1 Занятие №5

1.1 Указатели

Рассмотрим важнейший производный тип — *указательный*. Если *base* — это некоторый базовый тип, то тип указателя на тип *base* имеет своим множеством значений всевозможные адреса памяти, по которым на данной машине могут храниться значения типа *base*, плюс специальное значение **0**, которое не соответсвует никакому допустимому адресу объекта.

Другими словами в переменных указательного типа хранятся адреса объектов, то есть переменных или функций. Аналогом указательных типов в языке **Паскаль** являются ссылочные типы.

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 переменную `p` указательного на **`int`** типа и переменную `c` целого типа. Функция, возвращающая указатель, определяется похожим образом

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

Аналогично массив указателей

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

Указательный тип может иметь в качестве своего базового типа любой тип, в частности указательный, структурный, массивовый и т. д. Синтаксис таких сложных описаний (деклараторов) будет рассмотрен нами позднее.

Переменные указательного типа 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 значений одних указательных типов к другим. Язык **Си**, хотя и не поощеряет такие приведения типов, не запрещает их.

Для взятия адреса существующего 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 указательного значения (взятия значения по адресу) применяется унарная операция `*`. Например, `printf("%d\n", *p);`.

Даже если указатель `p` содержит 0 или недопустимое значение, выражение `&(*p)` всегда тождественно равно `p`. То же верно и для всякого выражения указательного типа. Обратное выражение `*&E`, очевидно, не всегда равно `E`.

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 (точнее, с константным выраженим, равным 0 какого-либо целого типа). 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 обозначает неявное сравнение значения выражения с нулевым указателем 0. Например цикл

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

будет работать до тех пор, пока p не обратиться в нуль.

В языке Си существует специальный тип обобщённого указателя, 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).

1.1.1 Массивы и указатели

В языке Си массивы и указатели тесно связаны друг с другом.

Во-первых, выражение, состоящее из имени массива, имеет тип константного указателя на первый элемент массива. То есть имя массива можно рассматривать как константный указатель на его первый элемент.

И наоборот, любое указательное значение можно рассматривать как адрес первого элемента некоторого массива, и соответственно, любое указательное значение можно индексировать.

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).

1.1.2 Арифметика указателей

На связи указателей и массивов построена арифметика указательных значений. Пусть переменная p указательного типа указывает на i-й элемент некоторого массива arr, то есть p = &arr[i], а переменная q — на j-й элемент того же массива arr.

Значение указательного типа можно складывать со значением целого типа, и из значения указательного типа можно вычитать значение целого типа. Тип результата этих операций совпадает с указательным типом, а результат — это указательное значение, указывающее 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];
```

Таким образом устанавливается тождественная связь между операцией индексирования и сложением указателя и целого значения:

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

Такие тождества рассматриваются в Си как определение операции индексирования.

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. Пусть p и q определены как указано выше.

```
p - q == i - j
```

Если p и q не указывают на элементы одного и того же массива, результат операции p - q не определён.

Если p — указатель, то

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

для любого типа на который указывает переменная p.

В качестве примера рассмотрим возможную реализацию функции strcpy.

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

1.2 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. По сути, памятью, выделяемой в куче,

нужно пользоваться, когда размер входных данных программы заранее неизвестен, то есть практически всегда.

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.

Функция `free` не требует параметра размера блока освобождаемой памяти, потому что размер блока памяти хранится в специальных структурах, поддерживаемых функциями работы с динамической памятью, непосредственно перед началом выделенного блока памяти. Поэтому запись за пределы выделенного блока, например, вследствие недостаточного контроля индексов массивов, может привести к разрушению служебной информации и краху программы. Как правило крах программы происходит при работе функций выделения или освобождения памяти. Поэтому, если Ваша програма завершилась с ошибкой `Segmentation fault`, и дамп стека показывает, что ошибка произошла, например, в функции `free`, это не значит, что `free` реализована неверно, но где-то в Вашей программе есть ошибка, которая приводит к записи вовне выделенного блока памяти и разрушению служебных структур.

```
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.

Функцию `realloc` можно использовать для реализации так называемых *расширяемых массивов*, то есть массивов, размер которых динамически увеличивается, когда очередная порция данных в них не может поместиться.

Рассмотрим следующий пример: на стандартном потоке ввода задаётся последовательность вещественных чисел. Ввод завершается концом файла. Необходимо распечатать на стандартный поток вывода только числа, не превышающие среднего значения всех введённых чисел. Для хранения чисел мы будем использовать расширяемый массив.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 int main(void)
 4 {
 5   int arr_a = 0, arr_u = 0, i;
 6   double *arr = 0;
 7   double s = 0, v;
 8   arr_a = 16;
 9   if (!(arr = (double*) malloc(arr_a * sizeof(arr[0]))))
10     goto out_of_mem;
11   while (scanf("%lf", &v) == 1) {
12     if (arr_u == arr_a) {
13       arr_a *= 2;
14       if (!(arr = (double*) realloc(arr, arr_a * sizeof(arr[0]))))
15         goto out_of_mem;
16     }
17     s += v;
18     arr[arr_u] = v;
19     arr_u++;
20   }
21   if (!arr_u) return 0;
22   s /= arr_u;
23   for (i = 0; i < arr_u; i++)
24     if (arr[i] <= s) printf("%g\n", arr[i]);
25   return 0;
26  out_of_mem:
27   fprintf(stderr, "memory␣exhausted\n");
28   return 1;
29 }
```

1.3 Упражнения

1. Используя только указательную арифметику, то есть не использования операций индексирования массива и переменных-индексов, напишите функцию `strlen` с прототипом:

```
int strlen(char *s1);
```

2. Используя только указательную арифметику, то есть не использования операций индексирования массива и переменных-индексов, напишите функцию `strspn` с прототипом:

```
int strspn(char *s1, char *s2);
```

   Функция просматривает строку `s1` и находит в ней первый символ, который входит в строку `s2`. Строка `s2` рассматривается, как множество символов, которые ищутся в `s1`. Если поиск завершился успешно, функция должна вернуть индекс первого такого символа. Если в строке `s1` нет символов из `s2`, функция возвращает `-1`.

3. Напишите функцию

```
char *concat(char *s1, char *s2);
```

   которая конкатенирует строки `s1` и `s2` и возвращает указатель на строку-результат. Память под строку-результат выделяется в куче.

4. Напишите функцию

```
char *getline(void);
```

   которая считывает строку текста со стандартного потока ввода. Строка завершается символом `'\n'` или признаком конца файла. В случае, если не считан ни один символ строки, и сразу получен конец файла, возвращается 0. Символ `'\n'` в строку не добавляется.

5. С использованием функции `getline` напишите программу, которая печатает в обратном порядке строки текста на стандартном потоке ввода.