Lab 3 (CS): C Language: Pointers, Arrays and Structures

Innopolis University
Course of Operating Systems

Week 03 - Lab

Recap.

- How to read a float number “grade” from stdin?

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?

Recap.

- How to read a float number “grade” from stdin?
  - $scanf(”\%f”, \&grade)$
  - $fscanf(”\%f”, \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])
- What are the stages of compiling the C program?

Recap.

- How to read a float number “grade” from stdin?
    - $scanf("\%f", \&grade)$
    - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
    - int main(int argc, char *args[])
- What are the stages of compiling the C program?
    - Preprocessing - Compiling - Assembling - Linking.

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])
- What are the stages of compiling the C program?
  - Preprocessing - Compiling - Assembling - Linking.
- How to define a string “username” in C (50 characters)?

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])
- What are the stages of compiling the C program?
  - Preprocessing - Compiling - Assembling - Linking.
- How to define a string “username” in C (50 characters)?
  - char username[50]; // (without using pointers)

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])
- What are the stages of compiling the C program?
  - Preprocessing - Compiling - Assembling - Linking.
- How to define a string “username” in C (50 characters)?
  - char username[50]; // (without using pointers)
- How to read a string “university” from stdin (4 characters)? (Which methods are (un)safe)

Recap.

- How to read a float number “grade” from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])
- What are the stages of compiling the C program?
  - Preprocessing - Compiling - Assembling - Linking.
- How to define a string “username” in C (50 characters)?
  - char username[50]; // (without using pointers)
- How to read a string “university” from stdin (4 characters)? (Which methods are (un)safe)
  - $fgets(university, 4, stdin)$
  - $scanf("\%s", university)$
  - $fscanf(stdin, "\%s", university)$
  - $gets(university)$

Recap.

- How to read a float number "grade" from stdin?
  - $scanf("\%f", \&grade)$
  - $fscanf("\%f", \&grade, stdin)$
- How to get command line parameters from the program?
  - int main(int argc, char *args[])
- What are the stages of compiling the C program?
  - Preprocessing - Compiling - Assembling - Linking.
- How to define a string "username" in C (50 characters)?
  - char username[50]; (without using pointers)
- How to read a string "university" from stdin (4 characters)? (Which methods are (un)safe)
  - $fgets(university, 4, stdin)$ // safe
  - $scanf("\%s", university)$ // unsafe "%4s"
  - $fscanf(stdin, "\%s", university)$ // unsafe
  - $gets(university)$ // unsafe

Lab Objectives

- Learn about pointers and arrays in C language.
- Learn about structures in C.
- Learn about function pointers and variadic functions.

Pointers

- Pointers
  - Special type of variables used to store the memory address of other variables. It is declared as:
    `<datatype>* var_name = NULL;`
  - NULL pointer is a reserved pointer in C and does not refer to any data objects.
  - Dynamic memory management functions: malloc, calloc, realloc, free (source: https://cplusplus.com/reference/cstdlib/).
- Constants
  - We can declare constants in C using the keyword **const** as follows:
    `const <datatype> var_name = constant_value;`
- Pointers to constants
  - `const <datatype>* var_name;`
- Constant pointers to nonconstants
  - `<datatype>* const var_name = constant_address;`
- Constant pointers to constants
  - `const <datatype>* const var_name = constant_address;`

Pointers

In brief:

- **Constant** pointer
  - Pointer value cannot be modified
  - Value pointed by the pointer can be modified
- Pointer to **constant**
  - Pointer value can be modified
  - Value pointed by the pointer cannot be modified
- **Constant** pointer to **constant**
  - Pointer value cannot be modified
  - Value pointed by the pointer cannot be modified

Source: https://codeforwin.org/2017/11/constant-pointer-and-pointer-to-constant-in-c.html

Pointers

Explain each line of the following code snippets:

- **//code snippet 1**
```
const int i = 10;
int * p = &i;
//*p = i;
(*p)++;
printf("%d\n", i);
```
- **//code snippet 2**
```
char * const p = malloc(6);
memset(p, 'A', 6);
(*p)++;
//p = NULL;
printf("%c\n", *p++);
free(p);
```
- **//code snippet 3**
```
double c = 1.5;
const double * const p = &c;
c+=0.5;
//*p = c;
//p = NULL;
printf("%f\n", *p);
```
- **//code snippet 4**
```
char * const p = malloc(6);
memset(p, 'A', 6);
(*(p+1))++;
*p = *(p+1);
//p = p;
//p = 0;
printf("%c\n", *p);
free(p);
```

Pointers

Can we modify the constants in C language?

