EPC

Программирование на C для встраиваемых систем

Основные вопросы

Для начала:

- История создания и распространения языка C
- Области применения C
- Чего ожидать от C
- Чего не следует ожидать от C

Давным-давно…

- В 1969 году Кен Томсон (Ken Thomson) из Bell создал первую версию ОС UNIX на ассемблере, которая была непортируемой
- В 1970 году он же разработал язык B для PDP/7, который состоял только из машинных инструкций, без типов переменных
- В 1974 году Деннис Ричи (Dennis M. Ritchie) переработал язык B и назвал его C. Он предназначался для PDP 11

Сейчас

- Сейчас C развился в интероперабельный язык
- Компиляторы C доступны для всех платформ: от микроконтроллеров до майнфреймов, и для всех ОС
- В процессе развития язык претерпевал много изменений, что потребовало стандартизации

Стандартизация

- В 1988 году Комитет ANSI издал свой стандарт языка, называемый сейчас ANSI C
- Все современные компиляторы должны быть ANSI-совместимы

Стандартизация

- Библия программиста: Брайан Керниган и Деннис Ричи «Язык программирования С»

  Brian W. Kernighan and Dennis M. Ritchie
  “The C Programming Language“

Стандартизация

- Вопрос:
  «Есть ли хорошая книга по C?»
- Ответ:
  «Да, одна!»

Стандартизация

Основные замечания

- C не является настоящим языком высокого уровня, он находится где-то между ассемблером и языками высокого уровня, типа Pascal
- C объединяет две уникальные особенности:
  - Хорошая адаптируемость к архитектуре процессора
  - Высокая переносимость

Нет больше тайн!

- Компилятор С генерирует очень эффективный код
  - **Благодаря оптимизации**
- Большинство задач решаются на С без применения ассемблера
- С – простейший язык программирования
  - **Снижается время разработки и выпуска программ**
- Приложение на С может быть перенесено на другой процессор
- Повышается читаемость программ
  - **Программа самодокументируемая**

Язык С применяется

- Для ПК, но есть тенденция к переходу на объектно-ориентированные языке, типа Java и C#
- В микроконтроллерах – переход от ассемблера к языкам высокого уровня
- С является дружественным к аппаратуре языком

Особенности С

- С – сравнительно примитивный язык и, поэтому, не столь дружественный
- В С поддерживается четыре типа данных:
  char, int, float и double
- Эти типы данных модифицируются с помощью: signed, short, long…
- Комплексные типы данных (строки, массивы, структуры и т.д.) получаются из простейших типов

Особенности С

- С имеет все необходимы инструменты для управления ходом выполнения программы:
  - Простое ветвление (if-else)
  - Множественное ветвление (switch-case)
  - Циклы (for)
  - Циклы с предусловием (while)
  - Циклы с постусловием (do-while)
- Поддерживается вызов функций
- Процедуры ввода/вывода не являются частью языка и реализуются через стандартные или пользовательские библиотеки

Особенности С

- В функцию передаются копии параметров, сами значения параметров не модифицируются
- Если передаваемый параметр является адресом, то функция может модифицировать значения ячеек, чей адрес ей передается
- Указатели являются мощным инструментом и ими нужно уметь пользоваться
- Преобразование типов данных прозрачное, но помимо гибкости может добавить и проблемы

Использование C

- C до сих пор набирает популярность, потому, что обеспечивает гибкую работу с аппаратурой, как ассемблер, и удобство языков высокого уровня
- C наиболее подходит для системного программирования
- Используя C для микроконтроллеров PIC, вы пишите операционную систему :-)

Использование C

- C используется всегда, когда необходима переносимость приложений
- Никакой другой язык не дает такой гибкости в переносимости

Итоги

- Легко адаптируется к архитектуре процессора
- Приложения просто переносятся на другие платформы
- Эффективный код
- Простое программирование
- Хорошо приспособлен для программирования микроконтроллеров

1079 EPC
Программирование на C

Вступление

Пример программы

```
/* Example 001 */
#include <stdio.h>
main()
{
  printf ("hello, world\n");
}
```

Пояснения к примеру

- Любая С программа состоит из функций и использует переменные
- Синтаксис функций:
  - **MyFunction()**
- Любая программа должна иметь функцию **`main()`**
- **`main()`** вызывает другие функции, описанные пользователем или содержащиеся в библиотеках

Пояснения к примеру

- Операторы функции заключаются в фигурные скобки **{}**
- **#include <stdio.h>**
  Эта директива сообщает компилятору об использовании стандартной библиотеки ввода/вывода
- Компилятор проходит по тексту программы линейно
- **main()**
  В функцию не передаются никакие параметры. В этом случае можно писать **()** , но лучше **(void)**

Пояснения к примеру

- `printf ("hello, world\n");`
  Вызов функции с передаваемым параметром строка
- `\n`
  Обозначение конца строки

```
printf ("hello, world
        ");
```

Нельзя вставлять новую строку в явном виде, это вызовет ошибку компиляции

Пояснения к примеру

```
printf ("hello, ");
printf ("world");
printf ("\n");
```

Эта группа операторов делает то же самое

- В С нет четкого форматирования текста, как в ассемблере, за исключением строк и директив **#defines**
- Операторы разделяются **';'**

Замечания

- Обычно микроконтроллер не имеет связи с внешним миром посредством **`printf()`**
- Функция **`printf()`** использует низкоуровневую функцию вывода одного символа **`putc()`**, которая работает с реальным железом
- Если вы не используете терминал, подключенный к USART, вам не повезло :-(

Замечания

- MPLAB® IDE позволяет использовать USART в режиме симуляции с выводом данных в окно
- Это важно знать и использовать как дополнительную возможность для отладки

Пример №2

```
/* Example 002 */
/* conversion table Fahrenheit-Celsius
   for fahr = 0, 20, ..., 300 */
#include <stdio.h>
main()
{
   int lower,upper,step;
   float fahr,celsius;
   lower = 0;   /* lower limit of temperature table */
   upper = 300; /* upper limit */
   step = 20;   /* Sstep size */
   fahr = lower;
   while (fahr <= upper) {
     celsius = (5.0/9.0) * (fahr-32.0);
     printf ("%4.0f %6.1f\n",fahr,celsius);
     fahr = fahr + step;
   }
}
```

Пояснения к примеру

- **`fahr = lower;`**
  Присваивается левой части правая. Заодно производится преобразование типов
- **`while(fahr <= upper)`**
  `{`
  `}`
  Операторы в фигурных скобках будут повторятся, пока отношение **`fahr <= upper`** будет верно

Пояснения к примеру

- `/* Example 002 */`
  Текст между `/*` и `*/` игнорируется компилятором и используется для комментариев
- Т.к. текст просматривается компилятором линейно, объявление переменных должно быть раньше их использования

Пояснения к примеру

- Объявление переменных: тип данных и список переменных

```
int lower,upper,step;
float fahr,celsius;
```

- `printf ("%4.0f %6.1f\n",fahr,celsius);`
  `printf` использует три аргумента. Вместо знака **`%`** при выводе подставляются следующие аргументы в требуемом формате

