Основы программирования на C, часть 2

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение, 2019

Содержание лекции

Функции

Программа на языке Си представляет собой набор функций.

- для того, чтобы программа была выполняемой, она должна содержать функцию с названием main , которой передается управление при запуске программы на выполнение;
- отсутствует понятие вложенных функций, все функции находятся на одном уровне видимости (глобальном), и их имена должны быть уникальны в рамках программы.

```
[<тип результата>]<имя функции>([список параметров])
{  [декларации]
   [операторы]
}
```

Возврат значения функцией

- return <выражение>;
- по умолчанию - возвращаемое значение типа int;
- void - отсутствие возвращаемого значения.

Прототим - описание функции, средство контроля за соответствием фактических параметров в вызове функции ее формальным параметрам.

```
/*прототип */
int f(char);
     ...
/* определение */
int f(char c)
{
 /*…тело функции…*/
}
```

новая нотация

```
/*прототип */
int f();
     ...
/* определение */
int f(c)
char c;
{
 /*…тело функции…*/
}
```

старая нотация

Прототип

- нужен, если вызов встречается до определения;
- если прототип отсутствует? функция, которая возвращает int;
- int foo() и int foo(void);
- аргументы всегда передаются в функцию 'по значению';

Задача. Написать функцию, суммирующую два вещественных значения в двух вариантах: сумма – возвращаемое значение функции и сумма – параметр функции.

```
 . . .
double sum1(double,double);/* прототип
                                    функции sum1*/
void sum2(double,double,double*); /* прототип
                                   функции sum2*/
main()
{ double a1=2.5, u;
  int a2=3;
  u=sum1(a1,a2);
  printf("u=%f\n",u); /* u=5.5 */
  sum2(a1,a2,&u);
  printf("u=%f\n",u); /* u=5.5 */
}
double sum1(double x,double y)
{ return x+y; }
void sum2(double x,double y,double *z)
{ *z=x+y; }
```

Список параметров переменной длины

```
int printf(const char *format, …);
```

Использование

- функции не известны ни типы передаваемых неименованных параметров, ни их количество;
- подключить стандартную библиотеку stdargs.h;
- описатьпеременную типа va_list;
- обратиться к макросу va_start;
- получать значение следующего параметра - va_arg;
- завершение - va_end;

```
int f(p1, p2, ...)
{
  va_list parg;
  va_start(parg, p2);/* p2 – последний
          именованный параметр функции f */
  int i = va_arg(parg, int);
     ...
  va_end(parg);
}
```

Общая структура программы

- может размещаться как в одном, так и в нескольких файлах;
- может содержать одну или несколько функций;
- одна из которых считается главной (main) - точка входа в программу;
- определение каждой функции должно полностью размещаться в одном файле;
- один этом файл может содержать несколько определений различных функций.

Характеристики переменных

- область видимости;
- область существования.

Область видимости переменной

определяет часть исходного текста программы, из любой точки которой доступна данная переменная.

- видимые в пределах блоков (локальные);
- видимые в пределах файла;
- видимые в пределах программы.

Для переменных, определенных в начале любого блока, областью видимости является весь этот блок.
В случае вложенных блоков переменные, определенные внутри вложенного блока, 'перекрывают' переменные с такими же именами, определенные в объемлющем блоке (и так для любой степени вложенности).

```
main()
{ int x = 1;
  if(x>0)
  {  int x = 2;
     printf("x = %d\n", ++x); /*выводит: x = 3 */
  }
  printf("x = %d\n", x); /*выводит: x = 1 */
}
```

Переменные, определенные внутри функции, “перекрывают” формальные параметры с теми же именами.

```
int f(int f)
{
  int f = 1;
     …
}
```

Переменные, определенные вне блоков (т.е., фактически, вне тела какой-либо функции), доступны с точки определения до конца файла. Если на такую переменную нужно сослаться до того, как она определена, необходимо ее описание со спецификатором extern.

```
int x;
main()
{
  extern int y;
  x = y = 10;
  printf("x=%d, y=%d\n",x,y); /* x=10, y=10 */
}
 …
int y;
```

В файле вне функций не может встречаться несколько определений переменных (возможно, разных типов) с одним и тем же именем.

```
int x;
main()
{
   ...
}
float x;/* ошибка: повторное определение x */
```

Указание static, примененное к нелокальной переменной или функции, ограничивает область их видимости концом файла.
Если используемая переменная определена в другом программном файле, она также должна быть описана со спецификатором extern.
При таком описании переменной память под нее не отводится, а только декларируется тип переменной, что позволяет компилятору осуществлять проверки типизации.

