**Лабораторная работа 7**

# Функции. Рекурсия.

Цель этой лабораторной работы — изучить понятие функции в языках Си и Си++ и научиться объявлять, определять и вы,зывать функции.

Функции — инструмент для управления сложностью. С помощь функций можно разделять программу на более удобные осмысленные части, избегать повторения кода, скрывать подробности низкого уровня. Функцию можно сравнить с мини-программой, которая имеет имя и решает определенную задачу. Чтобы ею пользоваться, вам не надо знать, как она устроена, достаточно знать, как она называется, как ей передать данные и как получить результат.

Например, если вы пишете программу для рисования графика температуры за год, можно написать функцию `getTemp(date)`, которая возвращает в виде числа температуру за указанную дату с одного из публичных серверов исторических погодных данных. После этого вы сможете вызывать ее для каждого из дней года и заняться вопросами красивого отображения графика, не волнуясь об адресе сервера, формате запроса, преобразовании ответа в число и тому подобных деталях.

Другой пример. Вот как могла бы выглядеть функция, которая находит длину строки:

```
int strlen(char* s) {
    int i = 0;
    while (s[i]!='\0')
        i++;
    return i;
}
```

Это *определение* функции. В общем случае определение выглядит так:

```
<тип> <имя функции>(<имена и типы параметров>) {
    <тело функции>
}
```

Тело функции должно содержать оператор `return`, который завершает ее выполнение, *возвращая* в точку вызова некоторое значение указанного типа. Единственное исключение — функции, которые ничего не возвращают, а просто выполняют какие-то действия. Такие функции часто называют *процедурами*, и для их объявления используется тип `void`.

*Параметры* можно воспринимать как локальные переменные, которые при вызове функции автоматически получают значения переданных в нее аргументов.

Рассмотрим пример:

```
// main.cpp: один из корней квадратного уравнения 2x^2+3x-9=0
#include <iostream>
#include <cmath>
double x1(double a, double b, double c) {
    double D = b*b - 4*a*c;
    return (-b - sqrt(D))/(2*a);
}
int main(){
    std::cout << x1(2, 3, -9) + 2;
}
```

Поскольку оператор `+` имеет более высокий приоритет, чем `<<`, сперва вычисляется значение суммы, а для этого должно быть сначала вычислено значение ее первого слагаемого: функция `main` приостанавливается и вызывает функцию `x1`. Это мини-программа, в ней есть собственная переменная `D`, а также переменные-параметры `a`, `b`, `c`, причем выполнение функции невидимо начинается с присваиваний `a=2; b=3; c=-9;` для этих «как бы» переменных. Вычисленное значение корня уравнения, `-3`, возвращается в `main`, которая продолжает свое исполнение, заменяя `x1(2, 3, -9)` полученным от нее значением `-3`. Итак, в выходной поток выводится вычисленное значение суммы $-3+2=-1$.

Для локальных переменных, то есть переменных, определенных внутри функции, применяется обычное правило: переменная видна только в ближайшем окружающем блоке, или фигурных скобках. В примере выше переменная `D` видна только внутри функции `x1` и недоступна из `main`. В разных функциях могут быть переменные с одинаковыми именами, но внутри каждой из функций это имя будет означать свою переменную.

Функции `main`, для того чтобы вызвать `x1`, ничего не надо знать о том, как она устроена. Надо лишь остановиться в месте вызова, запустить функцию `x1`, передать ей три `double`-числа и получить от нее одно число в ответ. Поэтому для компиляции файла `main.cpp` достаточно на самом деле лишь *определения*:

```
double x1(double a, double b, double c);
```

Оно сообщает всю необходимую информацию для вызова и возврата значения, но не содержит тела функции. Другое отличие объявления от определе-

ния в том, что одинаковых объявлений может быть много, а определение — только одно.

Разберем подробнее процесс построения исходного файла. Именно для организации этого процесса нужны проекты и решения в Visual Studio.

Хотя сборка в современной среде разработки обычно запускается одной кнопкой и может выглядеть как одно действие, в действительности процесс сборки состоит из нескольких этапов:

– препроцессинга,
– компиляции,
– компоновки.

В ходе *компиляции* файлы исходного кода на языке программирования преобразуются в объектные модули, состоящие из машинного кода. Для компиляции одного файла все используемые в нем функции должны быть объявлены, но не обязательно определены. В места вызова еще не определенных функций в объектные модули вставляются заглушки.