Подробнее о `printf()`

- **%d** Десятичное целое
  %6d Десятичное целое, 6 разрядов
  %f Число с плавающей точкой
  %6f Число с плавающей точкой, 6 разрядов
  %.2f Число с плавающей точкой, 2 разряда после запятой
  %6.2f Число с плавающей точкой, 6 целых и 2 дробных разряда
- Другие возможности: **%o** – восьмеричное, **%x** – шестнадцати-ричное, **%c** – символ, **%s** – строка, и %% для вывода %

Пример 3

```
/* Example 003 */
/* print table Fahrenheit-Celsius
   for fahr = 0, 20, ..., 300 */
#include <stdio.h>
main()
{
  int fahr;
  for(fahr = 0; fahr <= 300; fahr = fahr +20)
  {
    printf("%4.0f
  %6.1f\n",fahr,((5.0/9.0)*(fahr-32.0)));
  }
}
```

Замечания

- Программа выглядит короче, но нужно всегда осознавать, что вам еще нужно ее отлаживать
- Если вы не имеете переменной **`celsius`**, вы не сможете ее отследить

Пример 4

```
/* Example 004 */
/* print table Fahrenheit-Celsius /*
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP  20
main()
{
  int fahr;
  for(fahr = LOWER; fahr <= UPPER;
                              fahr = fahr + STEP)
  {
    printf("%4.0f %6.1f\n"
                  ,fahr,((5.0/9.0)*(fahr-32.0)));
  }
}
```

Программистские соглашения

- Профессиональные программисты придерживаются следующих правил при написании программ:
  - Использование структурированного подхода
  - Не переоценивайте свой талант
  - Keep It Simple, Stupid (KISS)
  - Если вы погибнете, ваша компания потерпит неприятности ;-)
  - Думайте о своих приемниках при написании программ

uglyform.c

```
/* 01 */  #include <stdio.h>
/* 02 */  int main()  /* Main program starts
/* 03 */  here */{printf("Good form
  ");printf("can aid in ");
/* 04 */  printf("understanding a program.\n")
/* 05 */  ;printf("And bad form ");printf("can
  make a program ");
/* 06 */  printf("unreadable.\n");return 0;}
/* 07 */
```

goodform.c

```
/*01*/  #include <stdio.h>
/*02*/
/*03*/  int main()  /* Main program starts here */
/*04*/ {
/*05*/    printf("Good form ");
/*06*/    printf   ("can aid in ");
/*07*/    printf      ("understanding a program.\n");
/*08*/    printf("And bad form ");
/*09*/    printf      ("can make a program ");
/*10*/    printf                 ("unreadable.\n");
/*11*/
/*12*/    return 0;
/*13*/
/*14*/ } /* end main() */
/*15*/
```

style01.c

```
...
  for (index = 0 ; index < 7 ; index = index + 1) {
   printf("The value of ");
   printf("index is %d\n", index);
   if (count < 5) {
    printf("The value of count is %d ", count);
    printf(" this is less than 5\n");
   } else {
      loop_count = 0;
      do {
       printf("The value of loop_count is %d\n", loop_count);
       loop_count = loop_count + 1;
      } while (loop_count < 3);
      printf("The value of count is %d ", count);
      printf(" this is not less than 5\n");
     }
   }
...
```

style02.c

```
...
  for (index = 0 ; index < 7 ; index = index + 1)
  {
   printf("The value of ");
   printf("index is %d\n", index);
   if (count < 5)
   {
    printf("The value of count is %d ", count);
    printf(" this is less than 5\n");
   }
   else
   {
    loop_count = 0;
    do
    {
     printf("The value of loop_count is %d\n", loop_count);
     loop_count = loop_count + 1;
    } while (loop_count < 3);
    printf("The value of count is %d ", count);
    printf(" this is not less than 5\n");
   }
  }
...
```

Самоорганизация

Самоорганизация

- Как правильно организовать свою программу:
  - Правильно учитывать область видимости переменных
  - Использовать возможности препроцессора

Область видимости

- Функции и переменные не обязательно компилируются в одно время
- Исходный текст программы можно разбить на несколько файлов
- Скомпилированные файлы могут загружаться из библиотек

Область видимости

- Возникает несколько вопросов
  - Как правильно объявлять переменные для компиляции?
  - Как исключить повторное объявление переменных и функций?
  - Как подключать внешние переменные?

Область видимости

- Переменные, описанные в теле функции – локальные!
- Локальные переменные с одинаковыми именами в разных функциях никак не пересекаются
- Глобальные или внешние переменные действуют с момента их объявления до конца файла

Область видимости

Пример (неправильный):

```
void main(void)
  {
    initialize();

    while(1)
    {
      mainfunction();
    }
  }

  void initialize(void)
    { ... varSytemState = IDLE; ... };
  void mainfunction(void)
    {int varSystemState; ... };
```

Область видимости

- Сначала компилятор найдет имя main и будет удовлетворен, т.к. никакой ошибки нет
- Затем он найдет **`initialize`** и **`mainfunction`**, и присвоит им тип **`int`** по умолчанию
- Затем он дойдет до определения функций и увидит **`void`** – т.е. тип возвращаемого значения поменялся!
- Также будет сообщение об ошибке о незнакомой переменной **`varSystemState`**, т.к. она нигде не описана

Область видимости

- Разобьем программу на несколько файлов:

```
/* file : main.c */
#include "initialize.h"
#include "mainfunction.h”
#include "mainfunction.c"
void main(void)
{
  initialize();

  while(1)
  {
    mainfunction();
  }
}
```

```
/* file: initialize.h */
void initialize(void);

/* file: initialize.c */
#include "initialize.h"
extern int varSystemState;
void initialize(void)
{ ...
  varSytemState = IDLE;
 ...
};
```

```
/* file: mainfunction.h */
void mainfunction(void);

/* file: mainfunction.c */
int varSystemState; /* global */
void mainfunction(void)
{  ...  };
```

Область видимости

- Все должно компилируется без ошибок благодаря работе препроцессора, который подставляет вместо **`include`** нужные файлы

Препроцессор C

Препроцессор

- Программа компилируется в несколько проходов
  - Препроцессор
  - Компилятор
  - Линкер

Препроцессор

- Препроцессор анализирует все, что начинается с #
- Компилятор транслирует текст программы в промежуточный объектный код
- Линкер собирает все объектные коды в один исполняемый файл
- В Windows он имеет расширение ".exe"
- Для микроконтроллеров используются кросс-компиляторы, на выходе которых получается файл прошивки ".hex"

Препроцессор

- Препроцессор использует следующие директивы:

```
#include
#define
#undef
#if !defined or #ifndef
#elif  (short for else if)
#endif
#error
#pragma
```

Препроцессор

- **`#include "filename"`** или **`#include <filename>`** подставляет содержимое в текст программы при компиляции
- Если имя файла указано в кавычках, то поиск производится в текущем каталоге
- Если имя указано в **< >**, то препроцессор ищет файл в стандартном каталоге

