Лекция 9.

9.1. *Функции*

9.1.1. ***Объявление функции***. Синтаксис:

тип_возвр_значения имя_функции`(`*тип параметр, тип параметр, …, тип параметр*`);`

Примеры:

```
int atoi(char s[]);
double atof(char s[]);
void QuickSort(char *items, int count);
```

Функция **`atoi`** преобразует строку десятичных цифр **`s`** в целое число, функция **`atof`** преобразует строку десятичных цифр **`s`** в число двойной точности, функция **`QuickSort`** выполняет сортировку массива **`items`**.

Тип возвращаемого значения **`void`** означает, что функция не возвращает значения.

9.1.2. ***Определение функции***. Синтаксис:

объявление_функции `{`*тело_функции*`}`

Пример:

```
/* вставление объявлений функций библиотеки ctype */
#include <ctype.h>
int atoi(char s[]) {
     int i, n, sign;
     /* пропуск пробельных символов */
     for(i = 0; isspace(s[i]); i++);
     /* определение знака числа по первому знаку */
     sign = (s[i] == '-') ? -1: 1;
     /* пропуск остальных знаков, если их несколько */
     if(s[i] == '+' || s[i] == '-') i++;
     /* вычисление значения числа */
     /* цикл должен начаться с текущего значения i */
     /* n = 0 "побочный эффект", позволяющий */
     /* сэкономить одну строку в программе */
     for(n = 0; isdigit(s[i]); i++)
          n = 10 * (s[i] - '0');
     return sign * n;
}
```

Стандартная библиотека **`ctype`** содержит такие функции, как **`isspace()`** , **`isdigit()`** и др.

Областью действия функции является *весь* программный файл, в котором она объявлена, начиная со строки, содержащей ее объявление. При этом определение функции должно присутствовать только в одном из программных файлов, в которых она объявлена, либо в одной из библиотек.

9.1.3. ***Вызов функции***.

Если функция **`f()`** возвращает значение типа *тип*, то вызов этой функции может иметь вид: **`v = f()`**, где **`v`** – переменная типа *тип*. Вызов функции может также входить в состав выражения типа *тип*.

Если функция **`f(`***параметр***`)`** не возвращает значений, вызов этой функции имеет вид: **`f(`***аргумент***`);`**

Если в программном файле вызывается какая-либо функция, она *обязательно должна быть объявлена в этом программном файле до ее вызова* (отсутствие объявления функции или объявление функции *после* ее вызова фиксируется компилятором как ошибка).

Директива препроцессора **#include <*имя_библиотеки*.h>** вставляет в программу (перед ее компиляцией) объявления всех функций соответствующей библиотеки (в частности, стандартной библиотеки), обеспечивая программе возможность вызова функций соответствующей библиотеки. Например, **#include <string.h>** вставляет объявления всех функций стандартной библиотеки **string**.

При вызове функции ей передаются ее аргументы. В языке Си все аргументы передаются по значению (т.е. передаются только значения аргументов, и эти значения копируются в память функции). Если аргументом является указатель, его значением является адрес объекта вызывающей функции, что обеспечивает вызываемой функции доступ к соответствующему объекту. Массив всегда передается с помощью указателя на его первый элемент (напомним, что указателем на первый элемент массива является, в частности, имя массива без индексов).

9.1.4. ***Пример***. Функция **void swap(int x, int y)**, меняющая местами значения переменных **x** и **y**.

Неправильный вариант:

```
void swap(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Правильный вариант:

```
void swap(int *px, int *py) {
     int tmp;
     tmp = *px;
     *px = *py;
     *py = tmp;
}
```

Неправильный вариант не учитывает того обстоятельства, что *все аргументы функции передаются только по значению*: операция **x = y;** не имеет смысла (ошибочна), так как значение **x** (целое число) не может быть левой частью присваивания.

Используя аргументы-указатели, функция может обращаться к объектам вызвавшей ее функции.

9.1.5. ***Возврат из функции***. Возврат из функции в точку вызвавшей ее функции, следующей за точкой вызова функции (это может быть следующий член выражения, если вызов функции входит в состав выражения, либо следующий оператор, если вызов функции не входит в состав выражения), осуществляется либо при выполнении оператора **return**, либо после выполнения последнего оператора функции, если она не содержит оператора **return**. Например, возврат из функции, печатающей строку **s** в обратном порядке.

