CH-230-A

Programming in C and C++

C/C++

Lecture 4

Dr. Kinga Lipskoch

Fall 2020

Local Variables

- Variables can be declared inside any function
  - These are called local variables
  - Local variables are created when the function is called (e.g., the control is transferred to the function) and are destroyed when the function terminates
- Local variables do not retain their values between different calls

The Concept of Scope

- The scope of a name (function, variable, constant) is the part of the program where that name can be used
- The scope of a local variable is the function where it is defined
  - From the point of its definition
- Names having different scopes do not clash

Global Scope

- The scope of the names of functions goes from the prototype/definition to the end of file
- After their name is known they can be used, i.e., called
- It is possible to define global variables, i.e., variables outside function
  - Their scope is from the point of definition to the end of the file
  - After their definition is given they can be used, i.e., written and read

Local and Global Scope

```c
#include <stdio.h>

//global variable
int x = 7;

void xlocal(int y)  {
  int x;
  x = y * y;
  printf("xlocal: %d\n", x);
  return;
}

void xglobal(int y)  {
  x = y * x;
  printf("xglobal: %d\n", x);
  return;
}
```

```c
int main() {
  //int x;
  // try to explain if not
  // commented out
  x = 8;
  printf("main: %d\n", x);
  xlocal(x);
  printf("main: %d\n", x);
  xglobal(x);
  printf("main: %d\n", x);
  return 0;
}
```

Do not Misuse Global Variables

- Global variables can be used to communicate parameters between functions
- They can introduce subtle bugs in your code
- In general try to avoid them unless enormous advantages can be gained at a price of low risk
  - Document why you insert them
- Bigger projects will avoid using global variables

Parameters

- Function parameters are treated as local variables
- Local variables within functions and parameters must have different names
- Therefore the scope of a parameter is its function

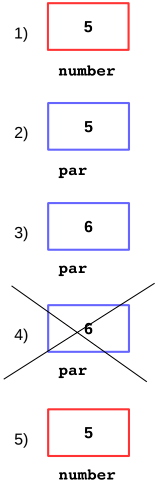
Parameters: by Value and by Reference

- **By value**: variables are copied to parameters
  - Changes made to parameters are not seen outside the function
- **By reference**: variables and parameters coincide
  - Changes made to parameters are seen outside the function
  - In C this is obtained by mean of pointers

Example: Passing by Value (1)

```c
#include <stdio.h>
void increase(int par) {
  par++;
}
/*  In this case no prototype:
    can you tell why? */
int main() {
  int number = 5;
  increase(number);
  printf("Increased number is %d\n", number);
  /* not as expected? */
  return 0;
}
```

Example: Passing by Value (2)



Parameters by Reference in C

- C passes only parameters by value
- For references it is necessary to provide a pointer to the variable
- In order to make a modification visible
- Outside it is necessary to use the dereference ($*$) operator

Example: Passing by Reference (1)

```c
#include <stdio.h>

void increase(int *par) {
  *par = *par + 1;
}

int main() {
  int number = 5;
  increase(&number); /* pass pointer */
  printf("Increased number is %d", number);
  return 0;
}
```

Example: Passing by Reference (1)

1) $\boxed{5}$

`number`

2) $\boxed{5}$

`par` is pointing to number `par = &number`
`par` is the copy of the memory address of `number`

3) $\boxed{6}$

`number` manipulated via pointer `par`

4) **`par` is deleted as the copy of the address**

5) $\boxed{6}$

`number`

Indentation Styles (1)

- Use spaces between operators: `a = b + 5;`
- Exception: `b++;`
- Do not use spaces if parentheses act as delimiter (functions)
  `printf("Number %d", b);`
- But use spaces before after `if`, `for`, `while`:
  `while (i <= 10)`
- Always put a space after comma
- Do not put a space before semicolon:
  `printf("Number %d", b);`

Indentation Styles (2)

- Put the opening brace either behind last word (including space) or put it on the next line
- Indent the block inside by tab or 4 (8) spaces
- The closing brace should be on the same column as the opening statement

```
for (i = 0; i < 10; i++) {    // K&R style
  printf("%d\n", i);
}
```

or

```
for (i = 0; i < 10; i++)     // Allman style
{
  printf("%d\n", i);
}
```

Strings

- A string is a sequence of characters
- Strings are often the main way used to communicate information to the user
- Many languages provide a string data type, but C does not
- In C strings are treated as arrays of characters
- `char my_string[30];`

C Strings

- A string is represented as a sequence of chars enclosed by double quotes
  - `"This is it"`
- String are stored in arrays of chars
  - An extra character is always added at the end to mark the end of the string
  - The extra character is the `'\0'` character i.e., the character whose ASCII code is 0

