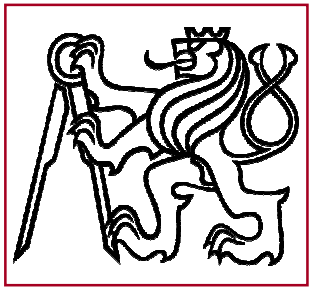
Y35PES

Programming for Embedded Systems

Ondřej Špinka, Pavel Němeček
spinkao@fel.cvut.cz nemecp1@fel.cvut.cz
http://dce.felk.cvut.cz/pes

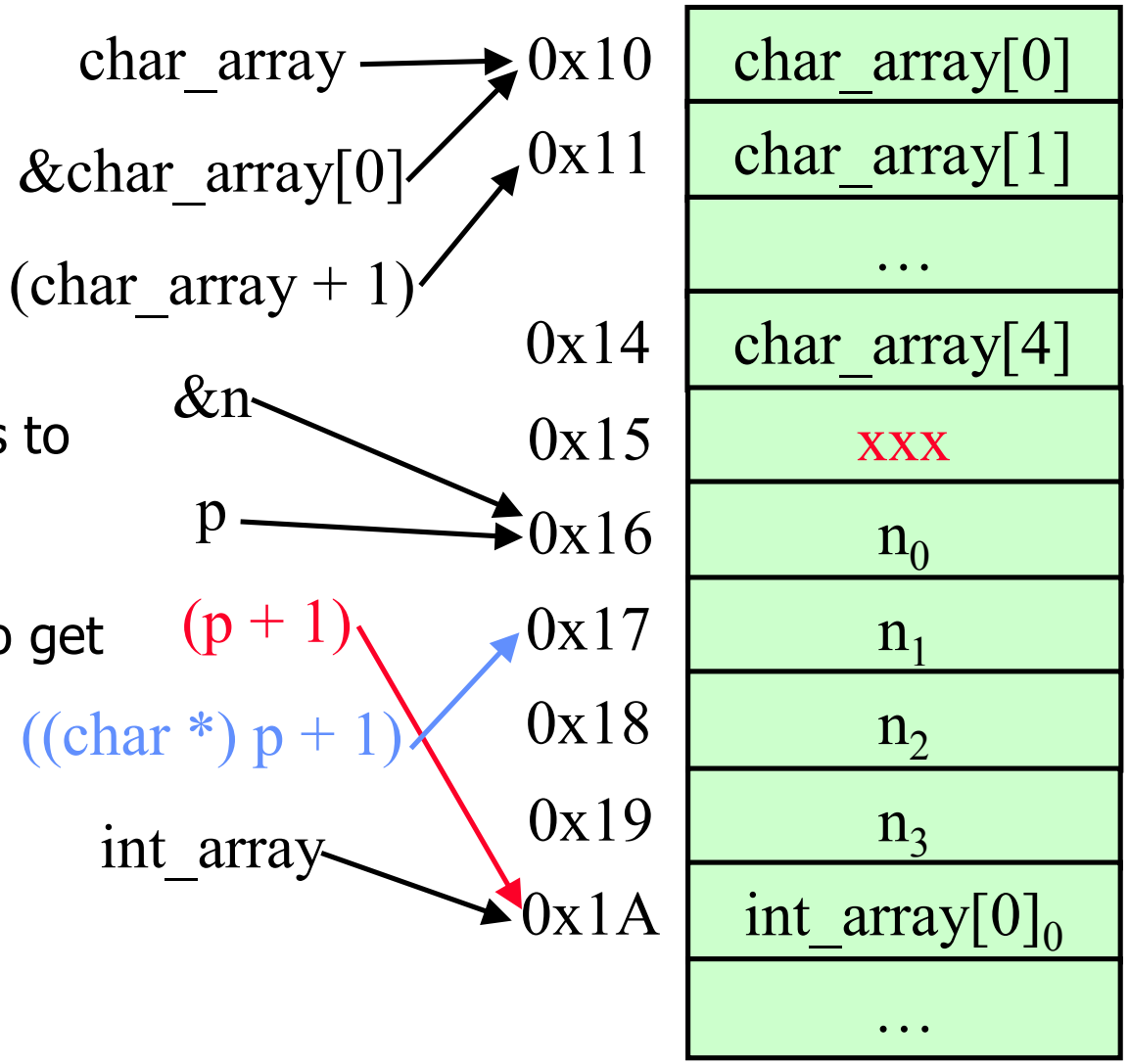


Arrays and Pointers in C (1)