Вы могли заметить, что мы не копировали объявления функций `scanf` / `printf` в код наших программ. Часто функции собираются *в библиотеку*, которая охватывает определенную область, например, как раз стандартный ввод-вывод. Было бы неудобно каждый раз выбирать и вставлять в каждый файл объявления используемых функций. Поэтому объявления всех функций, реализуемых библиотекой, собираются ее разработчиками в один общий заголовочный файл. И `stdio.h` представляет собой именно заголовочный файл. Чтобы не копипастить объявления, придумали `#include`. Все директивы, начинающиеся с решетки, обрабатываются на этапе *препроцессинга* (первый пункт в списке выше). Препроцессор просто заменяет `#include <stdio.h>` на содержимое файла `stdio.h`, то есть в вашу программу вставляются все объявления еще до того, как она дойдет до этапа компиляции.

Наконец, в ходе *компоновки* код нескольких объектных модулей (и библиотек) собирается в один исполняемый файл, а все заглушки заменяются на реальные вызовы соответствующих функций. чтобы отслеживать все зависимости и собирать настройки компиляции в одном удобном месте, и придумали системы сборки, одной из которых является MSBuild, графический интерфейс к которому и есть система проектов и решений Visual Studio.

На рисунке ниже проиллюстрирован процесс сборки гипотетической сетевой игры, в которой по отдельным файлам (светло-зеленые) разделены основной геймплей (my.cpp), логика противников (enemy.cpp) и сетевой функциионал (net.cpp). Используемые заголовочные файлы изображены темно-зеленым цветом, голубым выделены объектные модули, синим — библиотечные модули, оранжевым — итоговый исполняемый файл приложения.

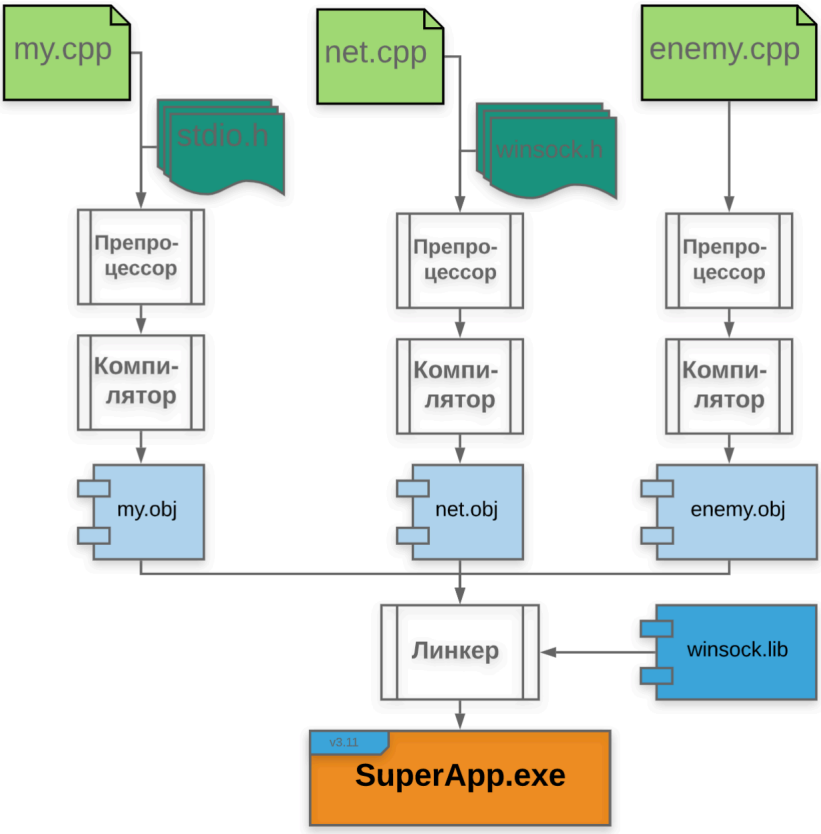


Рис. 1: Процесс сборки программы

*У попа была собака,*
*Он ее любил.*
*Она съела кусок мяса,*
*Он ее убил.*
*В землю закопал,*
*Надпись написал, что:*
*У попа была собака…*

Наверное вы догадались, к чему здесь этот жестокий стишок. Вы уже сталкивались с рекурсией при рассмотрении быстрой сортировки. В отличие от повтора или выбора действия, рекурсия редко встречается в повседневной жизни. Поэтому ею не так легко овладеть, однако такой способ мышления позволяет решать многие задачи, которые трудно с разбегу решить при помощи итерации. Поэтому навыки решения задач при помощи рекурсии — стоящее дополнение к вашему набору умений.

