Основы алгоритмизации и программирования

Лабораторная работа №4.

Цель работы - изучить функции (процедуры) в языке программирования Си.
Содержание отчёта:

1. Титульный лист
2. Формулировка цели и задач работы
3. Описание результатов выполнения
4. Выводы, согласованные с целью

Функции (процедуры)

Функции разбивают большие вычислительные задачи на более мелкие и позволяют воспользоваться тем, что уже сделано другими разработчиками, а не начинать создание программы каждый раз "с нуля". В выбранных должным образом функциях "упрятаны" несущественные для других частей программы детали их функционирования, что делает программу в целом более ясной и облегчает внесение в нее изменений.

Язык проектировался так, чтобы функции были эффективными и простыми в использовании. Обычно программы на Си состоят из большого числа небольших функций, а не из немногих больших. Программу можно располагать в одном или нескольких исходных файлах. Эти файлы можно компилировать отдельно, а загружать вместе, в том числе и с ранее откомпилированными библиотечными функциями. Процесс загрузки здесь не рассматривается, поскольку он различен в разных системах.

Объявление и определение функции —это та область, где стандартом ANSI в язык внесены самые существенные изменения. Синтаксис определения функции также изменен, так что теперь объявления и определения функций соответствуют друг другу. Это позволяет компилятору обнаруживать намного больше ошибок, чем раньше. Кроме того, если типы аргументов соответствующим образом объявлены, то необходимые преобразования аргументов выполняются автоматически.

Стандарт вносит ясность в правила, определяющие области видимости имен; в частности, он требует, чтобы для каждого внешнего объекта было только одно определение. В нем обобщены средства инициализации: теперь можно инициализировать автоматические массивы и структуры.

Улучшен также препроцессор Си. Он включает более широкий набор директив условной компиляции, предоставляет возможность из макроаргументов генерировать строки в кавычках, а кроме того, содержит более совершенный механизм управления процессом макрорасширения.

Определение любой функции имеет следующий вид:

```
тип-результата имя-функции(объявления аргументов){
        объявления и инструкции
}
```

Любая программа —это просто совокупность определений переменных и функций. Связи между функциями осуществляются через аргументы,возвращаемые значения и внешние переменные. В исходном файле функции могут располагаться в любом порядке;

исходную программу можно разбивать на любое число файлов, но так, чтобы ни одна из функций не оказалась разрезанной.

Инструкция return реализует механизм возврата результата от вызываемой функции к вызывающей. За словом return может следовать любое выражение:

```
return выражение;
```

Если потребуется, выражение будет приведено к возвращаемому типу функции. Часто выражение заключают в скобки, но они не обязательны.

Вызывающая функция вправе проигнорировать возвращаемое значение. Более того, выражение в return может отсутствовать, и тогда вообще никакое значение не будет возвращено в вызывающую функцию. Управление возвращается в вызывающую функцию без результирующего значения также и в том случае, когда вычисления достигли "конца" (т. е. последней закрывающей фигурной скобки функции). Не запрещена (но должна вызывать настороженность) ситуация, когда в одной и той же функции одни return имеют при себе выражения, а другие —не имеют. Во всех случаях, когда функция "забыла" передать результат в return, она обязательно выдаст "мусор".

До сих пор мы пользовались готовыми функциями вроде main, printf, теперь настала пора нам самим написать несколько функций. В Си нет оператора возведения в степень вроде **в Фортране. Поэтому проиллюстрируем механизм определения функции на примере функции power(m, n), которая возводит целое m в целую положительную степень n.Так, power(2, 5)имеет значение 32. На самом деле для практического применения эта функция малопригодна, так как оперирует лишь малыми целыми степенями, однако она вполне может послужить иллюстрацией.(В стандартной библиотеке есть функция pow(x, y), вычисляющая x^y.)

