Budapesti Műszaki és Gazdaságtudományi Egyetem

Közlekedés- és Járműirányítási Tanszék

Mechatronics and Microcontrollers

Szilárd Aradi PhD

Refresh of C

About the C programming language

- The C programming language is developed by Dennis M Ritchie in the beginning of the 70s
- One of the most popular generic programming languages.
- And the base of many other languages

Language structure

- The language is Case Sensitive

- Identifiers
  - English alphabet (’a’..’z’ , ’A’..’Z’)
  - Numbers (’0’..’9’)
  - ’_’ underscore character
  - Must not start with a number

  - Good: **Variable, _calulate, _5,a_, __**
  - Not good: **változó, 32szam,**

Language structure

- Literals
  - Given values, always has a type
- Character arrays
  - e.g. **”string with a new line character\n”**
- Keywords and flow control
- Operators
  - =, +, - stb.
- Other separators

Program structure

#include

#define

declarations:

        constants

        variables

        functions

int main(void){}

Function definitions

Types

char	One byte	0..255
short , or short int	The 16-bit short int data type	$-2^{15}..2^{15}-1$
int, or long int	The 32-bit int data type	$-2^{31}..2^{31}-1$
float	The float data type is the smallest of the three floating point types (32 bit)	
double	Double precision floating data type (64 bit)	

Type modifiers:

unsigned	for example: unsigned int is :0.. $2^{32}-1$
signed	for example: signed char is :-128..127
short	
long	for example long long int is 64 bit

Important! Atmel AVR uses fixed-point arithmetic, thus integer types are preferred.

Constants

Constants	
1234	int constant
1234L	long
1234UL	unsigned long
0x1f2	hexa int
0x1f2UL	hexa unsigned long
1234.5	double
1234.5f	float
’c’	char
”szoveg”	char[] (string)

Escape sequences	
\a	bell
\b	backspace
\f	formfeed
\n	new line
\r	carriage return
\t	horizontal tab, HTAB
\v	vertical tab, VTAB
\\	backlash
\?	?
\'	’
\"	”
\ooo	octal
\xhh	hexadecimal

Variable declarations

Declaration:

```
int i;
float f,g;
char c;
```

Initialization:

```
char c=a;
char s[]=”string”;
const int j=12;
```

Assignment operator

- *Assignment:*
- `i=2;`

- Compound assignment operators perform an operation involving both the left and right operands, and then assign the resulting expression to the left operand.
- `i=i+2;`
  Compound assignment operators format*:*
- `i+=2;`
  Assignment has a return value

Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression x % y produces the remainder when x is divided by y, and thus is zero when y divides x exactly.

The % operator cannot be applied to float or double. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

The binary + and - operators have the same precedence, which is lower than the precedence of *, /, and %, which is in turn lower than unary + and -. Arithmetic operators associate left to right

Relational and Logical Operators

The relational operators are: >, >=, <, <=.
They all have the same precedence. Just below them in precedence are the equality operators: ==, !=

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false

Expressions connected by && or II are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.

The unary negation operator I converts a non-zero operand into 0, and a zero operand into 1

Increment and Decrement Operators

C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1.

The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++n), or postfix (after the variable: n++). In both cases, the effect is to increment n. But the expression +-n increments n before its value is used, while n« + increments n after its value has been used. This means that in a context where the value is being used, not just the effect, ++n and n++ are different.

If n is 5, then

```
x = n++;
```

sets x to 5, but

```
x = ++n;
```

sets x to 6. In both cases, n becomes 6

Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, char, short, int, and long, whether signed or unsigned

&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

If-else

- The else is associating with the closest previous else-less if:

```
if (n>0)
    if (a>b)
     z=a;
    else
     z=b;
```

Switch example

```
char c;
switch (c){
 case 'a': case 'e': case 'i': case 'o':
case 'u':
   printf("maganhangzo"); break;
 case ' ':
   printf("Space"); break;
default: printf ("Egyik sem");
}
```

While example

```
char c=0;
    while(++c<10)
      {


      printf("%d ",c);
      }
```

Result: 1 2 3 4 5 6 7 8 9

While example with continue

```
char c=0;
    while(++c<10)
      {
      if (c==3) continue;

      printf("%d ",c);
      }
```

Result: 1 2 4 5 6 7 8 9

While example with break

```
char c=0;
    while(++c<10)
      {
      if (c==3) continue;
      if (c==6) break;
      printf("%d ",c);
      }
```

