Object Oriented Programming - Lecture 2

Diana Borza - diana.borza@ubbcluj.ro

March 7, 2024

Content

- Pointers
- Memory management
- Modular programming
- Abstract data types

Pointers

Pointers - the basics

- **Pointer** - a variable that stores the address of another variable; they allow us to manipulate data more flexibly;
  - **Declaring** a pointer: T* variable_name;

```
int* x; // pointer to integer
int** x; // pointer to a pointer of integer
struct date *d; // pointer to a struct
```

  - **Ampersand/"address of" operator &**: used to obtain the memory address of a variable;
  - **Dereferencing operator ***: used to obtain the *value* of the variable stored at the address specified by its operand.

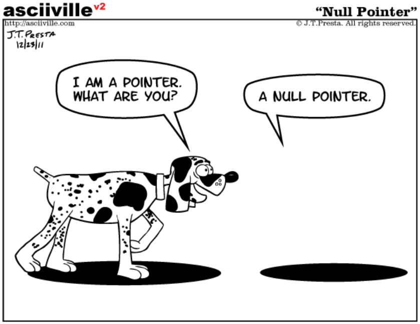
Null pointer

- pointer set to a special value: **0**; there is no memory location 0 invalid pointer.
- to set a pointer to null, we can use any of the following: 0, NULL, nullptr

```
#define NULL 0
//since C++11
#define NULL nullptr
```

The following are equivalent:

```
float *x = 0;
float *x = NULL;
float *x = nullptr;
float *x {nullptr};
```



Null pointer

Why would we use the null pointer?

- initialize a pointer variable when that pointer; variable isn't assigned any valid memory address;
- pass a null pointer to a function argument when we don't want to pass any valid memory address;
- we should check for a null pointer before accessing any pointer variable.

Null pointer

"In C++, the definition of NULL is 0, so there is only an aesthetic difference. I prefer to avoid macros, so I use 0. Another problem with NULL is that people sometimes mistakenly believe that it is different from 0 and/or not an integer. In pre-standard code, NULL was/is sometimes defined to something unsuitable and therefore had/has to be avoided. That's less common these days.
If you have to name the null pointer, call it nullptr; that's what it's called in C++11. Then, "nullptr" will be a keyword."

— Bjarne Stroustrup, `http://www.stroustrup.com/bs_faq2.html#null`

Dangling pointer

- a pointer that does not point to valid data: the data might have been erased from memory;
- the memory pointed to has undefined contents;
- **Dereferencing such a pointer will lead to undefined behaviour!**

Void pointers

- a **void pointer** is a pointer that has no associated *data type* with it;
- it can hold the address of a variable of any type; used in malloc, calloc;
- used to implement generic functions (e.g. qsort);
- void pointer cannot be dereferenced;
- the C standard does not allow pointer operations on void pointers.

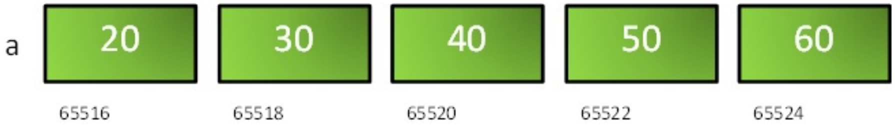
Constant pointers

- *Changeable pointer to constant data* - the pointed value cannot be changed, but the pointer can be changed to point to a different constant value.
  ```
  const int * p;
  ```
- *Constant pointer to changeable data* - the pointed value can be changed through this pointer, but the pointer cannot be changed to point to a different memory location.
  ```
  int * const p;
  ```
- *Constant pointer to constant data.*
  ```
  const int* const p;
  ```

Pointers and arrays

- Arrays can be seen as pointers to the first element of the array.
- Statically allocated arrays are *constant* pointers.
- When passed as function parameters, arrays are passed "by reference".

```
short arr[5] = {20, 30, 40, 50, 60};
```



Pointers and arrays

int a[5] = {1, 2, 3, 4, 5};

What is the effect of the following instructions ?

for(int i = 0; i < 5; i++)

- printf("%d", a[i]);
- printf("%d", *(a+i));
- printf("%d", a+i);
- printf("%d, *a);
- a++;

Pointers to functions

- a **function pointer** is a pointer which points to an address of a function;
- can be used for *dynamic (late) binding* (the function to use is decided at runtime, instead of compile time);
- functions can be used as parameters for other functions;
- do not need memory allocation/deallocation.