Препроцессор

```
C:\project
        +-\common
        |    +-systemconstants.h
        +-\libs
        |    +-adc.h
        +-\myfiles
             +--main.c
             ( #include <pic24xxx.h>
               #include "initialize.h"
               #include "mainfunction.h"
              ...)
             +-initialize.h
             +-initialize.c
             ( #include <pic24xxx.h>
               #include "initialize.h"
               #include "c:\project\common\systemconstants.h"
              ...)
             +-mainfunction.h
             +-mainfunction.c
             ( #include <pic24xxx.h>
               #include "mainfunction.h"
               #include "..\libs\adc.h"
              ..)
```

Препроцессор

- Определение:
  `#define name replacement_text`
- Простейшая форма определения макроса: везде в тексте программы **`name`** заменяется на **`replacement_text`**
- Область видимости: от директивы **`#define`** до конца файла

Препроцессор

- Примеры:

```
#define forever while(1)
#define forever_too for(;;)

#define NOT_ACTIVE_LOW 1
#define ACTIVE_LOW     0

#define MICRO_SECONDS  1
#define THOUSEND_       1000
#define MILLI_SECOND (THOUSEND * MICRO_SECONDS)

#define NOT_KEY    PORTAbits.bit0
#define TEMPERATURE_SENSOR  AD_CHNL_0

#define ON_OFF_LED LATBbits.bit1
```

Препроцессор

- Макросы могут быть с аргументами
- В таком случае, подставляемый текст будет меняться от раза к разу
- Пример:

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

- Это выглядит как функция, но при компиляции код подставляется при каждом вызове
- Это называется inline-компиляция

Препроцессор

- Примеры:

```
#define trigger() ((TRIGGER_PIN=1),(TRIGGER_PIN=0))

#define BIT(x)(1 << (x)) /*used by bit op macros   */

#define BIT0 0x01        /*used by bit op macros   */
#define BIT1 0x02

...
#define BIT6 0x40
#define BIT7 0x80

#define SETBIT(reg,bit) reg|=bit
#define CLRBIT(reg,bit) reg&=(~bit)
#define TGLBIT(reg,bit) reg^=bit
#define TSTBIT(reg,bit) (reg&bit)

/* to be used like
SETBIT(LATB, BIT(BIT0)); */
```

Препроцессор

- Для проверки условий используются директивы **`#if`** и **`#endif`**
- Выражение после **`#if`** выполняется, если выражение не нулевое, иначе выполняется выражение после **`#else`**
- Это позволяет избежать повторных включений файлов

Препроцессор

- Пример:

```
#if !defined HEADER_FIRST
#define HEADER_FIRST

/* content of header.h goes here */

#endif"
```

- В файле **header.h** определена переменная **HEADER_FIRST**

Препроцессор

Пример:

```
#define SOFT_SIM

/* simulation may take a while */

#ifdef SOFT_SIM
  DelayUs(5);
#else
  DelayMs(20);
#endif

#define LOWER_NIBBLE

#ifdef LOWER_NIBBLE
  use_lower_bits_for_lcd();
#else
  use_upper_bits_for_lcd();
#endif
```

1079 EPC
Программирование на C

Типы, операторы, выражения

Элементы исходного кода

- Программа состоит из слов:
  - Идентификаторы
  - Ключевые слова
  - Константы
  - Строковые константы
- Они комбинируются с операторами для выполнения тех или иных задач
- Разделителями являются пробелы, табуляции и символы концы строки

Элементы исходного кода

- С использует кодовую страницу ASCII (128 знаков):

  только латинские буквы `a-z, A-Z`

  цифры `0-9`

  и спецсимволы `+-*/\=,.; ?"#`
  `%&_' <>()[]{}`
  `|^~@:`

- Символы других языков не разрешены (только в комментариях)

Элементы исходного кода

- Различается верхний и нижний регистры:

  С чувствителен к регистру !!!

- Ключевые слова – строчные: **`if else int float`**
  Общепринятое написание:
  - Имя переменной: **`MixedCaseWords`**
  - Константа: **`UPPER_CASE_WORDS`**
- Операторы разделяются **`;`**

Ключевые слова

- В C есть 32 ключевых слова !!!

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	***goto***	**sizeof**	**volatile**
do	**if**	**static**	**while**

Имена переменных

- В именах можно использовать буквы, цифры и подчеркивание
- Обычно, имена не начинаются с цифр и подчеркиваний. С подчеркиваний начинаются имена библиотечных функций
- Важно применять адекватные имена переменных
  - **`InputTemperature, OutputTemperature`**
  - А не **`t1, t2`**

Имена переменных

- Максимальная длина имени зависит от компилятора
- Примеры:

```
InputTemperature /* OK */
1_value          /* wrong */
Caution          /* OK */
_caution         /* sub optimal*/
SystemState      /* OK */
if               /* wrong */
```

Имена переменных

- Везде, где есть «свободное место», можно вставлять комментарии
- Комментарии ограничиваются `/*` и `*/`
- Для комментирования одной строки можно применять `//`

  Но это не совместимо с ANSI
- Комментируйте только то, что требует пояснений, не нужно писать очевидных вещей
  - `c = a + b;  /* сложить a и b */`

Типы данных

- В С известно несколько типов данных
  - **`char`** один байт, один символ
  - **`int`** целое число, разрядность зависит от системы
  - **`float`** число с плавающей точкой
  - **`double`** число с плавающей точкой двойной точности
- Типы данных могут меняться с помощью квалификаторов
  - **`short`, `long`, `signed`** и **`unsigned`**

Типы данных

```
#ifndef _LIMITS_H
#define _LIMITS_H

#define MB_LEN_MAX 1

/* char properties */
#define CHAR_BIT   8
#define SCHAR_MAX  0x7f
#define SCHAR_MIN  (-SCHAR_MAX -1)
#define UCHAR_MAX  0xff
#define CHAR_MIN   SCHAR_MIN
#define CHAR_MAX   SCHAR_MAX

/* int properties */
#define SHRT_MAX   0x7fff
#define SHRT_MIN   (-SHRT_MAX -1)
#define USHRT_MAX  0xffff
#define INT_MAX    SHRT_MAX
#define INT_MIN    SHRT_MIN
#define UINT_MAX   USHRT_MAX

  /* long properties */
  #define LONG_MAX   0x7fffffff
  #define LONG_MIN   (-LONG_MAX -1)
  #define ULONG_MAX  0xffffffffU

  /* long long properties */
  #define                    LLONG_MAX
0x7fffffffffffffffLL
  #define LLONG_MIN  (-LLONG_MAX -1)
  #define                         ULLONG_MAX
0xffffffffffffffffUL

  #endif
```

Типы данных

```
unsigned char      :   0 – 255

signed   char      :   -128 + 127

unsigned int       :   0 – 65535

signed   int       :   -32768 – 32767

unsigned long      :   0 – 4294967295

signed long        :   -2147483648 - 2147483647

unsigned long long :   0 - 18446744073709551615

signed long long   :   -9223372036854775808 - 9223372036854775807
```

Константы

- Четыре типа констант
  - Целые
  - Дробные
  - Символьные
  - Строковые

Целые константы

- Целые константы состоят из цифр без первого нуля, они считаются десятичными
- **1234** – целое; Константа **long** заканчивается **l** или **L**
- Беззнаковые – **u** или **U** и **ul** или **UL**

Целые константы

- Целые константы можно также задавать в 8-ричном или 16-ричном виде
- **`01234`** – 8-ричныя, начинается с нуля
- **`0x1f, 0X1F`** – 16-ричные

