Функции

Артамонов Ю.Н.

Международный университет
природы, общества и человека "Дубна"
филиал Котельники

14 декабря 2017 г.

Содержание

Понятие программного модуля

Большинство реальных компьютерных программ достаточно велики по размерам. Часто объединить все действия в одну программу сложно и неудобно. Как показывает практика, наилучшим способом разработки и поддержки больших программ является их конструирование из небольших, относительно независимых частей - **модулей**. Данный принцип построения компьютерных программ известен как «разделяй и влавствуй».

Модули в языке С называются **функциями**.Программы обычно пишутся путем соединения новых функций, созданных программистом, с функциями, которые поставляются в составе *стандартной библиотеки С.*

Хотя функции стандартной библиотеки технически не являются частью языка С, они неизменно поставляются с системами ANSI С. Функции **printf, scanf, pow** являются функциями стандартной библиотеки. Программист может написать функции для решения своих задач, такие функции называются **функциями, определяемыми пользователем**.

Понятие программного модуля (продолжение)

Обращение к функции осуществляется посредством **вызова функции**. При этом одна функция будет вызывать другую. Например, при использовании функции **printf** в своей программе на самом деле функция **main** осуществляет вызов функции **printf**. Часто, но не всегда, вызов функций выстраивается в иерархическую структуру. Обычно к функции обращаются, записывая имя этой функции с последующей левой круглой скобкой, *аргументом функции* (или списком аргументов, отделяемых друг от друга запятыми) и правой круглой скобкоцй. Функции можно вкладывать одну в другую, например:

```
printf("%.2f ", sqrt(9.0)); //Вычисляет квадратный корень и выводит
  значение с точностью двух знаков
```

Функции стандартной математической библиотеки

Функции математической библиотеки позволяют программисту выполнять некоторые общие математические вычисления. При использовании функций математической библиотеки необходимо включить в программу соответствующий заголовочный файл с помощью директивы препроцессора:

```
#include <math.h>
```

(кроме этого, в Linux требуется компилировать программу с ключом -lm).
Важно! Все функции математической библиотеки возвращают результат **double** (это float двойной точности).
Сведем наиболее часто используемые функции в таблицу.

Функции стандартной математической библиотеки (продолжение)

№	Обозначение	Назначение
1	sqrt(x)	квадратый корень из x
2	exp(x)	экспоненциальная функция e^x
3	log(x)	натуральный логарифм x (основание e)
4	log10(x)	десятичный логарифм x
5	fabs(x)	абсолютное значение x
6	ceil(x)	округление до ближайшего большего целого x
7	floor(x)	округление до ближайшего меньшего целого x
8	pow(x,y)	x возводится в степень y
9	fmod(x,y)	остаток от деления (fmod(6.5,3)=0.5)
10	sin(x)	тригонометричекий синус от x в радианах
11	cos(x)	тригонометричекий косинус от x в радианах
12	tan(x)	тригонометричекий тангенс от x в радианах

Функции, определяемые пользователем

Функции позволяют программисту разбить программу на модули. Все переменные, объявленные в определениях функций, являются **локальными переменными** (они доступны только внутри функции, в которой определены).

Большинство функций имеют список **параметров**. Параметры позволяют функциям обмениваться информацией. Параметры функции - это также локальные переменные.

Раньше все наши программы состояли из одной функции main, которая вызывала другие стандартные функции. Определим теперь свою функцию - вычисления площади прямоугольника.

