

Ferenc Hegedus

Budapest University of Technology and Economics

Department of Control for Transportation and Vehicle Systems

hegedus.ferenc@edu.bme.hu

C Programming for beginners

Part 04

Preprocessor, arrays, pointers

C programming

Content for Part 04

- ▶ The C Preprocessor
- ▶ Arrays
- ▶ Pointers

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic look.

The C Preprocessor

C programming

The C Preprocessor - file inclusion

- ▶ File inclusion: any source line of the following form is replaced with the content of the corresponding file.
#include <filename> **#include <stdio.h>**
#include "filename" **#include "ownheader.h"**
- ▶ In case of " " notation, the searching for the file happens where the source program is located. If it's not found there, or the < > notation is used, searching follows an implementation-defined rule to find the file. This usually means that the file is searched among the compiler's standard library implementation.
- ▶ File inclusion is used to include common #define statements, external declarations and function prototypes.

C programming

The C Preprocessor - macro definition

- ▶ A definition for macro substitution has the following form:

```
#define name replacement_text
```

- ▶ Every occurrence of the token name will be replaced by the replacement_text.
- ▶ The *name* has the same form as a variable name, the *replacement text* can be arbitrary.
- ▶ The *replacement text* is normally the rest of the line, but long definitions may be continued by placing a \ sign at the end of the line.
- ▶ The scope of the *name* is from the point of definition to the end of the source file being compiled.
- ▶ Substitutions are made only for tokens and do not take place within quoted strings.
- ▶ It is also possible to define macros with arguments, so replacement text can be different in for different calls of the macro. E.g:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

C programming

The C Preprocessor - macro definition

- ▶ The usage of macros with arguments may not lead to the desired behavior.
- ▶ E.g. in case of macro `#define max(a, b) ((a) > (b) ? (a) : (b))`
the call `max(i++, j++)`
expands to `((i++) > (j++) ? (i++) : (j++))`
which will produce incorrect result.
- ▶ If the usage of a formal parameter in the macro is preceded with `#`, it will be substituted as a string. E.g. in case of macro `#define printvar(var) printf(#var " = %g\n", var)`
the call `printvar(d)`
expands to `printf("d" " = %g\n", d)`
- ▶ Directive `#undef` can be used to undefine a name that was previously defined.
- ▶ Let's see what we learnt about macros in an example!

C programming

The C Preprocessor - conditional compilation

- ▶ Conditional preprocessor directives can be used to control the preprocessing progress. This makes it possible to include code selectively, depending on the values of conditions evaluated during compilation.
- ▶ The **#if** directive evaluates a constant integer expression, and if the expression is non-zero, the following lines until an **#endif**, **#elif** (else if for preprocessor) or **#else** directive are included.
- ▶ The expression **defined(name)** in a **#if** is 1 if the name is defined, and 0 otherwise.
- ▶ The directive **#if defined** is equivalent with **#ifdef** and **#if !defined** with **#ifndef**.
- ▶ E.g. to make sure that the content of file header.h is included only once, one can write:

```
#if !defined(HEADER_H)
#define(HEADER_H)
    /* content goes here */
#endif
```

- ▶ Let's see conditional compilation in an example!

Arrays

C programming

Arrays - one dimensional

- ▶ Arrays are a data structure to store a fixed-size sequential collection of elements of same type.
- ▶ Arrays can also be think of as a collection of variables. Instead of declaring individual variables such as `x0`, `x1`, `x2`, `x3`, ... one can declare an array and use `x[0]`, `x[1]`, `x[2]`, `x[3]`, ...
- ▶ An array consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last one.

