

Embedded Real Time Control (ERTC)

C programming language

Alberto Morato

Pointers

- A pointer is a variable that contains the address of a variable

Simplified memory scheme

- A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups
- For example, a byte can contain a char, while four adjacent bytes can contain an int
- Similarly, a pointer is a group of (typically 4) bytes, that can hold an address

Pointers

Declaring a pointer

```
int *p;  
float *p;
```

Getting the address of a variable

Use the unary operator & like below:

```
int a = 10;  
int *p = &a;
```

In this case p is the pointer to a i.e. p contains the memory address of a

Pointers

Dereferencing a pointer

Dereferencing mean to get the value stored in the memory address pointed by a pointer

```
int a = 10;  
int *p = &a;  
int b = *p; // b contains the value of a;
```

Pointers

Dereferencing a pointer

- Every pointer points to a variable of a specific type
- it can be int, char or whatever
- void * is special: it can hold any type of pointer

⚠ dereferencing a void * does not make sense

```
int a = 10;
void *p = &a;
*p; // this leads to a warning from the compiler
int b = *p; // this is illegal
int b = *(int *)p; // this is OK
```

Pointers

Pointers and function arguments

- Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function.
- Example: swap.c
- Pointer arguments enable a function to access and change objects in the function that called it.

Pointers

Pointer arithmetic

Operators and expression can be used with pointers

```
int *p1;  
int *p2;  
p1++;  
p1 = p2 + 1;  
p2--;
```

Pointers

Pointer arithmetic

```
int num;  
int *p = &num; // assume that p = 0x2000  
p++;  
// now p = 0x2004
```

- `p` is incremented depending on the size of the type that the pointer point to: in the example above, we assume `sizeof(int)` equal to 4
- Example: `ptr-arithmetic.c`

Pointers

Pointers and arrays

- In C, there is a strong relationship between pointers and arrays
- Under the hood array subscripting use pointers

```
int a[10];  
int *pa;  
pa = &a[0];  
int x = *pa; // now x contains the value of a[0];
```

- thus, $pa + 1$ points to the element at the address successive to pa i.e. it points to $a[1]$.
- $pa + 1$ is equivalent to $\&a[1]$

Pointers

Pointers and arrays

In general, $pa + i$ points to $a[i]$ and so $*(pa + i)$ is the value of $a[i]$

```
int a[10];  
int *pa = a;
```

is equivalent to

```
int a[10];  
int *pa = &a[0];
```

since the name of the array is a synonym for the first element

Pointers


NULL pointer

A pointer can be assigned with the integer value 0 or with the value NULL to indicate the fact that it points to nothing

```
int *p = 0;
```

or

```
int *p = NULL;
```

 Dereferencing a NULL pointer cause a *segmentation fault* i.e. the program/device crash

Pointers

Multilevel pointers

- Pointers are variables and so pointers are associated to an address in memory
- This means that a pointer can point to other pointers
- To declare a multilevel pointer, for example:

```
int *p1  
int **p2;  
p2 = &p1;
```

- `int *****p;`
- Multilevel pointers can be used to manage multi-dimensional array

Structures

- A structure is a
 - collection of one or more variables
 - possibly of different types
 - grouped together under a single name for convenient handling.
- In C, structures, are called `struct`
- In some way a struct mimics an Object

Structures

- Let assume you want store cartesian coordinates
- It is possible to group these information inside a struct

```
struct coordinate
{
    int x;
    int y;
};
```

- where
 - *coordinate* is the **tag** of the structure
 - *x and y* are the members of the structure

Structures

- A struct declaration defines a **new** type
- To declare a variable of the type of a struct

```
struct coordinate point;
```

- To initialize the variable

```
struct coordinate point =  
{  
    300, 200  
};
```

- In general, access to the structure can happen using the `.` operator:

```
point.x = 300;
```

Structures

Structures can be nested, so that you can have situations like:

```
struct coordinate {
    int x;
    int y;
}
struct object {
    char name[20];
    struct coordinate point;
}
struct object spaceship = { "Spaceship", { -2, 230 } };
struct object planet;
planet.name = "Earth";
planet.point.x = 15;
```


Structures

Pointer to struct are possible

```
struct coordinate coord;  
struct coordinate *p_coord;  
p_coord = &coord;
```

There is a shortcut to access the the element of a struct directly from a pointer, using the operator ->

```
p_coord->x = 16;  
or  
int x;  
x = p_coord->x;
```

Structures

- You can pass struct or pointer to struct as an argument to a function
- If a structure has a big size, it can be more convenient to pass a pointer to the structure, as a function argument
- You can return struct as well
- You can make array of struct

```
int x;  
struct coordinate coords[10];  
x = coords[0].x;
```

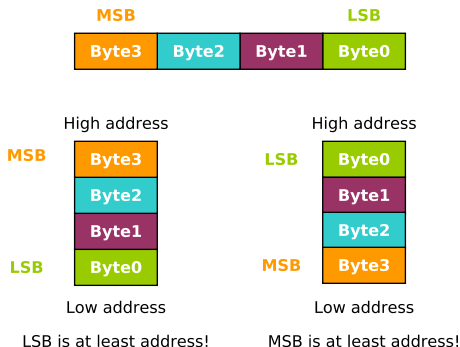
Unions

- Union types are similar to structures, except that the memory used by the member objects overlaps.
- Unions can contain an object of one type at one time, and an object of a different type at a different time, but never both objects at the same time, and are primarily used to save memory
- ⚠ Unions are endian-dependent, i.e. may not be portable between different architectures.