1. A const pointer in C is a shallow read-only reference that means marking a variable as const in C protects the variable, not the object the variable refers to, from being mutated.
2. Const in C is intended to create immutable objects but it does not provide such guarantee, and the ANSI/ISO standard states that modifying const variables through pointer will trigger ***Undefined Behavior***. Be careful!!. This means, it leaves the program unpredictable in terms of the expected results. This is a loophole. There is a stronger version of constness that might guarantee immutability. The qualifier is called *immutable* and it is available in the D programming language.
3. Most C experts encourage to use const since it provides type checking benefits at compile time but you should be aware of this loophole and avoid using constants if you cannot manage them since modifying constant objects may happen non-intentionally.

Exercise 1

- Write a program *ex1.c* to declare a pointer **q** to a constant integer x whose constant value is 1. Create three contiguous memory cells of type integer and are pointed by a constant pointer **p** to the first cell. Using the pointer **p**, fill the first two cells with the value of x and the third cell with the value of $2x$. Print the memory addresses of these cells to *stdout*. Check if the cells are contiguous.
- Write a function **const_tri** which accepts the pointer **p** and **n**. It calculates the Tribonacci number for **n** using only the cells pointed by **p**. The tribonacci numbers have the recurrence equation $T_n = T_{n-1} + T_{n-2} + T_{n-3}$. The function returns the result T_n.
- **Notes:**
  - Do not forget to free the allocated cells to avoid memory leaks. Do not reallocate the cells.
  - Submit the file *ex1.c* which contains the function **const_tri**.
  - do not use any additional pointers or variables. In order to not complicate the exercise, we can allow to use only one additional integer variable *temp* (it is not a pointer).

Arrays

Array:

- A contiguous collection of homogeneous elements that can be accessed using an index.
  - **Contiguous**: the elements of the array are adjacent to one another in memory with no gaps between them.
  - **Homogeneous**: all of the elements are of the same data type.
  - Has a fixed size.

    ```
    sizeof(arr)/sizeof(arr[0])
    ```
  - Can be in multiple dimensions (2d, 3d, ...etc).
- **Dynamic** array: a variable size array which allows to modify its size after creation by the support of the pointers' *realloc* functionality.

Arrays

Example:

- Create a static array of 5 characters to store the value “inno”. Can we increase the size of the string to hold the value “innopolis” without creating a new string? Repeat the previous steps using a dynamic array.
- Create a fixed size array **m** for storing 4 floats with the values {0.25, 0.5, 0.75, 1}. Declare a pointer **p** to a *float* and make it refers to the previous array **m**. What is the size of the array **m** and the pointer **p**? Can we consider the pointer to a `<datatype>` as an array? Check the size of the pointer to char? Is it the same as size of the pointer to float?
- Create 2-dimensional fixed size array **n** of (3x3) and fill the values with numbers from 0 to 10. Create a pointer **q** to the array **m** and replicate the values of the array **m** to make an array of size (2x4).

Structures

- contains a number of different data types grouped together.
  Decalred as:
  struct structure_name {

      ....
      fields;
      .....
  } var1, var2;
- Example: struct Person{char* name; int age; float height;} p1;
- Structure fields can be accessed through a structure variable using a dot (.) operator. (e.g. p1.age)
- Structure elements can be accessed through a pointer to a structure using the arrow (→) operator. (e.g. $(\&p1) \rightarrow age$)
- All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
  struct Person p2 = p1;

Structures

Example:

- Define a structure Person with the following fields: name as pointer, age as integer and height as float. Write a function as follows:
  struct Person *createPerson(char *name, int age, float height)
- Write a function as follows:
  showPerson(struct Person *)
  It prints the information of a person to standard output
- Create an array “friends” of size 3 and fill it up with your friends’ information.
- Print all the content in your friends array to the standard output
- **Hint:** use void *malloc(size_t size), void free(void *ptr)