Целые константы

- Примеры:

```
    1        01   0x1
   10       012   0xA
 4368    010420   0x1110

35000   0104270   0x8868 /* implizit
                            long */
35000l  0104270l  0x8868l
   10L      012L  0xAL
```

Дробные константы

- Дробные константы можно указывать с запятой или с экспонентой
- Они по умолчанию двойной точности
- Примеры

```
123.4
1e-2
```

Символьные константы

- Эти константы содержат один байт и эквивалентны типу данных **`char`**, пишутся в одинарных кавычках
- Используется кодовая страница ASCII
- Символ **`'0'`** имеет значение **`48`** или **`0x30`**
- Эти константы могут представляться как целые, что дает возможность выполнять некоторые операции. Например

  `'0' + 9`, в итоге получим ASCII код символа 9

Символьные константы

- Предопределенные escape последовательности

\a	alert (bell)	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	\ooo	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab	'\0'	null character

Строковые константы

- Строковая константа – последовательность символов, заключенная в двойные кавычки:

```
"Hello, world !\n"
"I am a string"
""      /* this is an empty string */
"Hello," "world !\n"  /* Объединение строк */
```

- Используется для объединения константы, написанной в нескольких строках

Строковые константы

- Физически строка – массив символов
- В конце массива стоит символ **'\0'**
- Поэтому в памяти строка занимает на один байт больше

Переменные

- Все переменные должны описываться перед использованием
- Пример:

```
int lower, upper, step;
char c, line[55], buffer[80];
int InputVoltage;   /* any comment */
int OutputVoltage;
```

Переменные

- При описании переменной они могут быть инициализированы:

```
char esc = '\\';
int i = 1;
int limit = MAXLINE + 1;
float eps = 1.0e-5;
```

- Явная инициализация происходит при каждом вызове функции или блока, где эта переменная описана

Переменные

- Внешние и статические переменные по умолчанию инициируются нулевым значением
- Квалификатор **const** используется для указания того, что переменная не будет меняться в процессе выполения программы

```
const double e = 2.718288182845905;
const char msg[] = "warning: ";
```

- Последняя запись создает строку в памяти программ
- Правильнее использовать константы:

```
#define MAXLINE 1000
#define e 2.718288182845905
```

Переменные

- Для описания класса памяти переменной используется:

```
auto
register
static
extern
typedef
```

Переменные auto

- Объявление переменной с типом памяти **`auto`** резервирует область памяти для ее размещения
- Если переменная **`auto`** объявлена в теле функции, то она заново размещается в памяти при каждом вызове и удаляется при завершении работы функции
- Такие переменные не инициализируются и не сохраняют своего предыдущего значения

Внешние переменные

- Внешняя переменная (тип памяти **`extern`**) доступна из любой функции в программе
- Внешняя переменная сохраняет свое значение в процессе выполнения программы
- Внешняя переменная – глобальная

Статические переменные

- Переменная типа **`static`** сохраняет свое значение между вызовами функций
- Переменные **`static`** могут использоваться как внутри функций, так и вне
- Как пример, такая переменная может применяться для подсчета количества вызовов функции

Регистровые переменные

- Переменная типа **`register`** эквивалентна типу **`auto`**, но она сообщает компилятору, что она используется часто и размещать ее нужно в РОН
- Размещение переменной зависит от типа процессора и компилятора. Некоторые компиляторы игнорируют этот тип

Тип typedef

- **`typedef`** только описывает тип переменной, но не размещает ее в памяти
- Аналогичное действие можно сделать с помощью директивы **`#define`**
- Более подробно – позже

Переменные и область видимости

- Переменные, описанные в функциях – локальные. Они не доступны для других функций
- Две функции могут иметь локальные переменные с одинаковыми именами
- Локальные переменные размещаются в памяти при вызове функции и удаляются при выходе из нее
- По умолчанию такие переменные имеют тип памяти **`auto`**

Переменные и область видимости

- Переменные **`extern`** доступны из любой функции
- Внешняя переменная **`extern`** должна быть объявлена вне функций один раз (глобальная, без ключевого слова **`extern`**)
- В каждой функции, где планируется ее применение, необходимо объявить эту переменную с ключевым словом **`extern`**

Переменные и область видимости

```
/* code snipped */
   int max;  /* global definition */
   function_1()
   {
  int i;  /* local declaration */
  extern int max;/* use max from global definition */
   }
function_2()
   {
     int i; /* different than i from function_1*/
   }
   ...
   /* different file */
   extern int max;
   function_3()
   {...
     max = a + b; /* uses global max */
   }
```

Операторы

- Арифметические операции
- Логические операции и отношения
- Битовые операции
- Выражения и присваивания
- Условные выражения
- Приоритет операций

Арифметические операции

- 6 арифметических операторов

отрицательное число	-
сложение	+
вычитание	-
умножение	*
деление	/
остаток от деления	%

Арифметические операции

- Примеры

```
int i=3, j=5, k=10;
int m;
float r=3., s=5.;
float u;
m = i+j+k;              /* 18 */
m = k-j-i;              /*  2 */
m = i-k;                /* -7 */
m = k/i;                /*  3 */
m = k/i+j;              /*  8 */
m = k/(i+j);            /*  1 */
m = k*i+j;              /* 35 */
m = k*(i+j);            /* 80 */
m = k%i;                /*  1 */
u = s+r;                /* 8.       */
u = s/r;                /* 1.666... */
u = s*r;                /*15.       */
u = k/r;                /* 3.333... */
u = k/r +s/r;           /* 5.       */
u = k/i + s/i;          /* 4.666... */
```

Унарные операции

- Инкремент `++ /* плюс 1 */`
- Декремент `-- /* минус 1 */`

Может использоваться как пред-, так и постинкремент/декремент:

```
n = 5;
x = n++;  /* x == 5, n == 6 */
x = ++n;  /* x == 7, n == 7 */
x = n--;  /* x == 7, n == 6 */
x = --n;  /* x == 5, n == 5 */
```

Унарные операции

```
int i;
int k;
i=1;
if (++i > 1)
  k = 5;              /* will be performed */
if (i++ > 2)
  k = 10;             /* will not be performed */
printf ("\nk equals: %d\n",k);    /* output: 5 */

int i,k;
int j;
i=2;
k=3;
j = i+++k;             /* i+ (++k); i=2,k=4,j=6  */
j = (i++)+k;           /*           i=3,k=4,j=6  */
j = i++ +k;            /*           i=4,k=4,j=7  */
j = (i+k)++;           /* forbidden !!!           */
j = i+k+1;
```

Отношения

Больше	>
Больше или равно	>=
Меньше	<
Меньше или равно	<=
Равно	==
Не равно	!=

Отношения

- Примеры

```
int a=5;
int b=12;
int c=7;
a < b                  /* true   */
b < c                  /* false */
a+c <= b               /* true   */
b-a >= c               /* true   */
a == b                 /* false */
a+c == b               /* true   */
a != b                 /* true   */
a = b<c;               /* may be: a=0 */
a = c<b;               /*   -"-    a=1 */
```