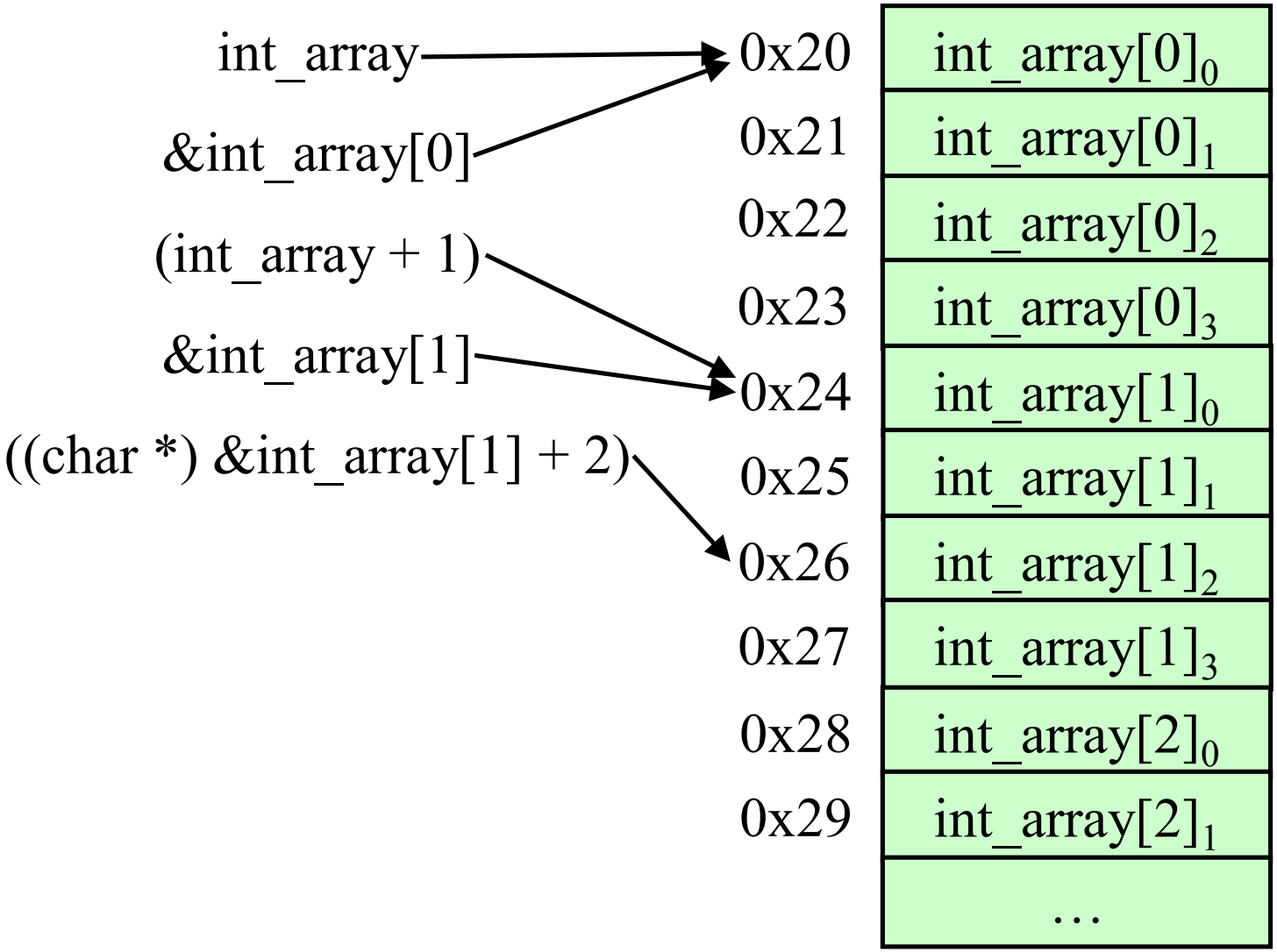
```
char char_array[5];
int n;
int *p = &n;
int int_array[10];
```

