Basics of C

Lecture 3

Summer 2018

Innopolis University
Eugene Zouev

Previous classes:

- C memory model
- Typical program structure
- Declarations & types
- Pointers, arrays
- Global/local & dynamic objects

Today:

- Pointers again ☺
- Static/auto & global/local objects
- Syntax of declarations

Problems with C pointers

From the previous lecture

```
T obj;
```

```
T* ptr;
```

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

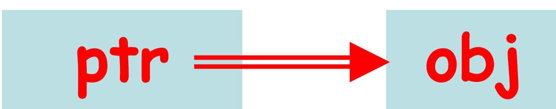
Exactly the same problems exist for ***C++ pointers*** as well!!

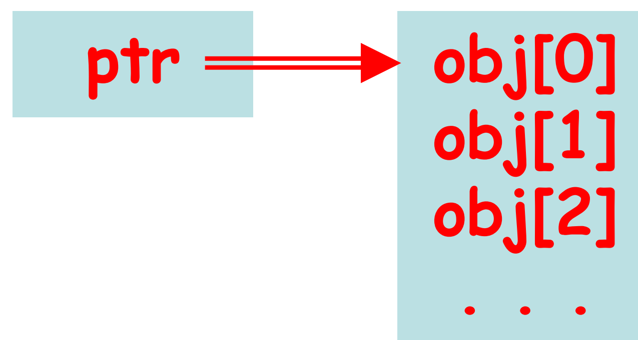
Problems with C pointers

From the previous lecture

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.






```
int x;

int A1[10];
int* A2 = &x;

int* A = cond ? A1 : A2;

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

Problems with C pointers

From the previous lecture

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?

```
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

From the previous lecture

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 `free()` 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

From the previous lecture

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;
}
```

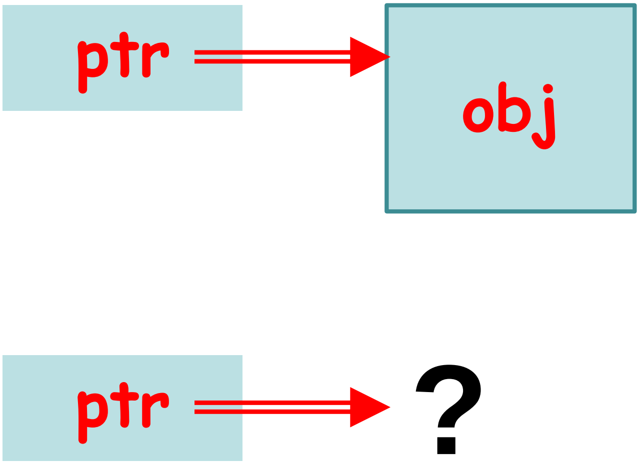
Problems with C pointers

From the previous lecture

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?
...
```



Problems with C pointers

From the previous lecture

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**

Problems with C pointers

Example: pointers & scopes

```
int* p;

void f() {
    int A[10];
    p = A+2;
}

void main() {
    f();
    *p = 777;
}
```

p is the **global object**; it's created on the program's start and exist until its end