Endianness

Definition

Endianness refers to the order in which bytes are stored in computer memory.



Unions

```
union u
{
    char character;
    int val;
};
union
u v;
v.character = 'c'; // character is used
v.val = 10; // val is used
```

Unions can contains struct and other types

Example: union.c

Structs vs Unions

Size of structures and unions

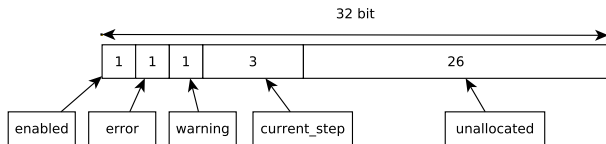
Example: `struct-union-size.c`

Bit Fields

- Used when storage space is a constraint
 - Used to pack several objects into a single machine word
 - For example flags

```
struct
{
    unsigned int enabled : 1;
    unsigned int error : 1;
    unsigned int warning : 1;
    unsigned int current_step : 3;
}
flags;
```

Bit Fields



⚠ memory alignment depends on the processor endianness

Accessing members of a bit field

```
flags.enabled = 1; // Can be 0 or 1
flags.error = 0;
is_warning_present = flags.warning;
flags.current_step = 5; // Can be 0 to 7
```


Enumerated types

- An enumeration consists of a set of named integer constants
- An enumeration type declaration gives the name of the (optional) enumeration tag and, it defines the set of named integer identifiers
- A variable of the enumeration type stores one of the values of the enumeration set defined by that type
- Variables of enum type can be used in indexing expressions and as operands of all arithmetic and relational operators.
- Enumerations provide an alternative to the `#define` pre-processor directive with the advantages that the values can be generated for you and obey normal scoping rules.

Enumerated types

- In ANSI C, the expressions that define the value of an enumerator constant always have int type.
- That means the storage associated with an enumeration variable is the storage required for a single int value.
- An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

Enumerated types

Example of enum definition

```
enum day_of_week
{
    MONDAY = 0,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

Create new type name

- C provides a facility called typedef for creating new data type names.
- For example

```
typedef int length;
```

makes the name `length` a synonym for `int`, so you can declare a variable with a custom type

```
length len
```

- Can be used with struct also

```
typedef struct coordinate coordinate  
coordinate c =  
{  
    400, -12  
};
```

The C Preprocessor

C provides certain language facilities by means of a preprocessor, which is conceptually a separate first step in compilation.

File inclusion

```
#include <filename>
```

or

```
#include "filename"
```

allow to replace the directive with the content of the file filename

The C preprocessor

Macro substitution

```
#define name replacement text
```

replaces occurrences of the word name with replacement text, for example

```
#define forever for (; ; )
forever
{
    printf("hello_ forever\n");
}
```

The C preprocessor

Macro substitution

- It is possible to define macro with arguments like:

```
#define max(a, b) ((a > b) ? a : b)
```

- Macros can lead to sneaky errors: consider the example

```
#define square(x) x * x
```

Example: square-macro.c

The C preprocessor

Conditional inclusion

Conditional statements evaluated during pre-processing

```
#if  
#ifdef  
#ifndef  
#elif
```

Example: cpp-example.c

Bitwise operators

- We use bitwise operators to manipulate the bits of an object or any integer expression
- Bitwise ($|$ $\&$ \wedge \sim) operators treat the bits as a pure binary model without concern for the values represented by these bits
Complement operator (\sim)
- The unary complement (\sim) operator works on a single operand of integer type and returns the bitwise complement of its operand; that is, a value in which each bit of the original value is flipped

Bitwise operators

Shift operators (<< and >>)

Shift operations shift the value of each bit of an operand of integer type by a specified number of positions

```
shift-expression << additive-expression
```

or

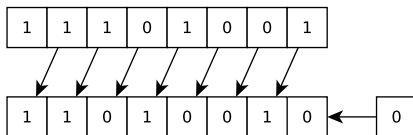
```
shift-expression >> additive-expression
```

for example

```
char var = 0x55;  
char shifed_var = var << 1; /* shifted_var contains  
* 0xAA */
```

Bitwise operators

Shift operators



left-shift by 1 position

Bitwise operators

Bitwise AND operator (&)

The binary bitwise AND (&) operator returns the bitwise AND of two operands of integer type

Bitwise OR operator (^)

The bitwise exclusive OR (^) operator returns the bitwise exclusive OR of the operands of integer type

Bitwise Inclusive OR operator (|)

The bitwise inclusive OR (|) operator operator returns the bitwise inclusive OR of the operands of integer type

Function pointers

In the future. . .