Result: 1 2 4 5

The C Preprocessor

- C provides certain language facilities by means of a preprocessor, which is conceptually a separate first step in compilation:
  - #include, to include the contents of a file during compilation
  - #define, to replace a token by an arbitrary sequence of characters
  - conditional compilation and
  - macros with arguments

Macro Substitution example

```
#define EOS '\0'
#define TRUE 1
#define boole int
int main(void)
{
  boole vege=0;
  char s[]="hello";
  int i=0;
  do
  {
     if (s[++i]==EOS)
vege=TRUE;
  }
  while (!vege);
  printf("%d",i);
  getchar();
}
```

```
int main(void)
{
  int vege=0;
  char s[]="hello";
  int i=0;
  do
  {
     if (s[++i]==’\0’)
vege=1;
  }
  while (!vege);
  printf("%d",i);
  getchar();
}
```

Problems, Threats

```
#define abs(x) ((x) < 0 ? (-(x)) : (x))
#define min(a, b) ((a) < (b) ? (a) : (b))

int i=-5; int j=-6;
    printf("%d",min(abs(i),abs(j)));
```

The previously defined macro uses the „parameters" twice, hence:

```
min(i++,j++)
```

substituted with:

```
((i++) < (j++) ? (i++) : (j++))
```

therefore one of the variables increments twice

Wrong usage

Further examples for wrong usage:

```
#define MARGIN 2
#define WIDTH 8
#define HEIGHT 10
#define FULLWIDTH WIDTH+MARGIN
#define AREA FULLWIDTH*HEIGHT
```

Wrong result, can be fixed with proper bracketing.

also,

```
#define square(x) x*x
square(i+1);
```

substituted with: `i+1*i+1`

4.11.2. Fix

Fixing with parenthesis:

```
#define MARGIN 2
#define WIDTH 8
#define HEIGHT 10
#define FULLWIDTH (WIDTH+MARGIN)
#define AREA FULLWIDTH*HEIGHT
```

also,

```
#define square(x) (x)*(x)
square(i+1);
```

substituted with: `(i+1)*(i+1)`

Arrays

```
int t[10];
// without intialization

int t[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// giving values to all elements

int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// same, the number of elements define array length

int t[10] = {1, 2, 3, 4};
// only the first 4 elements are initialized
```

5.7 Multi-dimensional Arrays

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

int t[3][4] = {0, 1, 2, 3,
               4, 5, 6, 7,
               8, 9,10,11};

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

Character Arrays

```
char s[200] = "Hello";
char s[200] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The two definitions are the same
Strings are terminated with the 0 byte (('\0') charater

```
char s[] = "Hello";
```

Array length will be set to 5+1 because of the 0

Basics of Structures

```c
struct pont{
    int x;
    int y; };
```

- This struct declaration defines a type, where the **point** is the structure tag and **x** and **y** are the members.
- Example for shorthand usage:

```c
struct point pt;
```

- Defines a point type variable pt. Initialization acan be done by listing member values:

```c
struct point pt={ 320 , 200 };
printf("%d, %d", pt.x, pt.y);
```

Functions Introduction

- After a while, the single `main(){}` program will be too large and complex.
- There are code sections and algorithms that need to be used at multiple code locations.
- Too complex algorithms reduce the readability of the program
- There is a need to structure our program
- Functions break large computing tasks into smaller ones, and enable people to build on what others have done instead of starting over from scratch. Appropriate functions hide details of operation from parts of the program that don't need to know about them, thus clarifying the whole, and easing the pain of making changes

Pointers

```c
int i=10;
int *ip;
ip=&i;
printf("%d\n",*ip);
i++;
printf("%d\n",*ip);
```

Console:

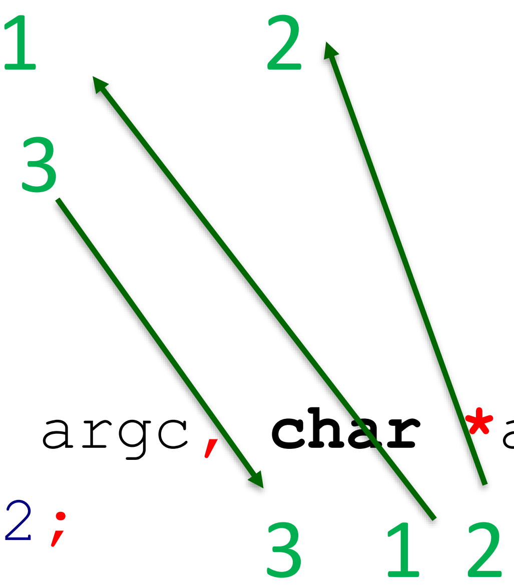
```
10
11
```

Memory address (byte num)	Variable	Value (int)
1000		?
1004	i	11
1008	ip	1004
1012		?
----	*ip	11
	&i	1004

Function example