- ▶ Arrays can be declared as

```
type array_name [array_size]      int numbers[100]
```

- ▶ The size of the array must be an integer constant greater than 0, and the type can be any valid data type in current program.
- ▶ Array are indexed from 0 in C, so one can use `array_name[0]` to access the first element, and `array_name[array_size - 1]` to access the last element.
- ▶ Once defined, the size of an array can be evaluated by

```
sizeof(array_name) / sizeof(type)
```

C programming

Arrays - multidimensional

- ▶ Multi-dimensional arrays can be declared as

```
type array_name [array_size0] [array_size1]...[array_sizeN]
```

- ▶ Initialization of a 2-dimensional arrays can happen like:

```
int a[3][5] =  
{  
    { 0, 1, 2, 3, 4 } ,      // a[0][*]  
    { 5, 6, 7, 8, 9 } ,      // a[1][*]  
    { 10, 11, 12, 13, 14 } ,  // a[2][*]  
};
```


- ▶ The value of `a[1][3]` is 8 in this case.
- ▶ Let's see arrays in an example!

Pointers

C programming

Pointers

- ▶ A typical machine has an array of consecutively addressed memory cells that can be manipulated individually or in contiguous groups. A single memory cell holds one byte of information.
- ▶ E.g. a single byte can be a **char**, a pair of one-byte cells can represent a **short int**, and four adjacent bytes can store an **int**.
- ▶ A pointer is a group of cells that can hold a memory address. In case of x86 compilation, a pointer requires 4 bytes, while in case of x64 builds it needs 8 bytes obviously.
- ▶ Let's **c** be a **char** and **p** a pointer to it (**char***). We can represent this situation as follows.



variable		char *p; p = &c;			char c; c = 0;		
value		p = 4 *p = 0			c = 0 &c = 4		
address	0	1	2	3	4	5	6

C programming

Pointers

- ▶ The unary operator `&` gives the address of an object, so the statement `p = &c;` assigns the address of `c` to the variable `p`. In this case `p` is said to "point to" `c`.
- ▶ The `&` operator can only be applied to objects in the memory; variables and array elements. It cannot be applied to expressions or constants.
- ▶ The unary operator `*` (indirection or dereferencing), when applied to a pointer, accesses the object that the pointer points to.
- ▶ If `int *ip` points to `int x`, then `*ip` can occur in any context where `x` could, e.g.:
 - ▶ `*ip = *ip + 10;` // increments `*ip (x)` by 10
 - ▶ `y = *ip + 1;` // takes what `ip` points at (`x`), adds 1 and assigns the result to `y`
 - ▶ `(*ip)++;` // increments what `ip` points to (`x`)
- ▶ Since pointers are variables, they can be used without dereferencing. If `int *iq` is another pointer, `iq = ip;` will make `iq` to point to wherever `ip` pointed to.
- ▶ Let's see the `&` and `*` operators in an example!

C programming

Pointers - function arguments

- ▶ Function arguments are always passed by value in C. This means that the called function gets only the copies of the caller function's variables that are passed as arguments.
- ▶ Because of this, there is no direct way for the called function to alter a variable in the calling function.
- ▶ Pointer arguments enable a function to access and change objects in the function that called it. This can be very useful when a function has multiple outputs.
- ▶ E.g. a function that has to swap two variables:

// correct

```
void swap(int *px, int *py)
```

```
{
```

```
    int temp = *px;
```

```
    *px = *py;
```

```
    *py = temp;
```

```
}
```

Programming - Ferenc Hegedus

// wrong: this only swaps the copies

```
void swap(int x, int y)
```

```
{
```

```
    int temp = x;
```

```
    x = y;
```

```
    y = temp;
```

```
}
```

