Tirgul 6 - Agenda

- The “right-left” rule
- Generic Programming in C
  - Why
  - void*
  - Pointers to functions
  - Promo to CPP

The "right-left" rule

So what is

```
int (*x)[7]
```

and how can we read it?

The "right-left" rule

1. Find the identifier
2. Look at the symbols on the right of the identifier
   - ( ) mean this is a function
   - [ ] mean this is an array
3. Look at the symbols to the left of the identifier
4. Go to step 2 until declaration is complete

* Parenthesis change the “natural” order

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is”	

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is” “arr is pointer to”	

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is” “arr is pointer to” “arr is pointer to array 5 of”	

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is” “arr is pointer to” “arr is pointer to array 5 of” “arr is pointer to array 5 of chars”	

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is pointer to array 5 of chars”	“arr is”

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is pointer to array 5 of chars”	“arr is” “arr is array 5 of”

The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is pointer to array 5 of chars”	“arr is” “arr is array 5 of” “arr is array 5 of pointers to”

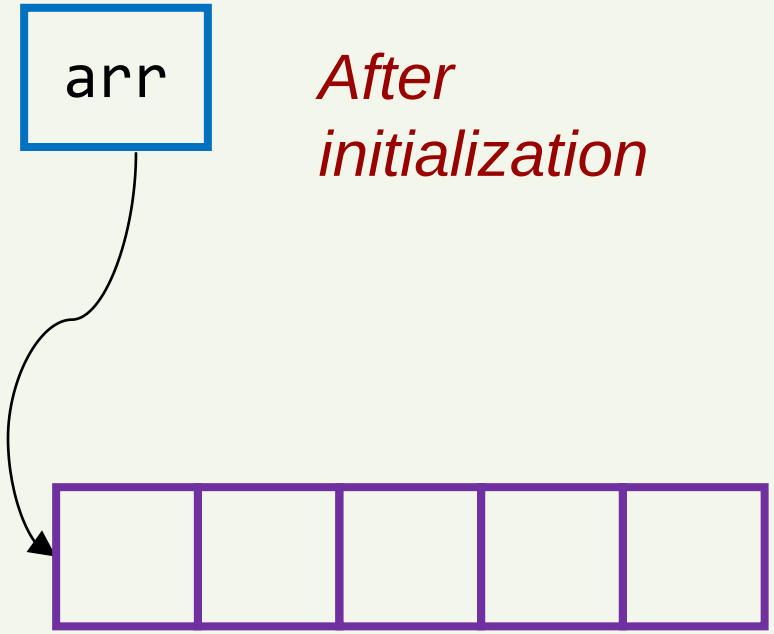
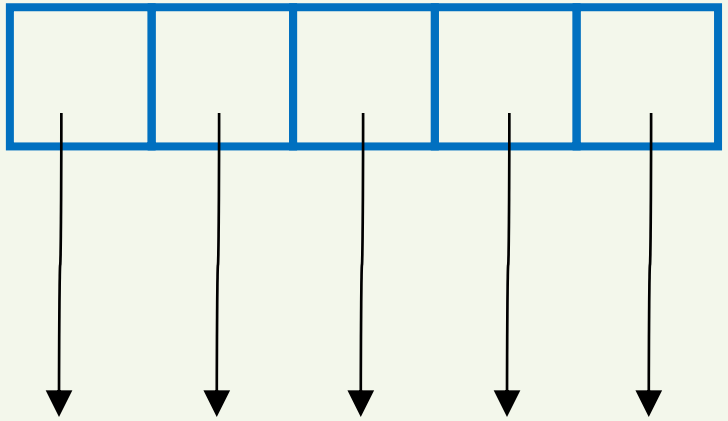
The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is pointer to array 5 of chars”	“arr is” “arr is array 5 of” “arr is array 5 of pointers to” “arr is array 5 of pointers to char”

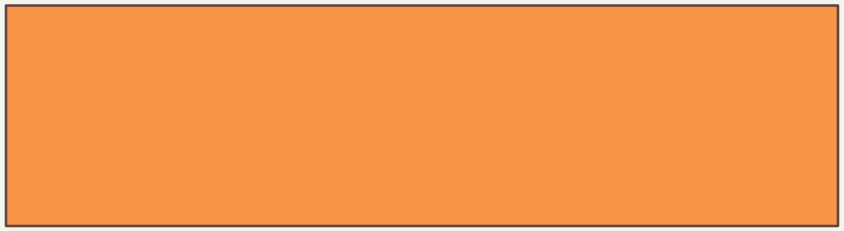
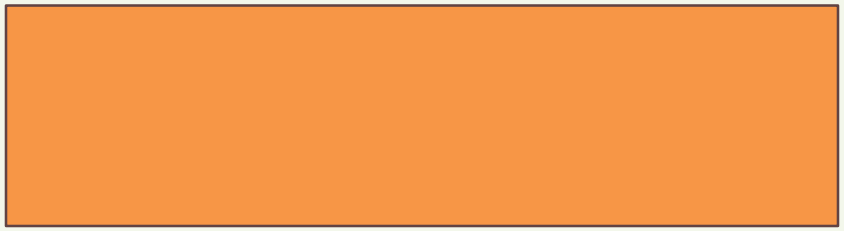
The "right-left" rule

