Basics of C
Lecture 2

Summer 2018
Innopolis University
Eugene Zouev

Last time:

- C memory model
- Typical program structure
- Declarations & types

Today:

- Pointers
- Global/local & dynamic objects
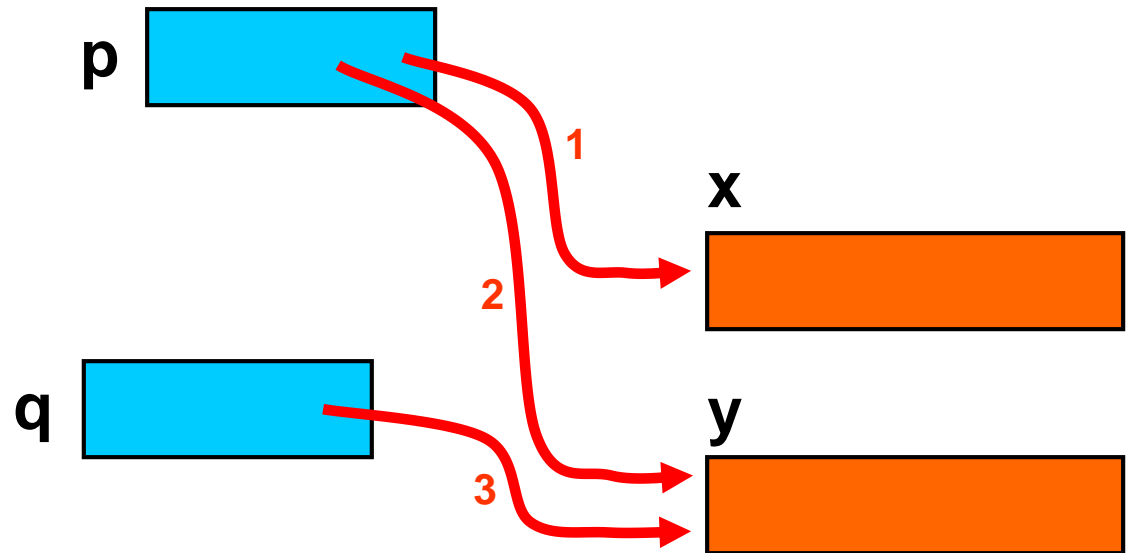- Arrays

Pointers

1. The Notion of Pointer:
An <u>object</u> containing <u>an address</u> of some other object

```
int x;
int* p;
...
p = &x;    1
```

Unary "address-of" operator

p

x

q

y

1
2
3

```
int y;
...
p = &y;  2
```

```
int* q;
...
q = p;   3
```

Pointers

2. Pointer types

$$T*\ \ p;$$

Declaration of an object of a pointer type, where T denotes a type pointed

Examples:

- **Pointers to (simple) variables;**   `int* pv;`

- **Pointers to objects of struct types;** `struct S* ps;`

- **Pointers to arrays;**   `int pa[10];`

- **Pointers to functions;**   `int (*pf)(int);`

- **Pointers to pointers;**   `int** p;`

- **Pointers to values of any type**   `void* p;`

Pointers

3. Operators on pointers

&object

Taking address of object:
Unary prefix operator

```
int x;
int* p;

...
p = &x;
```

*pointer

Dereferencing: Getting object pointed to by the "pointer"
Unary prefix operator

Notice
The same token * is used for two different purposes:
a) for specifying a pointer type
b) as dereferencing operator.

```
int x;
int* p = &x;

...
*p = 777;      // x is 777
int z = *p+1;  // z is 778
```

Pointers

4. Operators on pointers: <u>pointer arithmetic</u>

> Later today ☺

Pointers

5. Pointers & "Constness"

`T* ptr1;`

> Pointer to an object of type *T*; no restrictions on access to the object pointed to by `ptr1`