* - **dereference** – serves to get an object by its address

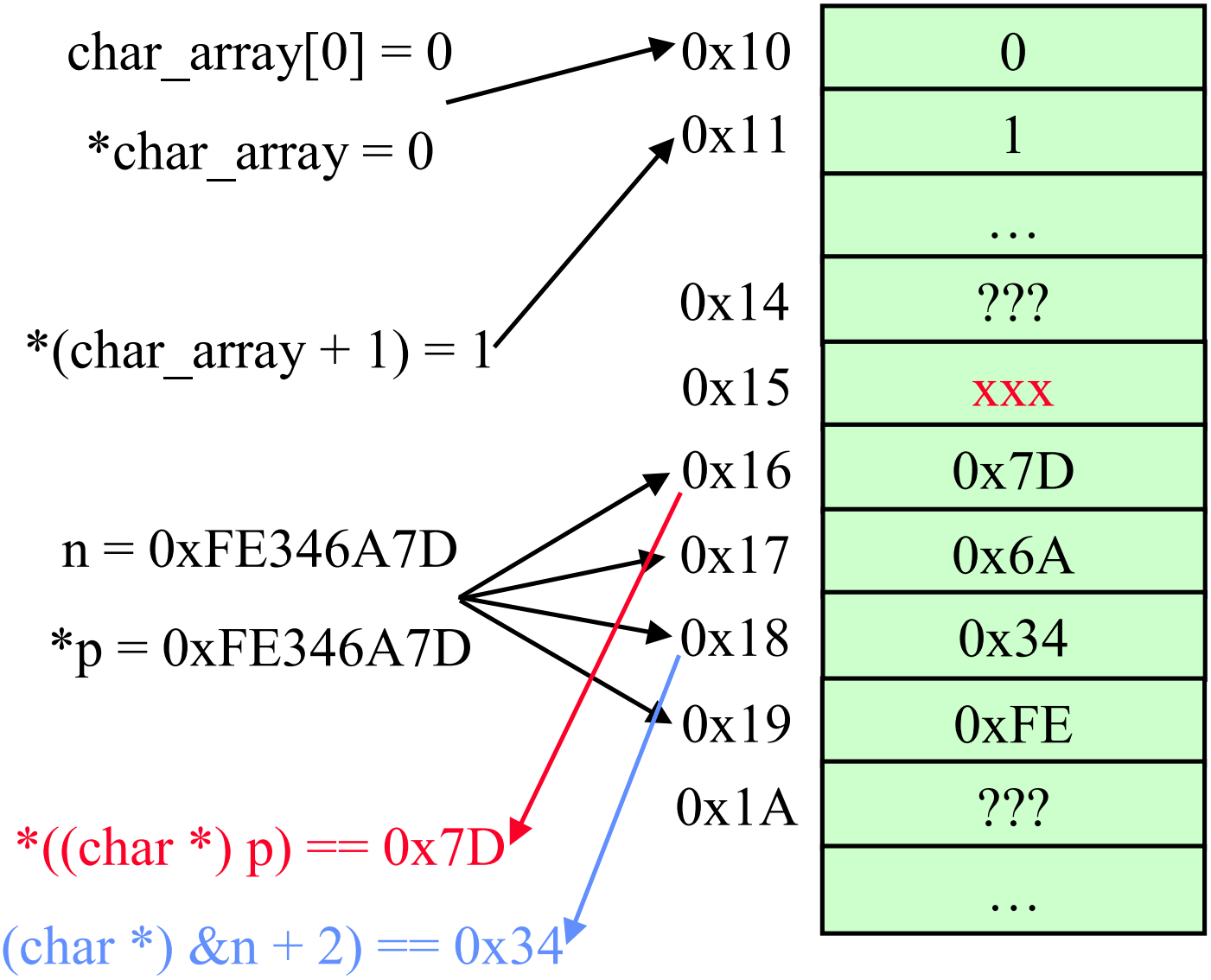
& - **reference** – serves to get address of an object



