Maciej Sobieraj

Lecture 9

Outline

1. Functions
    1. **Structures and strings as function parameters**
    2. Arrays as function parameters
2. Quiz

Function parameters that are structures

- Structures are always passed by *value* – just **like scalars**. This means that the following program → will output the text: 1 A

```c
#include <stdio.h>

struct STR {
   int    Int;
   char   Char;
};

void fun(struct STR s) {
   s.Int = 2;
   s.Char = 'B';
}

int main(void) {
   struct STR str = { 1, 'A' };

   fun(str);
   printf("%d %c", str.Int, str.Char);
   return 0;
}
```

Function parameters that are structures

- This function expects a **pointer to the structure**

```c
#include <stdio.h>

struct STR {
   int    Int;
   char   Char;
};

void funx(struct STR *p) {
   p -> Int = 2;
   p -> Char = 'B';
}

int main(void) {
   struct STR str = { 1, 'A' };

   funx(&str);
   printf("%d %c", str.Int, str.Char);
   return 0;
}
```

Function parameters

- **Arrays are always passed as a pointer to the first element.**
- It means that any change made to a “formal” array's element value **is immediately reflected** in an “actual” array.

Function parameters

```c
#include <stdio.h>

void mul2(int *arrptr) {
   int i;

   for(i = 0; i < 5; i++)
     arrptr[i] *= 2;
}

int main(void) {
   int arr[5] = { 1, 2, 3, 4, 5 };
   int i;

   for(i = 0; i < 5; i++)
      printf("%d ", arr[i]);
   printf("\n");
   mul2(arr);
   for(i = 0; i < 5; i++)
      printf("%d ", arr[i]);
   printf("\n");
   return 0;
}
```

Function parameters

- This program will output two lines of text, coming from the contents of the array before and after invoking the *mul2* function:

```
1 2 3 4 5
2 4 6 8 10
```

Function parameters

- Note that the **formal** parameter is declared as a pointer:
  - **void** mul2(**int** *arrptr)
- while the **actual** parameter is the name of the array:
  - mul2 (array);

Function parameters

- Note – the part of the code used to print the content of the array is duplicated.
- This is a great opportunity to make a new function and to invoke it, instead of performing loops.
- The new function will receive the pointer as the actual parameter, but the formal parameter is declared differently than before.

Function parameters

- Pay attention to two important circumstances:
  - note how clear the *main* function becomes when we **get rid of the repetitive code**; the name of the new function makes its purpose clear; this is what we call *self-commenting*; the word “print” affects the reader's imagination more strongly than any long-winded comment;
  - we’ve used a slightly different technique for declaring the formal parameter ¬– it suggests that we’re going to deal with an array, not a regular pointer:
    - int a[]

Function parameters

- The function that sends the pointer to the array as a parameter can't guess how many elements the array contains. An attempt to use the *sizeof* operator comes to nothing – we'll only get the size of the pointer itself.
- There are two possibilities:
  - the programmer only knows the size of the array (like in the programs cited here →, where we know that the array contains 5 elements)
  - information about the array's size is passed using another method (e.g. with the use of a different parameter)

Function parameters

```c
#include <stdio.h>
void print(int a[]) {
   int i;
   for(i = 0; i < 5; i++)
      printf("%d ", a[i]);
   printf("\n");
}
void mul2(int *arrptr) {
   int i;
   for(i = 0; i < 5; i++)
     arrptr[i] *= 2;
}
int main(void) {
   int arr[5] = { 1, 2, 3, 4, 5 };

   print(arr);
   mul2(arr);
   print(arr);
   return 0;
}
```

Function parameters that are strings

- A string is just a special type of array. We know that:
  - it's an array of type *char*
  - the array content must end with an empty character ('\0')
- We'll try to write our own implementation of the *strlen* function.

Function parameters that are strings

- We'll also make the following assumptions:
  - we require a **counter** with an initial value of zero
  - we **check** the character pointed to by *str* – if it's '\ 0', we'll leave the function returning the current *counter* value as the result
  - otherwise, we'll **increment** the *counter* and move the *str* pointer to the next character inside the string
  - we'll **return** to step 2
- Note that steps 2 through 4 form a loop. The condition is checked at the beginning of the loop's body, so we'll use the *while* loop for our algorithm.

Function parameters that are strings

```
int mystrlen(char *str) {
    int counter = 0;

    while(*str != '\0') {
        counter++;
        str++;
    }
    return counter;
}
```

Function parameters that are strings

- **Improvement #1:**
  - The original code:
    - **while**(*str != '\0') ...
  - Knowing the assumptions the “C” language makes regarding true and false values, we can shorten the condition and encode it in the following way:
    - **while**(*str) ...