Битовые операции

И	&
ИЛИ	|
Исключающее ИЛИ	^
Сдвиг влево	<<
Сдвиг вправо	>>
НЕ	~

Битовые операции

```
char i=7;          /* 0b00000111 */
char j=9;          /* 0b00001001 */
char k;
k = i & j;          /* k=1   */
k = i | j;          /* k=15  */
k = i ^ j;          /* k=14  */
k = ~0              /* k=max(unsigned int) */
k = ~i;             /* k=max(unsigned int) -7 */
k = i << 3;         /* k=56  */
k = i >> 3;         /* k=0   */
k = i & 0x03;       /* k=3; mask of all bits, but the
   lower two */
```

Выражения и присваивания

- Присваивание:
  правая часть – оператор – левая часть;

```
x = x + 1;
```

- Новое x равно старое x плюс 1 !!!
- Упрощенная форма

```
x += 1;
```

Условные выражения

- Использование условий:

```
if(a >b)
 z = a;   /* executed
             if condition true */
else
 z = b;   /* executed
             if condition false */
```

- Можно использовать тройной оператор " ? : ":

```
z = (a > b) ? a : b; /* z becomes
max(a,b) */
```

Приоритет операций

Оператор	Выполнение
()() [] -> .	Слева направо
! ~ ++ -- - (*type*) * & sizeof	Справа налево
* / %	Слева направо
+ -	Слева направо
<< >>	Слева направо
< <= > >=	Слева направо
== !=	Слева направо
&	Слева направо
^	Слева направо
|	Слева направо
&&	Слева направо
||	Слева направо
?:	Справа налево
= += -= += /= (etc.)	Справа налево
,	Слева направо

“void“

- **`void`** – описывает пустое значение, аргумент или результат

```
void main(void) ... /* get's no parameter,*/
void init(void) ... /* does not return a
                       value */
```

1079 EPC
Программирование на C

Выполнение программы

Операторы и блоки

Выражение типа

`x = 0` или **`i++`** или **`printf( … )`**

является оператором и завершается точкой с запятой.

`;` – разделитель между операторами

```
x = 0;
i++;
printf( … );
```

Операторы и блоки

Фигурные скобки используются для обозначения составного оператора или блока **{ }**. Составной оператор эквивалентен одиночному оператору

- Примеры
  - Тело функции
  - Операторы после **`if`**, **`else`**, **`while`**, **`for`**

if - else

`if – else` оператор простого условия:

```
if(expression)
  statement_1 /* expression is true */
else          /* optional */
  statement_2 /* expression is false */
```

true - false

ложь: выражение == 0

истина: выражение != 0

Любое ненулевое значение – ИСТИНА!!!

```
if(expression) /* is short for */
if(expression != 0) /* more readable */
```

else - if

Для множественного ветвления используется конструкция **`if – else if`**

```
if(expression)
  statement_1
else if(expression)
  statement_2
else if(expression)
  statement_3
else if(expression)
  statement_4
else
  statement_5
```

else - if

Выражения вычисляются по порядку; если выражение верно, то производится выполнение соответствующего оператора, после чего вычисление прекращается

```
if(TMR0IE && TMR0IF)
  TMR0_isr();
else if(INTE && INTF)
  INT_isr();
else if(RBIE && RBIF)
  PB_CHANGE_isr();
else
  Error_isr();
```

switch

Множественное ветвление организуется с помощью оператора **switch**:

```
switch(expression)
{
  case const_expression: statements
  case const_expression: statements
  default:               statements
}
```

switch

```
character = getchar(); /* case blind */
switch(character)      /* commands   */
{
  case 'a':
  case 'A': a_statements
            break;
  case 'b':
  case 'B': b_statements
            break;
  case 'c':
  case ‘C': c_statements
            break;
  default:  statements
            break;
}
```

break

- Оператор **break** используется для выхода из ветвления
- Операторы **case** аналогичны меткам; все операторы, следующие за **case**, выполняются до тех пор, пока не встретятся **break** или **return**
- **break** также применяется для выхода из циклов **while**, **for** и **do**

while

Оператор **while** используется для организации циклов с предусловием:

```
while(expression)
  statement    /* expression is true */
```

while

```
character = getchar();
while(   character == ' '
       || character == '\n'
       || character == '\t')
  ;  /* do nothing,
        skip white space
        characters */
```

for

Цикл **for** – цикл с инициализацией:

```
for(expression_1;expression_2;expression_3)
  statement
```

Это аналогично:

```
expression_1;
while(expression_2)
{
  statement;
  expression_3;
}
```

for

```
for(expression_1;expression_2;expression_3)
  statement

/* expression_1 - инициализация
   expression_2 – вычисляется в начале цикла
   если ПРАВДА, то выполняется оператор
   expression_3 - реинициализация
*/
```

for

```
for(i=0;i<n;i++)
  statement

for(i=MAX;i>0;i--)
  statement

void delay(int time)
{
  for(;time > 0;time--)
    NOP();
}

delay(20);
```

Бесконечный цикл

```
for(;;)      /* loops forever */
  statement

while(1)     /* loops forever, too */
  statement


void main(void)
{
  initialization();

  while(1)              /* the sad and boring */
    mainfunction(); /* life of an µC       */
}
```

do - while

do-while – цикл с постусловием, т.е. проверка условий производится после выполнения тела цикла. Если условие не выполняется, то следующего выполнения тела не происходит и программа выполняется дальше:

```
do
   statement
while(expression)  /* leave if expression is
false */
```

do - while

```
/* itoa: convert integer to characters */
void itoa(int n, char string[])
{ int i, sign;
  if((sign=n)<0)  /* record sign*/
    n=-n;         /* make n positive */
  i=0;
  /* generate digits in reverse order */
  do {
    string[i++] = n % 10 + '0'; /* get next */
  } while((n /= 10) > 0); /* delete it */
  if(sign < 0)
    string[i++] = '-';
  string[i] = '\0';
  reverse(string); /*foreign function used */
}
```

break - continue

- Оператор **break** вызывает безусловное прекращение выполнения цикла
- **continue** запускает следующую итерацию цикла **while**, **for** или **do**
- **continue** предназначен только для циклов, не для **switch**; если **switch** стоит в цикле, то **continue** вызовет следующую итерацию !!!

break - continue

```
for(time = value;time > 0;time--)
    NOP();

for(time = value;time > 0;time--)
    ; /* some optimizers do nothing */

for(time = value;time > 0;time--)
    continue;
time = value;
while(time-- > 0)
  continue;
```

goto и метки

- Оператор **`goto`** определен в C, но он не нужен
- Писать программы нужно без этого ужасного оператора :-)

1079 EPC
Программирование на C

Функции

Функции

- Функции разбивают большую вычислительную задачу на маленькие
- Программы могут использовать функции, написанные заранее
- Функции скрывают ненужные мелочи и позволяют видеть картину целиком
- Функции позволяют производить изменения в программе менее болезненно (особенно если они расположены в разных файлах)

Функции

- Функции позволяют решать проблему с программой сверху вниз
- Имеющиеся функции можно использовать для создания приложений снизу вверх

Функции

- Программа на С чаще всего состоит из нескольких функций
- Программа может состоять из нескольких файлов
- Исходные файлы программы могут компилироваться раздельно и собираться вместе только в самом конце процесса разработки ПО

