C and Assembler

- 2 documents on Canvas:
  - "Mixing C and assembly language programs"
  - "Mixing Assembly and C with AVRGCC"
- *Contains information about*
  - *how to declare functions and data that should be accessible*
  - *Which registers that are used in C parameters*

Differences between C and Java

- Some important concepts in C that are similar to Java:
  - The Array
  - The String
- Some important concepts in C that is not available (in the same way…) in Java:
  - The “Struct”
  - The “Union”
  - The “Pointer”

Arrays

- Although arrays in C look similar to the arrays used in Java, the rules are different. In C, you can declare an array like this:

```
type identifier[size];                    /* normal array */
type identifier[size1] ... [sizeN]; /* N-dim array */
type identifier[] = { value-list }; /* array with init */
```

- The *type* is the element type of the array, *identifier* is its name, and *size* the number of elements it contains.
- Note that when an array is declared, the array brackets must come after the name of the array (this is different in Java).
- An important difference between Java an C is that arrays in C are not references, but just a block of memory with a name attached to it

Arrays

- After the array is created, its size cannot be changed, because the memory it uses was reserved at compile time.
- For the same reason, the array can not be reassigned to point to a new array. (it is not a reference, just a named block of memory).
- When you create an array as a local variable, it will only exist as long as the function exists (placed on the stack).
- It would seem that arrays in C are not as flexible as the arrays we know from Java. Fortunately, there is a (more flexible) way to create arrays, by using the malloc call. This will be described in a later slide…

Arrays - examples

```
Int a1[5];                          /* OK */
int a2[] = { 1, 2, 3 };             /* OK (size calculated
                                       by compiler) */

int a5[9];
a5[10] = 7;    /* OOPS, C does not do any array bound */
a5[1000] = 9; /* checking. It accepts these statements *
a5[-5] = 5;    /* even though they are clearly WRONG */

int a3[];           /* ERROR, needs size */
a3 = a2;            /* ERROR, cannot assign arrays */
int []a4={1,2}      /* ERROR, wrong notation */
```

Strings

- In C, a string is simply an array of characters. Strings in C are expected to end with a special character '\0' (called nul). This character is used to find the end of the string. As a result, C-strings are always at least one character longer than the text they contain.
- Strings can be initialized in a number of ways:

```
char name0[6];
name0[0] = 'J';
name0[1] = 'a';
name0[2] = 's';
name0[3] = 'o';
name0[4] = 'n';
name0[5] = '\0';
char name1[] = { 'J', 'a', 's', 'o', 'n', '\0' };
```

Strings

- A string specific form of initialization can also be used:

```
char name2[6] = "Jason";
char name3[] = "Jason";
char name4[100] = "Not 100 characters long!";
```

- The string "Jason" will be translated by the compiler into special code which initializes the array. Note that it is not necessary to explicitly write down the '\0' character, this is done for you.
- As example name3 shows, it is also not necessary to specify the size of the string if it is immediately initialized.

Strings

- Functions to perform operations on strings can be imported using the header file `string.h`.
  - Examples of functionalities that are supported:
    - Search for the first occurrence of the character c in the first n bytes of string *str*.
    - Compare the first n bytes of *str1* and *str2*
    - Copy n characters from src to *dest*.
    - Append the string *src* to the end of the string *dest*.
    - Etc…

Enum

- Using an enumeration, a series of integer constants can be created. An enumeration is created as follows:

```
enum identifier {list};
```

- The identifier is a name used for the enumeration and is optional. The list is a list of constant integer variables to be created. The first variable is given the value of 0. Each variable is given the value of the previous variable plus 1. It is also possible to specify your own values.

Struct

- Structures provide a way to group a number of variables together. It is similar to a very simple Java object (without any methods). A structure can be defined as follows:

```
struct identifier {
      type variable_names;
      type variable_names; ...
} structure-variables,...;
```

- The identifier is a name used for the structure and is optional. This name can later be used to create variables of this structure type

Struct

```
struct {                              // Create a nameless
      int val1, val2;                 // struct and two
      double val3;                    // variables
} s1, s2;
s1.val1 = 5;
s2.val3 = 4.6;

struct ComplexNumber {                // Create a named
      double real, imag;              // struct and the
};
struct ComplexNumber num;             // variables
num.real = 2.5;                       // afterwards
num.imag = 0.3;
```

Struct

```
/* You can use structs in structs: */
 struct Parameters {
      ComplexNumber complex;
      double value;
};
struct Parameters param;
param.complex.real = 3.6;

/* ... and create arrays of structs: */
struct Parameters wow[5];
wow[3].complex.real = 3.6;
```