```
#include <string.h>
#include <stdio.h>
void str_reverse(char *s) {
     register int i;
     for(i = strlen(s) - 1; i >= 0; i--) putchar(s[i]);
}
```

выполняется сразу после окончания выполнения оператора цикла **for**.

Если у функции несколько операторов **return**, возврат осуществляется по тому из них, который будет выполнен первым.

9.1.6. ***Результат выполнения функции***. Все функции, кроме тех, которые относятся к типу **void**, возвращают значение. Это значение определяется выражением в операторе **return**. Согласно стандарту *C*99, в функции, тип которой отличен от **void**, каждый оператор **return** *обязательно* должен содержать возвращаемое значение[1]. Однако, если возврат из функции, тип которой отличен от **void**, выполняется не по оператору **return**, а после выполнения последнего оператора этой функции, то возвращается произвольное значение. Рекомендуется избегать таких случаев.

Помимо вычисления возвращаемого значения функция может изменять значения переменных вызывающей программы, осуществляя доступ к ним с помощью указателей, переданных ей в качестве аргументов, а также изменять значения глобальных переменных программы. Результаты вызова функции, не связанные непосредственно с вычислением возвращаемых значений, составляют *побочный эффект* функции.

В Си-программе можно использовать функции трех видов:

(1) Функции, которые выполняют операции над своими аргументами с единственной целью – вычислить возвращаемое значение.

(2) Функции, которые обрабатывают данные и возвращают значение, которое показывает, успешно ли была выполнена эта обработка. Основная работа такой функции заключается в ее побочном эффекте.

(3) Функции, не возвращающие значений. Все такие функции имеют тип **void**.

Примерами функций первого вида могут служить библиотечные функции **sqrt(x)** или **exp(x)**.

В качестве примера функции третьего вида рассмотрим функцию перемножения двух прямоугольных матриц.

```
void matr_prod(double *a, int m, int s, double *b, int
s,
                int n, double *c, int m, int n) {
    int i, j, k;
    for(i = 0; i < m; i++)
         for(j = 0; j < n; j++) {
             c[i, j] = 0;
             for(k = 0; k < s; k++)
                 c[i][j] += a[i][k]*b[k][j];
    }
}
```

Значения, возвращаемые функциями первого и второго вида, не обязательно должны быть использованы в программе (т.е. их можно вообще не использовать).

[1] В стандарте *C*89 в функциях типа, отличного от **void**, разрешаются операторы **return** без возвращаемого значения. В этих случаях вызывающей программе возвращается *произвольное* значение соответствующего типа.

Пример:

```
#include <stdio.h>

int mult(int a, int b);
int main() {
     int x, y, z;
     x = 10; y = 20;
     z = mult(x, y);
     printf("%d", mult(x, y));
     mult(x, y);
     return 0;
}

int mult(int a, int b) {
     return a * b;
}
```

Определение функции **mult** следует после определения. Но так как функция **mult** вызывается в функции **main**, до определения функции **main** функция **mult** должна быть объявлена до определения функции **main** (либо ее определение должно быть до определения функции **main**).

Несмотря на то, в 6-ой строке функции **main** вызывается функция **mult** , а ее значение никакой переменной не присваивается, никаких неприятностей при выполнении функции **main** не возникает. Рассматриваемый вызов функции **mult** является примером «мертвого кода», т.е. вычислений, результаты которых в программе не используются. Большая часть мертвого кода (но не весь мертвый код) исключается при компиляции программы.

Возвращаемым значением может быть указатель. Требуется, чтобы в объявлении такой функции тип возвращаемого указателя был объявлен точно: например, нельзя объявлять возвращаемый тип как **int *** , если функция возвращает указатель типа **char *** . Пример функции, возвращающей указатель (поиск первого вхождения символа **c** в строку **s**):

```
char *match(char, char *s) {
     while(c != *s && *s) s++;
     return (s);
}
```

9.1.7. ***Рекурсия***. В языке Си функция может быть *рекурсивной*, т.е. вызывать саму себя. Простым примером рекурсивной функции является функция, вычисляющая число Фибоначчи по его номеру:

```
int Fibrec(int n) {
     if(n == 1 || n == 2) return 1;
     else return ((Fibrec(n – 2) + Fibrec(n – 1));
}
```

Когда функция вызывает сама себя, в памяти в стеке размещается новый набор локальных переменных и параметров, а код функции выполняется над этими новыми переменными и параметрами с самого своего начала. При этом новая копия кода функции не создается, новыми являются только значения, над

которыми выполняется функция. При каждом возвращении из рекурсивного вызова старые локальные переменные и параметры извлекаются из стека и возобновляется работа функции с того места, которое идет сразу за рекурсивным вызовом.

Хотя может показаться, что рекурсивная функция более эффективна, на самом деле такое случается очень редко. Размер кода функции сокращается не очень сильно, эффективность работы с памятью возрастает также незначительно, зато добавляются накладные расходы на организацию рекурсивных вызовов. Кроме того, при большом количестве рекурсивных вызовов возникает переполнение стека.

Все это нетрудно заметить, сравнивая работу рекурсивной функции **Fibrec** и ее не рекурсивного варианта **Fib**, который приводится ниже:

