CHALMERS
UNIVERSITY OF TECHNOLOGY

Machine-Oriented Programming

C-Programming part 4

Pedro Trancoso
ppedro@chalmers.se

Original slides by Ulf Assarsson

Objectives

- Structs vs Union
- Enum
- Arrays vs Pointers
- Arrays of pointers, arrays of arrays, arrays of struct
- Program structure: multiple files
- Extern, static
- Compiler directives: #if, #ifdef, #ifndef
- Include guards

Number of bytes with sizeof()

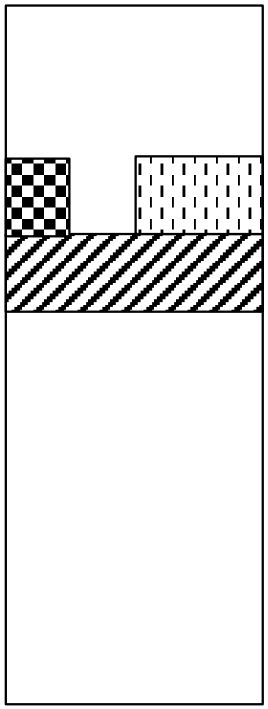
```
#include <stdio.h>

char* s1 = "Emilia";
char s2[] = "Emilia";

int main()
{
     printf("sizeof(char):  %d \n", sizeof(char) );
     printf("sizeof(char*): %d \n", sizeof(char*) );
     printf("sizeof(s1):    %d \n", sizeof(s1) );
     printf("sizeof(s2):    %d \n", sizeof(s2) );

     return 0;
}
```

```
sizeof(char):     1
sizeof(char*):    4
sizeof(s1):       4
sizeof(s2):       7
```

Sizeof evaluated at compile-time. One (of few) exceptions where arrays and pointers are different.

It is actually a “string” not an “array”

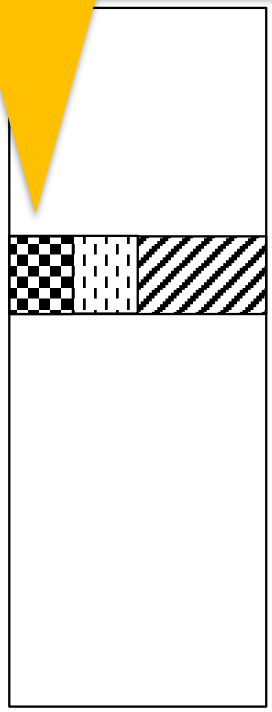
Struct vs Union



```
struct abc {
    int a;
    char b;
    short c;
};

struct abc x;

x.a = 2345678;
x.b = 'f';
x.c = 572;
```

```
union abc {
    int a;
    char b;
    short c;
};

union abc x;

x.a = 2345678;
x.b = 'f';
x.c = 572;
```

&x.a==&x.b==&x.c



```
printf("%d,%c,%d\n",
        x.a,x.b,x.c)
```

2345678,f,572

2294332,<,572

Byte-addressing with unions

For GPIO ports

```
// GPIO
typedef struct _gpio {
   uint32_t    moder;
   uint32_t    otyper;
   uint32_t    ospeedr;
   uint32_t    pupdr;
   union {
      uint32_t    idr;
      struct {
         byte    idrLow;
         byte    idrHigh;
         short  reserved;
      };
   };
   union {
      uint32_t    odr;
      struct {
         byte    odrLow;
         byte    odrHigh;
      };
   };
} GPIO;
#define GPIO_D (*((volatile GPIO*) 0x40020c00))
#define GPIO_E (*((volatile GPIO*) 0x40021000))
```

GPIO Input Data Register (IDR)

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic
0x10	RESERVERADE																r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	IDR
																	0x11								0x10								

