# Функции и структура программы

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение, 2020

# Содержание лекции

# Функции в языке C

Функции являются основным средством структуризации программы на языке C. Программа на языке C состоит из функций и объявлений переменных.
Функции позволяют:

- Сделать структуру программы более понятной, путем разбиения программы на части, заключенные внутри функций. Хорошо написанные функции скрывают подробности своей работы от тех частей программы, которым их знать не положено, таким образом проясняя задачу в целом и облегчая процесс внесения исправлений и дополнений.
- Повысить производительность с помощью повторного использования разработанных ранее функций.

# Функции и процедуры

В языке Pascal существовало два типа подпрограмм, отличающихся синтаксически: функции (возвращающие значение результата) и процедуры (не возвращающие значение).
В языке C существует единственное понятие - функция. Процедуры (подпрограммы, не возвращающие значения) являются функциями особого вида, имеющими тип результата `void`.

# Функции в языке C

В языке C существуют понятия объявления (декларации) и определения функции.
Объявление функции содержит только информацию о количестве и типе аргументов и типе возвращаемого значения и не содержит тела функции. Объявление функции заканчивается знаком ";". Объявление функции должно стоять в программе до ее первого вызова.
Необходимость использования объявлений функций в языке C вызвана тем, что язык разрабатывался исходя из требования разбора программы за один проход.

# Функции в языке C

Определение функции в языке C имеет формат:

```
возвр_тип имя_функции(список параметров)
{
    тело функции
}
```

Минимальная функция: `dummy() { }`
В отличие от переменных, все параметры функций должны объявляться отдельно, причем для каждого из них надо указывать и тип, и имя:

```
f(тип имя_переменной1, тип имя_переменной1, ..., тип имя_переменнойN)
```

```
f(int i, int k, int j)      /* правильно */
f(int i, k, float j)        /* не правильно, у переменной
                               k должен быть собственный
                               спецификатор типа*/
```

# Оператор `return`

Оператор `return` - это механизм возвращения значений из вызываемой функции в вызывающую. После `return` может идти любое выражение:

```
return выражение;
```

По мере необходимости выражение преобразуется в тип, возвращаемый функцией согласно ее объявлению и определению. Часто выражение заключают в круглые скобки, но это не обязательно.

# Пример функции

```
/* strindex: возвращает индекс строки t в s, -1 при
отсутствии */
int strindex( char s[], char t[]){
    int i, j, k;
    for (i = 0; s[i] != '\0'; i++){
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            if (k > 0 && t[k] == '\0')
                return i;
    }
    return -1;
}
```

# Аргументы функции

В языках программирования имеется два способа передачи значений подпрограмме. Первый из них - передача по значению. При его применении в формальный параметр подпрограммы копируется значение аргумента. В таком случае изменения параметра на аргумент не влияют.
Вторым способом передачи аргументов подпрограмме является передачи по ссылке. При его применении в параметр копируется адрес аргумента. Это значит, что, в отличие от передачи по значению, изменения значения параметра приводят к точно таким же изменениям значения аргумента.
За небольшим количеством исключений, в языке C для передачи аргументов используется передача по значению. Обычно это означает, что код, находящийся внутри функции, не может изменять значений аргументов, которые использовались при вызове функции.

# Примеры передачи значений функции

Передача по значению

```
int sqr(int x){
    x = x * x;
    return x;
}
```

Передача по ссылке

```
void swap(int *x, int *y){
    int temp;
    temp = *x;  /* сохранить значение по адресу x */
    *x = *y;  /* поместить y в x */
    *y = temp;  /* поместить x в y */
}
```

# Функция main

При запуске программы на языке C управление передается функции с именем main, которая должна присутствовать в каждой программе. В полном варианте функция main имеет следующий формат:

```
int main(int argc, char *argv[]){
    операторы
    return код_завершения;
}
```

С помощью аргументов argc и argv в программу передаются параметры командной строки, заданные при запуске программы. код_завершения представляет собой целое значение, которое передается в вызывающий процесс. Обычно при успешном (неаварийном) завершении программы код_завершения равен нулю. При необходимости результат функции main или ее аргументы могут отсутствовать, например:

```
int main(){
    операторы
}
```

# Аргументы функции main

С помощью аргументов функции main argc и argv в программу передаются параметры командной строки. Аргумент argc содержит число параметров командной строки. Значение argc всегда не меньше единицы, т.к. первым параметром всегда передается имя программы. Аргумент argv представляет собой массив указателей на строки, в каждой из которых передается параметр командной строки с соответствующим номером.
Пусть программный файл myprog был запущен командой
`myprog -ig file1.txt`