`const T* ptr2;`

> Pointer to a **constant object** of type *T*;
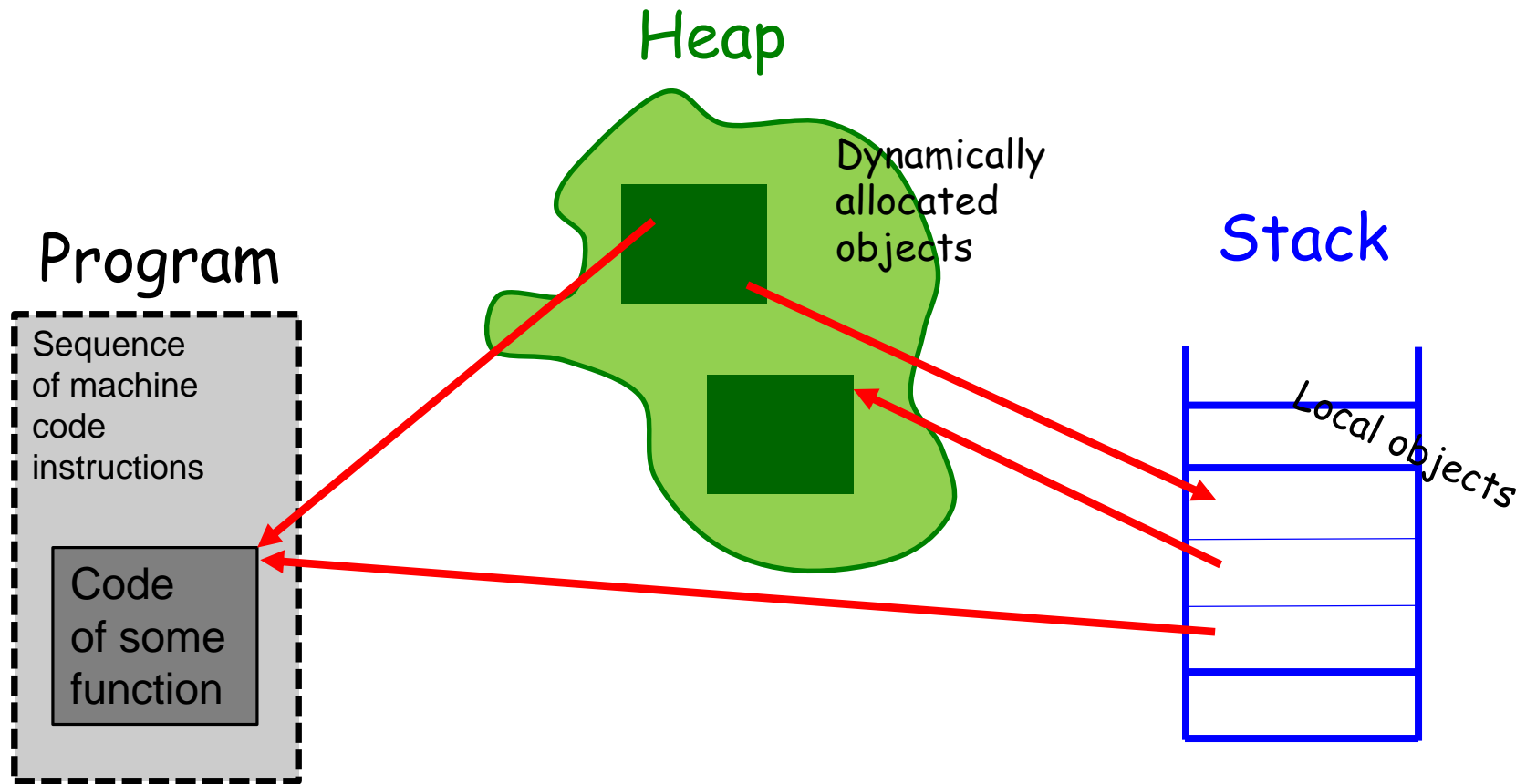> cannot use `ptr2` to modify object pointed to by it

