указатель, то

```
(p + 1) - p == 1
```

для любого типа на который указывает переменная p.

В языке Си определён обобщённый указатель, который записывается как

```
void *ptr;
```

то есть как указатель на «тип» **void**. Такому указателю можно присваивать значения указателей любого типа, и такой указатель можно присваивать указателю любого типа (только в Си, но не в Си++). *Указатель обобщённого типа нельзя разыменовывать и с ним нельзя проводить арифметические операции* (но с нулём сравнивать можно).

В качестве примера рассмотрим возможную реализацию функции strcpy.

```
char *
strcpy(char *str1, char *str2)
{
  char *d = str1;
  while ((*d++ = *str2++));
  return str1;
}
```

1.4 Работа с динамической памятью

В языке Си помимо области памяти, выделяемой во время компиляции программы для глобальных переменных, и области памяти в стеке, выделяемой для локальных переменных динамически при каждом вызове функции, существует область памяти (традиционно называемая «кучей»), представляющая собой нечто среднее между этими двумя типами памяти. Куча существует все время выполнения программы, но память в ней выделяется и освобождается динамически, по требованию программиста. Средства работы с динамической памятью не встроены в язык, а предоставляются стандартной библиотекой. Чтобы использовать функции работы с динамической памятью, необходимо подключить заголовочный файл

```
#include <stdlib.h>
```

Определены следующие стандартные функции

```
void *malloc(size_t size);
```

выделяет в куче область памяти размера `size` и возвращает указатель на начало этой области памяти. Здесь `size_t` — это тип, определённый стандартом для представления величин, задающих размер объектов. Операция **`sizeof`** вырабатывает значение именно этого типа. Обычно тип `size_t` определяется эквивалентным типу **`unsigned long int`**. Если `size` равно нулю, результат работы функции неопределён.

```
void *calloc(size_t nitems, size_t nsize);
```

выделяет в куче область памяти для размещения массива из `nitems` элементов, каждый из которых имеет размер `nsize`, и возвращает указатель на выделенную область памяти. Выделенная область памяти инициализируется нулями. Если функции `calloc` или `malloc` не могут выделить область памяти достаточного размера, они возвращают нулевой указатель (0 или `NULL`). *Программист должен проверять результат, возвращаемый этими функциями и предпринимать действия по обработке ошибок*, если был возвращён нулевой указатель.

```
void free(void *ptr);
```

освобождает ранее выделенный в куче блок памяти и делает его доступным для повторного использования. Переданный функции `free` указатель должен быть получен от функций `malloc`, `calloc` или `realloc`. Если аргумент функции не является таким указателем, результат работы функции `free` неопределён (чаще всего крах программы). Если блок памяти был уже ранее освобождён функцией `free`, повторное освобождение одного и того же блока памяти неопределено. После того, как блок был освобождён, с памятью в этом блоке нельзя проводить никаких операций, даже чтение. Не гарантируется, что значения, записанные в освобождаемой области памяти, будут сохранены после вызова `free`. Вызов `free(0)` безопасен и не производит никаких действий.

```
void *realloc(void *ptr, size_t newsize);
```

изменяет размер выделенного блока памяти. Если `ptr == NULL`, функция `realloc` работает в точности как `malloc`, а если `newsize == 0`, `realloc` работает как `free`. Иначе функция выделяет в динамической памяти блок размера `newsize` и копирует в него начало блока памяти по адресу `ptr`. Если новый блок больше старого, остаток памяти нового блока не инициализируется. Не гарантируется, что новый блок памяти будет начинаться с того же адреса памяти, что и предыдущий, даже если размер блока памяти был уменьшен. После функции `realloc` уже нельзя пользоваться старым блоком памяти, выделенным по адресу `ptr`. Если невозможно выделить область памяти заданного размера, функция возвращает нулевой указатель. В этом случае область памяти по старому адресу `ptr` не изменяется и по-прежнему доступна для использования.