Пример функции, определяемой пользователем

```
#include <stdio.h>
int area(int, int); //Это прототип функции

main()
{
  int a,b;
  printf("Длина прямоугольника равна a=");
  scanf("%d", &a);
  printf("\n Ширина прямоугольника равна b=");
  scanf("%d", &b);
  printf("\n Площадь прямоугольнака равна S=%d\n", area(a,b)); //
  Вызов функции
  return 0;
}
int area(int x, int y) //Определение функции
{
  return x*y;
}
```

Пояснения к программному коду функции, определяемой пользователем

Функция *area* вызывается внутри оператора:

```
printf("\n Площадь прямоугольнака равна S=%d\n", area(a,b));
```

с фактически введенными параметрами a, b.
Строка:

```
int area(int, int);
```

является **прототипом** функции. Слово int слева от имени функции информирует компилятор о том, что area возвращает результат целого типа. Два ключевых слова int в скобках указывают, что функция area имеет два аргумента, каждый из них имеет тип int.
Компилятор использует прототип функции для проверки того, что вызов area имеет коректный тип возвращаемых данных, корректное число аргументов, корректный тип аргументов и что аргументы следуют в заданном порядке.

Пояснения к программному коду функции, определяемой пользователем (продолжение)

В качестве имени функции может использоваться любой идентификатор (в нашем случае это area). Тип результата, возвращаемого функцией, указывается перед именем функции (в нашем случае int).

Если функция ничего не возвращает, то перед ее именем нужно писать слово **void**. Если тип возвращаемого результата не указан, то компилятор по умолчанию будет считать, что это типом int. Тип результата в прототипе должен совпадать с типом результата при определении функции, иначе будет выдано сообщение об ошибке.

Если функция должна что-то возвращать, а в функции return отсутствует, то это может привести к непредвиденным ошибкам!

Хотя пропущенный тип по умолчанию расценивается как int, следует всегда задавать его явно. Однако для функции main возвращаемый тип обычно опускается.

Пояснения к программному коду функции, определяемой пользователем (продолжение)

Список параметров - это список объявлений параметров, отделенных запятыми, получаемых функцией при ее вызове. Если функция не получает значения, список параметров обозначается ключевым словом void. Тип каждого параметра должен быть указан явно за исключением типа int. Если тип параметра не указан, то по умолчанию принимается тип int. Правила хорошего тона - включать тип каждого параметра в список параметров, даже если он относится к типу int, принятому по умолчанию. Повторное использование параметра функции как локальной переменной внутри функции приведет к синтаксической ошибке. Точка с запятой после правой круглой скобки, закрывающей список параметров в определении функции, является синтаксической ошибкой. При перечислении типов переменных в списке параметров, необходимо указывать тип каждой переменной отдельно. Например, нельзя указать float x,y. Это будет интерпретироваться, что переменная x имеет тип float, а переменная y имеет тип по умолчанию int.

Пояснения к программному коду функции, определяемой пользователем (продолжение)

Правильно писать float x, float y. *Объявления и операторы* внутри фигурных скобок образуют *тело функции*. Тело функции также называют **блоком**. Блок - это просто составной оператор, который включает в себя объявления. Переменные могут быть объявлены в любом блоке, а блоки вложены.

Однако, **важно помнить! В любом случае нельзя объявлять функцию внутри другой функции**.

Программы должны состоять в виде совокупности небольших функций. Это упрощает создание программ, их отладку, поддержку и модификацию. Желательно, чтобы длина функции не превышала одной страницы текста.

Функция, требующая большого количества параметров, вероятно пытается выполнить много задач. Ее целесообразно разбить на несколько функций.

Прототип функции, заголовок функции и вызов функции должны иметь взаимное соответствие по количеству, типу и порядку следования переменных.

Пояснения к программному коду функции, определяемой пользователем (продолжение)

Существуют три способа возвращения управления в ту точку программы, из которой была вызвана функция. Если функция не возвращает результат, управление просто передается при достижении правой фигурной скобки, завершающей функцию, или при исполнении оператора:

return;

Если функция возвращает результат, оператор:

return выражение;

возвращает вызывающей функции значение выражения.

Рассмотрим еще пример на использование функций - разработать функцию, находящую максимум из трех действительных чисел.