`T*const ptr3 = &v;`

> **Constant pointer** to an object of type *T*; cannot modify the value of `ptr3` (it must be initialized)

`const T*const ptr4 = &pc;`

> **Constant pointer to a constant object** of type *T*; cannot modify the value of `ptr4` (it must be initialized) and cannot use it to modify object pointed to by it

Pointers & The C Memory Model

Heap

Dynamically allocated objects

Program

Sequence of machine code instructions

Code of some function

Stack

Local objects

Program cannot modify this memory (self-modified programs are not allowed)

The discipline of using heap is defined by program **dynamic semantics**, i.e., at runtime (while program execution)

The discipline of using stack is defined by the (static) **program structure**

Global, Local & Dynamic Objects

Global objects **Stack**

   Are created on program's start and exist ("live") until program is completed.
   Live in the **global scope**.
   Are accessible ("visible") within the translation unit they are declared in, OR within the whole program.

Local objects **Stack**

   Are created when a function is invoked or when the control flow enters a block, and disappear on return or on exit from the block.

Dynamic objects **Heap**

   Are created and destroyed on arbitrary moments while program execution, following the program logic.

Global, Local & Dynamic Objects

How global & local objects are created?
- **By their declarations**

How dynamic objects are created?
- **Using special standard functions from the C library**

Globals & locals: example

```
int x;
int* ptr;

void f(int p)
{
    int* local = &x;
    if ( p > 0 )
    {
        float m = ;
        ...
    }
}
```

x & ptr are **global objects**; they are created on the program's start and exist until its end

p & local are **local objects**; they are created when f function is invoked and disappear on return form f

m is the **local object**; it is created when the control flow enters the then-branch of if and disappears on return from this block

We will consider this mechanism in details on the last lecture

Dynamic Objects

How dynamic objects are created (and destroyed)?

- Using special standard functions from the C library

```
void* malloc ( int size )
{

    ...
    Allocation algorithm
    ...
}
```

```
void free ( void* ptr )
{

    ...
    Deallocation algorithm
    ...
}
```

- Specification is a bit <u>simplified</u>.
- <u>The function allocates space for an object whose size (in bytes) is passed via the parameter</u>.
- The function returns a pointer to the memory allocated.
- The pointer is "untyped" (void*).
- There are more allocation functions in the library.

Library Organization

Each translation unit is represented
by two source files:
- with forward declarations ("interface");
- with full declarations ("implementation").

To remind...

```
void* malloc(int size);
void free(void* ptr);
...
And many other function
headers ("prototypes")
...
```

stdlib.h

```
void* malloc(int size)
{
    ...
    Implementation
    ...
}
...
And implementations
of many other standard
functions
...
```

stdlib.c

Precompiled

Dynamic Objects

How dynamic objects are created?
- **Using special standard functions from the C library**

Example

```
#include <stdlib.h>

struct S { int a, b; }

void* ptr = malloc(sizeof(struct S));

struct S* s = (struct S*)ptr;

s->a = 5;
...
```

In order to use `malloc`, we should add its header

This is struct type declaration

Here, we dynamically allocate memory suitable to keep objects of type `struct S`…

…and **convert** the void pointer type to the type of pointer to `struct S`.

After that, we can use `s` to get access to elements of `struct S`.

Arrays

`T A[size];`

`T` is the type of array elements

`A` is the array identifier

`size` specifies the number of array elements; this is an expression of an integer type

In general, `size` should be a constant; however, in some cases, `size` can be omitted, or be replaced for * or be a non-constant.
Will see some cases later.

```
int Array[10];

const int x = 7;
void* Ptrs[x*2+5];

int Matrix[10][100];
```

The only operator on arrays:
- **Getting access to an element**

```
int el5 = Array[5];

Array[7] = 7;
```

Arrays & Pointers

```
int Array[10];
```

Array →

Element 0
Element 1
Element 2
...
Element 9

By definition, array name is treated as a **pointer** to the first array element.