T	**h**	**i**	**s**		**i**	**s**		**i**	**t**	**\0**

fgets versus gets (1)

- `gets` does not check if you type more characters than allowed:

```
char inputString[50];
gets(inputString);
```

- `fgets` allows additional parameters:

```
char line[50];
fgets(line, sizeof(line), stdin);
```

  - Reads up to 49 characters from the input stream
  - The 50th one is used to store the null character '\0'

`fgets` versus `gets` (2)

- `gets` replaces the trailing '\n' with a '\0'
- `fgets` does not replace '\n', but it leaves it in the string
- Read the man pages for learning more on these functions
  - `man gets`
  - `man fgets`
- To make your life easier use `fgets` and convert to integer via `sscanf`
- Avoid using `gets`, it is unsafe

fgets and scanf together

- `scanf` and `fgets` do not work well together
- Your code should look like this, if you use both

```
scanf("%d", &number);
getchar();
...
fgets(line, sizeof(line), stdin);
sscanf(line, "%d", &number);
```

String Functions

- Defined in `string.h`
- `strlen` Determines the length of a string
- `strcat` Concatenates two strings
- `strcpy` Copies one string into another
- `strcmp` Compares two strings
- `strchr` Searches a char in a string
- See man pages
  - Do not reinvent the wheel, there are many many functions that will help you

Passing Arrays to Functions

- An array does not store its size
- This has to be provided as a parameter, or by making assumptions on the contents of the array (like for strings)
- The name of an array is a pointer to the first element of the array, i.e., when an array is passed to a function, a copy of the address of the first element is given
- Modifications to the elements are seen outside
- Modifications to the array are not seen outside
- Can you explain why?

Passing Arrays to Functions: Example

```
#include <stdio.h>
#include <stdlib.h>
void strange_function(int v[], int dim) {
  int i;
  for (i = 0; i < dim; i++)
    v[i] = 287;
  // v = (int *) malloc(sizeof(int) * 1000);
}
int main() {
  int array[] = {1, 2, 9, 16};
  int *p = &array[0];
  strange_function(array, 4);
  printf("%d %p %p\n", array[0], p, array);
  return 0;
}
```

same

Dynamic Memory Allocation

- What if we do not know the dimension of the array while coding?
- Dynamic memory allocation allows you to solve this problem
  - And many others
  - But can also cause a lot of troubles if you misuse it

Pointers and Arrays

There is a strong relation between pointers and arrays

- Indeed an array is nothing but a pointer to the first element in the sequence
- We are looking at this in detail

Specifying the Dimension on the Fly

To specify the dimension on the fly you can use the `malloc()` function defined in the header file `stdlib.h`

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *dyn_array, how_many, i;
  printf("How many elements? ");
  scanf("%d", &how_many);
  dyn_array =
    (int*) malloc(sizeof(int) * how_many);
  if (dyn_array == NULL)
    exit(1);
  for (i = 0 ; i < how_many; i++) {
    printf("\nInput number %d:", i);
    scanf("%d", &dyn_array[i]);
  } return 0;
}
```

The `malloc()` Function (1)

- `void * malloc(unsigned int);`
- `malloc` reserves a chunk of memory
- The parameter specifies how many bytes are requested
- `malloc` returns a pointer to the first byte of such a sequence
- The returned pointer must be forced (cast) to the required type

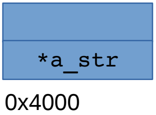
The malloc() Function (2)

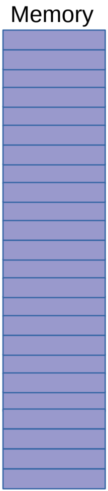
```
pointer   = (cast) malloc(number of bytes);


char* a_str;
a_str = (char*) malloc(sizeof(char) * how_many);
```

- `malloc` returns a `void *` pointer (i.e., a generic pointer) and this is assigned to a non `void *` pointer
- If you omit the casting you will get a warning concerning a possible incorrect assignment

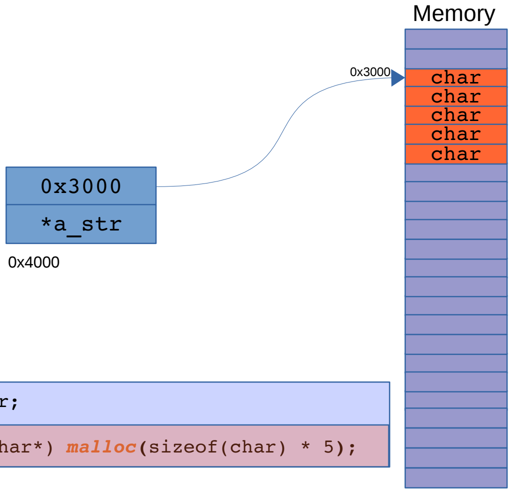
Dynamically Allocating Space for an Array of `char`





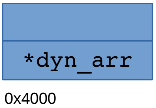