Функции

Объявление функции имеет вид:

```
returntype functionname (argument
  declarations)
  {
    declarations and statements
  }
```

Некоторые части могут отсутствовать:

```
dummy(){}
```

Функция ничего не делает и ничего не возвращает :-)

Функции

- Основная программа превращается в набор описаний переменных и функций
- Обмен данными между функциями осуществляется посредством:
  - Передаваемых аргументов
  - Возвращаемых результатов
- Функции могут вызываться в любом порядке
- Программа может быть разбита на несколько файлов, функция должна быть описана в одном файле

Функции

- Если не указан тип возвращаемых данных, то по умолчанию считается **`int`**
- Оператор **`return`** предназначен для возврата результата работы функции
- Он имеет вид:

```
return expression;
```

или

```
return (expression);
```

- Результат может быть преобразован в нужный тип данных

Функции

- Вызывающая функция может проигнорировать возвращаемое значение
- Оператор **`return`** также может не иметь аргумента

```
return;
```

- Управление возвращается вызывающей функции также и при достижении конца вызываемой функции

Примеры

```
#include <stdio.h>
#define MAXLINE 100

main()
{
  char line[MAXLINE];
  while(getline(line, MAXLINE) > 0)
    if(index(line,"the")
      printf("%s",line);
}
/* to be continued ... */
```

Примеры

```
int getline(char s[], int limit)
{  int c, i;
   i = 0;

   while(--limit>0 && (c=getchar()) != EOF && c != '\n')
     s[i++] = c;

   if(c == '\n')
     s[i++] = c;

   s[i] = '\0';

   return(i);
   }
}
   /* to be continued ... */
```

Примеры

```
index(char s[], char t[])
/* return the position of t in s, -1 if not found */
{  int i, j, k;
   for(i = 0; s[i] != '\0';i++)
   {
     for(j=1, k=0; t[k] != '\0' && s[j] == t[k]; j++, k++)
       ;

     if(t[k] == '\0'
       return(i);
   }
   return(-1);
}
```

Примеры

```
#include <p24fxxxx.h>
#define PORTB_IO_MASK      0xFF00

/* we need to tell the compiler, that the two functions are of
   type void, because it assumes int */

void initialize(void);  /* prototypes */
void mainfunction(void);

int main(void)
{  initialize();

   while(1)
   {
     mainfunction();
   }
}
```

Примеры

```
void initialize(void)
{
  TRISB = PORTB_IO_MASK;
}

void mainfunction(void)
{
  NOP(); /* breakpoint possible for timing control ??*/
  LATBbits.LATB0 = ~LATBbits.LATB0;
}

/* see the bit toggle in MPLABs watch window J */
```

Несколько файлов

- Для более удобной работы с большой программой есть смысл разбивать ее на несколько файлов
- При линковке всего проекта в один исполняемый файл не обязательно компилировать все входящие в него файлы, достаточно компилировать только некомпелированные (функция Make в MPLAB IDE)

Несколько файлов

- Программа разбивается на три файла – основная программа, инициализация и основной алгоритм
  - main.c
  - initialize.c
  - mainfunction.c

Несколько файлов

```
/* content of file main.c */
#include <p24fxxxx.h>

/* we need to tell the compiler, that the two functions are of
   type void, because it assumes int */

#include "initialize.h"
#include "mainfunction.h"

int main(void)
{  initialize();

   while(1)
   {
     mainfunction();
   }
}
```

Несколько файлов

```
/* content of file initialize.c */
#include <p24fxxxx.h>
#include "initialize.h"
void initialize(void)
{
  TRISB = TRISB_MASK;
}

/* content of file initialize.h */
#ifndef INIT_FIRST
#define INIT_FIRST
#define TRISB_MASK  0xFF00

void initialize(void);
#endif
```

Несколько файлов

```
/* content of file mainfunction.c */
#include <p24fxxxx.h>
#include "mainfunction.h"
void mainfunction(void)
{
  NOP(); /* breakpoint possible for timing control ??*/
  LATBbits.LATB0 = ~LATBbits.LATB0;
}

/* content of file mainfunction.h */
#ifndef MAIN_FNCTN_FIRST
#define MAIN_FNCTN_FIRST

void mainfunction(void);

#endif
```

1079 EPC
Программирование на C

Массивы и указатели

Массивы

- Массив – группа объектов одного типа
- Массивы могут быть
  - Одномерные
    - строки
  - Многомерные
- Объекты располагаются в памяти линейно
- Для описания массива после идентификатора ставятся квадратные скобки [ ]
- Имя массива адресует его первый элемент в памяти

Массивы

- Размер массива указывается в скобках []
- Примеры

```
char s;          /* 1 character */
char sf[20];     /* 20 characters */
int i;           /* 1 integer */
int iff[100]; /* 100 integers
                 => 200 bytes ! */
```

Массивы

- Для доступа к элементам массива используется имя и номер элемента в скобках:

```
int j, k;
j = 5;

sf[8] = 'a';
sf[j] = sf[19];
k = iff[j];
```

- Индексы идут от 0 до MAX

Массивы

- Первый элемент массива имеет индекс нуль
- Элементы могут иметь любой тип данных
  - char
  - int
  - long
  - float
- Но все элементы одного типа

Массивы

- Массивы с классом памяти **`auto`** не инициализируются; задавать начальные значения можно только глобальным и внешним массивам
- Массивы не могут иметь класс памяти **`register`**
- Элементы неинициализированных массивов инициируются нулями
- Начальные значения можно задать в фигурных скобках **`{}`**

Массивы

- Примеры инициализации массивов

```
int iff[10] = {1,2,3,4,5,6,7,8,9,10};

int jf[5] = {0,1,3};
/* only the first three initialized
   explicitly, all others = 0*/

int kf[] = {1,3,5,7,11,13};
/* size of array specified through
   number of values */
```

Массивы

```
/* filling an array with the sum of natural
numbers up to this field */

#define MAX 100;
int sum[MAX], i, j;
for (i=0, j=0; i < MAX; i++)
{
  j = j + i + 1;
  sum[i] = j;
}
/* doing the same without using the first
element */

#define MAX 101;
for (i=1,j=0; i < MAX; i++)
{
  j = j + i + 1;
  sum[i] = j;
}
```

Строки

- Строки – специальный тип одномерного массива
- Строка – группа данных типа **`char`**
- Можно инициализировать даже для класса **`auto`**
- Этот массив всегда заканчивается **`\0`**
- Поэтому в памяти занимают на один байт больше

Строки

- Пример инициализации строки:

```
char s[] =
       {'s','t','r','i','n','g','\0'};

char s[] = "string";
/* '\0' appended automaticly;
   size of array is 7 !!! */

char s[10] = "string";
/* rest of array filled with zeros */
```

Строки

- Учитывая эти особенности, строку можно считать массивом
- К элементам строк можно обращаться как к элементам массива:

s[0] имеет значение 's'

s[3] имеет значение 'i'