Definition

<return type> (∗ <name>)(<parameter types>)

E.g.:

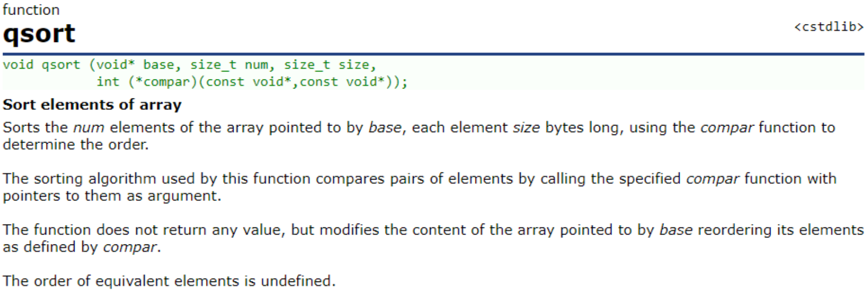
```
double (* operation) ( double , double ) ;

void *(*func)(double *);
```

Pointers to functions

- **Declaration:**
  void (∗func_pointer)(int);
- **Initialization:**
  func_pointer = my_int_func; // where my void my_int_func(int v) is a function you wrote , **or**
  func_pointer = my_int_func
- **Invocation:**
  func_pointer(arg), **or**
  (*func_pointer)(arg)

Pointers to functions - examples

- *callback* functions: void create_button(int x, int y, const char ∗text, void (∗callback_func));
- qsort - `http://www.cplusplus.com/reference/cstdlib/qsort/`



function
qsort
<cstdlib>

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

Sort elements of array

Sorts the *num* elements of the array pointed to by *base*, each element *size* bytes long, using the *compar* function to determine the order.

The sorting algorithm used by this function compares pairs of elements by calling the specified *compar* function with pointers to them as argument.

The function does not return any value, but modifies the content of the array pointed to by *base* reordering its elements as defined by *compar*.

The order of equivalent elements is undefined.

Pointer arithmetic

The following arithmetic operators can be applied on pointers:

- increment (++) and decrement (- -);
- addition of a number (+) and subtraction of a number (-);

$$new\_address = current\_address - (number * sizeof(data\_type)) \quad (1)$$

- subtraction of two pointers;

$$address2 - address1 = (address2 - address1)/sizeof(data\_type) \quad (2)$$

- pointer comparison ($<, >, >=, <=, ==$).

Are pointers really that hard?

```
void (*(*f[])())()
```

Let’s decipher it!

Are pointers really that hard?

void (*(*f[])())()

f - f

f[] is an array

*f[] of pointers

(*f[])() to functions

*(*f[])() returning pointers

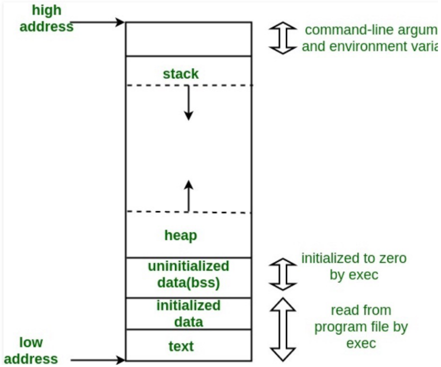
(*(*f[])())() to functions

void (*(*f[])())(); returning void

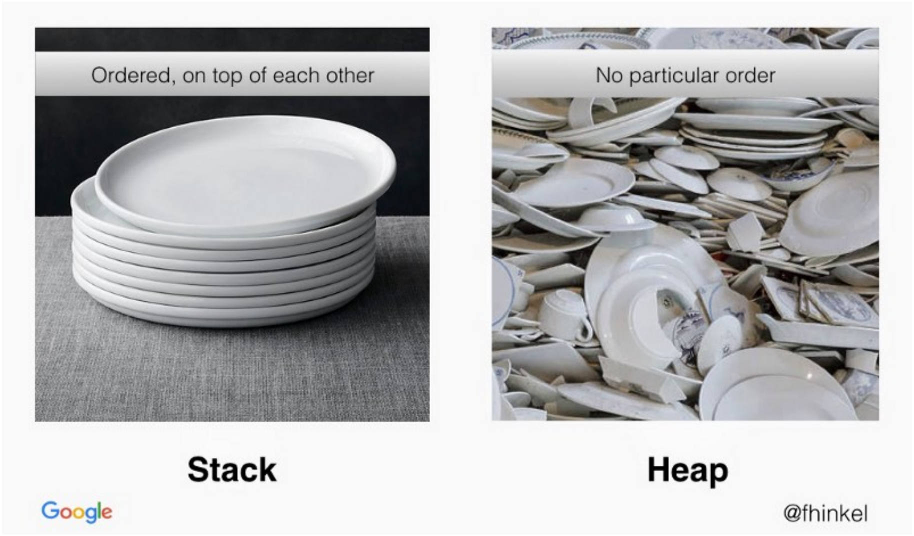
Memory segments

Memory assigned to a program can be divided into 5 parts:

- *initialized data segment*: global and static initialized variables;
- *uninitialized data segment (bss)*: all global and static variables that are initialized to zero or do not have explicit initialization;
- *code/text segment* - the actual machine instructions of our program (read-only);
- *stack*;
- *heap*.