В этом случае аргументы будут иметь следующие значения:

```
argc = 3,
argv[0] = “myprog”,
argv[1] = “-ig”,
argv[2] = “file1.txt”
argv[3] = 0
```

# Пример

```c
/* Программа счета в обратном порядке. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
int main(int argc, char *argv[]) {
    int disp, count;
    if(argc<2) {
        printf("В командной строке необходимо ввести число, с которого\n");
        printf("начинается отсчет. Попробуйте снова.\n");
        exit(1);
    }

    if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
    else disp = 0;

    for(count=atoi(argv[1]); count; --count)
        if(disp) printf("%d\n", count);

    putchar('\a'); /* здесь подается звуковой сигнал */
    printf("Счет закончен");
    return 0;
}
```

# Препроцессор C

Некоторые средства языка C реализованы с помощью препроцессора, работа которого по сути является первым этапом компиляции. Два наиболее часто используемых средства — это директива `#include`, включающая все содержимое заданного файла в компилируемый код, и директива `#define`, заменяющая некий идентификатор заданной последовательностью символов. Другие средства, которые будут рассмотрены, — это условная компиляция и макросы с аргументами.

# Включение файлов. Директива #include

Включение файлов задается одной из следующих строк в исходном коде:

```
#include "имя_файла"
#include <имя_файла>
```

При обработке текста строка заменяется содержимым файла `имя_файла`. Если `имя_файла` указано в кавычках, поиск файла обычно начинается с того места, где находится исходный текст программы. Если имя файла заключено в угловые скобки < и >, то поиск файла продолжается по правилам, зависящим от реализации языка, как правило в каталоге, предназначенном для хранения заголовочных файлов. Обычно, стандартные файлы в директиве `#include` указываются в угловых скобках, а файлы пользователя – в кавычках. Включаемый файл может в свою очередь содержать директивы `#include`.

# Макроподстановки

Следующая конструкция задает макроопределение, или макрос:

```
#define имя_макроса текст_для_замены
```

Это - макроопределение простейшего вида; всякий раз, когда `имя_макроса` встретится в тексте программы после этого определения, оно будет заменено на `текст_для_замены`.
Имя в `define` имеет ту же форму, что и имена переменных, а текст для замены может быть произвольным. Обычно `имя_макроса` пишут заглавными буквами.
Обычно текст для замены умещается на одной строке, но если он слишком длинный, его можно продлить на несколько строк, поставив в конце каждой строки символ продолжения \.
Текст для подстановки может быть совершенно произвольным. Например, таким образом можно определить имя `forever`, обозначающее бесконечный цикл:

```
#define forever for (;;) /* бесконечный цикл */
```

# Макросы с аргументами

Можно также определять макросы с аргументами, чтобы изменять текст подстановки в зависимости от способа вызова макроса. Например, так определяется макрос с именем max:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Хотя внешне он напоминает вызов функции, на самом деле max разворачивается прямо в тексте программы путем подстановки. Вместо формальных параметров (A и B) будут подставлены фактические аргументы, обнаруженные в тексте. Возьмем, например, следующий макровызов:

```
x = max(p+q, r+s);
```

При компиляции он будет заменен на следующую строку:

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Если всегда задавать при вызове соответствующие по типу аргументы, нет необходимости иметь несколько различных макросов max для различных типов данных, как это было бы с функциями.

# Побочные эффекты макросов с аргументами

Если внимательно изучить способ подстановки макроса `max`, можно заметить ряд "ловушек".
Так, выражения в развернутой форме макроса вычисляются дважды, и это даст неправильный результат, если в них присутствуют побочные эффекты наподобие инкремента-декремента или ввода-вывода.
Например, в следующем макровызове большее значение будет инкрементировано два раза:

```
max(i++, j++) /* НЕПРАВИЛЬНО */
```

Необходимо также аккуратно использовать скобки, чтобы гарантировать правильный порядок вычислений. Представьте себе, что произойдет, если следующий макрос вызвать в виде `square(z + 1)`:

```
#define square(x) x * x /* НЕПРАВИЛЬНО */
```

# Директива #undef

Директива #undef удаляет ранее заданное определение имени макроса, то есть "аннулирует"его определение; само имя макроса должно находиться после директивы.
В общем виде директива #undef выглядит таким образом:

```
#undef имя_макроса
```

Директива #undef используется в основном для того, чтобы локализовать имена макросов в тех участках кода, где они нужны.