Массивы

```
/* two versions filling an array with square
   numbers */

#define MAX 100;
int mul[MAX], i;

for (i=0; i<MAX; i++)
  mul[i] = (i + 1) * (i + 1);

/*
 */

#define MAX 100;
 int mul[MAX], i, j;
for (i=0,j=0; i < MAX; i++, j++ )
 mul[i] = j * j;
```

Многомерные массивы

- Многомерные массивы имеют несколько квадратных скобок [ ]
- Многомерный массив – массив массивов

Многомерные массивы

```
/* declaration may be extern or in the
calling function */

int md[5] [10];  /* extern (global) */
funct(md,5);

/* in the function */

void funct(int mdf[][10], int howmany)
{
  int i, z;
  for (i=0; i<howmany; i++)
    for (j=0; j<10; j++)
      mdf[i][j] = i + j;
}
/* with howmany, we forward the actual size
of the first array */
```

Многомерные массивы

- Для многомерного массива также можно задавать начальные значения:

```
static int md[][] =
{
 {0,1,2,3,4,5, 6, 7, 8, 9}
,{1,2,3,4,5,6, 7, 8, 9,10}
,{2,3,4,5,6,7, 8, 9,10,11}
,{3,4,5,6,7,8, 9,10,11,12}
,{4,5,6,7,8,9,10,11,12,13}
};
```

Многомерные массивы

- Пример:

```
const char multistring[][40] =
{
 {"Error !!!"}
,{"Well done !"}
,{"Temperature too high"}
,{"The time is: "}
,{"… and the winner is: "}
};
```

Многомерные массивы

- Пример:

```
static int day_tab[2][13]=
{{0,31,28,31,30,31,30,31,31,30,31,30,31}
,{0,31,29,31,30,31,30,31,31,30,31,30,31}
};

/* calculate the day of the year from month
and day */

day_of_year(int year, int month, int day)
{ int i, leap;
  leap = year%4 == 0 && year%100 != 0
                     || year%400 == 0;
  for(i=1; i<month; i++)
    day = day + day_tab[leap][i];
  return(day);
}
```

Многомерные массивы

- Пример с указателями !?:

```
/* calculate month and day from the day of
the year */

month_day(int year, int yearday, int
*pmonth, int *pday)
{ int i, leap;
  leap = year%4 == 0 && year%100 != 0
                     || year%400 == 0;
  for(i=1; yearday > day_tab[leap][i]; i++)
    yearday = yearday - day_tab[leap][i];
  *pmonth = i;
  *pday = yearday;
}
```

А где же указатели?

Указатели

- Указатель – переменная, содержащая адрес переменной !!!
- Указатели используются для доступа к массивам и структурам
- Указатели совместно с операторами **`goto`** – прекрасный путь к созданию абсолютно непонятной программы :-)

Указатели

- Указатели и адреса
- Арифметика указателей
- Указатели и массивы
- Функции и указатели

Указатели

- Для получения адреса переменной используется оператор **&**

```
pVar = &anyVar
```

pVar содержит адрес переменной **anyVar**

Указатели

- Унарный оператор * - оператор косвенного доступа
- Если этот оператор применяется к указателю, то происходит доступ к объекту, на который указатель указывает

Указатели

- Примеры:

```
pV= &anyVar; /* pV points to anyVar */
anyVar = 1;  /* anyVar becomes 1 */
W = *pV;     /* W becomes 1 */
*pV = 0;     /* anyVar becomes 0 */
```

Указатели

- Объявление указателей

```
int *pVar1;  /* this pointer points to
                an integer Variable */
char *pVar2; /* this pointer points to
                a char */
long *pVar3; /* this pointer points to
                a long integer */
```

Указатели

- Указатели указывают на определенные типы данных
- Указатели объявляются с использованием оператора *
- Арифметика указателей знает, как с ними работать

Арифметика указателей

- Если **p** – указатель, а **n** – целое число, то **p+n** указывает на n-й элемент после элемента, на который указывает **p**
- **n** не просто прибавляется к **p**: **n** умножается на размер типа данных, а полученное значение прибавляется к **p**
- Тип указываемых данных известен из объявления указателя **p**

Арифметика указателей

- Т.к. адрес – целое число, все операции с указателями - целочисленные
- Указатели можно инкрементировать и декрементировать
- Указатели могут вычитаться друг из друга. Это имеет смысл лишь в том случае, если они указывают на одну группу объектов

Арифметика указателей

- Указатели можно сравнивать друг с другом
  `>, >0, <, <=, !=, ==`
- Опять же, это имеет смысл при указании на одну группу

Арифметика указателей

- Никакие другие операции типа умножение, деление, сдвиги, и т.д. не разрешены !!!

Указатели и массивы

- Массив – группа объектов одного типа
- Имя массива – его начальный адрес
- Т.к. начальный адрес зафиксирован, то имя массива – константа

Указатели и массивы

- Доступ к объектам массива может осуществляться посредством указателей

Указатели и адреса

- Указатели (***pVar**) могут применяться везде, где применяются переменные, на которые они указывают

```
y = *px + 10;
y = *px + *px;
printf("%d\n", *px);
*px = 0;
py = px;
      /* given py is a pointer, too */
```

Указатели и адреса

- Приоритет операций над указателями

```
y = *px + 1; /* y =(content of px) + 1 */
y = *(px+1); /* y = cntnt of Address px+1 */
*px += 1; /* cntnt of px = cntnt of px + 1*/
(*px)++;  /* increment content of px */
*px++;    /* *(px); px = px + 1;  */
*++px;    /* pre-increment; px = px + 1;
             content of address px + 1;
```

Указатели и адреса

- Примеры

```
int a[10];      /* array of integer, 10 elements */
int *pa, y, i; /* pointer to an integer object */
pa = &a[0];     /* pa = address of first element */
pa = a;         /* same as above,
                   because a is address of array */
y = *pa;        /* y = first element */
y = *(pa + 1); /* y = second element */
y = *px++;      /* y = content of px; px = px + 1; */
i = 0;
y = a[0];       /* y = first element */
y = a[i];       /* dito */
Y = a[++i];     /* ??? */
```

Указатели и адреса

- Еще примеры

```
a[i] is equivalent to *(a+i)
&a[i]          "          (a+i)
pa[i ]         "          *(pa+i)
```

Если не хотите, то можете не использовать указатели

Но часто они делают жизнь проще ;-)

Указатели и адреса

- Разрешенные и запрещенные операции

```
pa = a;  /* left side becomes constant */
pa++;    /* increment pointer variable */

a = pa;   /* ??? Address = variable ??? */
a++;      /* ??? Increment a constant ??? */
pa = &a;  /* ??? pa = address of address ??? */
```

Указатели и адреса

- Пример

```
strlen(char *s)  /* get the
length of a string */
{
  int n;
  for(n=0; *s != '\0'; s++)
    n++;
  return(n);
}
```

Указатели и адреса

- 4 способа скопировать строку

```
strcpy(char s[], char t[])  /* 1.: using arrays */
{
   int n;
   i = 0;
   while((s[i] = t[i]) != '\0' )
     i++;
}

strcpy(char *s, char *t)  /* 2.: using pointer */
{
   while((*s = *t) != '\0' )
   {
     s++;
     t++;
   }
}
```

Указатели и адреса

- 4 способа скопировать строку

```
strcpy(char *s, char *t)  /* 3.: using pointer */
{
   while((*s++ = *t++) != '\0' );
}

strcpy(char *s, char *t)  /* 4.: using pointer */
{
   while(*s++ = *t++);     /* confused ??? */
}
```

1079 EPC
Программирование на C

Структуры

Структуры

- Массивы содержат объекты одного типа
- Структуры группируют данные разных типов
- Структуры используются для работы со сложными типами данных, например, записи в БД, пакеты TCP/IP, кадры CAN и т.д.