Function parameters that are strings

- **Improvement #1:**

```
int mystrlen(char *str) {
   int counter = 0;

   while(*str) {
      counter++;
      str++;
   }
   return counter;
}
```

Function parameters that are strings

- **Improvement #2:**
  - If we take into account the priorities of the * and ++ operators, we'll come to the conclusion that the condition and the body of the *while* loop:
    - **while**(*str) {
      counter++;
      str++;
      }
  - may be encoded in a much more comprehensive form

Function parameters that are strings

- **Improvement #2:**

```
int mystrlen(char *str) {
   int counter = 0;

   while(*str++)
      counter++;
   return counter;
}
```

Function parameters that are strings

- **Improvement #3:**
  - Try to see the following interesting dependency:
    - **initialization**: *counter* = 0
    - **checking**: **str*++
    - **modifying**: *counter*++
  - Does it remind you of anything?

Function parameters that are strings

- **Improvement #3:**

```
int mystrlen(char *str) {
   int counter;

   for(counter =0; *str++; counter++)
      ;
   return counter;
}
```

Function parameters that are strings

- **Improvement #4:**
  - Note the redundancy – we increment two dependent values (counter and pointer). We'll try to simplify our algorithm and increase one variable. The first option will increment the counter only

Function parameters that are strings

- **Improvement #4:**

```
int mystrlen(char *str) {
   int counter;

   for(counter = 0; str[counter]; counter++)
      ;
   return counter;
}
```

Function parameters that are strings

- **Improvement #5:**
  - Now we'll increment the pointer and remove the counter from the code completely. We'll need a second pointer, though, which will be responsible for pointing to the beginning of the string

Function parameters that are strings

- **Improvement #5:**

```
int mystrlen(char *str) {
   char *begin;

   for(begin = str; *str; str++)
      ;
   return str – begin;
}
```

Function parameters that are strings

- Now we're going to implement our version of the *strcpy* function. The original function has the following prototype:
  - **char** *strcpy(**char** *destination, **char** *source);
- The function **copies** all the characters from the string pointed to by *source* into the string pointed to by *destination* and returns the *destination* pointer as the result.

Function parameters that are strings

- We can describe the algorithm in the following way:
  - **store** the destination pointer in the *res* variable
  - **copy** one character from the char pointed to by *source* to the char pointed to by *destination*
  - **check** if the copied char was the empty char; if yes, exit the function and return the *res* variable as the function result; otherwise, increase both *source* and *destination* pointers
  - **return** to step #2

Function parameters that are strings

```
char *mystrcpy(char *destination, char *source) {
  char *res = destination;

  for( ; ; ) {          /* a finite infinite loop */
     *destination = *source;
    if(*source == '\0')
      break;
    destination++;
    source++;
  }
  return res;
}
```

Function parameters that are strings

- The first modification is based on an interesting property of the = operator.
- The assignment operator sets its left argument with a value of the right argument, but also **returns the value** which has been assigned already.
- The value of the *2+2* expression is *4*, and in the same way the value of the expression
  - i = 2 * 3

    is 6.

Function parameters that are strings

- This feature of the = operator is often used by expressions similar to the following one
  - i = j = k = 0;
- which should be interpreted as follows
  - i = (j = (k = 0));
- and which will result in the substitution of zero into the variables *k, j* and *i* (in that order), which is obviously more elegant than:

```
k = 0;
j = 0;
i = 0;
```

Function parameters that are strings

```
char *mystrcpy(char *destination, char *source) {
   char *res = destination;

   for( ; ; ) {          /* a finite infinite loop */
      if(!(*destination = *source))
         break;
      destination++;
      source++;
   }
   return res;
}
```

Function parameters that are strings

```
char *mystrcpy(char *destination, char *source) {
  char *res;

  for(res = destination; (*destination = *source);) {
    destination++;
    source++;
  }
  return res;
}
```

- Here's a small code fragment to test it:

```
char str1[10] = "not so";
char str2[10] = "best test";
mystrcpy(str1, str2);
```

Function parameters that are strings

- We would even go as far as to say that there's no way of writing it in a more condensed way.

```
char *mystrcpy(char *destination, char *source) {
   char *res;

   for(res = destination; (*destination++ = *source++); ) ;
   return res;
}
```

Function parameters that are strings

```
char *mystrcat(char *destination, char *source) {
   char *res;

   for(res = destination; *destination++; ) ;
   for(--destination; (*destination++ = *source++); ) ;
   return res;
}
```

Outline

1. Functions
    1. Structures and strings as function parameters
    2. **Arrays as function parameters**
2. Quiz

Function parameters

- **Function parameters which are multidimensional arrays**
- As you already know, multidimensional arrays may exist:
  - in a true and pure form when they're declared with a complete set of dimensions, like this:
    - **int** array[3][3];

which creates an array of nine elements arranged in three rows and three columns;

Function parameters

- As you already know, multidimensional arrays may exist:
  - as an array of pointers, i.e. as the **vector which contains pointers** to the rows of the array and is declared like this:
    - **int** *ptrarr[3];
  - Although references to elements of both arrays are syntactically identical and look like this:
    - array[2][2] = ptrarr[2][2];
  - they differ significantly in terms of semantics.