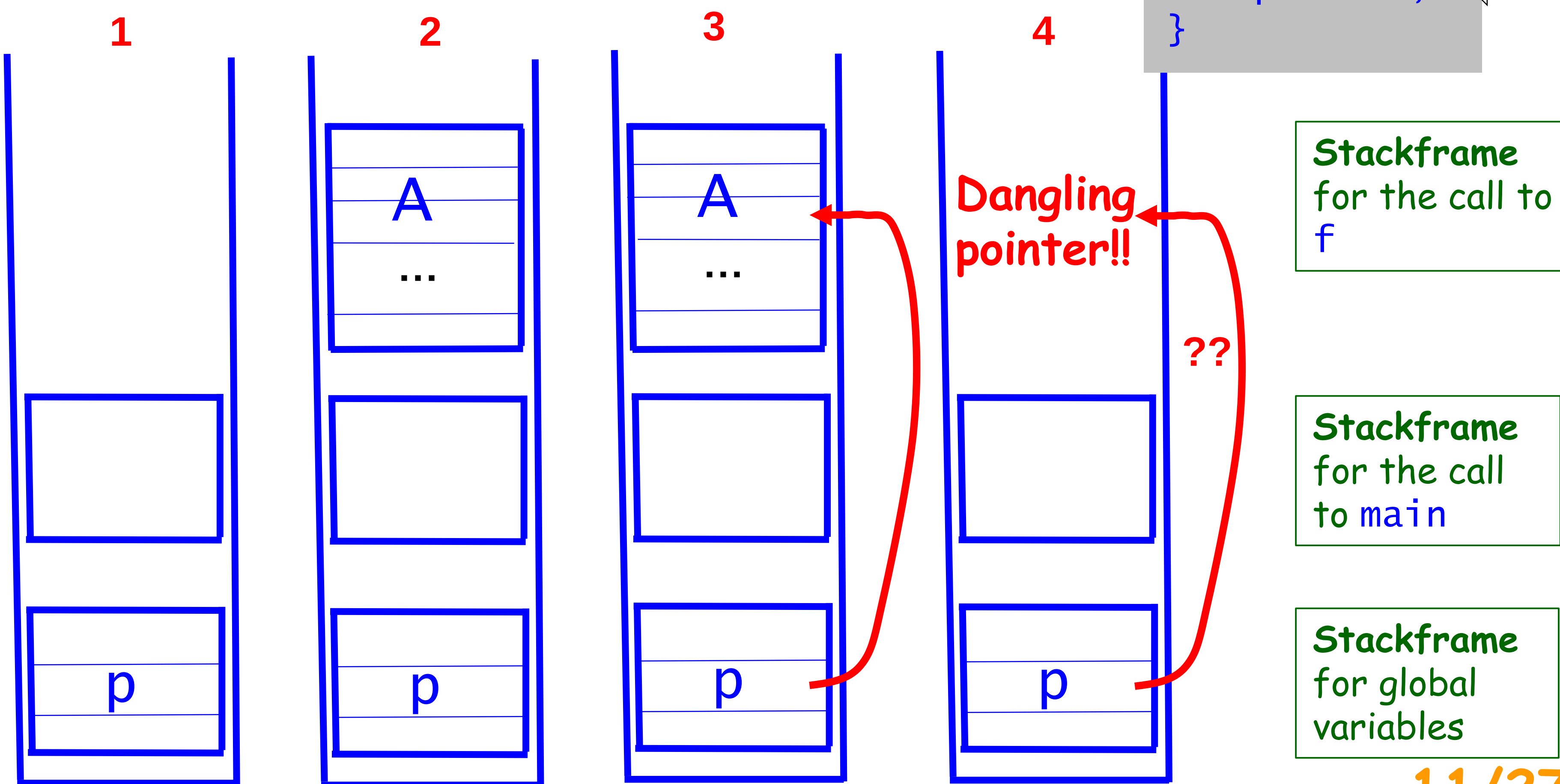
A is the **local object**; it's created on the f's start and exists **until exit from** f

What the hell will happen here?!

Problems with C pointers

```
int* p;
void f() {
    int A[10];
    p = A+2;
}
void main() {
    f();
    *p = 777;
}
```