Union (not needed in this course…)

- Unions look very similar to structures.
- The difference between them is, that all variables in a union use the same memory location. When a union variable is created, enough space is allocated for the largest variable in the union. All other variables share the same memory. (Can be used for e.g. type conversion…)

```
union identifier {
      type variable_names;
      type variable_names; ...
} union-variables,...;
```

Union (not needed in this course…)

- Create a union of double and int. */

```
union MyUnion {
        int i_value;
        double d_value;
};
union MyUnion u;
u.i_value = 6;
u.d_value = 5.4; /* Overwrites the i_value !! */
```

K&R: “…a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the ”widest" member, and the alignment is appropriate for all of the types in the union”.

Union example

```
union {                         // convert between
    uint32_t  integer;          // 32 bit integer and
    uint8_t   byte[4];          // a 4-byte array
} temp32bitint;

typedef union                           // This is a union
{   struct                              // of a struct
    {   BYTE FlagBootloaderMode;        // 4 bytes
        WORD Version;                   // 2 bytes
        float Gain;                     // 4 bytes
        float GlobalGain;               // 4 bytes
        WORD Filter_Input;              // 2 bytes
    };
    char Data[16];                      // and a array of char.
} EEPROM_DATA_t;
```

Pointers

- In C when we define a pointer variable we do so by preceding its name with an asterisk ('*').
- In C we also give our pointer a type which refers to the type of data stored at the address we will be storing in our pointer.

```
int *ptr;
int number;
int *p_number;
```

Note that the asterisk is written together with the variable identifier! (in some coding standards it is directly after the type)

(Compare to the use of the X, Y and Z registers in AVR Assembler…)

Pointers

- Now, definition does NOT mean that variable is initialised!
- We want to store in ptr the address of our integer variable k. To do this we use the unary '&' operator

```
int k;
int *ptr;
ptr = &k;
int number;
int *p_number = &number;
```

- What the '&' operator does is retrieve the address of k. Now, ptr is said to "point to" k.

Pointers

- The ”dereferencing operator” is the asterisk and it is used as follows:

```
*ptr = 7;
```

- will copy 7 to the address pointed to by ptr. Thus if ptr ”points to” (contains the address of) k, the above statement will set the value of k to 7. Thus:

```
int k;
int *ptr;
ptr = &k;
*ptr = 7;
```

Pointers – parameters to functions

- In C, like in Java, functions always receive a copy of their parameters. Take the following example

```
void swap(int a, int b) {              /* WRONG */
      int temp = a;
      a = b;
      b = temp;
}
int main(void) {
      int j = 2;
      int k = 4;
      swap(j, k);
}
```

Pointers – parameters to functions

- When swap is invoked by main, the local variables j and k are passed as its parameters - copies of their values!
- `swap` only changes the values of the parameters itself. The values of j and k (in main) remain unchanged.
- Create a swap function that receives pointers to the variables it must swap as parameters:

```
void swap(int *a, int *b) {                    /* OK */
   int temp = *a;
   *a = *b;
   *b = temp;
}
swap(&j, &k);                          /* in main… */
```

Pointers – parameters to functions

- In litterature, these different ways of sending parameters to functions are called:
  - Call-by-Value
  - Call-by-Reference

*Note! (trivia knowledge..) A difference between C and C++ is that an Array parameter in C is **always** a pointer, but if Call-by-value is used in C++, the whole array is copied to the stack!*

Example of pointer to string (1/2)

Below is how you might use a *character pointer* to keep track of a string.

```
char  label[] = "Single";
char  *labelPtr;
labelPtr = label;          // note that & is not used!
```

We would have something like the following in memory (e.g., assuming the array label started at memory address 2000):

```
label @2000:      ------------------------------
                  | S | i | n | g | l | e | \0 |
                  ------------------------------

labelPtr @4000:   --------
                  | 2000 |
                  --------
```

Example of pointer to string (2/2)

Note: Since we assigned the pointer the address of an *array of characters*, the pointer must be a *character pointer* -- the types must match.

Also, to assign the address of an array to a pointer, we do not use the *address-of* (&) operator since the name of an array (like label) behaves like the address of that array in this context.

