Robotics and Embedded Systems

Technische Universität München

C-Programming
More C

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

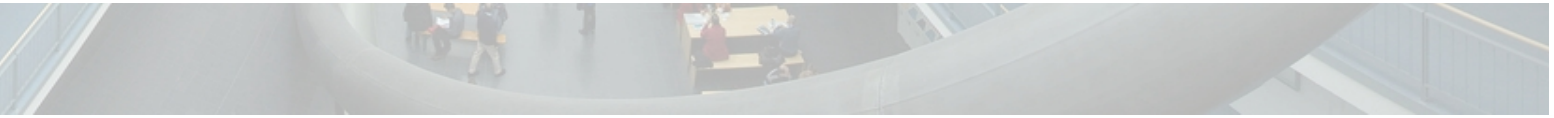
Sequencing

```c
#include <stdio.h>

void foo( void )
{
    int a = 41;
    a = a++;
    printf( "a = %d\n", a )
}

int main( void )
{
    foo();
    return 0;
}
```

Always compile with –*Wall* option

```
$> gcc -o sequencing sequencing.c -Wall
sequencing.c: In function 'foo':
sequencing.c:6: warning: operation on 'a' may be undefined
```

similar code:
int a=3,b;
b=(++a)+(++a);



Include guard

- Usually you will organize your code in several header and source files
- Header files contain declarations (and sometimes inline definitions)
- Header file might be included by several other headers:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

void     myfunc( int, int, int );
float    foo();
void     bar();

#endif
```

Bitcount #1

- Most simple approach

```
int bit_count( uint32_t v )
{
    uint32_t sum;
    for( sum = 0; v; v>>=1 )
        sum += v & 1;
    return sum;
}
```

- Move bits to 0 position and check if bit is set
- `v>>=1`
- if the bit is set, sum will be incresed by 1

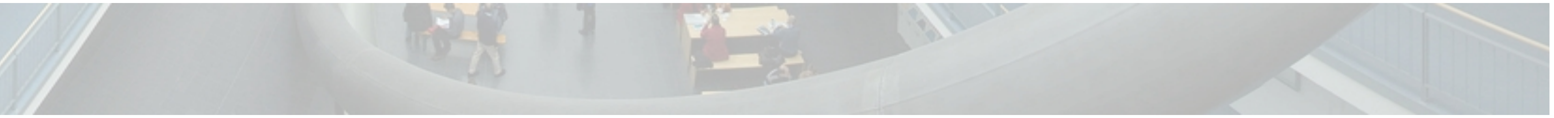
Bitcount #2

- Linear in number of bits set:

```
int bit_count( uint32_t v )
{
    uint32_t sum;
    for( sum = 0; v; sum++ )
        v &= v - 1; /* clear right most bit set */
    return sum;
}
```

6:0110
5:0101
6&5:0100

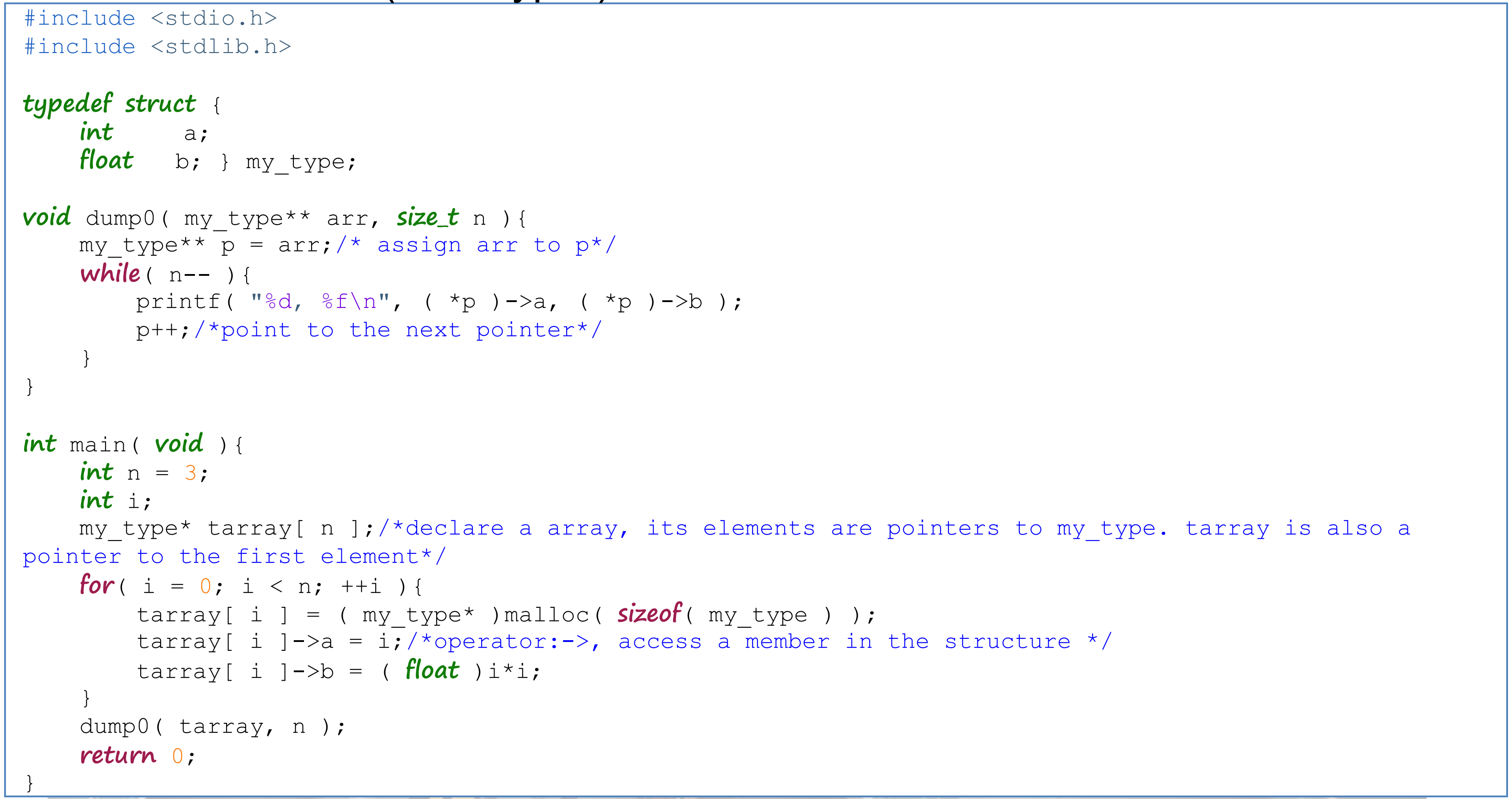
- bit operation, removes the right most set bit



Pointer to Pointer

Syntax: DataType** Name = PointerAddress

You can see it as: (DataType*) * Name= PointerAddress



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

typedef struct {
    int      a;
    float    b; } my_type;

void dump0( my_type** arr, size_t n ){
    my_type** p = arr;/* assign arr to p*/
    while( n-- ){
        printf( "%d, %f\n", ( *p )->a, ( *p )->b );
        p++;/*point to the next pointer*/
    }
}

int main( void ){
    int n = 3;
    int i;
    my_type* tarray[ n ];/*declare a array, its elements are pointers to my_type. tarray is also a
pointer to the first element*/
    for( i = 0; i < n; ++i ){
        tarray[ i ] = ( my_type* )malloc( sizeof( my_type ) );
        tarray[ i ]->a = i;/*operator:->, access a member in the structure */
        tarray[ i ]->b = ( float )i*i;
    }
    dump0( tarray, n );
    return 0;
}
```