Итак,мы имеем функцию power и главную функцию main, пользующуюся ее услугами, так что вся программа выглядит следующим образом:

```
#include <stdio.h>

int power(int m, int n);

/* тест функции power */
int main(){
        int i;
        for (i = 0; i < 10; ++i) {
                printf("%d %d %d\n", i, power(2,i), power(-3, i));
        }
        return 0;
}

/* возводит base в n-ю степень; n>= 0 */
int power(int base, int n){
        int i, p;
        P = 1;
        for (i = 1; i <= n; ++i) {
                p = p * base;
        }
        return p;
}
```

Определения функций могут располагаться в любом порядке в одном или в нескольких исходных файлах, но любая функция должна быть целиком расположена в каком-то одном. Если исходный текст программы распределен по нескольким файлам, то, чтобы ее скомпилировать и загрузить, вам придется сказать несколько больше, чем при использовании одного файла; но это уже относится к операционной системе, а не к языку. Пока мы предполагаем, что обе функции находятся в одном файле, так что будет достаточно тех знаний, которые вы уже получили относительно запуска программ на Си.

В следующей строке из функции main к power обращаются дважды.

```
printf("%d %d %d\n", i, power(2,i), power(-3,1));
```

При каждом вызове функции power передаются два аргумента, и каждый раз главная программа main в ответ получает целое число, которое затем приводится к должному формату и печатается. Внутри выражения power(2,1)представляет собой целое значение точно так же, как 2или i.

В первой строке определения power:

```
int power(int base, int n)
```

указываются типы параметров, имя функции и тип результата. Имена параметров локальны внутри power, это значит, что они скрыты для любой другой функции, так что остальные подпрограммы могут свободно пользоваться теми же именами для своих целей. Последнее утверждение справедливо также для переменных іи р: i в power и i в main не имеют между собой ничего общего.

Далее параметром мы будем называть переменную из списка параметров, заключенного в круглые скобки и заданного в определении функции, а аргументом— значение, используемое при обращении к функции.Иногда в том же смысле мы будем употреблять термины формальный аргумент и фактический аргумент.

Функция не обязательно возвращает какое-нибудь значение. Инструкция return без выражения только передает управление в ту программу,которая ее вызвала, не передавая ей никакого результирующего значения. То же самое происходит, если в процессе вычислений мы выходим на конец функции, обозначенный в тексте последней закрывающей фигурной скобкой. Возможна ситуация, когда вызывающая функция игнорирует возвращаемый ей результат.

Вы, вероятно, обратили внимание на инструкцию return в конце main. Поскольку main есть функция, как и любая другая,она может вернуть результирующее значение тому, кто ее вызвал, —фактически в ту среду, из которой была запущена программа. Обычно возвращается нулевое значение, что говорит о нормальном завершении выполнения. Ненулевое значение сигнализирует о необычном или ошибочном завершении. Программы должны сообщать о состоянии своего завершения в операционную систему.

Объявление

```
int power(int m, int n);
```

стоящее непосредственно перед main, сообщает, что функция power ожидает двух аргументов типа intи возвращает результат типа int. Это объявление, называемое прототипом функции, должно быть согласовано с определением и всеми вызовами power. Если определение функции или вызов не соответствует своему прототипу, это ошибка.

Имена параметров не требуют согласования. Фактически в прототипе они могут быть произвольными или вообще отсутствовать, т.е. прототип можно было бы записать и так:

```
int power(int, int);
```

Однако удачно подобранные имена поясняют программу, и мы будем часто этим пользоваться.

Одно свойство функций в Си, вероятно, будет в новинку для программистов, которые уже пользовались другими языками, в частности Фортраном. В Си все аргументы функции передаются "по значению". Это следует понимать так, что вызываемой функции посылаются значения ее аргументов во временных переменных, а не сами аргументы. Такой способ передачи аргументов несколько отличается от "вызова по ссылке" в Фортране и спецификации var при параметре в Паскале, которые позволяют подпрограмме иметь доступ к самим аргументам,а не к их локальным копиям.

Главное отличие заключается в том, что в Си вызываемая функция не может непосредственно изменить переменную вызывающей функции: она может изменить только ее частную, временную копию.