# Условная компиляция

Существует возможность управлять самой препроцессорной обработкой, используя условные директивы, которые выполняются по ходу этой обработки.
Это делает возможным условное включение фрагментов кода в программу в зависимости от условий, выполняющихся в момент компиляции.
К директивам условной компиляции относятся директивы:

- `#if`,
- `#else`,
- `#elif`,
- `#endif`,
- `#ifdef`,
- `#ifndef`.

# Директивы if и endif

Директивы #if и #endif дают возможность в зависимости от значения константного выражения включать или исключать те или иные части кода. В простейшем виде директива #if выглядит таким образом:

```
#if константное_выражение
    последовательность операторов
#endif
```

Если находящееся за #if константное выражение истинно, то компилируется код, который находится между этим выражением и #endif.В противном случае этот промежуточный код пропускается. Директива #endif обозначает конец блока #if.

# Директивы #else и #elif

Директива #else позволяет в блоке #if-#endif выбрать для включения в программу один из двух возможных фрагментов кода в зависимости от истинности или ложности заданного константного выражения

```
#if константное выражение
    последовательность операторов 1 /* выражение истинно */
#else
    последовательность операторов 2 /* выражение ложно */
#endif
```

Директива #elif эквивалентна комбинации директив #else и #if. Директива #elif позволяет организовать выбор одного из нескольких фрагментов кода для включения в программу. При этом фрагмент кода включается в программу когда истинно соответствующее ему константное выражение.

# Формат директивы #elif

```
#if выражение
    последовательность операторов
#elif выражение 1
    последовательность операторов
#elif выражение 2
    последовательность операторов
#elif выражение 3
    последовательность операторов
#elif выражение 4
. . .
#elif выражение N
    последовательность операторов
#endif
```

# Пример условной компиляции

В следующем фрагменте кода анализируется SYSTEM и принимается решение, какую версию заголовочного файла включать в программу:

```
#if SYSTEM == WIN_9X
    #define HDR “win9x.h"
#elif SYSTEM == WIN_NT
    #define HDR “winnt.h"
#elif SYSTEM == LINUX
    #define HDR “linux.h"
#else
    #define HDR "default.h"
#endif

#include HDR
```

# Директивы #ifdef и #ifndef

Другой способ условной компиляции — это использование директив #ifdef и #ifndef, которые соответственно означают "if defined"(если определено) и "if not defined"(если не определено). В общем виде #ifdef выглядит таким образом:

```
#ifdef имя_макроса
    последовательность операторов
#endif
```

Блок кода будет компилироваться, если имя макроса было определено ранее в директиве #define.
В общем виде директива #ifndef выглядит таким образом:

```
#ifndef имя_макроса
    последовательность операторов
#endif
```

Блок кода будет компилироваться, если имя макроса еще не определено в директиве #define. И в #ifdef, и в #ifndef можно использовать директивы #else или #elif.

# Заголовочные файлы

Большую программу затруднительно разместить в одном файле исходного кода. Ее приходится разбивать на несколько файлов исходного кода, содержащих переменные и функции, которые могут компилироваться отдельно. При этом необходимо дать ответы на следующие вопросы:

- Как записать объявления переменных, чтобы они правильно воспринимались во время компиляции?
- Как организовать объявления так, чтобы при компоновке программы все ее части правильно объединились в одно целое?
- Как организовать объявления так, чтобы каждая переменная присутствовала в одном экземпляре?
- Как инициализируются внешние переменные?

В языке C принят способ выделения части объектов программы в так называемые заголовочные файлы (имеющие расширение *.h). В заголовочные файлы выносят объявления переменных и функций, а их определения размещают в обычных файлах исходного кода (*.c).

# Заголовочные файлы

В заголовочном файле могут содержаться:

- Определения типов `struct point  int x, y; ;`
- Описания функций `extern int strlen(const char*);`
- Описания данных `extern int a;`
- Определения констант `const float pi = 3.141593;`
- Перечисления `enum bool  false, true ;`
- Директивы include `#include`
- Определения макросов `#define Case break;case`
- Комментарии `/* проверка на конец файла */`

В заголовочных файлах НЕ содержатся:

- Определения обычных функций `char get()  return *p++;`
- Определения данных `int a;`
- Определения сложных константных объектов `const tbl[] =  /* ... */`