Область существования переменной

множество всех точек программы, при приходе управления на которые переменная существует, т.е. для нее выделена память.

Статические переменные

- существуют на всем протяжении работы программы;
- выделяется на этапе редактирования внешних связей и загрузки программы;
- тогда же происходит и инициализация статических переменных;
- инициализатором для статической переменной может служить только константное выражение;
- при отсутствии инициализатора статические переменные по умолчанию инициализируются нулем;
- все переменные, определенные вне функций, являются статическими.

Определение статической переменной, локализованной в блоке

```
int max; /*статическая переменная вне блока*/
int f(int param)
{
  static int min; /* статическая переменная,
                     определенная внутри блока */
      …
}
```

Сохранение значений статической переменной при выходе из блока

```
#include <stdio.h>
void print_a();
main()
{ int i;
  for(i=0; i<5; i++)
     print_a();
}

void print_a()
{
  static int a = 1;
  printf("a = %d\n", a++);

}
```

в результате напечатается:

```
a = 1
a = 2
a = 3
a = 4
a = 5
```

Автоматические переменные

- все переменные определенные внутри блока (функции) и не являющиеся статическими;
- существуют на протяжении работы блока, в котором они определены, включая блоки, вложенные в данный.
- выделение памяти под автоматические переменные и их инициализация осуществляется каждый раз при входе в блок;
- инициализация для автоматических переменных, фактически, эквивалентна присваиванию, т.е. в качестве инициализирующего выражения может выступать любое выражение, которое может стоять в правой части оператора присваивания;
- при отсутствии инициализатора начальное значение по умолчанию для автоматических переменных не определено.

```
#include <stdio.h>
void print_a();
main()
{ int i;
  for(i=0; i<5; i++)
     print_a();
}

void print_a()
{
  int a = 1;
  printf("a = %d\n", a++);

}
```

в результате напечатается:

```
a = 1
a = 1
a = 1
a = 1
a = 1
```

Регистровые переменные

- квалификатор register в определении переменных указывает компилятору, что данную переменную в целях ускорения программы имеет смысл разместить на регистрах, однако компилятор может проигнорировать это указание;
- может применяться только к автоматическим переменным и формальным параметрам функций;
- для регистровых переменных не определено понятие адреса (т.е. не определена операция &).

...

a = 2

адрес возврата

сохраненный fp

сохраненные регистры

auto переменная i

auto переменная r
(если ее не удалось разместить на регистрах)

a = 1

адрес возврата

сохраненный fp

сохраненные регистры

auto переменная i

auto переменная r
(если ее не удалось разместить на регистрах)

```
int f(int a){
  int i;
  register int r;
     ...
  r = a--;
  if(r){
     return f(r);
  }
  return r;
}

main(){
  ...
  int b = f(2);
  ...
}
```

стековый кадр – непрерывная область памяти стека, используемая функцией для своей работы

fp (*frame pointer*) – указатель стекового кадра

Этапы стандратной схемы трансляции Си-программы

- препроцессирование;
- компиляция.

Препроцессор

предварительная стадия обработки исходного текста программы, по окончании работы которой модифицированный исходный текст поступает на вход компилятору.

Задачи препроцессора

- удаление комментариев;
- «склеивание» строчек, последним символом которых является \, со следующей за ними строкой;
- конкатенация рядом стоящих строковых литералов;
- макроподстановка;
- условная компиляция;
- подключение файлов.

```
#include <stdio.h>  /* подключение файла */
#define  a "max=%d\n" /* макроподстановка */
int x=15;
max(int);
main()
{ int y, u;
  scanf("%d",&y);
  u=max(y);
  printf(a,u);
}
max(int f)
{ int k;
  k=(x>f)?x:f;
  return k;
}
```

Включение файлов: на место директивы #include препроцессор подставляет содержимое указанного в ней файла.

- Если имя файла заключено в $< >$, то он ищется в стандартном каталоге подключаемых файлов.
- Если же имя файла заключено в кавычки и не является полным (абсолютным) путевым именем файла, то препроцессор рассматри вает его как относительный путь от местонахождения того файла, в котором встретился #include.

```
#include <stdio.h>
#include "myfile.h"
#include "/usr/stuff/myfile.h"
```

Макроподстановка: #define имя подставляемый_текст.