Однако вызов по значению следует отнести к достоинствам языка, а не к его недостаткам. Благодаря этому свойству обычно удается написать более компактную программу, содержащую меньшее число посторонних переменных,поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные вызванной подпрограммы. В качестве примера приведем еще одну версию функции power, в которой как раз использовано это свойство.

```
int power(int base, int n){
        int i, p;
        P = 1;
        for (i = 1; i <= n; ++i) {
                p = p * base;
        }
        return p;
}
```

Параметр n выступает здесь в роли временной переменной, в которой циклом for в убывающем порядке ведется счет числа шагов до тех пор, пока ее значение не станет нулем. При этом отпадает надобность в дополнительной переменной i для счетчика цикла. Что бы мы ни делали с n внутри power, это не окажет никакого влияния на сам аргумент,копия которого была передана функции power при ее вызове.

При желании можно сделать так, чтобы функция смогла изменить переменную в вызывающей программе. Для этого последняя должна передать адрес подлежащей изменению переменной (указательна переменную), а в вызываемой функции следует объявить соответствующий параметр как указатель и организовать через него косвенный доступ к этой переменной.

Механизм передачи массива в качестве аргумента несколько иной. Когда аргументом является имя массива, то функции передается значение, которое является адресом начала этого массива; никакие элементы массива не копируются. С помощью индексирования относительно полученного значения функция имеет доступ к любому элементу массива. Разговор об этом пойдет в следующем параграфе.

В предыдущих примерах функции либо вообще не возвращали результирующих значений (void), либо возвращали значения типа int. А как быть, когда результат функции должен иметь другой тип? Многие вычислительные функции, как, например, sqrt, sinи cos, возвращают значения типа double; другие специальные функции могут выдавать значения еще каких-то типов. Чтобы проиллюстрировать, каким образом функция может возвратить нецелое значение, напишем функцию atof(s), которая переводит строку s в соответствующее число с плавающей точкой двойной точности. Она имеет дело со знаком (которого может и не быть), с десятичной точкой, а также с целой и дробной частями, одна из которых может отсутствовать. Наша версия не является высококачественной программой преобразования вводимых чисел; такая программа потребовала бы заметно больше памяти.Функция atof входит в стандартную библиотеку программ; ее описание содержится в заголовочном файле <stdlib.h>.

Прежде всего отметим, что объявлять тип возвращаемого значения должна сама atof, так как этот тип не есть int. Указатель типа задается перед именем функции.

```
#include<ctype.h>

/* atof: преобразование строки sв double */
double atof (char s[]){
        double val, power;
        int i, sign;
        for (i = 0; isspace (s[i]); i++);
                /* игнорирование левых символов-разделителей */
                sign = (s[i] == '-') ? -1 : 1;
                if (s[i]== '+' || s[i] == '-') {
                        i++;
                }
                for (val = 0.0; isdigit(s[i]); i++) {
                        val = 10.0 * val + (s[i] -'0');
                }
                if (s[i] =='.') {
                        i++;
                }
                for (power = 1.0; isdigit(s[i]); i++) {
                        val = 10.0 * val + (s[i] -'0');
                        power *= 10.0;
                }
                return sign * val / power;
}
```

Объявление и определение функции atof должны соответствовать друг другу. Если в одном исходном файле сама функция atof и обращение к ней в main имеют разные типы, то это несоответствие будет зафиксировано компилятором как ошибка. Но если функция atof была скомпилирована отдельно (что более вероятно), то несоответствие типов не будет обнаружено, и atof возвратит значение типа double, которое функция main воспримет как int, что приведет к бессмысленному результату.

Располагая соответствующим образом описанной функцией atof, мы можем написать функцию atoi, преобразующую строку символов в целое значение, следующим образом:

```
/* atoi: преобразование строки s в int с помощью atof */
int atoi (char s[]){
        double atof (char s[]);
        return(int) atof(s);
}
```

Обратите внимание на вид объявления и инструкции return. Значение выражения в

```
return выражение;
```

перед тем, как оно будет возвращено в качестве результата, приводится к типу функции. Следовательно, поскольку функция atoi возвращает значение int, результат вычисления atof типа double в инструкции return автоматически преобразуется в тип int. При преобразовании возможна потеря информации, и некоторые компиляторы предупреждают об этом.Оператор приведения явно указывает на необходимость преобразования типа и подавляет любое предупреждающее сообщение.

В Си допускается рекурсивное обращение к функциям, т.е. функция может обращаться сама к себе, прямо или косвенно. Рассмотрим печать числа в виде строки символов. Как мы упоминали ранее, цифры генерируются в обратном порядке —младшие цифры получаются раньше старших, а печататься они должны в правильной последовательности.