`char (*arr)[5];`	`char *arr[5];`
“arr is pointer to array 5 of chars”	“arr is array 5 of pointers to char”

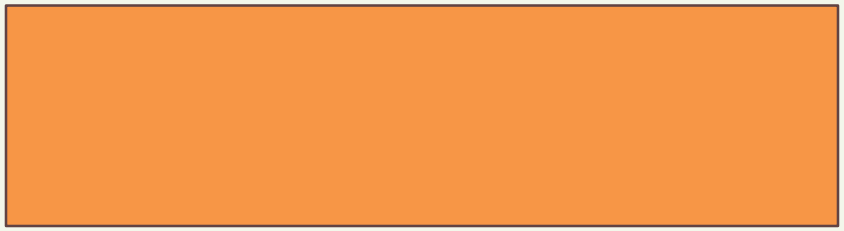
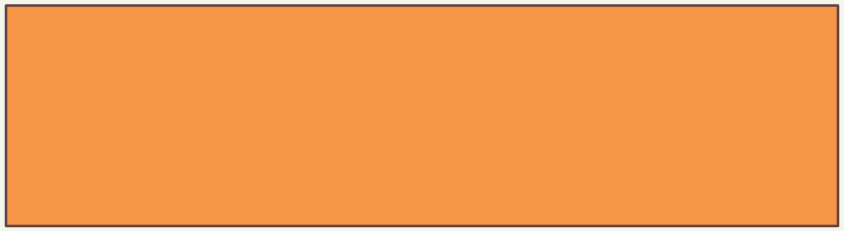
Pointer to array and array of pointers

`char (*arr)[5];`	`char *arr[5];`
	

Pointer to array and array of pointers

`char (*arr)[5];`	`char *arr[5];`
`sizeof (arr) =` 	`sizeof (arr) =` 
`sizeof (*arr) =` 	`sizeof (*arr) =` 

Pointer to array and array of pointers

`char (*arr)[5];`	`char *arr[5];`
`sizeof (arr) = sizeof(void*)` `sizeof (*arr) =` 	`sizeof (arr) =`  `sizeof (*arr) =` 

Pointer to array and array of pointers

`char (*arr)[5];`	`char *arr[5];`

`sizeof (arr) = sizeof(void*)`
 	
`sizeof (*arr) =`
	`sizeof (arr) = 5*sizeof (char*) = 5*sizeof (void*)`
	
	`sizeof (*arr) =`

Pointer to array and array of pointers

`char (*arr)[5];`	`char *arr[5];`

`sizeof (arr) = sizeof(void*)`
 	
`sizeof (*arr) = 5*sizeof(char)`
	`sizeof (arr) = 5*sizeof (char*) = 5*sizeof (void*)`
	
	`sizeof (*arr) =`

Pointer to array and array of pointers

`char (*arr)[5];`	`char *arr[5];`

`sizeof (arr) = sizeof(void*)`
 	
`sizeof (*arr) = 5*sizeof(char)`
	`sizeof (arr) = 5*sizeof (char*) = 5*sizeof (void*)`
	
	`sizeof (*arr) = sizeof (char*)= sizeof (void*)`

Pointer to array: using typedef

Explicit declaration:

```
char (*arr_2d)[20];
                // arr_2d is a pointer
                // not initialized
```

Using `typedef`:

```
typdef char arr_2d_20[20];
arr_2d_20 *p_arr_2d;
```

Generic Programming

- The goal:
  To write code once that works on a variety of types

- The tools in C:
  - pointers to functions
  - void*

- The tools in C++:
  - polymorphism (not called Generic programming)
  - templates (“pure” Generic programming)
  - overloading

void*

- A way to pass data of an arbitrary type
- void* is a generic pointer capable of representing any pointer type

```
int          i= 5;
double       f= 3.14;
void*        p;

p= &i;       // ok
p= &f;       // ok
```

Examples:

- We saw void* usage before in stdlib – where? **malloc**
- **void *memset(void *str, int c, size_t n)**
- **void *memcpy(void *str1, const void *str2, size_t n)**

“Rules” of void*

- a void* pointer cannot be dereferenced

```
int i= 5;
double f= 3.14;
void* p;

p= &i;                      // ok
p= &f;                      // ok

printf("%f\n", *p);         // compilation error
```

- Why? Because we don’t know what is there, thus we don’t know how to read it!