Bits 16 to 31 are not used and will hold its RESET value, that is 0

Now idrHigh can be addressed with:

```
byte c = GPIO_E.idrHigh;
```

Instead of with:

```
byte c = *((byte*)&(GPIO_E.idr) + 1));
```

As for example:

```
GPIO_E.odrLow &= ( ~B_SELECT & ~x);
```

Instead of:

```
*((byte*)&(GPIO_E.odr)) &=(~B_SELECT & ~x);
```

Enumerations: enum

```
enum type_name { value1, value2,..., valueN }; //type_name optional. By default, value1 = 0, value2 = value1 + 1, etc.

enum type_name { value1 = 0, value2, value3 = 0, value4 }; //However, with gcc values: 0, 1, 0, 1 – not intuitive.

enum day {monday=1, tuesday, wednesday, thursday, friday, saturday, sunday};
enum day today; // day becomes a char, short or int
today=wednesday;
printf("%d:th day",today+1); // output: "4:th day"

typedef enum { false, true } bool;
bool ok = true;
--------------------------------------------------------------------------------
#define B_E         0x40         // using enums
#define B_RST       0x20
#define B_CS2       0x10
#define B_CS1       8
#define B_SELECT    4
#define B_RW        2
#define B_RS        1
```

Can be replaced with:

```
enum {B_RS=1, B_RW=2, B_SELECT=4, B_CS1=8, B_CS2=0x10, B_RST=0x20, B_E=0x40};
```

Enum

```
#include <stdio.h>

typedef enum { one=1, two, three, four, five=10, six, seven, eight, nine, ten } numbers;

int main() {
  printf("two=%d eight=%d\n", two, eight );
}
```

```
two=2 eight=13
```

```
#include <stdio.h>

typedef enum { monday, tuesday, wednesday, thursday, friday, saturday, sunday } daysofweek;
char *daysname[] = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" };

int main() {
  for(int i = monday; i <= wednesday; i++)
    printf("Day = %s\n", daysname[i]);
}
```

```
Day = Mon
Day = Tue
Day = Wed
```

Indexing: Same for array / pointers

x[y] is translated to *(x + y) and is thus a way to derive a pointer.
Indexing is the same for pointers as for the array.
So are arrays pointers? No…

```c
#include <stdio.h>

char* s1 = "Emilia";
char s2[] = "Emilia";

int main()
{
    // two equivalent ways to dereference a pointer
    printf("'l' in Emilia (version 1): %c \n", *(s1+3));
    printf("'l' in Emilia (version 2): %c \n", s1[3]);

    // two equivalent ways to index an array
    printf("'l' in Emilia (version 1): %c \n", s2[3]);
    printf("'l' in Emilia (version 2): %c \n", *(s2+3));

    return 0;
}
```

Arrays vs Pointers: Similarities and Differences

```
char* s1 = "Emilia";
char s2[] = "Emilia";
```

- Both have and address and a type.
  - `char s2[] = "Emilia";`
    - `sizeof(s2) = 7`
  - `char* s1 = "Emilia";`
    - `sizeof(s1) = sizeof(char*) = 4`

```
s1++; // is allowed
s2++; // is NOT allowed
```

- Indexing has the same result.
  - s1[0] → 'E'
  - s2[0] → 'E'
  - *s1 → 'E'
  - *s2 → 'E' (because s2 is an address, we can dereference it just like a pointer)

Arrays vs Pointers: Similarities and Differences

```
char* s1   = "Emilia";
char  s2[] = "Emilia";
```

	s2	s1
Type:	Array	Pointer variable
Addressing:	**&s2 is not possible -** s2 is just a symbol s2 = symbol = array's start address. s2 = &(s2[0]) s2[0] ≡ *s2 → 'E'	&s1 = address for variable s1. s1 = s1's value = string's start address. s1 = &(s1[0]) s1[0] ≡ *s1 → 'E'
Pointer arithmetic:	**s2++ is not possible** (s2+1)[0] is OK	s1++ is OK (s1+1)[0] is OK
Size of type:	sizeof(s2) = 7 bytes	sizeof(s1) = sizeof(char*) = 4 bytes

s2 is a symbol (not a variable) for an address which is known at compile time.
Because s2 is an address we can dereference it exactly as a pointer: *s2 → 'E'.

Arrays as function parameters become pointers

```
void foo(int i[]);
```

```
void foo(int *i);
```

[ ] – the notation exists but it means pointer!

Avoids the entire array to be copied. *Length not always known at compile time*. The address of the array is added to the stack and accessed via the stack variable i.

