Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Tutorial 3

Eugene Zouev
Fall Semester 2020
Innopolis University

Outline

- Pointers: a remark
- Six kinds of problems with pointers
- External declarations
- Type & declaration syntax
- Typedef declaration

Arrays & Pointers (again)

Well, as we know, the two following constructs are semantically the same:

`Array[N]` and `*(Array+N)`

Arrays & Pointers (again)

Well, as we know, the two following constructs are semantically the same:

`Array[N]` and `*(Array+N)`

Interesting conclusion: does it mean that

`*(Array+N)` and `*(N+Array)`

Are also identical?

Arrays & Pointers (again)

Well, as we know, the two following constructs are semantically the same:

`Array[N]` and `*(Array+N)`

Interesting conclusion: does it mean that

`*(Array+N)` and `*(N+Array)`

Are also identical? And, therefore,

`Array[N]` and `N[Array]`

might be also identical? ☺

Check this at home!

Arrays & Pointers (again)

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).

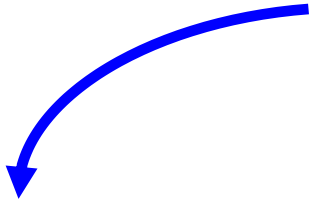
C++ Standard:

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

Pointers in C++

Breaking News:

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



The standardization committee came to a conclusion two weeks ago that pointers in C++ 20 are to be declared **obsolete** and will be highly likely **removed** from C++ 23.

«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/

Breaking News:

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

April 1st

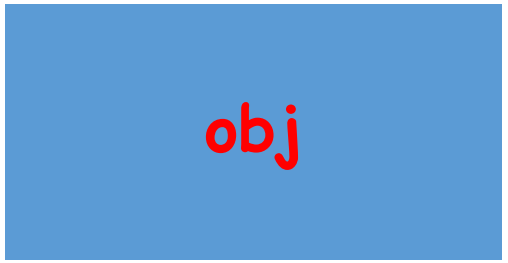
The standardization committee came to a conclusion two weeks ago that pointers in C++ 20 are to be declared **obsolete** and will be highly likely **removed** from C++ 23.

«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.»

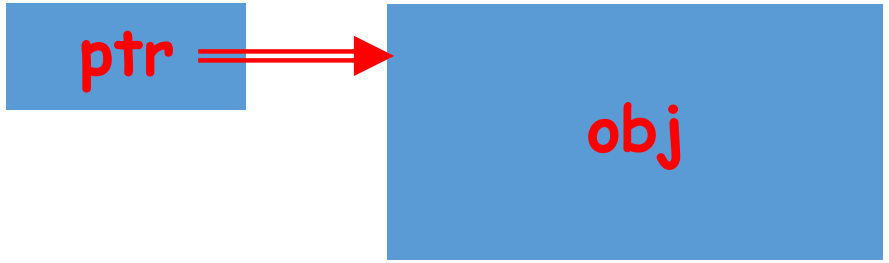
https://habrahabr.ru/post/352570/

Problems with C pointers

```
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

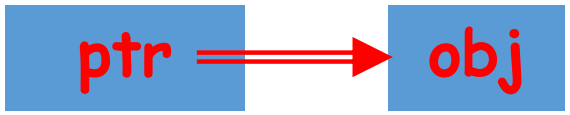
Scott Meyer:
6 kinds of problems with pointers

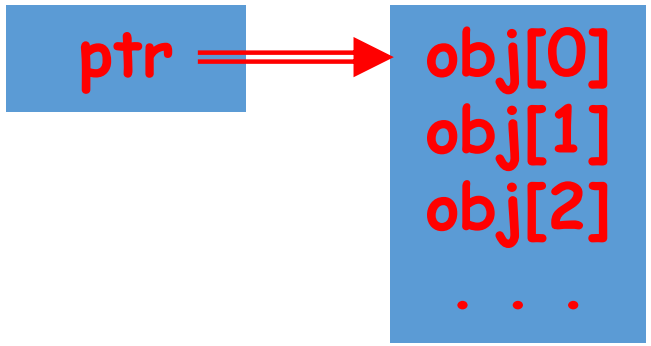
Problems 1 & 4:
A pointer can point either to a **single object**, *or to an* **array**. *– And there's no way to distinguish betw these.*

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.*