```
int add(int a, int b)
              1      2
{
 return a+b; 3
}

int main(int argc, char *argv[])
{ int x=1,y=2;     3  1 2
  printf("x+y=%d",add(x,y));
  return 0;
}
```



Call by value example

```c
void swap(int a, int b)
                 1      2
{
   int c;
   c=a;a=b;b=c;
   return;
}
int main(int argc, char *argv[])
{
  int x=1,y=2;
  swap(x,y);     1  2
  printf("x:%d y:%d",x,y);
  return 0;
}
```

a=2, b=1, just for the copied variables

„x:1 y:2” swap has no effect outside

Pointers and Function Arguments

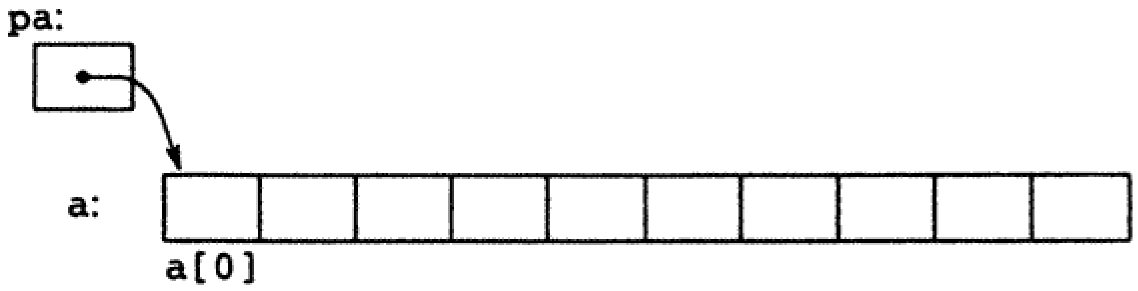
Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above only swaps copies of a and b. The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

```
swap(&a, &b);

void swap(int *px, int *py) /* Right   */
{ int temp=*px;
  *px = *py;
  *py = temp;
 }
```

5.3. Pointers and Arrays

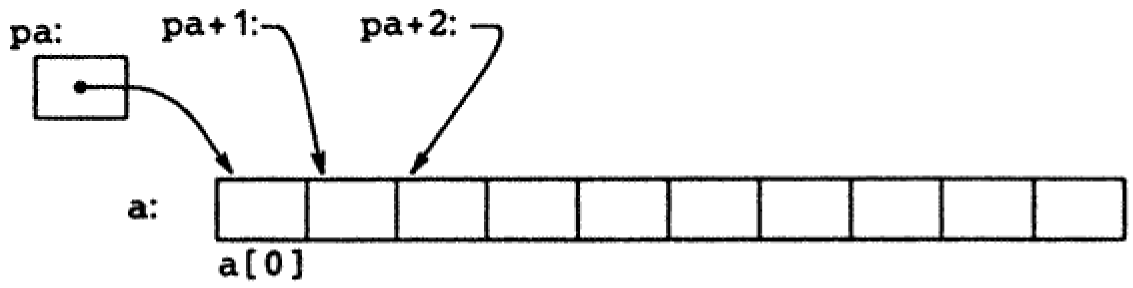
- The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as `int *pa;` then the assignment `pa = &a[0]` sets pa to point to element zero of a; that is, pa contains the address of `a[0]`.
- Now the assignment `x=*pa;` will copy the contents of `a[0]` into x.



5.3. Pointers and Arrays

Thus, if pa points to `a[0]`, `*(pa+1)` refers to the contents of `a[1]`,

`pa+i` is the address of `a[i]`, and `*(pa+i)` is the contents of `a[i]`.



Argument passing with arrays

Same as with pointers.

```
                 „hamu”
void confuse(char s[]) {
   s[0]='m'; return; „mamu”
}
int main(int argc, char *argv[]) {
   char s[]="hamu";
   printf("%s\n",s);  „hamu”
   confuse(s);
   printf("%s\n",s);  „mamu”
   return;
}
```