NOTE: A **struct** is copied and placed on the stack.

```
int sum_elements(int *a, int l)
{
    int sum = 0;
    for (int i=0; i<l; i++) {
      sum += a[i];
    }
    return sum;
}
…
int array[] = {5,4,3,2,1};
int x;
x = sum_elements(array, 5);
```

Array of pointers

```c
#include <stdio.h>

char *manyName[] = {"Emil", "Emilia", "Droopy"};

int main()
{
    printf("%s, %s, %s\n", manyName[2], manyName[1], manyName[0]);

    return 0;
}
```

```
Droopy, Emilia, Emil

sizeof(manyName) = 12; // 3*sizeof(char*) = 3*4 = 12
```

Array of arrays

```
#include <stdio.h>

char shortName[][4] = {"Tor", "Ulf", "Per", "Ian" };

int main()
{
     printf("%s, %s, %s\n", shortName[2], shortName[1], shortName[0]);

     return 0;
}
```

```
Per, Ulf, Tor

sizeof(shortName) = …
```

Array of arrays

```
#include <stdio.h>

int arrayOfArrays[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };

int main()
{
    int i,j;
    for( i=0; i<3; i++) {
        printf("arrayOfArray[%d] = ", i);
        for ( j=0; j<4; j++)
            printf("%d ", arrayOfArrays[i][j]);
        printf("\n");
    }

    return 0;
}
```

arrayOfArrays[i][j] = arrayOfArrays + (i*4+j)*sizeof(int)

Array of structs

```c
#include <stdio.h>

typedef struct {
     char* name;
     float credits;
} Student;

Student stud1 = {"Per", 200.0f};
Student stud2 = {"Tor", 200.0f};
Student stud3 = {"Ulf",  20.0f};

Student students[3] = {stud1, stud2, stud3};
// or simply:
Student students[] = {{"Per", 200.0f}, {"Tor", 200.0f}, {"Ulf",  20.0f}};

int main()
{
     printf("%s, %s, %s\n", students[0].name, students[1].name, students[2].name);
     return 0;
}
```

```
Per, Tor, Ulf
```

Program Structure

If you have long programs then you should not put everything in a single main.c file. You should split the functionalities between different files. Header h-files should include function prototypes (and user-defined type definitions). Different c-files can include different functions grouped by topic.

```c
// main.c
#include <stdio.h>
#include "foo.h"

int main()
{
    printf("x is %d\n", foo(0));
    return 0;
}
```