To be more precise, array name is a **constant pointer**

Therefore, these two constructs are semantically identical:
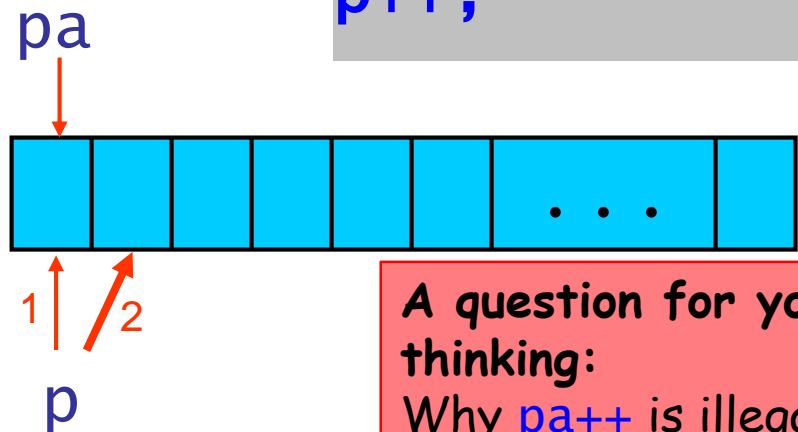
```
Array[0]
```

```
*Array
```

Pointers

4. Operators on pointers: <u>pointer arithmetic</u>

```
pointer+i
pointer-i
pointer++
pointer--
ptr1-ptr2
```

```
int pa[10];
int p = pa;          1

p++;                 2
```

pa

1   2

p

A question for your home thinking:
Why pa++ is illegal?

```
T* p;
p+i // the same as
    // (T*)((char*)p+sizeof(T)*i)
```

Arrays & Pointers (again)

`int Array[10];`

As we saw, these two constructs are semantically identical:

`Array[0]`   `*Array`

Going on: these two constructs are also semantically identical:

`Array[1]`   `*(Array+1)`

General form:

`Array[N]` ⬅➡ `*(Array+N)`

Array ➡ Element 0
| Element 1
| Element 2
| ...
| Element 9

Arrays & Pointers (again)

Interesting conclusion:

`Array[N]` and `*(Array+N)` are identical. But the second if the same as `*(N+Array)`, isn't it?

Therefore, `Array[N]` and `N[Array]` might be also identical? ☺

Check this at home!

C Standard:

6.5.2.1 Array subscripting

…

Semantics

2 A postfix expression followed by an expression in square brackets **[]** is a subscripted designation of an element of an array object. The definition of the subscript operator **[]** is that **E1[E2]** is identical to **(*((E1)+(E2)))**. Because of the conversion rules that apply to the binary **+** operator, if **E1** is an array object (equivalently, a pointer to the initial element of an array object) and **E2** is an integer, **E1[E2]** designates the **E2**-th element of **E1** (counting from zero).

C++ Standard:

…Therefore, despite its asymmetric appearance, subscripting is a commutative operation…

Pointers in C++

Breaking News:

Pointers are to be removed from the C++2023!!!

«Комитет по стандартизации языка в Джексонвиле две недели назад принял решение о том, что **указатели будут объявлены устаревшими** в C++20 и с **большой долей вероятности будут удалены** из C++23.»

