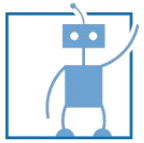
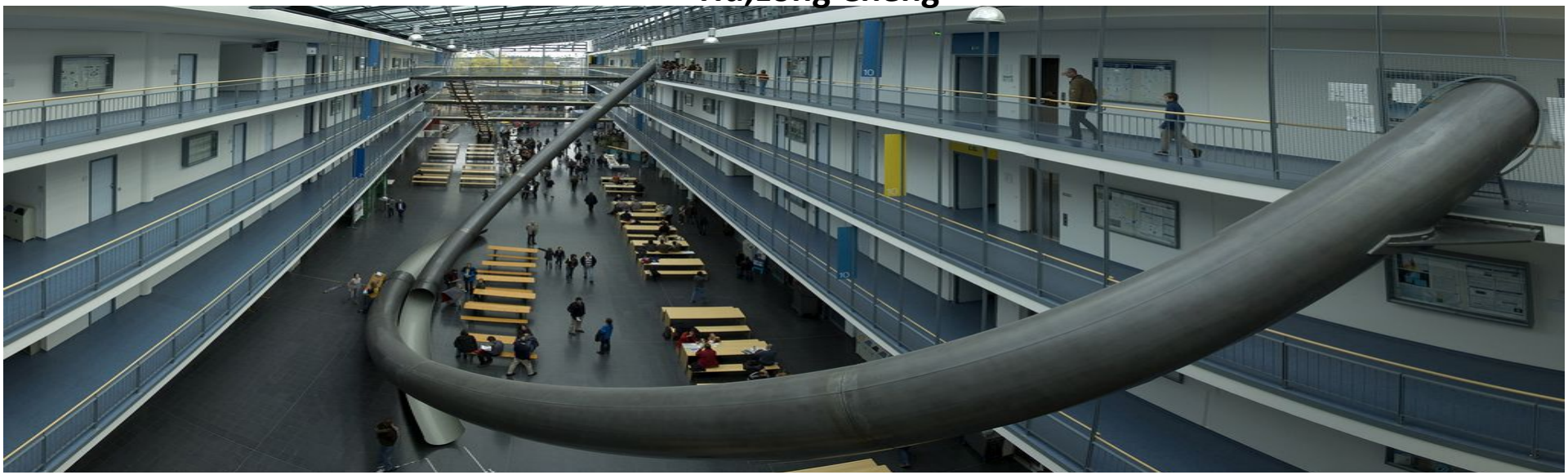


Robotics and Embedded Systems

Technische Universität München

C-Programming
More C

Kai Huang, Sebastian Klose, Gang Chen, Hardik Shah， Biao Hu,Long Cheng



Constants

- *A constant number may be used many times in a programm, for example,* $\pi = 3.1415$

- *defines:*

```
#define MY_CONSTANT ( 10 )
```

- using **const** keyword:

```
static const int my_constant = 10;
```

What's that static?

static

- Global variables and functions
  ➔ set‘s scope to compilation unit
- Local variables: store variable in the static allocated memory (usually compile time)

```c
#include <stdio.h>

int count()
{
    static int call_times;
    call_times++;

    printf( "count has been called %d times\n", call_times );
}

int main( int argc, char* argv[] )
{
    count();
    count();
    count();
    return 0;
}
```

What ‘s the output?

static

- Global variables and functions
  ➔ set‘s scope to compilation unit
- Local variables: store variable in the static allocated memory (usually compile time)

```c
#include <stdio.h>

int count()
{
    static int call_times;
    call_times++;

    printf( "count has been called %d times\n", call_times );
}

int main( int argc, char* argv[] )
{
    count();
    count();
    count();
    return 0;
}
```

```
count has been called 1 times
count has been called 2 times
count has been called 3 times
```

Character arrays

- Strings are represented by array of chars in C
- End symbol: '\0'

```c
#include <stdio.h>

int main( void )
{
    char buf[ 6 ];
    buf[ 0 ] = 'H';
    buf[ 1 ] = 'E';
    buf[ 2 ] = 'L';
    buf[ 3 ] = 'L';
    buf[ 4 ] = 'O';
    buf[ 5 ] = '\0';

    char buf2[] = { 'H','E','L','L','O','\0' };
    char buf3[] = { 'H','E','L','L','O','\0','H' };
    char* buf4 = "HELLO";

    printf( "%s -> %zu\n", buf, sizeof( buf ) );
    printf( "%s -> %zu\n", buf2, sizeof( buf2 ) );
    printf( "%s -> %zu\n", buf3, sizeof( buf3 ) );
    printf( "%s -> %zu\n", buf4, sizeof( buf4 ) );

    return 0;
}
```

```
HELLO -> 6
HELLO -> 6
HELLO -> 7
HELLO -> 4
```

Segmentation faults

- By using pointers you can access the memory and change the content.
- Once you start using pointers you‘ll definitely run into them!

```c
#include <stdio.h>

int main( void )
{
    int x = 5;
    int arr[] = { 0, 1, 2, 3, 4 };

    int* p = arr;

    int i = 0;
    while( i <= 5 ){
        printf( "%p: [ %04d ]\n", p, *p );
        ++i;
        ++p;
    }

    return 0;
}
```

```
0x7fff59415960: [ 0000 ]
0x7fff59415964: [ 0001 ]
0x7fff59415968: [ 0002 ]
0x7fff5941596c: [ 0003 ]
0x7fff59415970: [ 0004 ]
0x7fff59415974: [ 0005 ]
```

Gosh!!