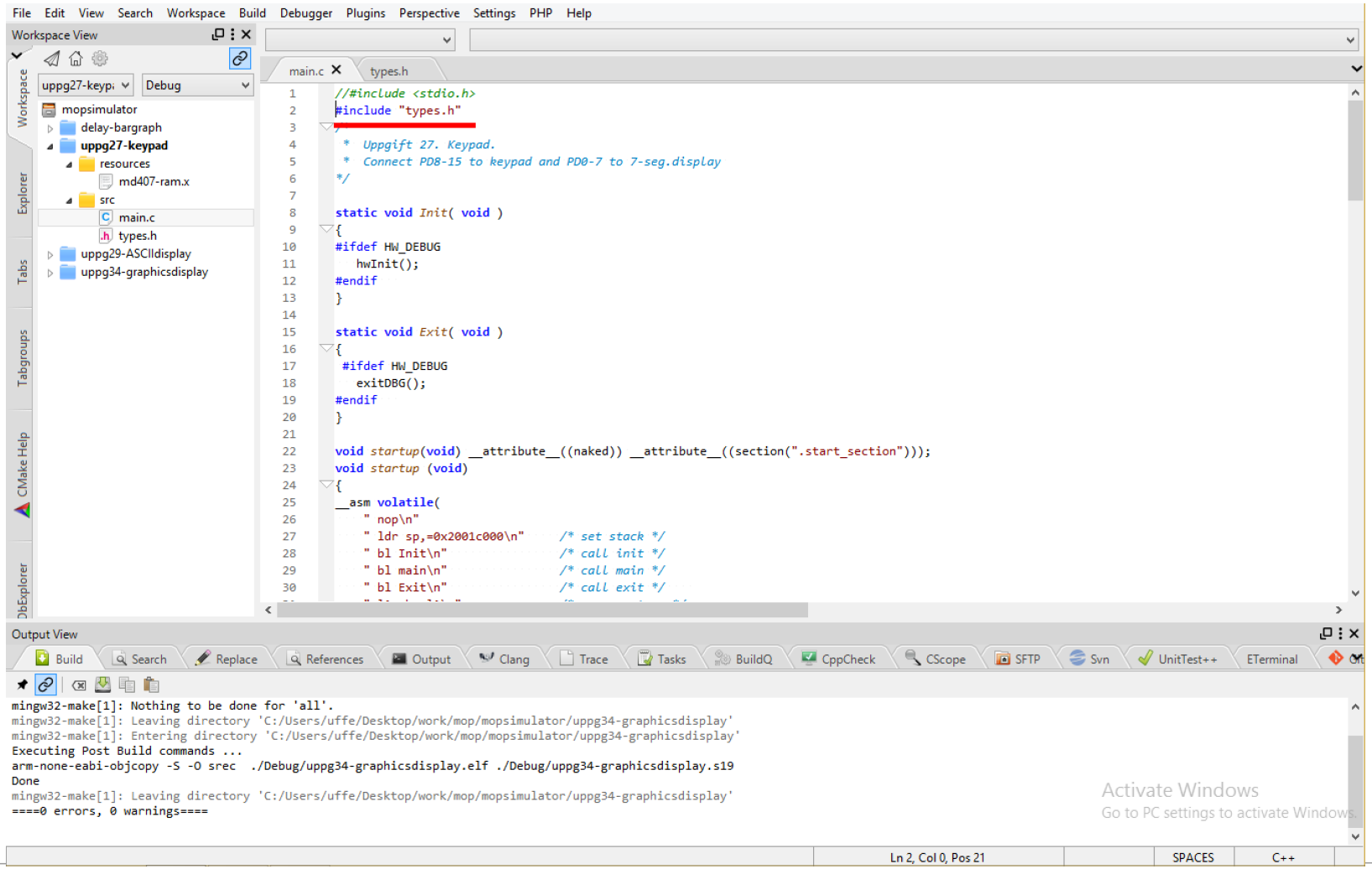
c-file
Includes header files

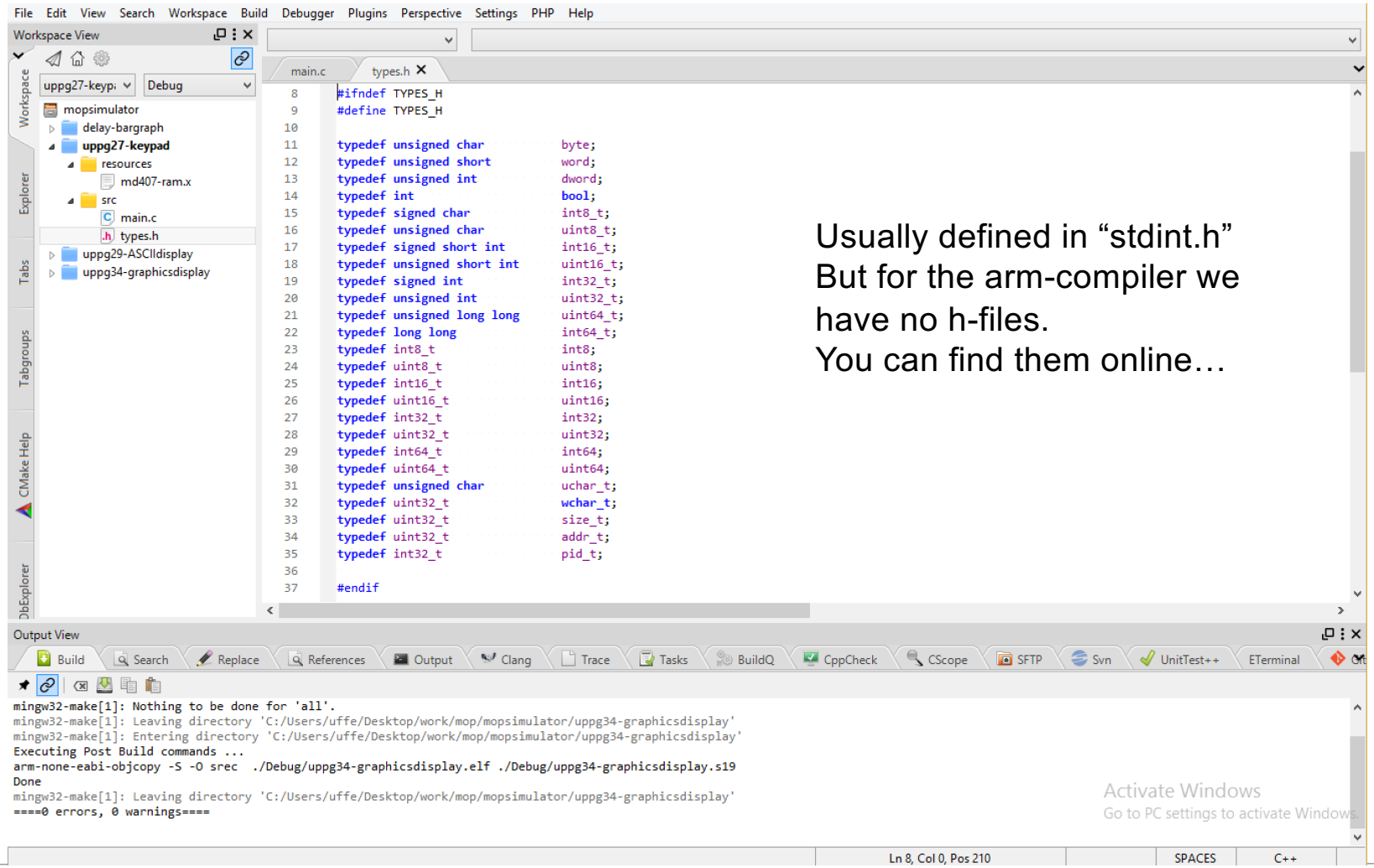
```c
// foo.h

int foo(int x);
```

header-file
Contains the
function prototypes

```c
// foo.c
#include <stdlib.h>

int foo(int x)
{
    if( x == 0 ){
        int x = 4;
        return x;
    }
    return x;
}
```

c-file


File Edit View Search Workspace Build Debugger Plugins Perspective Settings PHP Help
Workspace View
uppg27-keypa
Debug
mopsimulator
delay-bargraph
uppg27-keypad
resources
md407-ram.x
src
main.c
types.h
uppg29-ASCIIdisplay
uppg34-graphicsdisplay
main.c
types.h
//#include <stdio.h>
#include "types.h"
* Uppgift 27. Keypad.
* Connect PD8-15 to keypad and PD0-7 to 7-seg.display
static void Init(void)
#ifdef HW_DEBUG
hwInit();
#endif
static void Exit(void)
#ifdef HW_DEBUG
exitDBG();
#endif
void startup(void) __attribute__((naked)) __attribute__((section(".start_section")));
void startup (void)
__asm volatile(
" nop\n"
" ldr sp,=0x2001c000\n" /* set stack */
" bl Init\n" /* call init */
" bl main\n" /* call main */
" bl Exit\n" /* call exit */
Output View
Build Search Replace References Output Clang Trace Tasks BuildQ CppCheck CScope SFTP Svn UnitTest++ ETerminal
mingw32-make[1]: Nothing to be done for 'all'.