Stack vs. the heap



Stack

- continuous block of memory consisting of several stack frames;
- LIFO (*Last In, First Out*) order ;
- *stack frame/activation record* - collection of all data on the stack associated with one function call: return address, function arguments, local variables $\rightarrow$ stack variables are local in nature;
- a stack frame is constructed and pushed onto the stack for each function call; at the end of the function the stack frame is popped out of the stack (local variables destroyed) $\rightarrow$ memory managed for you;
- stack pointer;
- limited size defined (around 2Mb) (Linux: ulimit -a);

Heap

- can be controlled dynamically by the programmer;
- no size restrictions (except the obvious ones imposed by the size of the RAM);
- allocation: free list + additional book-keeping;
- the data in the heap must be managed by the programmer, it is not automatically managed for you;
- variables created on the heap are accessible by any function, anywhere in your program $\rightarrow$ heap variables are essentially global in scope.

Static vs. dynamic allocation

Memory can be allocated in two ways:

- *Statically (compile time)*
  - by declaring variables;
  - the size must be known at compile time;
  - there is no control over the lifetime of variables.
- *Dynamically (”on the fly”, during runtime)*
  - on the heap;
  - the size does not have to be known in advance by the compiler;
  - is achieved using pointers;
  - the programmer controls the size and lifetime of the variables.

Dynamic allocation and de-allocation I

- **C** - use the functions defined in **stdlib.h**:
  - malloc - finds a specified size of free memory and returns a **void** pointer to it (memory is uninitialised).
  - calloc - allocates space for an array of elements, initializes them to zero and then returns a void pointer to the memory.
  - realloc - reallocates the given area of memory (either by expanding or contracting or by allocating a new memory block).
  - free - releases previously allocated memory.

Dynamic allocation and de-allocation II

- **C++** - new and delete operators.
- **new T**
  - memory is allocated to store a value of type T;
  - it returns the address of the memory location;
  - the return value has type T*.
- **delete p**
  - deallocates memory that was previously allocated using new;
  - precondition: p is of type T*;
  - the memory space allocated to the variable p is freed.

Stack vs. Heap - recap

Stack

- very fast access
- don't have to explicitly de-allocate variables
- space is managed efficiently by CPU, memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time
- you must manage memory
- variables can be resized using realloc()

Stack vs Heap - when to use which?

Stack

- is the preferred way to allocate variables : it's easier, less memory errors and faster
- functions with variables that only persist within the lifetime of the function

Heap

- for variables that can change size dynamically
- for variables you need to persist throughout the lifetime of your application
- when you need to allocate a large block of memory (e.g. store a texture of 15Mb)

Memory errors

- Memory leaks - memory is allocated, but not released (Visual Studio: <**crtdbg.h**> and **CrtDumpMemoryLeaks();**).
- Invalid memory access - unallocated or deallocated memory is accessed.
- Mismatched Allocation/Deallocation - deallocation is attempted with a function that is not the logical counterpart of the allocation function used.
- Freeing memory that was never allocated.
- Repeated frees - freeing memory which has already been freed.

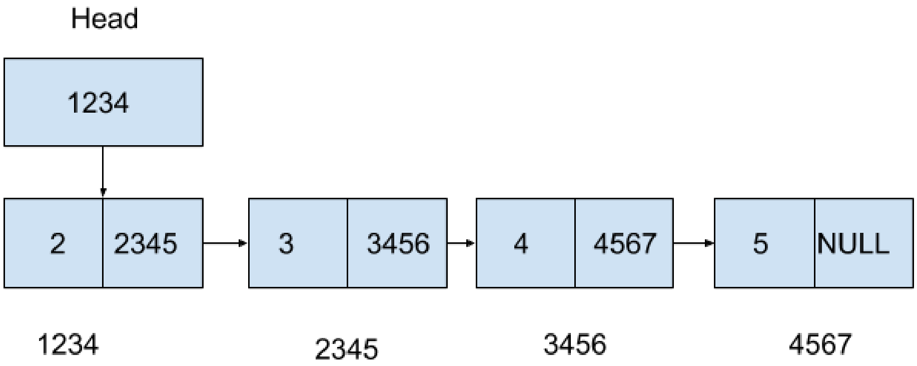
Why do we need pointers?

1. the only way to dynamically allocate memory in C/C++ - most common use;
2. arrays are implemented using pointers; we can use pointers to iterate through an array (instead of [] operator);
3. they allow us to pass a large amount of data to a function without copying the data;
4. pass a function as a parameter to another function;
5. used to achieved polymorphism (covered later in the course);
6. in advanced data structures: they can be used to have one struct/class point at another struct/class to form a chain; e.g. linked lists, trees;
7. when we need "pass by reference" in C.