```
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?

```
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 `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

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;
}
```

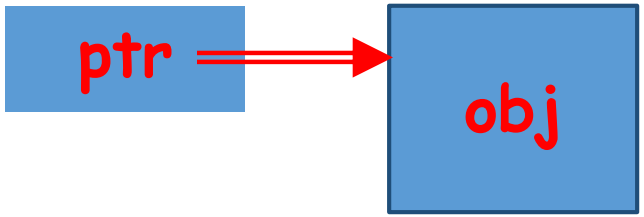
```
void user_fun()
{
  T* ptr = malloc(sizeof(T));
  // The function owns its object.

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

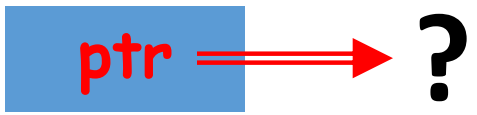
Problems with C pointers

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?
...
```





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

Declarations: syntax & semantics

Four kinds of information are given in a declaration:

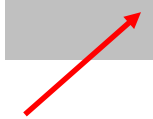
```
static int a = 777;
```

Declarations: syntax & semantics

Four kinds of information are given in a declaration:

- *Object storage class*

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



Storage
class
specifier

Declarations: syntax & semantics

Four kinds of information are given in a declaration:

- ***Object storage class***
- ***Entity name***

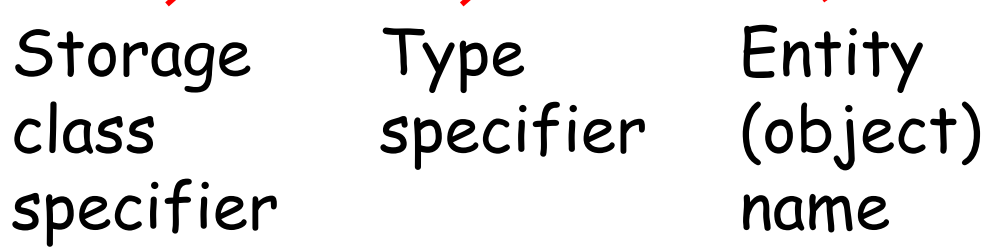
```
static int a = 777;
```

Declarations: syntax & semantics

Four kinds of information are given in a declaration:

- ***Object storage class***
- ***Entity name***
- ***Entity type***

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



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

External declarations

Translation Unit 1

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

Translation Unit 2

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

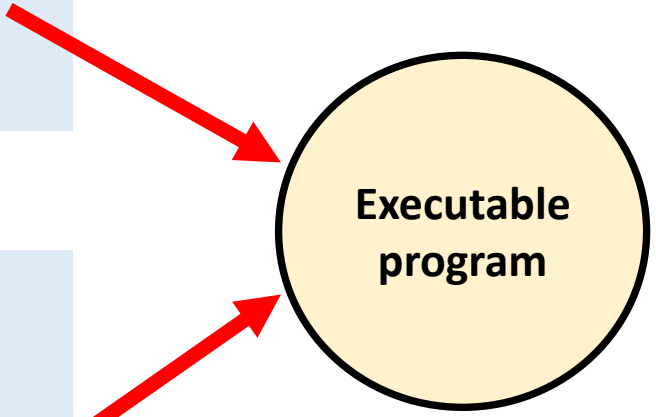
External declarations

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

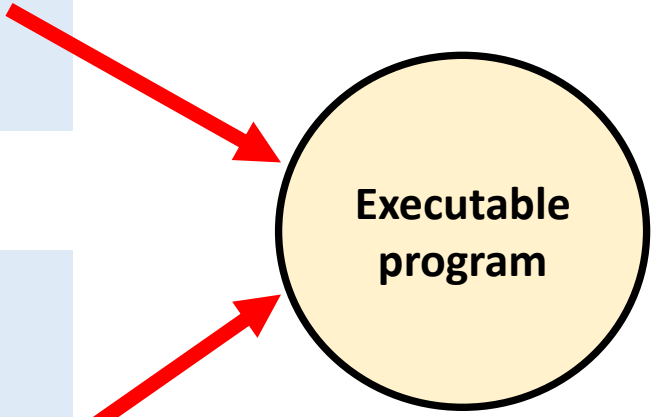
External declarations

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;
...
```

Translation Unit 2

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

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 marks a as allocated somewhere else

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 marks 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

More on external declarations

Are there local (nested) functions in C?