mingw32-make[1]: Leaving directory 'C:/Users/uffe/Desktop/work/mop/mopsimulator/uppg34-graphicsdisplay'
mingw32-make[1]: Entering directory 'C:/Users/uffe/Desktop/work/mop/mopsimulator/uppg34-graphicsdisplay'
Executing Post Build commands ...
arm-none-eabi-objcopy -S -O srec ./Debug/uppg34-graphicsdisplay.elf ./Debug/uppg34-graphicsdisplay.s19
Done
mingw32-make[1]: Leaving directory 'C:/Users/uffe/Desktop/work/mop/mopsimulator/uppg34-graphicsdisplay'
====0 errors, 0 warnings====
Activate Windows
Go to PC settings to activate Windows.
Ln 2, Col 0, Pos 21
SPACES
C++



Usually defined in “stdint.h”
But for the arm-compiler we have no h-files.
You can find them online…

Extern

`extern` in C means that the symbol is coming from another file.

```
// main.c
#include <stdio.h>

extern int g_var;
void myFunc();

void main()
{
    g_var = 5;
    myFunc();
}
```

Just declaration – without definition.
g_var defined later – for example in another .c-file

`extern` not necessary for functions, the prototype means: the code exists at link-time.

```
// myFunc.c
int g_var;
void myFunc()
{
    printf("hej");
};
```

Extern

Note: If you code a larger program for lab5, you may not want all global items/variables declared in the main file. Then you need to use external.