Execution Stack



Problems with C pointers

Example: dynamic objects, pointers & scopes

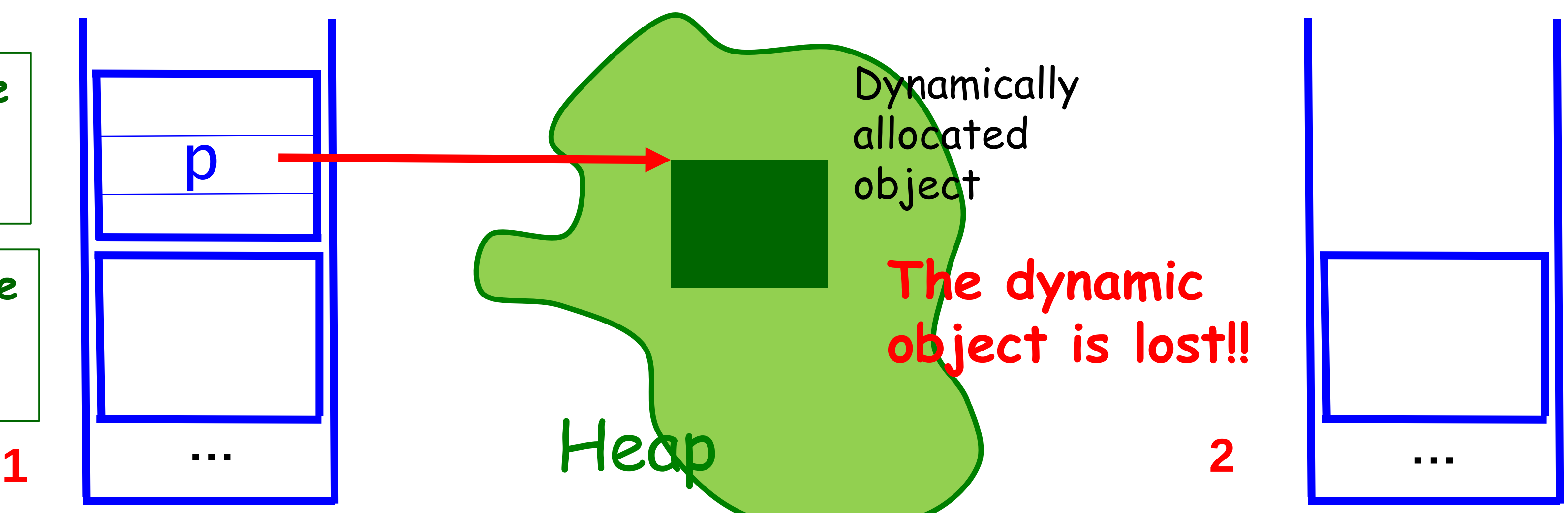
```
void f() {
    int* p = (int*)malloc(10);
}

void main() {
    f();
    ...
}
```

p is the **local object**; it lives only within the f function

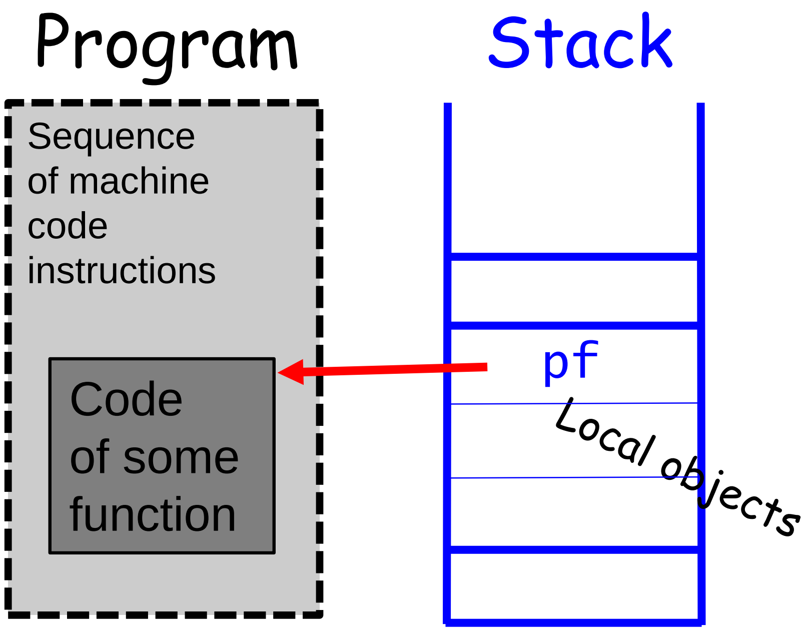
The unnamed object created by malloc() is **dynamic object**; it doesn't follow the scoping rules!

Stackframe for the call to f

Stackframe for the call to main



Pointers to Functions




```
void f(int p)
{
    ...
}
...
void (*pf)(int) = f;
...
void main()
{
    f(1);  // call to f
    pf(1); // call to f!
}
```

Automatic & static objects

```
void f() {
    int x = 0;
    ...
    x += 1;
    printf("%d", x);
}

void main() {
    for(int i=1; i<=100; i++)
        f();
}
```

x is the **local object**
- it "belongs" to the f function;
- it is **available** only from within the f function;
- it is created and gets initialized **each time** the f function is invoked

The loop prints the same value 1 on each iteration.