- Если имя файла заключено в < >, то он ищется в стандартном каталоге подключаемых файлов.
- Если же имя файла заключено в кавычки и не является полным (абсолютным) путевым именем файла, то препроцессор рассматри вает его как относительный путь от местонахождения того файла, в котором встретился #include.

```
#define NUMBER 10
#define DOUBLED_NUMBER NUMBER*2 /*в #define-
определении можно использовать более ранние
определения11 */
#define PrintHello printf("Hello,\
world"); /* "\" используется для продолжения
определения на следующей строке */
```

"Забыть"определенное имя - #undef имя

Условная компиляция

управление выборочным включением того или иного текста в программу в зависимости от вычисляемого на этапе препроцессирования условия.

```
if-директива выражение_или_идентификатор
...
[{#elif выражение}
...
#else
... ]
#endif
```

где **if-директива** – это:

```
#if константное_выражение
```

Пример

```
#ifdef имя /* 1, если имя уже определено */
#ifndef имя /*1, если имя не было определено */
```

Пример

```
#ifndef MYFILE
#define MYFILE

  …содержимое
  подключаемого
  файла…

#endif
```

или

```
#if !defined(MYFILE)
#define MYFILE

  …содержимое
  подключаемого
  файла…

#endif
```

Указатель

переменная, значение которой есть адрес некоторой области памяти.

Пример

```
char c, *p;
```

Операции:

- & - операция взятия адреса объекта;
- * - операция косвенной адресации (или раскрытия ссылки).

Пример

```
double x=1.5,y=7.1,*dp;
dp=&x;
dp++;
y=*dp-3;
```

Нетипизированный указатель

указатель на неопределенный тип: void *vp

Пример

```
int *ip, k;
void vp;
ip=&k;
vp=ip;          // Правильно
ip=vp;          // Неправильно
ip=(int*)vp;    // Правильно
```

Пример

```
k+=*(int*)vp;   // Правильно
(int*)vp++;        // Неправильно
((int*)vp)++;   // Правильно
```

Массив

собой совокупность элементов одного и того же типа.

Пример:

- int v[10];
- int m[3][4];
- операция индексации: m[i][j]
- инициализация: int m1[5]=0,1,2,3, m2[10]=0;
- char str1[ ]='a','b','c','\0';
- char str2[ ]="abc"
- int w[3][3]=1,2,3,4,5;

Связь массивов и указателей

Имя массива

константный указатель, содержащий адрес его нулевого элемента.

Пример: double m[10], *dp;;

- dp = m;
- dp = &m[0];

```
dp=m;       - адрес m[0]
dp+1        - адрес m[1]
dp+i        - адрес m[i]
*(dp+i)     - эквивалентно m[i]
m++;        /* ошибка, константный указатель
            изменять нельзя */
```

Операция индексации E1[E2] определена как *(E1+E2).

В вещественном массиве найти максимальное значение

```
double m[100], *p, max;
.   .   .
for( p=m+1,max=*m;p<m+100;p++)
  if(max<*p) max=*p;
```

Написать функцию суммирования двух целочисленных векторов

Размер векторов и результирующий массив передаются в качестве параметров

```
   . . .
void sum_vec(int*,int*,int*,int);
main()
{ int m1[20],m2[20],m3[20],i;
   . . .
  sum_vec(m1,m2,m3,20);
  for(i=0;i<20;i++)
     printf("m3[%d]=%d\n",i,m3[i]);
}
void sum_vec(int x[],int *y,int z[],int k)
{ int i;
  for(i=0;i<k;i++)
     *z++=x[i]+y[i]; /*z[i]=x[i]+y[i];*/
}
```

Написать функцию, определяющую длину строки

```
int len_str(char * str) {
      int len;
      char * p = str;
      while(*p) p++; /* Си-строка оканчивается
                        нулем */
      len=p-str;}
      return len;
}
```

Написать функцию копирования строк

```
void copy (char source[], char dest[]) {
      while( *dest++=*source++);
}
main() {
      char s[100], t[100];
      copy (s, t);
}
```

Распечатать последний символ каждой строки

Написать функцию, которой в качестве аргумента передается массив указателей на строки (признак конца – нулевой указатель). Распечатать последний символ каждой строки.

```
void fp(char *s[])/* void fp(char **s)
                              тождественно */
{
      int i=0;
      while(s[i]!=NULL) {
            printf("%c\n", *(s[i]+ strlen(s[i])-1));
            i++;
      }
}
```