Структуры

- Для объявления структуры используется ключевое слово **`struct`**:

```
struct structure_name
{
  structure_member(s)
}variable_name(s) initialization
```

Структуры

- Тег структуры может использоваться для объявления данных, как специальный тип
- Элементы структуры описываются как обычные данные
- Имена структуры и элементов может совпадать с другими переменными; компилятор знает, как их различать
- Как только структуре присвоено имя, под нее выделяется область в памяти

Пример

```
struct date
{ int day;
  int month;
  int year;
  int day_of_year;
  char name_of_month[4];
};  /* defines the prototype structure */

struct date
{ int day;
  int month;
  int year;
  int day_of_year;
  char name_of_month[4];
}birth_date, today;
  /* creates the structures birth_date
     and today of type date */
```

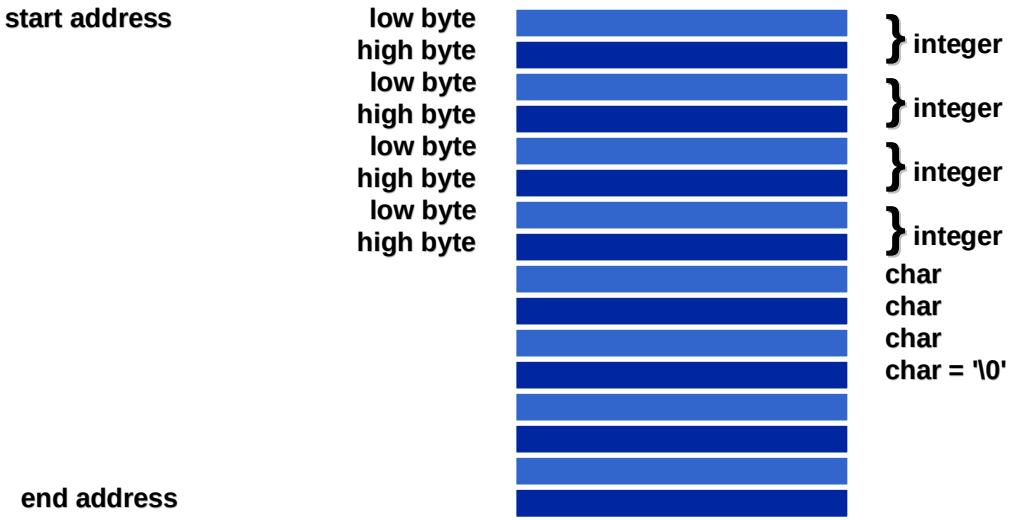
Пример

```
struct date birth_date, today;
  /* also creates the structures birth_date
     and today of type date */

struct date today =
  { 24
  , 10
  , 2006
  , 0
  , “oct"
  };
/* creates and initializes the structures birth_date
     and today of type date */

extern structure date today;
/* we should use a structure today of type date
   defined somewhere else */
```

Структуры в памяти



Структуры

- Структуры могут содержать структуры, но не рекурсивно

```
struct customer
{ char name[NAME_LENGTH];
  char address[ADDR_LENGTH];
  int customer_number;
  struct date delivery_date;
  struct date billing_date;
  struct date payment_date;
};

struct customer customer_1, customer_2, … ;
struct customer arrayCustomer[CUST_MAX];
```

Структуры

- Более приближенный к микроконтроллерам пример:

```
struct message
{ int target_address;
  int target_command;
  int msg_data_length;
  char data_buffer[BUFFER_MAX];
  int crc;
};
```

Доступ к данным структур

- Для структур определены две операции
  - Доступ к элементам
    - Прямой
    - Косвенный
  - Получение начального адреса структуры

Доступ к данным структур

- Получение начального адреса производится с помощью оператора **&**, как обычно
- Прямой доступ к элементам структуры осуществляется с помощью оператора **.**

```
structure_variable.component
```

Доступ к данным структур

- Пример прямого доступа:

```
birth_date.year
today.day
today.name_of_month
customer_1.customer_number
arrayCustomers[3].name
customer_2.delivery_date.month
```

Доступ к данным структур

- Для косвенного доступа к данным структуры необходимо иметь указатель на структуру (оператор **&**) и использовать оператор **->**

```
pointer_var_to_structure -> component
```

Доступ к данным структур

- Пример косвенного доступа:

```
struct date *pdat;
struct customer *pcust;
pdat  = today;
pcust = arrayCustomers[4];

pdat  -> day
pdat  -> name_of_month[1]
pcust -> address
puts -> billing_date.month
```

typedef

- **`typedef`** позволяет создать новое имя для типа данных, но не новый тип данных
- **`typedef`** похож **`#define`**
- **`typedef`** используется для устранения проблем несовместимости типов данных при переносе приложений

typedef

- Например, тип int не на всех системах одинаковый
- Для устранения проблем можно использовать

```
typedef uint8_t  unsigned char;
typedef int8_t   signed char;
typedef uint16_t unsigned int;
typedef int16_t  signed int;
typedef uint32_t unsigned long;
typedef int32_t  signed long;
```

bits

- В С нет типа данных БИТ!!!
- Работа с битами осуществляется только посредством битовых операций **&, |, ^, ~, <<, >>**
- Но есть более простой способ

bits

- C предлагает альтернативный способ работы с битами, используя структуры, где каждый элемент - бит
- КАК???

```
struct eightbits
{  unsigned int bit0 : 1;
   unsigned int bit1 : 1;
   unsigned int bit2 : 1;
   unsigned int bit3 : 1;
   unsigned int bit4 : 1;
   unsigned int bit5 : 1;
   unsigned int bit6 : 1;
   unsigned int bit7 : 1;
 };
```

bits

- Тип данных битового поля – int
- Число после двоеточия – количество бит для каждой переменной

```
struct other_eightbits
{  unsigned int bit0   : 1;
   unsigned int bit1   : 1;
   unsigned int unused : 4;
   unsigned int bit3   : 1;
   unsigned int bit7   : 1;
 };
```

bits

- Можно указывать число бит без имени элемента
- Если указано число битов 0, то производится расширение на следующую ячейку памяти (**int**)

```
struct
{  unsigned int bit0    : 1;
   unsigned int bit1    : 1;
                        : 4;
   unsigned int bit3    : 1;
   unsigned int bit7    : 1;
                        : 0;
 }bits;
```

bits

- Доступ к битам осуществляются также, как к элементам структуры

```
struct eightbits errorFlags
                  , semaphores ;

errorFlags.bit0 = 0; /* {0, 1} */

#define semaphoreFifo semaphores.bit1;

if(semaphoreFifo == 1)
  LATBbits.LATB0 = ~LATBbits.LATB0;
```

КОНЕЦ

Спасибо за внимание!