Pointers in advanced data structures

Pointers used for defining linked lists:

- each element contains a *value* and the *address* of the next element
- Head - pointer to the first element of the list



Modular programming I

A **module** is collection of functions and variables that implements a well defined functionality.

Goals:

- separate the interface from the implementation; function prototypes (function declarations) are grouped into a separate file called *header file*.
- hide the implementation details.

Modular programming II

The code of a C/C++ program is split into several source files .h and .c/.cpp:

- .h files - contain the function declarations (the interfaces);
- .c/.cpp files - contain the function implementations.

Ensures **abstraction** - highlights the essential features of something while ignoring its details.

The header file is a **contract** between the developer and the client of the library that describes the data structures and states the arguments and return values for function calls.

The compiler enforces the contract by requiring the declarations for all structures and functions before they are used (this is why the header file must be included).

Modular programming - advantages

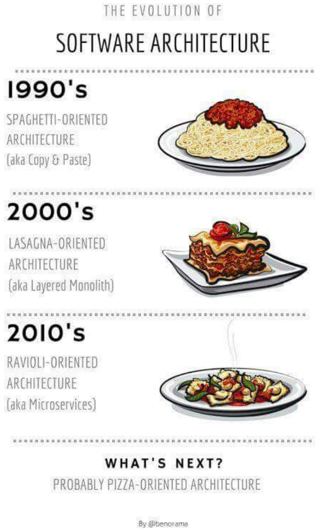
- code reuse;
- programs can be designed more easily a small team deals with only a small part of the entire code;
- code is short, simple and easy to understand;
- the .c/.cpp files can be compiled separately (for error checking and testing);
- errors can easily be identified, as they are localized to a subroutine or function;
- the scoping of variables can easily be controlled.

Module design guidelines

- Separate the interface from the implementation:
  - *header* - type declarations and function prototypes;
  - *source files* - actual definition of the functions;
- Include a short description of the module (comment);
- *Cohesion*: module has a *single responsibility* and the functions within the module are related;
- Self contained headers: they include all the modules on which they depend (no less, no more);
- Layered architecture
  - Layers: model, validation, repository, controller, ui.
  - Manage dependencies: each layer depends only on the "previous" layer.
- Protect against multiple inclusion (include guards).

Layered architecture

- clarity and separation for different functional blocks in systems;
- organize the system as a set of layers each of each provides a set of services to the layer "above" and serving as a client to the layer "below";
- usually, elements from a layer "see" only elements for the same layers and from the layers below;
- advantages: *independence*, *re-usability*, *flexibility*.


THE EVOLUTION OF
SOFTWARE ARCHITECTURE
1990's
SPAGHETTI-ORIENTED ARCHITECTURE (aka Copy & Paste)
2000's
LASAGNA-ORIENTED ARCHITECTURE (aka Layered Monolith)
2010's
RAVIOLI-ORIENTED ARCHITECTURE (aka Microservices)
WHAT'S NEXT?
PROBABLY PIZZA-ORIENTED ARCHITECTURE
By @benorama

Libraries

- A **library** is a set of functions, exposed for use by other programs.
- Libraries are generally distributed as:
    - a header file (.h, .hpp) containing the function prototypes
    - binary file (.dll,.lib, .so, .a) containing the compiled implementation.
    - the source code does not need to be shared
- Static vs dynamic linking
    - *Static linking* - happens at compile time and the .lib or .a binary file is completely "included" in the executable
    - *Dynamic linking* - includes only the information needed at run time to locate and load the .dll or .so that contains a data item or function.

Preprocessor directives I

- lines in code preceded by a hash sign (#).
- are executed by the preprocessor, before compilation.

Directive	**Meaning**
#include *header file*	opens the header file and insert its contents in the current file; <>(search in systems directory) vs "" (search in current directory, then system directories)
#define *identifier replacement*	any occurrence of *identifier* in the code is replaced by *replacement*

Preprocessor directives II

Directive	**Meaning**
#ifdef *macro*, ... ,#endif	the code between the two directives is compiled only if the specified macro has been defined.
#ifndef *macro*, ... ,#endif	the code between the two directives is compiled only if the specified macro has been **not** defined; together with #define can be used as *include guard*
#pragma once	the current file will be included only once in a single compilation (not standard, but widely supported)

Abstract data types (ADT)

An ADT is a data type which:

- exports a name (type);
- defines the domain of possible values;
- establishes an interface to work with objects of this type (operations);
- restricts the access to the object components through the operations defined in its interface;
- hides the implementation.

Any program entity that satisfies the requirements from the ADT definition is considered to be an implementation of the ADT.

ADT implementation in C/C++:

- interface - header file (.h, .hpp);
- implementation - source file (.c/.cpp).