Другой пример функции, определяемой пользователем

```
#include <stdio.h>
double maximum(double, double, double);
main()
{
   float a, b, c, max;
   printf("Введите три числа:\n");
   scanf("%f%f%f", &a, &b, &c);
   max = maximum(a,b,c);
   printf("Максимальное число: %f\n", max);
   return 0;
}
double maximum(double x, double y, double z)
{
   double max;
   max = x;
   if (max<y) max = y;
   if (max<z) max = z;
   return max;
}
```

Прототипы функций, заголовочные файлы

Одной из наиболее важных особенностей ANSI C являются прототипы функций, которые позаимствованы комитетом ANSI C у разработчиков C++. Прототип функции сообщает компилятору тип данных, возвращаемых функцией, число параметров, получаемых функцией, тип и порядок следования параметров. Компилятор использует прототипы функций для проверки корректности обращений к функции. Предыдущие версии C не выполняли этот вид проверок. Поэтому существовала возможность неправильного вызова функции, при котором компилятор не регистрировал ошибки.

Использование прототипа не является обязательным, но его всегда целесообразно включать.

Прототипы функций, заголовочные файлы (продолжение)

Кроме проверки ошибок, использование прототипа позволяет осуществлять *автоматическое приведение аргументов*. Например, в нашей задаче нахождения максимума из трех чисел, программа будет корректно работать и с целыми числами, хотя аргументы функции maximum имеют тип double. Это происходит из за того, что функция maximum согласно своему прототипу принудительно приводит свои аргуметны к типу double.

Приведение аргументов осуществляется в соответствии с *правилами возведения типов C*. Правила возведения определяют, каким образом однин типы могут быть преобразованы в другие типы без потери данных. Однако такие преобразования возможны не всегда. Например, преобразование типа double в тип int отбрасывает дробную часть значения double.

Прототипы функций, заголовочные файлы (продолжение)

Правила возведения автоматически применяются к выражениям, содержащим значения двух и более типов данных (такие выражения называются *выражениями смешанного типа*).
Каждое значение в выражении смешанного типа автоматически возводится к наивысшему типу выражения. Однако нужно иметь в виду, что процедура возведения типов применяется последовательно к каждой паре операндов текущей операции по ходу оценки выражения. Например, вычисление 3/2*3.14 даст в результате 3.14, поскольку 3 и 2 целые числа и будет выполнено целочисленное деление. В таблице ниже приведены типы данных в порядке от наивысшего приоритета к низшему, также в таблице указаны соответствующие спецификации для printf, scanf.

Прототипы функций, заголовочные файлы (продолжение)

Тип данных	**Спецификация для printf**	**Спецификация для scanf**
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c

Прототипы функций, заголовочные файлы (продолжение)

Преобразование значений к более низким типам обычно приводит потере данных. Поэтому такое приведение должно быть указано явно.
Если прототип для данной функции не был включен в программу, компилятор формирует собственный прототип функции, используя ее первое вхождение в программу: или определение, или вызов функции. По умолчанию компилятор предполагает, что функция возвращает тип int, ничего не предполагая относительно аргументов функции.

Прототипы функций, заголовочные файлы (продолжение)

Кроме этого, прототипы можно выносить в отдельный файл, подключая его директивой #include по аналогии с подключением функций из стандартных библиотек. Для этого используется понятие **заголовочного файла**.
Каждая стандартная библиотека имеет свой заголовочный файл, содержащий прототипы для всех функций данной библиотеки, а также опредеения различных типов данных и констант, еобходимых этим функциям. Перечислим имена некоторых заголовочных файлом стандартных библиотек.

- <math.h> содержит прототипы функций математической библиотеки;
- <stdio.h> содержит прототипы функций для ввода/вывода и информацию, используемую ими;
- <stdlib.h> содержит прототипы функций преобразования чсел в текст и обратно, прототипы функций генерации случайных чисел и др.;
- <string.h> содержит прототипы для функций обработки строк.

Прототипы функций, заголовочные файлы (продолжение)

При создании специализированного заголовочного файла программист должен сохранить его с расширением .h там же, где находится сам файл программы. Данный заголовочный файл следует включать директивой:
#include "name.h”

Задание 5.1

Реализовать функцию подсчета количества цифр заданного числа.