Arrays and Pointers in C (2)



Arrays and Pointers in C (3)



Arrays and Pointers in C (4)

```
int i;
int *p = &i;
int *parray;
int my_array[30];
```

Following statements are equivalent:

```
i = i + 1;
i++;
*p = *p + 1;
(*p)++;
p[0]++;

p++;
p = (int *) ( (char *)p + sizeof(int) );
```

```
parray = my_array;
parray = &my_array[0];

for ( int i = 0; i < 30; i++ ) my_array[i]++;
for ( int i = 0; i < 30; i++ ) (*parray++)++;
```

Arrays and Pointers in C (5)

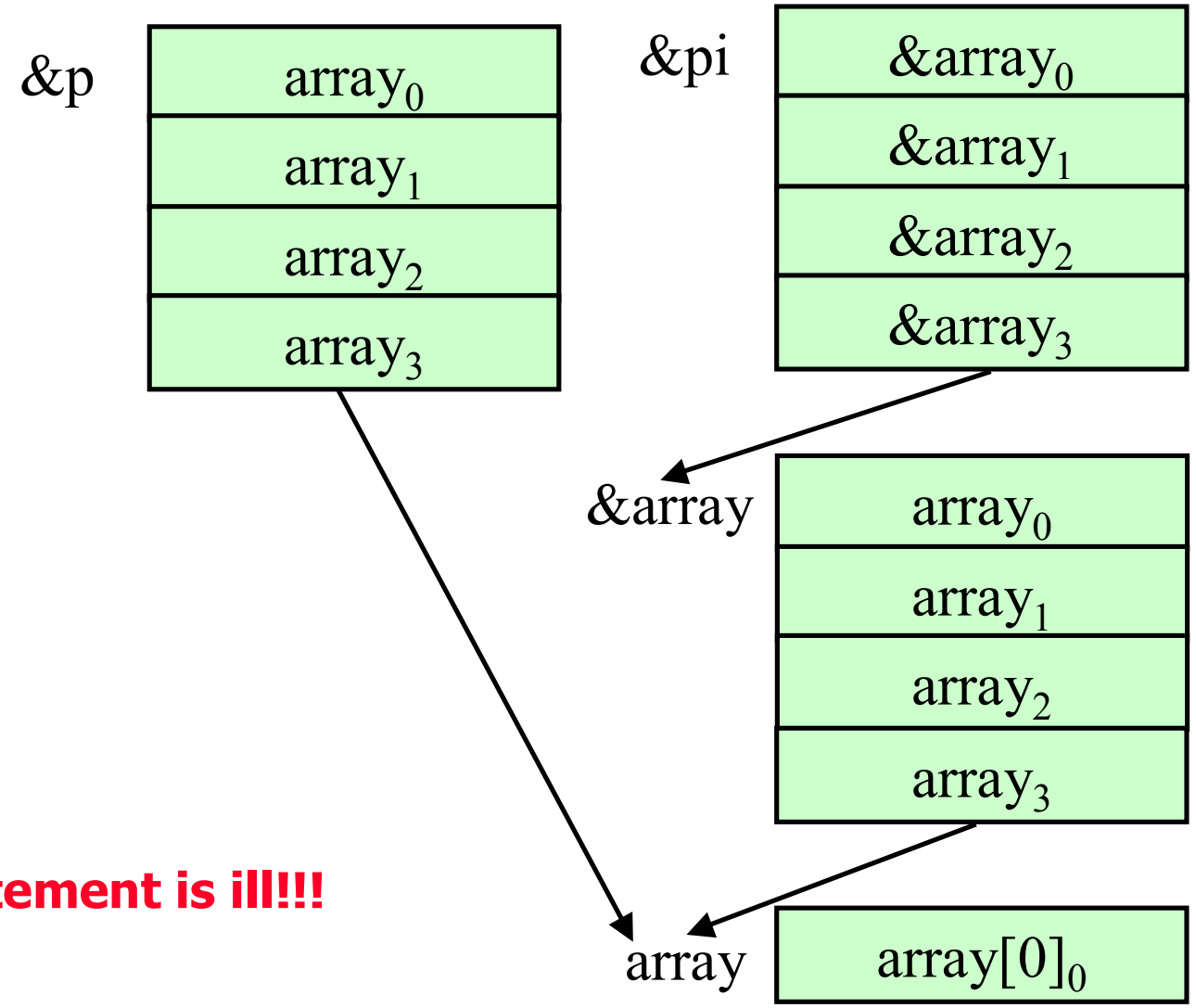
```
void *p;
int **pi;
int array[30];

p = (void *) array;
pi = &array;

array[0] == **pi;
array[0] == *p;
array[0] == *pi[0];
array[0] == p[0];
```