Проблему можно решить двумя способами. Первый —запомнить цифры в некотором массиве в том порядке, как они получались, а затем напечатать их в обратном порядке. Второй способ —воспользоваться рекурсией, при которой printd сначала вызывает себя, чтобы напечатать все старшие цифры, и затем печатает последнюю младшую цифру. Эта программа, как и предыдущий ее вариант, при использовании самого большого по модулю отрицательного числа работает неправильно.

```
#include<stdio.h>

/* printd: печатает nкак целое десятичное число */
void printd(int n){
        if (n < 0) {
                putchar('-');
                n = -n;
        }
        if (n / 10) {
                printd(n / 10);
        }
        putchar(n % 10 + '0');
}
```

Когда функция рекурсивно обращается сама к себе, каждое следующее обращение сопровождается получением ею нового полного набора автоматических переменных, независимых от предыдущих наборов. Так,в обращении printd(123)при первом вызове аргумент n= 123, при втором —printd получает аргумент 12, при третьем вызове —значение

1. Функция printd на третьем уровне вызова печатает 1 и возвращается на второй уровень, после чего печатает цифру 2 и возвращается на первый уровень.Здесь она печатает 3 и заканчивает работу.

Рекурсивная программа не обеспечивает ни экономии памяти, поскольку требуется где-то поддерживать стек значений, подлежащих обработке, ни быстродействия; но по сравнению со своим нерекурсивным эквивалентом она часто короче, а часто намного легче для написания и понимания. Такого рода программы особенно удобны для обработки рекурсивно определяемых структур данных вроде деревьев.

Используемые источники:

1. Язык программирования Си. Авторы: Б. Керниган, Д. Ритчи.
2. Онлайн компилятор языка Си: https://www.onlinegdb.com/online_c_compiler
3. Онлайн блок-схемы: https://app.diagrams.net

Задание:

При решении каждой задачи, необходимо создать блок-схемы. Решением работы должен служить файл-отчёт, в котором будут располагаться скриншоты кода, выполнения программ, а также сами блок-схемы.

1. Дана функция НОД (наибольшего общего делителя). Создайте на основе неё процедуру в языке программирования Си.

НОД(a, b) = a, если b = 0,

НОД(a, b) = НОД(b, a mod b).

2. Дана функция фиббоначи. Создайте на основе неё процедуру в языке программирования Си.

ФИБ(a) = 0, если $a \leq 1$,

ФИБ(a) = ФИБ(a-1) + ФИБ(a-2).

3. Дана функция факториала. Создайте на основе неё процедуру в языке программирования Си.

ФАКТ(a) = 1, если $a \leq 1$,

ФАКТ(a) = a * ФАКТ(a-1).

4. Дана функция суммы числа. Создайте на основе неё процедуру в языке программирования Си.

СУМ(a) = 0, если $a \leq 0$,

СУМ(a) = a + СУМ(a-1).

5. Дана функция нахождение чисел Ферма, применяющихся в криптографии ($a \leq 4$). Создайте на основе неё процедуру в языке программирования Си.

ФЕРМ(a) = $2^{2^a} + 1$.

6. Дана функция теста Ферма для нахождения или проверки простых чисел. Создайте на основе неё процедуру в языке программирования Си.

ТЕСТ(p) = 1, если $a^{p-1} \bmod p = 1$, где a - это случайное число от 2 до p-2.

ТЕСТ(p) = 0, если $a^{p-1} \bmod p \neq 1$.

7. Дана функция вычисления дискриминанта. Создайте на основе неё процедуру в языке программирования Си.

ДИСК(b, a, c) = $b^2 - 4 * a * c$.

8. Дана функция символа Лежандра, определеющая квадратичные вычеты. Создайте на основе неё процедуру в языке программирования Си.

ЛЕЖ(a, p) = 0, если $a \bmod p = 0$, где p - простое число,

ЛЕЖ(a, p) = 1, если $a^{(p-1)/2} \bmod p = 1$,

ЛЕЖ(a, p) = -1, если $a^{(p-1)/2} \bmod p = p-1$.