Thus the statement in prevoius slide:

```
labelPtr = label;
```

…and not:

```
labelPtr = &label;
```

Pointers and Strings

- A string-copy function with pointers:

```
char *my_strcpy(char *destination, char *source) {
   char *p = destination;
   while (*source != '\0') {
       *p = *source;
       p++;
       source++;
   }
   *p = '\0';
   return destination;
 }
```

Pointers and Strings

- A string-copy function with array notation:

```
char *my_strcpy(char dest[], char source[]) {
   int i = 0;
       while (source[i] != '\0') {
       dest[i] = source[i];
       i++;
       }
   dest[i] = '\0';
   return dest;
 }
```

- Here, you need to know that call-by-reference will be used, since arrays are never sent as call-by-value… (and it would not work with []…)

Pointers and Strings

- Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what actually gets passed is the address of the first element of each array.
- Thus, the numerical value of the parameter passed is the same whether we use a character pointer or an array name as a parameter.
- This would tend to imply that source[i] is the same as *(p+i)!
- => a[i] can be replaced with *(a + i)

Array as a pointer…

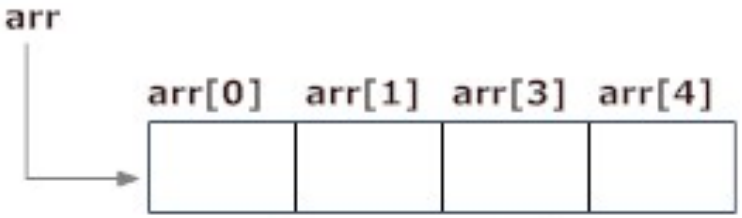


Figure: Array as Pointer

Consider the array:

```
int arr[4];
```

In C, the name of the array always points to the address of the first element of an array. In the above example, **arr** and **&arr[0]** points to the address of the first element, so:

```
&arr[0] is equivalent to arr
```

Since the addresses of both are the same, the values of arr and &arr[0] are also the same.

```
arr[0] is equivalent to *arr (value of an address of the pointer)
```

Pointers and Structures

```
struct tag {
        char  lname[20];  // last name
        char  fname[20];  // first name
        int   age;
        float rate;
 };
struct tag my_struct;
struct tag *st_ptr;
st_ptr = &my_struct;     // point to my_struct
(*st_ptr).age = 63       // This way is…
st_ptr->age = 63         // equivalent to this way!
```

Dynamic allocation of memory

- When you ask för allocated dynamic memory, you get a pointer to it:

```
myType *iptr;
   iptr = malloc(10 * sizeof(myType));
if (iptr == NULL) {
    /* handle error here */
}
```

- The allocating function (such as malloc()) returns a pointer. The type of this pointer is “void” (i.e., it points to something without a type). This void pointer can be assigned to a pointer variable of any type.

Summary C concepts

- What can we do when we put all these concepts together?
- Well, we can actually mimic the object orientation, by:
  - Use **Structs** as **Classes**. Elements in the struct would then correspond to attributes in the object.
  - **Pointers** to these structs would be **object refs**
  - Use **Functions** as **methods**. These need to take the object reference as a parameter, in order to “know” which object it is representing.
  - Structs and functions can be placed in **files** corresponding to **Classes**.
  - Allocate data using `malloc`, so **instances** can be created

Precedence and order of evaluation

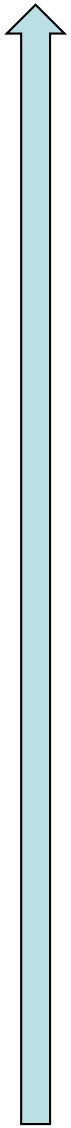
- `x = 50 + 10 / 2 - 20 * 4;`
- Answer is 40 (left-to-right) or -25 (C)?
- Depends on precedence and order of evaluation!
- Math may be clear, but precedence in other situations are not always easy to handle…
- ***DO NOT* memorize the Table of Operator Precedence and Associatively in C.**
  ***DO* use '(' and ')' to make your program clear!**

```
x = 50 + 10 / 2 - 20 * 4;   or
x = 50 + (10 / 2) - (20 * 4); ?
```