Multidimensional Arrays (1)

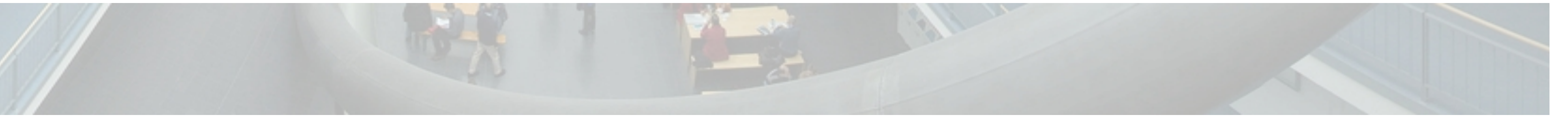
- Syntax: type a[n][m]   e.g.   float rain[5][12]
- float rain[5]   [12]   a array with 5 elements
- float rain[5]   [12]   every element is a array with 12 floats

```
float matrix[ 3 ][ 3 ];

matrix[ 0 ][ 0 ] = 1.0f;
matrix[ 0 ][ 1 ] = 0.0f;
matrix[ 0 ][ 2 ] = 0.0f;

matrix[ 1 ][ 0 ] = 0.0f;
matrix[ 1 ][ 1 ] = 1.0f;
matrix[ 1 ][ 2 ] = 0.0f;

matrix[ 2 ][ 0 ] = 0.0f;
matrix[ 2 ][ 1 ] = 0.0f;
matrix[ 2 ][ 2 ] = 1.0f;
```

matrix[i] :a pointer to matrix[i][0]

matrix:a pointer to matrix[0]

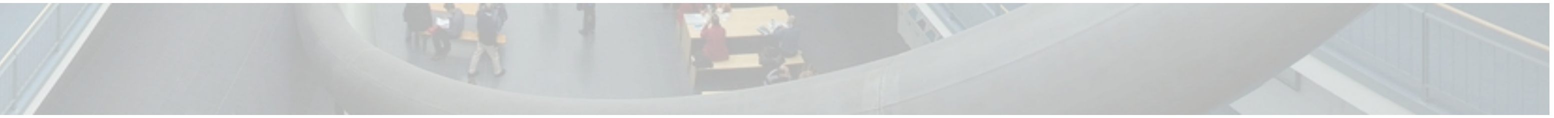
"matrix" is a pointer to pointer.



Multidimensional Arrays (2)

- Passing to a function:
  - Can only leave off dimension of first parameter, this means the argument is a pointer.
  - Other dimensions tell the data type

```
void func2( int param[][ 10 ] );
void func3( int param[][ 20 ][ 10 ] );
```



Arithmetic issues

- Int arithmetic caveats

```
float a = 1 / 2;
float b = 1.0f / 2;
float c = 1 / 2.0f;

printf( "a = %0.2f\n", a );
printf( "b = %0.2f\n", b );
printf( "b = %0.2f\n", b );
```

```
a = 0.00
b = 0.50
b = 0.50
```

- Big & small numbers

```
float  f(){ return 10000.1234f * 10.0f; }
double d(){ return 10000.1234  * 10.0; }

int main( void ){
    printf( "f() = %0.8f\n", f() );
    printf( "d() = %0.8f\n", d() );
    return 0;
}
```

```
f() = 100001.23437500
d() = 100001.23400000
```

Compiler options

- Debugging symbols
  `-g`
- Compiler Optimizations
- `-O0        No optimization`
- `-O -O1     Optimize`
- `-O2        Optimize even more`
- `-Os        Level 2.5`
- `enables all -O2 optimizations that do not increase code size`
- `-O3        Optimize yet more`
- Turn on compiler all warnings
  `-Wall`