Ключевой момент здесь — умение выделить в задаче меньшую задачу такого же типа. Иногда упоминают метафору «сделать наименьшее количество работы, чтобы тебя нельзя было назвать бездельником, и спустить остальное подчиненным». Иногда наоборот, удобнее смотреть на задачу, предполагая, что любую меньшую задачу мы уже умеем решать. В любом случае в тексте программы рекурсия выглядит как вызов изнутри функции самой этой функции, но с другими, обычно меньшими, параметрами. Поскольку такая цепочка вызовов продолжаться бесконечно не может (даже у самого большого начальника нет бесконечного числа подчиненных, поп писать устанет), какой-то самый простой случай задачи должен решаться без рекурсивного вызова.

Рассмотрим пример. Нужно найти сумму целочисленного массива размера $N$, написав функцию `int sum(int* A, unsigned N)`. Для интереса попробуем придумать рекурсивное решение. Представим, что нам лень выполнять сложение больше одного раза. Сумму какого массива мы можем найти вообще без сложения? Массива размера 1. Теперь предположим, что мы уже умеем решать все меньшие задачи, то есть у нас есть помощник `sum(...)`. Заставим его просуммировать все $N-1$ элементов массива, оставив себе лишь последний. Когда он вычислит почти всю сумму, прибавим к ней последний элемент и присвоим все лавры себе. Получилось рекурсивное решение:

```
int sum(int* A, unsigned N){
    if (N==1)
        return A[0];
    else
        return sum(A, N-1) + A[N-1];
}
```

## ОБЩИЕ ЗАДАНИЯ

1. Напишите функцию возведения вещественного числа в целую неотрицательную степень `double power(double x, unsigned k)`, которая вычисляет $x^k$. Напишите с ее помощью программу, которая вводит целое значение показателя степени $m$, и выводит через пробел $m$-ые степени чисел 1, 2, …, 10. Заголовочные файлы `math.h/cmath` не использовать.

   Пример ввода: 2, правильный вывод `1 4 9 16 25 36 49 64 81 100`.

2. Напишите функцию `double dist(x1, y1, x2, y2)`, которая принимает вещественные декартовы координаты двух точек на плоскости и возвращает расстояние между ними. Напишите с ее помощью программу, которая вводит вещественные координаты $x_1$, $y_1$, $x_2$, $y_2$, $x_3$, $y_3$ трех точек на плоскости и выводит длину самой длинной стороны треугольника с вершинами в этих точках, или –1, если такого треугольника не существует.

   Пример ввода: `0 0 6 0 3 2`, правильный вывод `6`.

3. Напишите функцию `unsigned word_end(char* s, unsigned i)`, которая принимает строку $s$, состоящую из одних латинских букв и пробелов, и индекс $i$ в ней $\left(0 \leqslant i < \texttt{strlen}(s)\right)$. Функция должна возвращать индекс символа, следующего за последним символом слова, которому принадлежит символ $s[i]$. Напишите с помощью этой функции программу, которая вводит число $k$, затем строку, состоящую из одних латинских букв и пробелов, и выводит $k$-е слово строки (считая с 1).

   Примеры. Для строки $s =$ `"This is fine"` значение `word_end(s, 0)` равно 4, значение `word_end(s, 9)` равно 12. Для $k = 3$ с таким $s$ программа должна ответить `fine`.

4. Напишите рекурсивный вариант вычисления минимума целочисленного массива в виде функции `int min(int* A, unsigned n)`, находящей минимум среди первых $n$ элементов массива $A$. Напишите с помощью этой функции программу, которая вводит число $N$, динамически создает массив размера $N$, заполняет его с клавиатуры и выводит минимум массива.

5. Напишите рекурсивный вариант функции, определеющей, является ли подстрока строки $s$, начиающаяся в $i$-м и завершающаяся перед $j$-м символом, палиндромом. Функция должна быть реализована как `bool`

`isPalindrome(char* s, unsigned i, unsigned j)`. Напишите с помощью этой функции программу, которая вводит строку *s*, длины не превышающей 100, и отвечает на вопрос, является ли она палиндромом.