https://habrahabr.ru/post/352570/

1 апреля

Problems with C pointers

`T obj;`

`T* ptr;`

obj

ptr ⟹

obj

The problems with pointers come from its low-level nature…

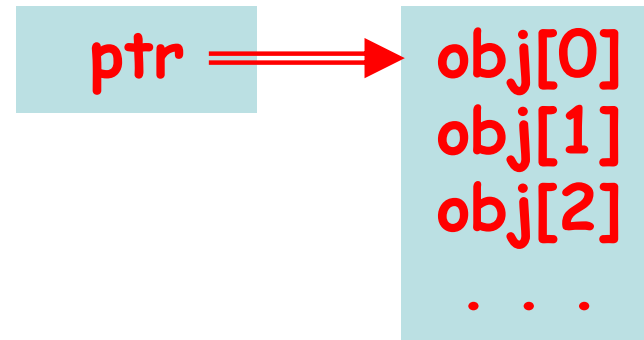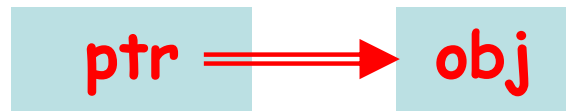*Exactly the same problems exist for **C++ pointers** as well!!*

Problems with C pointers

Scott Meyer:
6 kinds of problems with pointers

Problems 1 & 4:
A pointer can point either to a **single object**, or to an **array**. – And there's no way to distinguish betw these.

ptr ⟶ obj   ptr ⟶ obj[0]
                                    obj[1]
                                    obj[2]

                                    . . .

```
int x;
int A1[10];
int* A2 = &x;
int* A = cond ? A1 : A2;

int res = A[5]; // ???????
```

Problems with C pointers

Problem 2:
A declaration of a pointer tells nothing whether we must destroy the object pointed after the work is completed.

Or: does the pointer **owns the object** pointed?

```cpp
void fun(T* ptr)
{
  // Some work with an object
  // pointed to by ptr.

  // Should we destroy the object
  // before return?
  return;
}
```

Problems with C pointers

Problem 3:
Even if we know that we should destroy the object pointed to by a pointer – in general we don't know **how to do that**!

I.e., either just to apply **delete** or use some special function for that?

```
void fun(T* ptr)
{
  // Some work with an object
  // pointed to by ptr.

  // We know that fun should destroy
  // the object before return.
  free(ptr);
  return;
}
```

…or perhaps:
```
myDealloc(ptr)
```

Problems with C pointers

Problem 5 (a consequence from problem 2): Even if we **own** the object pointed to by a pointer it's hard (or even impossible) provide **exactly one** act of destroy.

I.e., it's quite easy either to leave the object live, or to try to destroy it twice or more.

```
void lib_fun(T* ptr)
{
  // This library performs some
  // actions on the object passed
  // as parameter.

  // The function doesn't destroy
  // the object before return.
  return;
}
```

```
void user_fun()
{
  T* ptr = malloc(sizeof(T));
  // The function owns its object.

  lib_fun(ptr);
  // Should we destroy the object
  // before return, OR lib_fun has
  // already destroyed it??
  return;
}
```
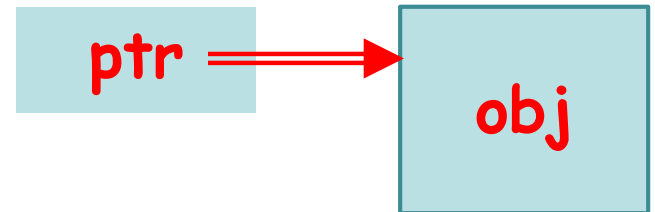
Problems with C pointers

Problem 6:
There is no way to check whether a pointer actually points to a real object.

Or: to check whether the pointer is "dangling pointer".

```
T* ptr = (T*)malloc(sizeof(T));
...
if ( condition ) free(ptr);
...
// Long code…
...
// How to know whether ptr
// still points to an object?
...
```

ptr → **obj**

ptr → **?**

Problems with C pointers

Problem 7 (in addition to Scott Meyers' ☺):
There is no way to ensure that an object gets destroyed when the single pointer to it disappears.

```
if ( condition )
{
    T* ptr = (T*)malloc(sizeof(T));
    ...
    // No free(ptr)
}
...        ⟵————  Here, ptr doesn't exist,
                  but the object itself still does:
                  memory leak
```