The x variable is often called as **automatic local variable.**

```
...
auto int x = 0;
...
```

Automatic & static objects

```
void f() {
    static int x = 0;
    ...
    x += 1;
    printf("%d", x);
}

void main() {
    for(int i=1; i<=100; i++)
        f();
}
```

x is still the **local object**
- it "belongs" to the f function;
- it is **available** only from within the f function;
- it is created and gets initialized **only once**: before the very first call to the function it belongs to.

The loop prints values 1,2,3,… on each iteration.

The x variable is often called as **static local variable.**

Algol-60:
Own variables

Automatic & static objects

Example: Fibonacci numbers

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-2) + Fib(n-1)

```
long long Fib(int N)
{
    if ( N == 0 || N == 1 ) return N;
    return Fib(N-2) + Fib(N-1);
}
```

The stateless function

```
long long Fib() {
    static long long first = 0;
    static long long second = 1;
    long long out = first + second;
    first = second;
    second = out;
    return out;
}
```

The function with its own state, OR

Finite automat

Automatic/static, globals/locals

```
int a;
static char b;

void f()
{
    auto float c;
    double d;
    static int e;
}
```

a is the **global non-static object**
- it "belongs" to the whole program;
- it is available throughout the program;
- it is created only once: before the program starts.

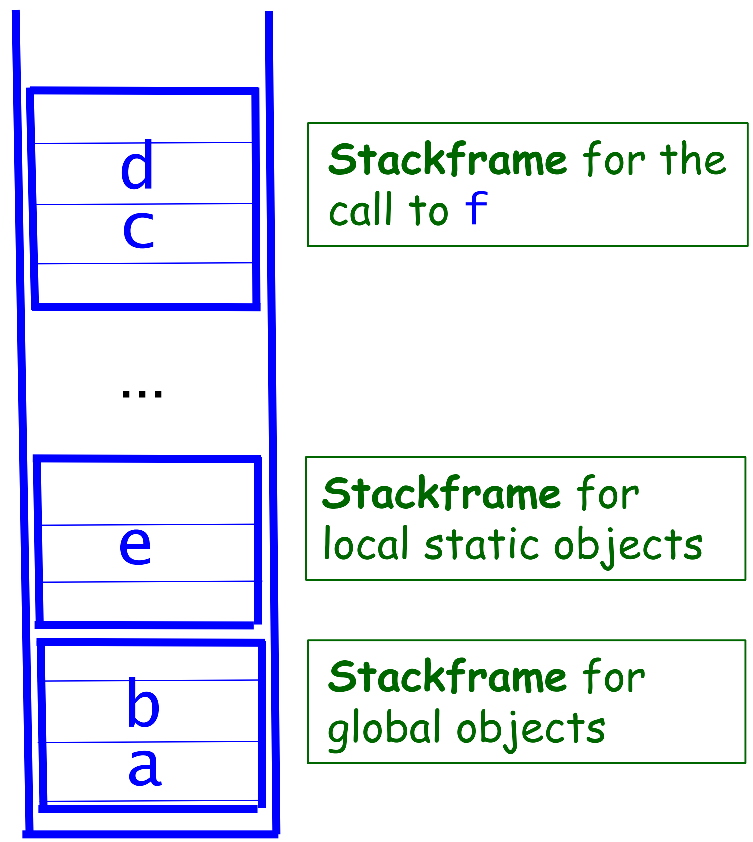
b is the **global static object**
- it "belongs" to the whole program;
- it is available **only from within the translation unit** it belongs to;
- it is created only once: before the program starts.

c and d are **automatic local objects**
- they "belong" to the function in which they are declared;
- they are available only from within the function (i.e., they are local to the function);
- they are created each time the function only once: before the program starts.

e is the **local static object**
- it "belongs" to the function in which it's created;
- it is available only from within the function;
- it is created only once: before the program starts.