Задание 5.2

Реализовать функцию, определяющую, является ли заданное число квадратом некоторого числа.

Задание 5.3

Реализовать функцию, вычисляющую двойной факториал:

$$N!! = 1 \cdot 3 \cdot 5 \ldots N, N \equiv 1 \mod 2$$
$$N!! = 2 \cdot 4 \cdot 6 \ldots N, N \equiv 0 \mod 2$$

Задание 5.4

Реализовать функцию, вычисляющую числа Фибоначчи:

$$F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2}$$

Задание 5.5

Реализовать функцию, вычисляющую $\sin(x)$ с заданной точностью e:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Атрибуты идентификаторов

Любой *идентификатор* в программе (переменные, функции) имеет следующие атрибуты:

- имя;
- тип;
- значение;
- класс памяти;
- период хранения;
- область действия;
- тип компоновки.

С первыми тремя мы знакомы. Рассмотрим некоторые остальные.

Классы памяти

Язык C поддерживает четыре *класса памяти*: **auto**, **register**, **extern**, **static**. Класс памяти идентификатора позволяет определить период его хранения, область действия и тип компановки.

Период хранения идентификатора - это время, в течение которого данный идентификатор существует в памяти. Некоторые идентификаторы существуют короткое время, некоторые неоднократно создаются и разрушаются, другие существуют в течение всего времени выполнения программы.

Область действия идентификатора характеризует возможность обращения к нему из различных частей программы. Некоторые идентификаторы доступны во всей программе, другие - только в отдельных ее частях.

Классы памяти (продолжение)

Тип компоновки определяется для программ, состоящих из нескольких исходных файлов, объединенных на этапе компоновки. Этот атрибут показывает, известен ли идентификатор только в текущем файле или в любом файле с соответствующими объявлениями (подробно этот атрибут мы будем рассматривать в дальнейшем).

Четыре спецификатора класса памяти могут быть разбиты на два типа по периоду хранения: *автоматический период хранения* и *статический период хранения*.

Для объявления переменных с автоматическим периодом хранения служат ключевые слова **auto**, **register**. Переменные с автоматическим хранением создаются, когда управление получает программный блок, в котором они объявлены. Переменные этого типа существуют, пока блок активен, и уничтожаются, когда происходит выход из блока.

Классы памяти (продолжение)

Автоматический период хранения могут иметь только переменные. Локальные переменные функций (объявленные в списке параметров или теле функции) обычно имеют автоматический период хранения. Ключевое слово **auto** объявляет переменные с автоматическим хранением явным образом. Например, объявление

```
auto float x,y;
```

указывает, что переменные x, y существуют только в теле функции, в которой находится данное объявление.

Локальные переменные имеют автоматический период хрaрнения по умолчанию, поэтому ключевое слово **auto** в явном виде используется редко. Автоматическое хранение способствует экономии памяти, поскольку такие переменные существуют только тогда, когда они необходимы. Они создаются при запуске функции, в которой объявлены, и уничтожаются, когда происходит выход из функции.

Классы памяти (продолжение)

Данные исполняемой программы могут храниться на винчестере, в оперативной памяти, в регистрах процессора. Наиболее быстрый доступ к данным возможен при их хранении в регистрах процессора. Если перед объявлением переменной используется ключевое слово **register**, это как раз рекомендация компилятору разместить эти данные в регистрах процессора. Рекомендацию разместить переменную в регистрах следует использовать, если к такой переменной осуществляется интенсивное обращение внутри программы. Например, переменные счетчики, или аккумулирующие переменные следует размещать в регистрах процессора:

```
register int counter = 0;
```

Ключевое слово **register** можно использовать только с переменными, имеющими автоматический период хранения. При этом компилятор может такую рекомендацию проигнорировать, если на текущий момент все регистры заняты. Современные компиляторы сами способны распознавать, где использовать **register**, не требуя от программиста объявления **register**.