Exercise 2

- Define a structure **Point** with 2 real number fields **x** and **y**
- Provide an implementation of a function *distance* that computes the euclidean distance between two points.
- Write a function *area* that will compute the area of the triangle whose vertices are A(x1, y1), B(x2, y2), and C(x3, y3).
- Write a main function to define A(2.5, 6), B(1, 2.2) and C(10, 6) as the vertices of the triangle ABC. Find the distance between A and B, then calculate the area of ABC.
- Save the program as **ex2.c** and submit.
- Save the script to run the program as **ex2.sh** and submit.
- **Hint:** Use the following formula for calculating the area of the triangle ABC where A(x1, y1), B(x2, y2), and C(x3, y3):

$$area = \frac{1}{2}|x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_1 - x_1y_3|$$

Exercise 3 (Optional) (1/4)

- Files and directories in most operating systems are organized in hierarchical manner. In the Linux-based OS, you have a root directory (/) which does not have a parent directory and contains all other sub-directories and files in the system. In this exercise, you will create a simple system for organizing your files and directories hierarchically using C structures* and pointers.
- Create a `struct File` which represents a file with the fields (**id**: unique number assigned to each file, **name**: name of the file, **size**: current size of the file data, **data**: the actual textual content of the file as a string, **directory**: the directory of type `struct Directory` where the file is in). The structure supports the following operations on files:
  - **overwrite_to_file(struct File* file, const char* str)** which overwrites the file content *file* with the new content *str*.
  - **append_to_file(struct File* file, const char* str)** which appends the new content *str* to the end of the file *file*.
  - **printp_file(struct File* file)** prints to *stdout* the path of file *file*.

*https://en.wikipedia.org/wiki/Struct_(C_programming_language)

Exercise 3 (Optional) (2/4)

- Create a `struct Directory` which represents a directory with the fields (**name**: the directory name,**files**: array of files, **directories**: array of sub-directories, **nf**: number of files in the directory, **nd**: number of sub-directories in the current directory, **path**: the absolute path of this directory). It supports the following operations on directories:
  - **add_file(struct File* file, struct Directory* dir)** which adds a new file $file$ to the current directory dir.
- write a program *ex3.c* contains a *main* function.
  - Create the root directory (/) with two subdirectories **home** and **bin**.
  - Add a file **bash** to the directory **bin**.
  - Add two files ex3_1.c and ex3_2.c to the directory **home**. The file ex3_1.c contains the code: "int printf(const char * format, ...);" And the file ex3_2.c contains the code: "//This is a comment in C language"
  - Add the content "Bourne Again Shell!!" to the file **bash**.

Exercise 3 (Optional) (3/4)

- Append the content "int main(){printf("Hello World!")}" to the file **ex3_1.c**
- Print the path of all files in the system by calling the function **printp**$_{f}ile$.
- **Notes:**
    - the datatype of nf and nd in `struct Directory` is *unsigned char*. We did not specify the datatype of **id** field. It is up to you to determine the datatype taking into consideration the maximum number of files that the system can hold.
    - the maximum length of the file name is 63 characters.
    - the maximum size of the path is 2048.
    - the maximum size of file data is 1024.
    - The path in this system starts with "/".
    - size field in `struct File` is the current size of the file data or the length including the null character. Here we do not mean the maximum size of this field.
    - Submit the file *ex3.c* and a supplement script *ex3.sh* to run it.

Exercise 3 (Optional) (4/4)

Given code snippets:

```
// Prints the name of the File file
void show_file(File *file)
{
    printf("%s ", file->name);
}
// Displays the content of the Directory dir
void show_dir(Directory *dir)
{
    printf("\nDIRECTORY\n");
    printf(" path: %s\n", dir->path);
    printf(" files:\n");
    printf(" [ ");
    for (int i = 0; i < dir->nf; i++)
    {
        show_file(&(dir->files[i]));
    }
    printf("]\n");
    printf(" directories:\n");
    printf(" { ");

    for (int i = 0; i < dir->nd; i++)
    {
        show_dir(dir->sub_dirs[i]);
    }
    printf("}\n");
}
```

```
#define MAX_PATH 2048

// Adds the subdirectory dir1 to the directory dir2
void add_dir(Directory *dir1, Directory *dir2)
{
    if (dir1 && dir2)
    {
        dir2->sub_dirs[dir2->nd] = dir1;
        dir2->nd++;
        char temp_path[MAX_PATH];
        if (strcmp(dir2->path, "/"))
        {
            strcpy(temp_path, dir2->path);
            strcat(temp_path, "/");
            strcat(temp_path, dir1->name);
            strcpy(dir1->path, temp_path);
        }
        else
        {
            strcpy(temp_path, "/");
            strcat(temp_path, dir1->name);
            strcpy(dir1->path, temp_path);
        }
    }
}
```