Automatic/static, globals/locals

```
int a;
static char b;

void f()
{
    auto float c;
    double d;
    static int e;
}
```



External declarations

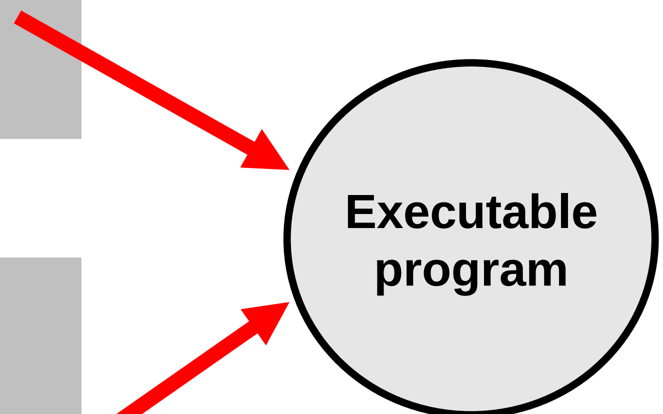
Translation Unit 1

```
...
int a = 777;
...
```

Translation Unit 2

```
...
int a = 999;
...
void main() {
  printf("%d",a);
}
```

The effect: two identical global objects co-exist in the same program



What will the program do? Options:

- Prints 777
- Prints 999
- Compilation error
- Linkage error
- Undefined behavior

Make an experiment ☺

External declarations

Solution

Translation Unit 1

```
...
int a = 777;
...
```

The compiler will allocate memory for the variable a

Translation Unit 2

```
...
extern int a;
...
void main() {
  printf("%d",a);
}
```

The compiler won't allocate memory for the variable a but mark a as allocated somewhere else

…And the linker will resolve the reference to a in printf as the reference to a declared in the Translation unit 1

Declarations: syntax & semantics

Four kinds of information are given in a declaration:

- ***Object storage class***
- ***Entity name***
- ***Entity type***
- ***An object initializer***

All parts are optional ☺

```
static int a = 777;
```

Storage class specifier

Type specifier

Entity (object) name

Object initializer

Declarations: syntax & semantics

The main design idea behind declaration syntax:

Syntax rules for declarations are conceptually similar to <u>expression</u> rules: associativity + precedence + <u>grouping</u>

```
int a = 777;
double* b;
float** c;
```

- Type of a is integer
- Type of b is pointer to double
- Type of c is pointer to pointer to float

```
int* f1();
double* a1[10];
```

- Type of f1 is function without params returning pointer to integer
- Type of a1 is array of pointers to doubles

Declarations: syntax & semantics

```
int *f2(int);
```

- Type of f2 is **function** that accepts one integer parameter and returns pointer to integer

```
int (*f3)(int);
```

- Type of f3 is **pointer to function** that accepts one integer parameter and returns integer

```
double* a2[10];
```

- Type of a2 is **array** of 10 elements whose type is pointer to double

```
double (*a3)[10];
```

- Type of a3 is **pointer to array** of 10 elements whose type is double

Declarations: syntax & semantics

```
int* (f4(int))[10];
```

Is it legal? Check!

- Type of f4 is **function** that accepts one integer parameter and returns array of pointers to integers

```
int (*(a4[10]))(int);
```

- Type of a4 is **array** of 10 elements of type pointer to function with one integer parameter returning integer type

Tasks for your home thinking:
- **Write a small but real program that works with f4 and a4.**
- **Declare an array of pointers to pointers to doubles.**
- **Declare the variable of the type "pointer to a function with no parameters returning a pointer to integer".**

Declarations: syntax & semantics

```
int f(double d, int, float*);
```

- Forward function declaration OR just function declaration, OR function prototype declaration.
- The declaration specifies function that accepts three parameters and returns a result.
- The type of the value returning by the function is integer.
- The types of function parameters are double, integer and pointer to float.
- The fist parameter is specified with its name; the second and third parameters are specified **without names**.

Declarations: syntax & semantics

```
long double f(double*, int, float (*)(double));
```

- Very similar to the previous prototype declaration.
- Three **unnamed** parameters; first two is the typical C way for representing **arrays**; the third one is the **pointer to a function**.

The task:
- **Write a reasonable function (full declaration) that applies the function pointed to by the 3rd parameter to each element of the array – and returns some result.**
- **Declare an array and some function and call function `f` passing array & function to it.**

Typedef Declarations

The way to simplify specifications of complex types

```
int (*(a4[10]))(int);
```

- Type of a4 is **array** of 10 elements of type pointer to function with one integer parameter and return type is integer

```
typedef int (*PtrFun)(int);

PtrFun a4[10];
```

Here, PtrFun is **not an object** but a **synonym of some type** – namely the type "pointer to function".

```
struct S { int a, b; };

struct S s1, s2;
```

```
typedef struct { int a, b; } S;

S s1, s2;
```