Классы памяти (продолжение)

Ключевые слова **extern**, **static** используются для объявления идентификаторов переменных и функций со статическим периодом хранения. Идентификаторы статического периода хранения существуют с того момента, как программа начинает выполняться. Однако область видимости этих идентификаторов не совпадает с периодом их хранения. То есть переменная может уже существовать, но будет не видна в программе. Существуют два типа идентификаторов со статическим периодом хранения: внешние идентификаторы (глобальные переменные, имена функций) и локальные переменные, объявленные со спецификатором класса памяти **static**.

Глобальные переменные и имена функций имеют по умолчанию класс памяти **extern**. Глобальные переменные создаются при помещении их объявлений вне любого определения функции, они сохраняют свои значения в течение всего времени выполнения программы. Обращение к глобальной переменной возможно из любой функции, которая следует после объявления глобальной переменной.

Классы памяти (продолжение)

Локальные переменные, объявленные с ключевым словом **static**, остаются известными только из той функции, в которой они определены, но в отличие от автоматических статические локальные переменные сохраняют свое значение и после выхода из функции. При следующем вызове статическая переменная будет содержать то значение, которое она имела при последнем выходе из функции.

```
static int counter = 1;
```

Если статическая переменная не инициализирована программистом, она по умолчанию инициализируется нулем.

Правила области видимости

Область видимости идентификатора - это та часть программы, в которой возможно обращение к этому идентификатору. Область видимости идентификатора делится на четыре вида:

- область видимости функции;
- область видимости файла;
- область видимости блока;
- область видимости прототипа функции.

Область видимости функции имеют метки case в операторе switch, на них нельзя сослаться вне тела функции.

Идентификатор, объявленный вне любой функции, имеет область видимости файла. Такой идентификатор известен всем функциям, начиная с того места, где он объявлен, и до конца файла. Глобальные переменные, определения функций, прототипы функций, помещенные вне функций, - все они имеют область видимости файла.

Правила области видимости (продолжение)

Идентификаторы, объявленные внутри блока, имеют область видимости блока. Такая область видимости заканчивается завершающей правой фигурной скобкой блока. Локальные переменные, объявленные в начале функции имеют область видимости блока. Любой блок может содержать свои объявления переменных.

Если блоки вложены, а идентификатор во внешнем блоке имеет такое же имя, как во внутреннем блоке, то виден только идентификатор внутреннего блока. Локальные переменные, объявленные как static внутри блока, существуют с момента выполнения программы, но видны только внутри блока. Таким образом, область видимости и период хранения не совпадают.

Единственным идентификатором с областью видимости прототипа функции являются идентификаторы, которые используются в списке праметров прототипа функции.

Пример на область видимости

```
#include <stdio.h>
void a(void); //Прототипы функций
void b(void);
void c(void);
int x = 1; //Глобальная переменная
main()
{
  int x = 2; //Локальная переменная в main
  printf("%d\n",x);
  //Начало области видимости блока
  {
    int x = 3;
    printf("%d\n",x); //Печатает 3
  }
  printf("%d\n",x); //Печатает 2
  a(); b(); c();
  a(); b(); c();
  return 0;
}
```

Пример на область видимости

```
void a(void)
{int x=4;
   printf("%d\n",x);
   x++;
   printf("%d\n",x);
}
void b(void)
{static int x=5;
   printf("%d\n",x);
   x++;
   printf("%d\n",x);
}
void c(void)
{
   printf("%d\n",x);
   x++;
   printf("%d\n",x);
}
```

Пример на область видимости

```
2
3
2
4
5
5
6
1
2
4
5
6
7
2
3
```

Рекурсивные функции

До сих пор мы рассматривали функции, которые вызывали другие функции, подчиняясь строгой иерархии вызовов. Однако в некоторых задачах более естественным определением функции может быть вызов этой функцией самой себя.

Рекурсивная функция - это функция, которая вызывает саму себя или непосредственно или косвенно через другие функции.