Beware!!! This statement is ill!!!

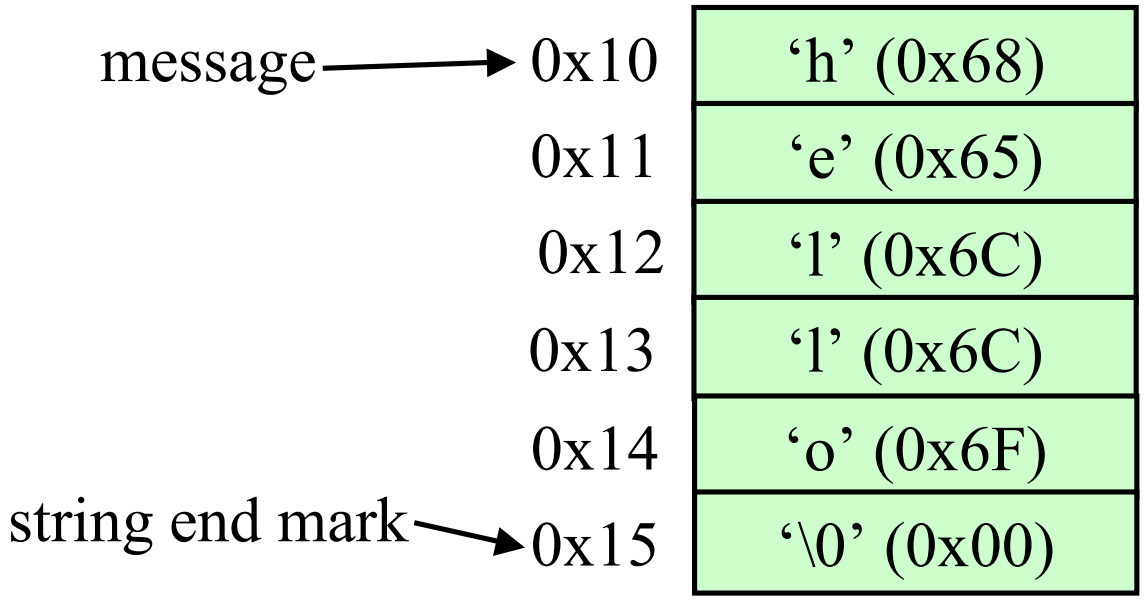
```
int *p;
*p = 10;
```



Strings in C

```
char *message = 'hello';
char *aux;
char temp[6];

.
.
.
```




```
printf ( "%s\n", message );
aux = message;
printf ( "%s\n", aux );
strcpy ( temp, aux );
printf ( "%s\n", temp );
```