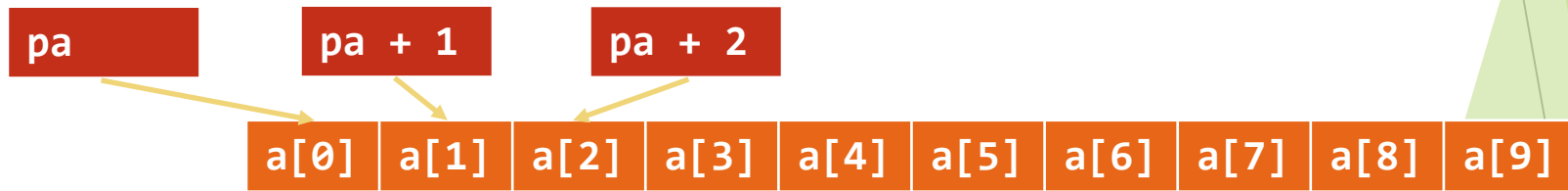
C programming

Pointers - relation to arrays

- ▶ In C, there is strong relationship between arrays and pointers. Any operation that can be achieved by array subscripting can also be done with pointers.
- ▶ The declaration `int a[10]` defines an array of size 10, a block of consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`.



- ▶ If `int *pa` is a pointer to an integer, then the assignment `pa = &a[0];` sets `pa` to point to element zero of `a`.



- ▶ If `pa` points to a particular element of an array, then by definition `pa + i` points `i` elements after, and `pa - i` points `i` elements before `pa`.

C programming

Pointers - relation to arrays

- ▶ If `pa` points to `a[0]`, the following notations are equivalent:
 - ▶ `*pa` `a[0]`
 - ▶ `*(pa + 1)` `a[1]`
 - ▶ `*(pa + i)` `a[i]`
- ▶ The remarks above are true regardless of the type or size of the variable in the array `a`. The meaning adding `i` to a pointer is that `pa + i` points to the i^{th} object after `pa`.
- ▶ By definition, the value of a variable or expression of type array is the address of element zero of the array. This means that the following are equivalent:
 - ▶ `a` `&a[0]`
- ▶ Consequently, the reference `a[i]` can always be written as `*(a + i)`. The two forms are equivalent.
- ▶ There is one difference between an array name and a pointer. Since a pointer is a variable, expressions like `pa = a` or `pa++` are legal. But an array name is not a variable, so constructions like `a = pa` or `a++` are illegal.
- ▶ Let's see the relation of pointers and arrays in an example!

C programming

Pointers - arrays as function arguments

- ▶ When an array name is passed to a function, what is passed is the location of the initial element.
- ▶ Within the called function, this argument is a local variable, so an array name parameter is a pointer.
 - ▶ As a pointer, it can be incremented or decremented freely.
- ▶ As formal parameters in a function definition, the following are equivalent:
 - ▶ `char s[]` `char *s`
- ▶ It is possible to pass just a part of an array to a function by passing a pointer to the beginning of the subarray:
 - ▶ `fcn(&a[2])` and `fcn(a+2)` both pass to function `fcn` the subarray that starts at `a[2]`.
- ▶ Let's see array as function arguments in an example!

C programming

Pointers - character pointers

- ▶ A string constant, written as `"This is a string!"` is an array of characters. It is also terminated with the null character `'\0'` so the length of storage is one more than the number of characters.
- ▶ When a string constant is used as a function argument, the function receives a pointer to the beginning of the character array.
- ▶ If there is a character pointer `char *pstring`, the following statement assigns the beginning of the character array to `pstring`:

```
pstring = "This is a string again.";
```
- ▶ It is important that this operation is not a string copy. Only pointers are involved. C does not provide any operators for processing an entire string of characters as one unit.

C programming

Pointers - character pointers

- ▶ There is an important difference between the two definitions below:

```
char amessage[] = "This is an array.";
```

```
char *pmessage = "This is a pointer.";
```

- ▶ The variable `amessage` is an array, just big enough to store the string. The characters of the string may be modified, but `amessage` will always refer to the same storage location.
- ▶ The variable `pmessage` is a pointer, initialized to point to the beginning of a string constant. The pointer may later be modified to point somewhere else, but the behavior is undefined if one tries to modify the string's content.
- ▶ Let's see the difference in an example!