“Rules” of void*

- void* can be explicitly cast to another pointer type

```
int i= 5;
double f= 3.14;
void* p;

p= &i;                                  // ok
p= &f;                                  // ok

printf("%f\n", *((double*)p)); // ok
```

Example – generic swap

```
void memswap_memcpy(void* p1, void* p2, size_t size)
```

memswap - #version 1

```
void memswap_arr(void* p1, void* p2, size_t size)
{
        size_t  i;
        char* pc1= (char*)p1;
        char* pc2= (char*)p2;
        char  ch;
        printf("In memswap_arr\n ");
        for (i = 0; i < sizeof(p1); ++ i)
        {
                ch=pc1[i];
                pc1[i]= pc2[i];
                pc2[i]= ch;
        }
}
```

memswap - #version 1

```
void memswap_arr(void* p1, void* p2, size_t size)
{
        size_t  i;
        char* pc1= (char*)p1;
        char* pc2= (char*)p2;
        char  ch;
        printf("In memswap_arr\n ");
        for (i = 0; i < size; ++ i)
        {
                ch=pc1[i];
                pc1[i]= pc2[i];
                pc2[i]= ch;
        }
}
```

memswap - #version 2

```
void memswap_ptr(void* p1, void* p2, size_t size)
{
        size_t  i;
        char* pc1= (char*)p1;
        char* pc2= (char*)p2;
        char  ch;
        printf("In memswap_ptr\n ");
        for (i = 0; i < size; ++ i)
        {
                ch=*pc1;
                *pc1= *pc2;
                *pc2= ch;
                ++(*pc1);
                ++(*pc2);
        }
}
```

memswap - #version 2

```
void memswap_ptr(void* p1, void* p2, size_t size)
{
        size_t  i;
        char* pc1= (char*)p1;
        char* pc2= (char*)p2;
        char  ch;
        printf("In memswap_ptr\n ");
        for (i = 0; i < size; ++ i)
        {
                ch=*pc1;
                *pc1= *pc2;
                *pc2= ch;
                ++pc1;
                ++pc2;
        }
}
```

memswap - #version 3

```
void memswap_memcpy(void* p1, void* p2, size_t size)
{
        void* tmp;
        tmp= malloc(size);
        printf("In memswap_memcpy\n ");

        memcpy(tmp, p1  , size);
        memcpy(p1 , p2  , size);
        memcpy(p2 , tmp, size);
        return;
}
```

memswap - #version 3

```
void memswap_memcpy(void* p1, void* p2, size_t size)
{
        void* tmp;
        tmp= malloc(size);
        printf("In memswap_memcpy\n ");

        memcpy(tmp, p1  , size);
        memcpy(p1 , p2  , size);
        memcpy(p2 , tmp, size);
        free(tmp);
        return;
}
```

Pointers to Functions

- Are you serious?!
- Yes!
- A friendly tutorial: http://www.newty.de/fpt/index.html

- Assuming that function f is defined:
  &f and **f** are pointers to the function
- i.e.: The address where the function's **definition** begins in memory

Recommendation

- Use typedef

```
typedef int (* TwoIntsFunc) (int, int);

TwoIntsFunc f1;
f1 = &avg;
...
f1 = &sum;
```

And again… “right-left” rule:

```
// What does this signature means?


float (*GetPoiter1(const char op))(float, float)
{

                                ...


}
```

What is it good for?

- Generic programming - pass a function name to another function as a parameter
- Once upon a time… it was a way to make a c struct kind of a poor class

Classical Example – qsort

void* & pointer to function example

```
#include <stdlib.h>

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

Start of array to be sorted

Number of elements in array

sizeof of each element in array

Pointer to the comparison function
Returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second

Using qsort

```
int compareInt(void const *p, void const *q)
{
   int a = *(int const*)p;
   int b = *(int const*)q;
   if( a < b )
   {
      return -1;
   {
   return a > b;
}

int array[10] = { ...} ;
qsort( array, 10, sizeof(int), compareInt );
```

qsort problems or C++ “promo”

- Many parameters (function pointers)
- Not user friendly
- Type safety problems

- C++ improves all these

So.. That’s it!

You know C!

Good luck in the exam ☺