Beware!!! Those statements are ill!!!

```
temp = message;
aux = 'hello';
strcpy (aux, message);
```

Scope of a Variable (1)

Scope of a variable determines the region of a program within which the variable is accessible. The scope of a variable is determined by its declaration placement and always starts at the point of declaration. Basically, there are four types of scope:

- **File scope** – A variable that is declared outside of all functional blocks has a scope that terminates at the end of the translation unit (**global variable**).
- **Block scope** – A variable that is declared within a block {} of code has a scope terminating at the end of the block (**local variable**).
- **Function scope** – Only labels have this scope. A label can be accessed from anywhere within the block in which it is declared, regardless of the point of declaration (**local variable**).
- **Function prototype scope** – Terminates at the end of the function declarator (**local variable**).

Scope of a Variable (2)

- **Global variable** is created at the program start-up in the RAM and would be maintained here until the program ending.
- **Local variable** is dynamically created on the **stack** whenever the respective functional block is entered, unless declared as **static**. In that case it would be created and maintained in the memory (such as a global variable would be), but it would be inaccessible outside of its scope.
- **Function parameters** are always treated as local variables (parameters are valid within the function block).
- **Return value** of a function is given to the calling function using stack.

Scope of a Variable (3)

If there is a block scope variable with the same name as a file scope variable, the file scope variable (global) is **overshadowed** by the block scope variable (local) in the respective block.

If it is desirable to share a global variable between multiple files, it has to be declared as the type **extern**.

Scope of a Variable (4)

```
int a = 0;
int b = 1;

int myfunc ( int a, int b ) {
    return a + b;
}

void main ( ) {
    int a = 2;
    printf  ( "myfunc = %d\n", myfunc ( a, b ) );
}
```

Scope of a Variable (5)

```
int a = 0;
int b = 1;

int myfunc ( int a, int b ) {
    static int result = 0;
    result += a + b;
    return result;
}

void main ( ) {
    myfunc ( a, b );
    int a = 2;
    printf  ( " myfunc = %d\n", myfunc ( a, b ) );
}
```

Passing Parameters to a Function (1)

The parameters of a function are local variables, with scope ranging from beginning to the end of the function block. The parameters are passed via stack. A function may return one value of arbitrary type, which is passed to the calling function via stack:

```
int myfunc ( int a, int b ) {
    return a + b;
}

void main ( ) {
    int a = 2;
    printf  ( "myfunc = %d\n", myfunc ( a, b ) );
}
```

Passing Parameters to a Function (2)

It is sometimes convenient to pass parameters to and from the function via pointers. This holds for cases when it is desirable to either change the value of the variable given as a parameter by the calling function, or when passing large amount of data:

```
void myfunc ( int *a, int b ) {
    *a += b;
}

void main ( ) {
    int a = 2, b = 1;
    myfunc ( &a, b );
    printf  ( "myfunc = %d\n", a );
}
```

Passing Parameters to a Function (3)

```
#define ARRAY_LENGTH 5

int myfunc ( int *a, int b ) {
    int result = 0;
    int i;
    for ( i = 0; i < b; i++ ) result += a[i];
    return result;
}

void main ( ) {
    int array[ARRAY_LENGTH] = {1, 2, 2, 6, 10};
    printf  ( "myfunc = %d\n", myfunc ( array, ARRAY_LENGTH ) );
}
```

Lecture Summary – Essential Things to Remember

- **Pointers** and **arrays** understanding forms the basis of C programming skills – learn to use them thoroughly.
- Strings in C are just arrays of ACII codes closed by *'\0'*.
- **Scope** of a variable determines the region of a program within which the variable is accessible. The scope of a variable is determined by its declaration placement and always starts at the point of declaration.
- The **parameters** of a function are local variables, with scope ranging from beginning to the end of the function block. The parameters are passed via stack.
- Large parameters can be issued to and from the function via pointers.