```
char *a_str;
a_str = (char*) malloc(sizeof(char) * 5);
```

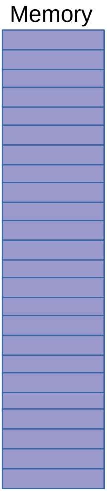
Dynamically Allocating Space for an Array of `char`




```
char *a_str;
a_str = (char*) malloc(sizeof(char) * 5);
```

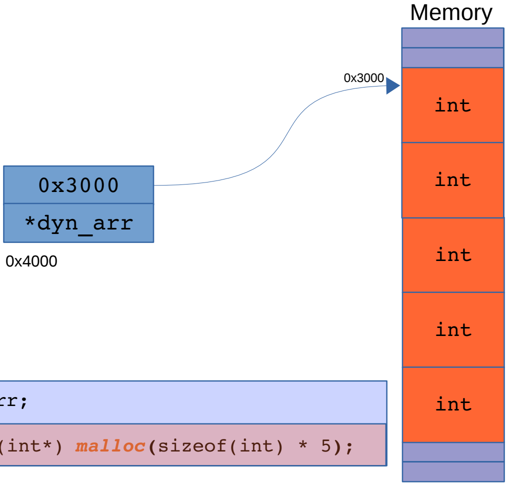
Dynamically Allocating Space for an Array of `int`






```
int *dyn_arr;
dyn_arr = (int*) malloc(sizeof(int) * 5);
```

Dynamically Allocating Space for an Array of `int`




```
int *dyn_arr;
dyn_arr = (int*) malloc(sizeof(int) * 5);
```

malloc() and free()

- All the memory you reserve via `malloc`, must be released by using the `free` function
- If you keep reserving memory without freeing, you will run out of memory

```
float *ptr;
int number;
...
ptr = (float*) malloc(sizeof(float) * number);
...
free(ptr);
```

Rules for `malloc()` and `free()`

- The following points are up to you (the compiler does not perform any control)
  1. Always check if `malloc` returned a valid pointer (i.e., not `NULL`)
  2. Free allocated memory just once
  3. Free only dynamically allocated memory
- Not following these rules will cause endless troubles
- `sizeof()` is compile time operator, it does not work on allocated memory

Review: Pointers, Arrays, Values

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int length[2] = {7, 9};
  int *ptr1, *ptr2; int n1, n2;
  ptr1 = &length[0];
  // &length[0] is pointer to first elem
  ptr2 = length;
  // length is pointer to first elem therefore
  // same as above
  n1 = length[0];
  // length[0] is value
  n2 = *ptr2;
  // *ptr2 is value therefore same as above
  printf("ptr1: %p, ptr2: %p\n", ptr1, ptr2);
  printf("n1: %d, n2: %d\n", n1, n2);
  return 0;
}
```

Multi-dimensional Arrays

- It is possible to define multi-dimensional arrays
  - Mostly used are bidimensional arrays, i.e., tables or matrices
- As for arrays, to access an element it is necessary to provide an index for each dimension
  - Think of matrices in mathematics

Multi-dimensional Arrays in C

- It is necessary to specify the size of each dimension
  - Dimensions must be constants
  - In each dimension the first element is at position 0

```
int matrix[10][20];    /* 10 rows, 20 cols */
float cube[5][5][5];   /* 125 elements */
```

- Every index goes between brackets

```
matrix[0][0] = 5;
```

Multi-dimensional Arrays in C: Example

```c
#include <stdio.h>
int main() {
  int table[50][50];
  int i, j, row, col;
  scanf("%d", &row);
  scanf("%d", &col);
  for (i = 0; i < row; i++)
    for (j = 0; j < col; j++)
      table[i][j] = i * j;
  for (i = 0; i < row; i++)
  {
    for (j = 0; j < col; j++)
      printf("%d ", table[i][j]);
    printf("\n");
  }
  return 0;
}
```

The `main` Function (1)

- Can return an `int` to the operating system
  - Program exit code (can be omitted)
  - print exit code in shell: `$> echo $?`
- Can accept two parameters:
  - An integer (usually called `argc`)
  - A vector of strings (usually called `argv`)
  - `argc` specifies how many strings contains `argv`

The main Function (2)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for (i = 1; i < argc; i++)
    printf("%d %s\n", i, argv[i]);
  return 0;
}
```

- Compile it and call the executable `paramscounter`
- Execute it as follows:

  `$> ./paramscounter first what this`
- It will print first, what and this, one word per line
- Note that argc is always greater or equal than one
- The first parameter is the program's name