`extern` in C means that the symbol comes from another file.

```
// main.c
#include "fractalmountain.h”
…

POBJECT objects[] = {&player, &fmountain};
unsigned int nObjects = 2;

void main()
{
   …
}
```

```
// fractalmountain.h
#ifndef FRACTALMOUNTAIN_H
#define FRACTALMOUNTAIN_H

#include "object.h"


extern OBJECT fmountain;

#endif //FRACTALMOUNTAIN_H
```

```
// fractalmountain.c
#include "fractalmountain.h"

OBJECT fmountain = {
  0,             // geometry - no
  0,0,           // direction vector
  0,0,           // initial startposition
  drawMountain, // draw method
  0,             // clear_object – unused
  moveMountain, // move method
  set_object_speed // set-speed method
};
```

extern = declare without defining
(without extern we would create a **new** variable.)

C: visibility for declarations

All declarations (variables, functions) and expressions like typedefs + defines are visible first below them - not above them. Source files are processed from top to bottom.

Example:

```
void fkn1(short param)
{
     …
     if(…) {

        return fkn2('a'); // fkn2 is here unknowned for the C-compiler so it results in a compilation error
     }
     …
     return;
}

void fkn2(char c)
{
     …
     return;
}
```

C: visibility for declarations

All declarations (variables, functions) and expressions like typedefs + defines are visible first below them - not above them. Source files are processed from top to bottom.
Example:

```
void fkn2(char c);          // Fix to make the declaration of fkn2 already known here.
void fkn2(char c);          // We may have the declaration how many times we want
void fkn1(short param)
{
     …
     if(…) {

        return fkn2('a'); // now fkn2 is known here
     }
     …
     return;
}

void fkn2(char c)           // The definition of fkn2
{
     …
     return;
}
```

C: visibility for declarations

All declarations (variables, functions) and expressions like typedefs + defines are visible first below them - not above them. Source files are processed from top to bottom.
Example:

```
void fkn2(char c);         // Fix to make the declaration of fkn2 already known here.
                           // We may have the declaration how many times we want
void fkn1(short param)
{
     void fkn2(char c);       // We can even have it here instead
     if(…) {
        void fkn2(char c);    // or here!
        return fkn2('a’); // now fkn2 is known here
     }
     …
     return;
}

void fkn2(char c)          // The definition of fkn2
{
     …
     return;
}
```

`static`

`static` in C can do two things:

1. Give visibility only in its own and inner scopes
   - For example, symbols (variables / features) can not be seen outside its .c file.
2. A static variable is in the data segment as opposed to volatile on the stack
   - Allows you to create local variables that retain their value between the function calls

See also homework assignment 4

Static variant 1

```
static int var;
```

The variable *var* can not be seen from another file, regardless of whether you use `extern`.

In ”Java terminology” this is close to ”private” but the scope is the file.

Static variant 2

```
#include <stdio.h>

void testFkt()
{
     int var1 = 0;
     static int var2 = 0;
     var1++;
     var2++;
     printf("var1: %i, var2: %i \n", var1, var2);
}

int main()
{
     testFkt();
     testFkt();
     testFkt();
}
```

Output:

```
var1: 1, var2: 1
var1: 1, var2: 2
var1: 1, var2: 3
```

var2 is initialized to zero only the first time we call the function, but then retains its value between calls.

var2 is a **global variable** that has only local visibility in its scope – in this case testFkt().

Compiler Pragma Directives

- #include
- #if, #ifdef, #ifndef, #endif
- #define
- #pragma

Preprocessor directives: #if, #ifdef, #ifndef

Preprocessor conditional inclusion

```
#define X  1       // syntax: #define [identifier name] [value], where [value] is optional

#if X == 0         // syntax: #if <value>, where 0=false and !0==true. Integer arithmetic OK.
    …              // any C-code
#elif X-1 == 1     // elif means else if
    …
#else
    …
#endif

#if 0 // good for temporarily commenting out large blocks of code.
 …    // any C-code
#endif

#define HW_DEBUG
…
#ifdef HW_DEBUG
    … // any C-code, for example:
    #undef SIMULATOR
#endif
```

```
void delay_500ns(void)
{
#ifndef SIMULATOR
       delay_250ns();
       delay_250ns();
#endif
}
```

Include guards

- Include guards are used to automatically avoid to include a header file (.h-file) more than once per .c-file. This may happen if a .c-file includes several .h-files which in turn include the same .h-file