C programming

Pointers - address arithmetic

- ▶ C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal abnormal events.
- ▶ Pointers and integers are not interchangeable, except for the value zero. Constant zero (0) may be assigned to a pointer, and a pointer may be compared with constant zero (0).
- ▶ The macro `NULL` defined in `stdio.h` is usually used instead of zero to indicate that this is a special value for a pointer.
- ▶ If `p` and `q` point to members of the same array, relations like `==`, `!=`, `<`, `<=`, etc. work properly. E.g.:
 - ▶ `p < q` is true when `p` points to an earlier member of the array than `q` does.
- ▶ Behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array. (Exception: the address of the first element after the end of an array can be used in pointer arithmetic.)
- ▶ If `p` and `q` point to members of the same array and `p < q`, then `q - p + 1` is the number of elements from `p` to `q` inclusively.

C programming

Pointers - address arithmetic

- ▶ Pointer arithmetic is consistent, it is independent from the type of the data that is pointed to. All pointer manipulations automatically take into account the size of the object pointed to. E.g.:

```
int i[2] = { 0, 1, 2 };    // an int is 4 bytes
int *pi = i;               // pi points to 0 (to i[0])
pi++;                      // pi points to 1 (to i[1] which is 4 bytes after i[0])
double d[2] = { 0.0, 1.0, 2.0 }; // a double is 8 bytes
double *pd = d;            // pd points to 0.0 (to d[0])
pd++;                      // pd points to 1.0 (to d[1] which is 8 bytes after d[0])
```

- ▶ The valid pointer operations are:
 - ▶ assignment of pointers of same type,
 - ▶ adding and subtracting a pointer and an integer,
 - ▶ subtracting and comparing two pointers to members of same array,
 - ▶ assigning and comparing to zero.

C programming

Pointers - pointer arrays

- ▶ Since pointers are variables themselves, they can be stored in arrays just like any other variables.
- ▶ A simple example where a pointer is needed: store multiple variable-length strings in an array. E.g.:

```
char *month(int n)
{
    static char *months[] = { "Not a valid month", "January", "February", "March", "April",
                              "May", "June", "July", "August", "September", "October",
                              "November", "December" };

    char *result = months[0];
    if((n > 0) && (n < 13))
    {
        result = months[n];
    }
    return result;
}
```

C programming

Pointers - difference between pointer and multidimensional arrays

- ▶ Consider the following definitions:

```
int a[10][20];
```

```
int *b[10];
```

- ▶ The references `a[3][4]` and `b[3][4]` are both syntactically legal.
- ▶ Variable `a` is a real 2D array and storage is allocated for 200 `int`.
- ▶ For `b`, only 10 pointers are allocated, initialization must be done explicitly statically or by code.
- ▶ Assuming that each element of `b` points to a 20 element array, storage is allocated for 200 `ints`, plus for the 10 pointers.
- ▶ In case of a pointer array, the rows can be of different length.

C programming

Pointers - pointers to functions

- ▶ A function itself is not a variable, but it is possible to define pointers to functions.

- ▶ A definition of a function pointer looks like:

```
return_type (*fcn_name)(arg_type1, arg_type2, ...)
```

- ▶ E.g. a pointer that points to a function that has two double arguments and returns an int looks like:

```
int (*fcn)(double, double)
```

- ▶ To assign the address of a function to a function pointer one can write:

```
fcn = &function_to_point_to; // the & is optional
```

- ▶ To call a function that a function pointer points to one can write:

```
(*fcn)(1.0, 2.0) // the * with the parentheses is optional
```

- ▶ Let's see function pointers in an example!

C programming

Pointers - command line arguments

- ▶ The standard prototype of the main function looks like:

```
int main(int argc, char *argv[])
```

- ▶ The first argument **argc** (argument count) is the number of command line arguments that the program was invoked with.
- ▶ The second argument **argv** (argument vector) is an array of character strings that contain the arguments, one per string.
- ▶ By convention, **argv[0]** is the name by which the program was invoked, this means **argc** is at least 1.
- ▶ Let's see command line arguments in an example!



Thank
you