```
int Fbn(int n) {
   if ((n == 1) || (n == 2))
     return 1;
   else {
     g = h = 1;
     for(k = 2; k < n; k++) {
          Fb = g + h;
          h = g;
          g = Fb;
     }
     return Fb;
   }
}
```

9.1.8. ***Ключевое слово*** **inline**. В стандарт *C*99 добавлено ключевое слово, которое применяется к функциям и определяет так называемую *подставляемую функцию*. Смысл подставляемой функции (и ключевого слова **inline**) станет ясен из следующего примера. Рассмотрим Си-программу

```
#include <stdio.h>

inline int max(int a, int b) {
     return a > b ? a : b;
}
int main() {
     int x = 5, y = 17;
     printf("Наибольшим из чисел %d и %d является %d\n",
          x, y, max(x, y));
     return 0;
}
```

При типичной реализации **inline** приведенная программа эквивалентна следующей:

```
#include <stdio.h>

inline int max(int a, int b) {
     return a > b ? a : b;
}
```

```
int main() {
    int x = 5, y = 17;
    printf("Наибольшим из чисел %d и %d является %d\n",
        x, y, (x > y ? x : y));
    return 0;
}
```

Подставляемые функции помогают создавать более эффективный код (подробности – в следующем семестре).

9.2. *Структуры*

9.2.1. Структура – это совокупность нескольких переменных, часто разных типов, сгруппированных под одним именем для удобства. С помощью структур удобно организовывать сложные типы данных, так как они позволяют группу переменных, связанных между собой информационно и по размещению в памяти, рассматривать и обрабатывать как единое целое, а не как разрозненный набор элементов. Переменные, перечисленные в объявлении структуры, называются ее *полями*, *элементами*, или *членами*.

9.2.2. ***Объявление структуры*** начинается с ключевого слова **`struct`**, за которым следует необязательный идентификатор, называемый *меткой структуры*, а далее в фигурных скобках перечисляются объявления полей структуры. Пример: структура, описывающая точку на координатной плоскости:

```
struct point {
    int x;
    int y;
}; f, g;
```

Здесь **`point`** – метка структуры[2], **`x`** и **`y`** – имена полей структуры.

Объявление структуры фактически вводит новый тип данных. После объявления структуры может стоять список переменных, каждая из которых является именем экземпляра структуры. Переменные **`f`** и **`g`** – это две точки (экземпляры структуры **`struct point`**). Объявления новых экземпляров структуры **`struct point`** можно записать в виде:

```
struct point h, maxpt = {320, 320}
```

Объявление структуры, после которого нет списка переменных, не выделяет никакой памяти для объектов, а просто описывает «шаблон» структуры. Если в объявлении структуры есть метка, ее можно использовать для сокращенного обозначения структуры.

Поля структуры могут иметь любой тип, например, тип массива или тип другой структуры, или даже тип той же самой структуры. Например, объявление

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

определяет прямоугольник, заданный своей диагональю (**`pt1`** и **`pt2`** – концы диагонали).

[2] Метка структуры не является ее именем, более того, в функции может быть объявлена переменная, имя которой совпадает с меткой одной из структур, объявленных в той же функции.

9.2.3. ***Доступ к полям структуры*** осуществляется с помощью операции *точка* "." Например, **`f.x`**, **`f.y`**, **`g.y`** и т.п.

9.2.4. Массивы структур.

9.2.5. Указатели на структуры.

9.3. ***Перечисления***

Перечисления или перечислимые типы относятся к целочисленным типам данных. Они не встроены в язык, а определяются программистом. При определении перечислимого типа вслед за ключевым словом **`enum`** в фигурных скобках задается упорядоченное множество *имен значений* путем перечисления этих имен через запятые. Разрешается перед фигурными скобками поставить имя, которое будет трактоваться как *имя* соответствующего перечислимого *типа*. Каждому имени значения присваивается *порядковый номер*: первому имени – номер 0, второму – 1 и т.д. Например:

```
enum colors {red, orange, yellow, green, azure, blue, violet};
```

или

```
enum {red, orange, yellow, green, cyan, blue, violet};
```

Программист может сопоставить каждому имени перечисления целое значение, отличное от его порядкового номера: **`enum {red, orange, yellow, green, cyan = 75, blue, violet}`**; В этом случае порядковые номера значений **`blue`** и **`violet`** будут, соответственно 76 и 77. Отметим, что можно написать и так:

`enum {red, orange = 23, yellow = 23, green, cyan = 75, blue = 75, violet}`; (если программист не видит разницы между оранжевым и желтым, а также между голубым и синим).

Еще примеры:

```
enum months {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

```
enum Boolean {no, yes};
```

(если вдруг возникло желание вместо **`false`** и **`true`** писать **`no`** и **`yes`**)

Если, например, умножить **`no`** на **`yes`**, получится **`0`**, а если их сложить – **`1`**. Имя в типе **`enum`** – это сокращение, употребляемое вместо **`{…}`**.

Перечислимые типы задают наборы целочисленных констант (тип этих констант **`int`**).

9.4. ***Средство*** **`typedef`**.