Function parameters

- We have a simple *main* function, containing a declaration of a small 2-dimensional array

```
int main(void) {
   int arr[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   /* we want to invoke a smart function here which is able
     to print the content of the tab in a nice way */

   return 0;
}
```

Function parameters

- We'll design the **skeleton** of this function, neglecting completely the way in which the formal parameter has to be declared and replacing it with three question marks. We only assume that the formal parameter's name will be *t*
- We'd like to invoke this function as follows:
  - printarr(arr);

Function parameters

```
void printarr( ??? ) {
    int i,j;
    for(i = 0; i<3; i++) {
        for(j = 0; j<3; j++)
            printf("%d\t",t[i][j]);
        printf("\n");
    }
}
```

Function parameters

- Let's analyze how the *printarr* function operates on the t array. We assume that we want to reach the element *t[1][1].*
- So:
  - we have to “skip” the whole first row of the t array to get to the beginning of the second row;
  - once in the second, row we must “jump” to the second element.

Function parameters

- **Question**: To successfully go through the steps described above, do we need to know how **many rows** the array contains?
- **Answer**: **No**, we don't, because when we want to “jump” to the beginning of the row number x, and so we don't care about how many rows exist in the array.

- **Question**: To successfully go through the steps described above, do we need to know **how many columns** the array contains (that is, how many elements exist in each row)?
- **Answer**: **Yes**, because without this information we won't be able to skip *x-1* lines; we need to know how long each row is.

Function parameters

- The declaration is **invalid**, because *t* is not a pointer to an array of pointers, but a pointer to a real array.

~~void printarr(int **t);~~

Function parameters

- The following declaration is **prohibited** too.
- There's a suggestion that *t* is a two-dimensional array, but the compiler doesn't know how many rows it contains

```
void printarr(int t[][]);
```

Function parameters

- This is good, although we've now provided more information than the compiler really needs. The information stating that t has three rows is not useful to the compiler at all.

```
void printarr(int t[3][3]);
```

Function parameters

- This means that a form of the declaration which is both **acceptable** and **sufficient** goes as follows:

```
void printarr(int t[][3]);
```

Function parameters

```c
#include <stdio.h>

void printarr(int t[][3]) {
   int i,j;
   for(i = 0; i<3; i++) {
      for(j = 0; j<3; j++)
         printf("%d\t",t[i][j]);
      printf("\n");
   }
}

int main(void) {
   int arr[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

   printarr(arr);
   return 0;
}
```

Function parameters

```
int main(void) {
   int *arrptr[3];
   int i,j;

   for(i = 0; i < 3; i++) {
      arrptr[i] = (int *) malloc(3 * sizeof(int));
      for(j = 0; j < 3; j++)
         arrptr[i][j] = (3 * i) + j + 1;
   }

   /* we want to invoke a smart function here which is able
            to print the content of the tab in a nice way */

   for(i = 0; i < 3; i++)
      free(arrptr[i]);
   return 0;
}
```

Function parameters

- In this case, no difficulty arises in the declaration. It may take one of the following two forms:

```
void printarrptr(int *t[]);
void printarrptr(int **t);
```

Function parameters

```
#include <stdio.h>
#include <stdlib.h>
void printarrptr(int **t) {
   int i,j;
   for(i = 0; i<3; i++) {
      for(j = 0; j<3; j++)
         printf("%d\t",t[i][j]);
      printf("\n");
   }
}
int main(void) {
   int *arrptr[3];
   int i,j;

   for(i = 0; i < 3; i++) {
      arrptr[i] = (int *) malloc(3 * sizeof(int));
      for(j = 0; j < 3; j++)
         arrptr[i][j] = (3 * i) + j + 1;
   }
   printarrptr(arrptr);
   for(i = 0; i < 3; i++)
      free(arrptr[i]);
   return 0;
}
```

Outline

1. Functions
    1. Structures and strings as function parameters
    2. Arrays as function parameters
2. **Quiz**

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <string.h>
void f(char *s) {
   s[1] = '\0';
}
int main(void) {
   char p1[] = "ABC", p2[] = "XYZ";
   f(p1);
   f(p2);
   printf("%d",strlen(p1) + strlen(p2));
   return 0;
}
```

- ○ the program outputs 2
- ○ the program outputs 0
- ○ the program outputs 1

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <string.h>
void cp(char *p1,char *p2) {
   int i,j;

   j = 0;
   for(i = 0; i < strlen(p2); i += 2)
      p1[j++] = p2[i];
   p1[j] = '\0';
}
int main(void) {
   char str[100];

   cp(str,"ABCD");
   printf("%s",str);
   return 0;
}
```

- ○ the program outputs AB
- ○ the program outputs AD
- ○ the program outputs AC

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
char fun(char *n, int m) {
   return *(n + 2 * m);
}
int main(void) {
   printf("%c",fun("aAbBcCdD",1));
   return 0;
}
```

- ○ the program outputs b
- ○ the program outputs 0
- ○ the program outputs a

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
char fun(char *n, int m) {
   return (m+2)[n];
}
int main(void) {
   printf("%c",fun("aAbBcCdD",1));
   return 0;
}
```

- the program outputs A
- the program outputs B
- the program outputs C