Function pointers

Function pointer:

- A pointer that holds the address of a function. Declared as:
  `<return type> (*func_name)(`*`types`*`);`
- Examples:
  - `double (*f1)(int); // passed an int and returns a double`

  -
    ```
    int* (*f2)(char*); // passed a pointer to char and
                       // returns a pointer to an integer
    ```

Variadic function:

- A function which accepts a variable number of arguments. Declared as:
  `<return type> func_name(`*`first_params`*`, ...)`
- The parameter list ends with ellipsis (...)
- Examples:
  ```
  int sum(int n1, ...); // returns the summation
                        // of passed integers
  ```

Function pointers

Example: (assume that we are expecting double or integer pointers)

```
#include <stdio.h>
#include <stdlib.h>

void my_print_func(void* x, int size){ // define an enum to check the datatype
    if (size==sizeof(int)){
        int* y = (int*) x;
        printf( "%d\n", *y);
    }else{
        double* y = (double*) x;
        printf( "%f\n", *y);
    }
}

int main(){
    void (*foo)(void*, int);
    foo = &my_print_func;

    int* p = malloc(sizeof(int)); *p = 20;
    double* q = malloc(sizeof(double)); *q = 1.5;

    foo(p, sizeof(int));
    (*foo)(p, sizeof(int));

    foo(q, sizeof(double));

    return 0;
}
```

Variadic functions

Example:

```
#include <stdio.h>
#include <stdarg.h>

int variadic_addition (int count, ...)
{
  va_list args;
  int i, sum;

  va_start (args, count); /* Save arguments in list. */

  sum = 0;
  for (i = 0; i < count; i++){
    sum += va_arg(args, int); /* Get the next argument value. */
  }

  va_end (args); /* Stop traversal. */
  return sum;
}

int main(){
  // call 1: 4 arguments
  printf("Sum: %d\n", variadic_addition(3, 10, 20, 30));

  //call 2: 6 arguments
  printf("Sum: %d\n", variadic_addition(5, 10, 20, 30, 40, 50));

  return 0;
}
```

Exercise 4 (1/2)

- Write a function **aggregate** that applies an aggregation operation on the elements of an array of **double** and **integer** types. The supported aggregation operations are addition, multiplication and the max of the elements. The function accepts **base** as a pointer to any type, **size** as the size of array datatype in bytes, **int n** as the number of items of the array, **initial_value** as pointer to the inital value of the aggregation operation and **opr** as function pointer of two paramaters and used to apply the corresponding operation on the parameters . It has the following header:

```
void* aggregate(void* base, size_t size, int n, void* initial_value,
                                void* (*opr)(const void*, const void*))
```

- Write a program *ex4.c* to test the previous function on an array of 5 doubles and another array of 5 integers and print the result for each array to **stdout**.

Exercise 4 (2/2)

- **Notes:**
  - The return type of **aggregate** function is void*
  - You cannot change the header of the function **aggregate**.
  - We assume that we have only 2 types of arrays (double and integer arrays).
  - It is up to you to set the values of the arrays but the size of each array should be 5.
  - the initial value of the addition operation is 0 and the initial value of the multiplication operation is 1. For the max operation, you should use the minimum number for the chosen datatype as the initial value.
  - Submit the file *ex4.c*.
- **Hint:** Use *sizeof* to check the specific datatype of the void* pointer.

Exercise 5

- Write a variadic function **vaggregate** which provides the same functionality as does the function **aggregate**.

Exercise 6 (optional)

- Implement a `Quicksort* algorithm:`
  - Pick an element, called a pivot, from the array
  - Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation
  - Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values

*https://en.wikipedia.org/wiki/Quicksort

References

- Lecture Notes - Practical Programming in C
- Quick Sorting algorithm

End of Lab 3 (OS)