Include guards

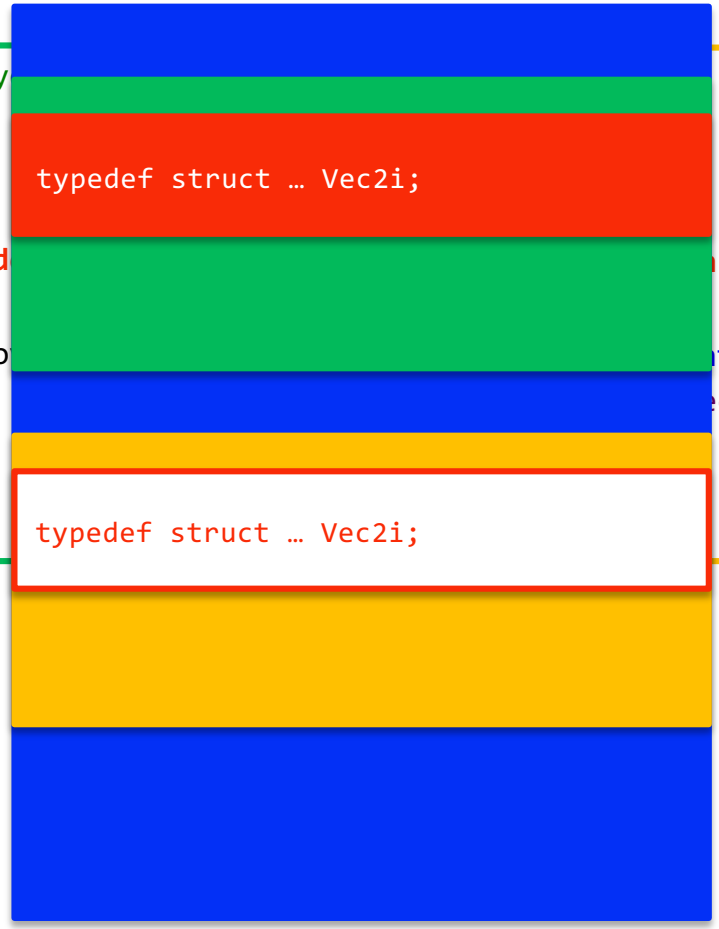
```c
// main.c
#include "player.h"
#include "enemies.h"

void main()
{
   …
}
```

c-file

```c
// vecmath.h



typedef struct {
   union {
      int v[2];
      struct {int x, y;};
   };
} Vec2i;

Vec2i add2i(Vec2i a, Vec2i b);
char isEqual(Vec2i a, Vec2i b);
```

```c
// play

#includ

void mo
```

```c
.h"

nt i,
ec2i v);
```



Include guards

```c
// main.c
#include "player.h"
#include "enemies.h"

void main()
{
   …
}
```

c-file

```c
// vecmath.h
#ifndef VECMATH_H
#define VECMATH_H

typedef struct {
   union {
      int v[2];
      struct {int x, y;};
   };
} Vec2i;

Vec2i add2i(Vec2i a, Vec2i b);
char isEqual(Vec2i a, Vec2i b);

#endif //VECMATH_H
```

```c
// player.h
#ifndef PLAYER_H
#define PLAYER_H

#include "vecmath.h"

void movePlayer(Vec2i v);


#endif //PLAYER_H
```

```c
// enemies.h
#ifndef ENEMIES_H
#define ENEMIES_H

#include "vecmath.h"

void moveEnemy(int i,
               Vec2i v);

#endif //ENEMIES_H
```

We should have include-guards on all .h files - although in this example is not required for player.h as well as enemies.h

Without including guards, vecmath.h is included twice for main.c. The second time, the compiler will complain that `Vec2i` is already defined.

.c-files are compiled separately. The preprocessor and compiler are executed individually per .c-file.

Vector addition and unions

- **Problem: Want to use a simple 32bit integer addition to add 2 vectors of 4 values each being 8bit integers. You should use a single operation to perform all additions at the same time i.e.**
  - **unsigned char a0, a1, a2, a3, b0, b1, b2, b3**
  - **(c0 c1 c2 c3) = (a0 a1 a2 a3) + (b0 b1 b2 b3)**
  - **Where c0=a0+b0, c1=a1+b1…**
  - **Use union!**

Next C Lecture – pick a topic!

- Big Endian vs Little Endian
- Constant and Inlining
- Constants and Macros with #define & #pragma
- C and C++ code – extern “C”
- Command line arguments to main()
- Object-oriented in C
- Recursive function
- Linked lists
- Storage, packing, padding
- List of Don’t s

- Any other wish list items?