Pure luck, that the accesses memory is part of our stack

Segmentation faults

- Another result
- Be careful!

```c
#include <stdio.h>

int main( void )
{
    int x = 5;
    int arr[] = { 0, 1, 2, 3, 4 };

    int* p = arr;

    int i = 0;
    while( i <= 5 ){
        printf( "%p: [ %04d ]\n", p, *p );
        ++i;
        ++p;
    }

    return 0;
}
```

```
0xbf9c9830: [ 0000 ]
0xbf9c9834: [ 0001 ]
0xbf9c9838: [ 0002 ]
0xbf9c983c: [ 0003 ]
0xbf9c9840: [ 0004 ]
0xbf9c9844: [ -1080256444 ]
```

meaningless

Macros

- Useful for small calculations, definitions, etc

```
#define MACRO_NAME(arg1, arg2, ...) [code to expand to]
```

- Be careful when using multiple statements:

```
#define TWO_PRINT( STR0, STR1 ) printf( "%s\n", STR0 ); printf( "%s\n", STR1 );

int main( void ){
    TWO_PRINT( "FOO", "BAR" )
    int x = 1;
    if( x > 2 )
        TWO_PRINT( "A", "SURPRISE" );
    return 0;
}
```

```
FOO
BAR
SURPRISE
```

- Enclose the statement in braces to remove 'SURPRISE'
- Enclose multiple statements into
  `do { ... } while( 0 );`

X Macro trick

- Useful for basic code generation:

```
#include <stdio.h>
#include <stdlib.h>

// the enum
#define X( name, ... ) name,

typedef enum {
    #include "errors.h"
} Errors;
#undef X

/* the array of error string */
#define X( name, errstr ) errstr,
static const char* error_strings[] = {
    #include "errors.h"
};
#undef X

int main( void ){
    printf( "%s\n", error_strings[ IO_ERROR ] );
    printf( "%s\n", error_strings[ FILE_ERROR ] );
    printf( "%s\n", error_strings[ STANDARD_ERROR ] );
}
```

```
errors.h
// <ENUM> <ERROR_STRING>
X( STANDARD_ERROR,   "standard error" )
X( FILE_ERROR,       "file error" )
X( IO_ERROR,         "io error" )
```

```
enum:
STANDARD_ERROR 0
FILE_ERROR     1
IO_ERROR       2
```

```
io error
file error
standard error
```

Macro – stringify / concatenate

```c
#include <stdio.h>

#define str( x ) #x
#define idef( name, y ) int ivar_ ## name = y


int main( void ){
    char* string = str( 123445567 );

    idef( foo, 100 );

    printf( "%s\n", string );
    printf( "%d\n", ivar_foo );
    return 0;
}
```

```
123445567
100
```

stringify argument #abc=="abc"

concatenate

Function pointers

- Pointers to functions are useful, e.g. for interrupt callbacks
- Give a new type name to function pointers that point to same kind functions
- Syntax examples:

```
typedef void ( *VoidFunc )( void );
typedef float ( *OtherFunc )( int a, float b );
```

- Address operator can also be used for functions, to assign a function pointer
- Can be called just as normal functions or by using the derefence operator (classic style)
- put *and pointer in parentheses: (*pointer)(*void*)

Example

```
#include <stdio.h>

void func_a(){ printf( "%s\n", __func__ ); }
void func_b(){ printf( "%s\n", __func__ ); }

float other( int a, float b ){ return ( float )a + b; }

typedef void ( *VoidFunc )( void );
typedef float ( *OtherFunc )( int a, float b );

int main( int argc, char* argv[] ){
    VoidFunc myFunc = &func_a;
    myFunc();

    myFunc = &func_b;
    myFunc();

    OtherFunc myOther = &other;
    printf( "%0.2f\n", myOther( 1, 2.0f ) );

    return 0;
}
```

```
func_a
func_b
3.00
```

foreach

```c
#include <stdio.h>

typedef void ( *FuncPtr )( int );

void foreach_int( const FuncPtr func, int* iarr, size_t n ){
    while( n-- )
        func( *iarr++ );
}

void print( int i ){ printf( "%d\n", i ); }
void sqr( int i )  { printf( "%d\n", i*i ); }

int main( int argc, char* argv[] ){
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    FuncPtr f = &print;
    foreach_int( f, arr, sizeof( arr ) / sizeof( int ) );

    f = &sqr;
    foreach_int( f, arr, sizeof( arr ) / sizeof( int ) );
    return 0;
}
```

```
1
2
3
4
5
6
7
8
1
4
9
16
25
36
49
64
```

Function pointer arrays

```c
#include <stdio.h>

typedef int ( *FuncPtr )( int );

int times2( int i ){ return i << 1; }
int sqr( int i ){ return i * i; }
int not( int i ){ return ~i; }

int main( int argc, char* argv[] ){
    FuncPtr functions[] = {
        &times2,
        &sqr,
        &not
    };

    size_t nfuncs = sizeof( functions ) / sizeof( FuncPtr );
    int a = 10;

    int i;
    for( i = 0; i < nfuncs; i++ ){
        int x = functions[ i ]( a );
        printf( "Function: %d -> %d\n", i, x );
    }
    return 0;
}
```

```
Function: 0 -> 20
Function: 1 -> 100
Function: 2 -> -11
```