В качестве примера, когда с помощью рекурсии действительно проще определить функции, рассмотрим задачу вычисления факториала:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

$$n! = (n-1)! \cdot n$$

Если определить $F(n) = n!$, то получаем:

$$F(n) = F(n-1) \cdot n$$

Рекурсивные функции (продолжение)

Однако этого недостаточно, чтобы посчитать значение функции для какого-то конкретного числа. Например, пусть требуется вычислить $F(4)$. Пользуясь определением функции много раз получаем:

$$F(4) = F(3) \cdot 4 = F(2) \cdot 3 \cdot 4 = \ldots = F(-1) \cdot 0 \cdot 1 \cdot 2 \cdot 3 \cdot 4 = \ldots$$

Видно, что этот процесс вычислений не получается закончить. Нужна точка останова - когда значение функции не требуется вычислять, оно уже известно. Например, можно определить, что $F(0) = 0! = 1$. Такая точка останова, которая не требует рекурсивного вызова функции называется **базой рекурсии**. Тогда получаем следующий вычислительный процесс:

$$F(4) = F(3) \cdot 4 = F(2) \cdot 3 \cdot 4 = F(1) \cdot 2 \cdot 3 \cdot 4 = F(0) \cdot 1 \cdot 2 \cdot 3 \cdot 4 = 1 \cdot 1 \cdot 2 \cdot 3 \cdot 4$$

$$F(4) = 24$$

Пример рекурсии

Рассмотрим, как такие вычисления организовать в C.

```
#include<stdio.h>
long int Fact(long int);
long int Fact(long int n)
{
  if (n == 0)
    return 1;
  else
    return n*Fact(n-1);
}
main()
{
  long int a;
  printf("n = ");
  scanf("%ld",&a);
  printf("n! =%ld\n", Fact(a));
  return 0;
}
```

Еще пример рекурсии

Рассмотрим опять последовательность чисел Фибоначчи: $F(0) = 0, F(1) = 1; F(n) = F(n-1) + F(n-2)$

```
#include<stdio.h>
long int Fib(long int);
long int Fib(long int n)
{
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return Fib(n-1)+Fib(n-2);
}
```

Еще пример рекурсии (продолжение)

```
main()
{
   long int a;
   printf("n = ");
   scanf("%ld",&a);
   printf("Fib(%ld) =%d\n", a,Fib(a));
   return 0;
}
```

Определение функций с помощью рекурсии часто более наглядно, однако рекурсия, как правило, приводит к существенному росту времени работы алгоритма (попробуйте вычислить Fib(100) с помощью нашей программы).

Задание 5.6

Целое число называется *простым*, если оно делится только на 1 и на себя. Например, 2, 3, 5, 7 - простые числа, 4, 6, 8, 9 - составные числа.
Написать функцию, которая определяет, является ли число простым. С помощью этой функции вывести на экран таблицу всех простых чисел от 1 до 1000.

Задание 5.7

Написать функцию invers, которая для заданного числа возвращает число с обратным порядком цифр. Например, $invers(1234)$ функция должна вернуть 4321. Использовать эту функцию для определения функции sub:

$$sub(n) = |n - invers(n)|$$

Для заданного числа n организовать вычисление $sub(sub(sub(sub(\ldots n)))\ldots)$ до тех пор, пока не получится нуль. Определеть для заданного числа n, сколько потребуется вызовов функции sub.

Например, |321-123|=198, |891-198|=693, |693-396|=297, |797-297|=495, |594-495|=99, |99-99|=0. Т.е. потребуется 6 вызовов. Найдите числа, для которых количество вызовов бесконечно.

Задание 5.8

Реализовать функцию, которая моделирует подбрасывание шестигранной кости (каждая грань может выпадать с равной вероятностью).
Построить игровую программу:

1. Игрок делает ставку (не более суммы в банке).
2. Программа подбрасывает две кости и суммирует количество набранных очков игрока (с учетом очков, набранных ранее).
3. Если игрок набрал 20 очков, он получает свою удвоенную ставку.
4. Если игрок набирает более 20 очков, он проигрывает, его деньги остаются в банке.
5. В противном случае игрок может закончить играть, тогда ему возвращается его ставка, или продолжить играть.