```
void fun()
{
    void nested(int x) { ... }
    ...
    nested(7);
    ...
}
```

More on external declarations

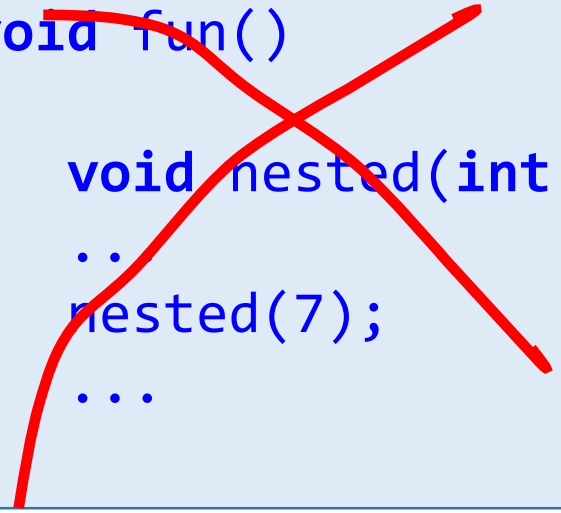
Are there local (nested) functions in C?

```
void fun()
{
    void nested(int x) { ... }
    ...
    nested(7);
    ...
}
```

More on external declarations

Are there local (nested) functions in C?

```
void fun()
{
    void nested(int x) { ... }
    ...
    nested(7);
    ...
}
```

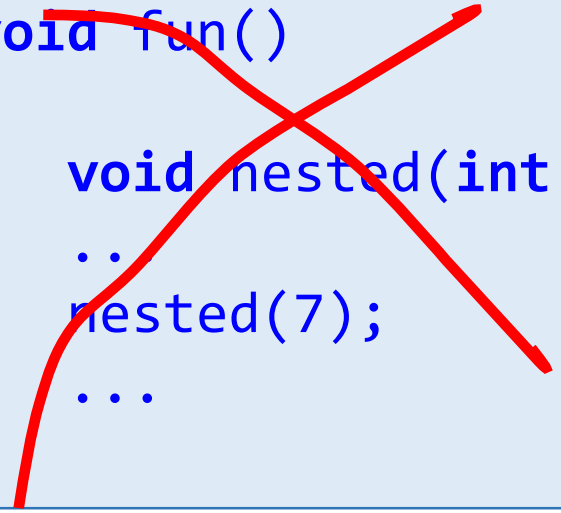


```
void fun()
{
    extern void extra(int x);
    ...
    extra(7);
    ...
}
```

More on external declarations

Are there local (nested) functions in C?

```
void fun()
{
    void nested(int x) { ... }
    ...
    nested(7);
    ...
}
```



```
void fun()
{
    extern void extra(int x);
    ...
    extra(7);
    ...
}
```

- `extra` is declared as a **local entity**.
  The name is known only within its scope.
- `extra` denotes a function.
- The declaration introduces only "forward" function declaration.
- The `extra`'s storage class specifier says that the full definition of extra is provided in some other TU.

Declarations: syntax & semantics

The main design idea behind declaration syntax:

Syntax rules for declarations are conceptually similar to expression rules: associativity + precedence + grouping

Declarations: syntax & semantics

The main design idea behind declaration syntax:

Syntax rules for declarations are conceptually similar to expression rules: associativity + precedence + grouping

```
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

Declarations: syntax & semantics

The main design idea behind declaration syntax:

Syntax rules for declarations are conceptually similar to expression rules: associativity + precedence + grouping

```
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);
```

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

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

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];
```

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

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];
```

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

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

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* (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".**

http://c-faq.com/decl/spiral.anderson.html
[This was posted to comp.lang.c by its author, David Anderson, on **1994-05-06**.]

The "Clockwise/Spiral Rule"

By David Anderson

- How to "bootstrap" C declarations ☺

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**.

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 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);
```

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

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".

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 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;
```

Unnamed struct