Kernighan & Ritchie Table 2.1, p 48

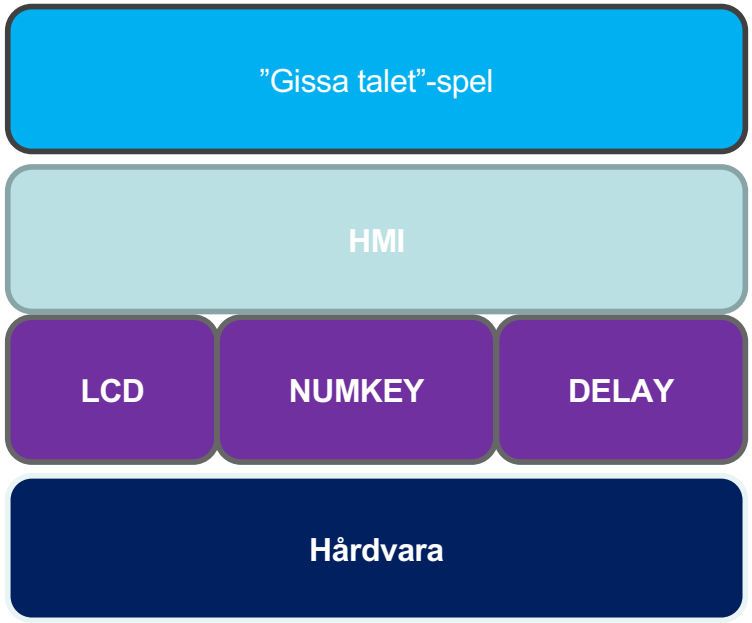
Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, *, /, and % all have the same precedence, which is higher than that of binary + and -.

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right



Unary & +, -, and * have higher precedence than the binary forms.

A layered approach (architecture) to programming:



- Split the program into different ”layers”, each responsible for a specific type of operation, e.g.:
  - HW drivers
  - Support functions (HMI in our example)
  - Application logic

Code development

- When developing code for embedded systems, a layered approach is often used.
- Each layer;
  - uses services provided by lower layers (or same layer)
  - Provides services to higher layers (and same layer)
- This way, the level of abstraction is higher in the upper layers.
- If, for example, the HW needs to be replaced, this will only affect the lowest layer(s)
- Bottom-up (easy to foresee what is needed) or Top-down design (problem needs to be broken down into smaller/simpler problems can be used

Extrabilder

ÖK: All access to array data is handled using pointer arithmetic. Consider the following code:

When you *subscript* an array the number in the brackets indicates the number of elements past the first element in the array. The *first* element in a C array is number 0. This is because the notation `myArray[0]` is interpreted as a jump 0 elements past the first element in the `myArray` memory block.

```
int myIntArray[8];
int *myIntPtr;
myIntArray[0] = 5; //first element of array set to 5
myIntArray[1] = 10; //second element of array set to 10
// set myIntPtr to point to first element in myIntArray
myIntPtr = &myIntArray[0];
myIntPtr = myIntArray; //same effect as preceding line

 // this equivalency expression is true:
*(myIntPtr + 1) == myIntArray[1];
```

ÖK: Comments to previous slide

- The tricky part of the **Example above** is the last statement. Pointer arithmetic allows us to specify the int sized block of memory next to myIntPtr with the expression myIntPtr + 1. Since we know that arrays are always stored in contiguous blocks of memory, it follows that the int sized block of memory next to myIntArray[0] must be myIntArray[1].
- In general, the expressions *(myIntPtr + *x*) and myIntArray[*x*] are equivalent when myIntPtr points to the first member of myIntArray[]. Because the subscript square brackets are an operator, the expressions myIntPtr[*x*] and myIntArray[*x*] are also equivalent. The subscript operator checks the underlying type of myIntPtr and, finding that it points to an int, jumps over *x* int sized blocks.
- Be careful! The apparent symmetry between pointers and arrays emerges from the way their related operators work. Arrays and pointers are not fundamentally the same.

ÖK: Restate the various rules you need to follow when using pointers (from Purdum): (1/2)

A pointer variable must be defined using an asterisk in the definition, such as:

```
int *ptr;
```

which defines a pointer that is used with an `int` variable.

The scale of the pointer is determined at the time the pointer variable is defined. The pointer's type specifier determines the scalar. The scalar is used to determine how many bytes are to be manipulated by the pointer.

A pointer never points to anything useful until it is initialized. The address of operator is used to initialize a pointer with the lvalue of what is being pointed to:

```
ptr = &myVariable;
```

ÖK: Restate the various rules you need to follow when using pointers: (2/2)

The address of operator (&) causes the lvalue of the variable (myVariable) to be fetched and that value is then assigned to the rvalue of the pointer variable (ptr).

After a pointer is initialized, you can use indirection to change the rvalue of the variable being pointed to. Therefore, the statements:

- `int myVariable;`
- `int *ptr;`
- `ptr = &myVariable;`
- `*ptr = 10;`

have the effect of assigning the value 10 into myVariable using the indirection operator (*) and ptr.