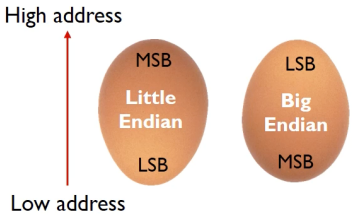
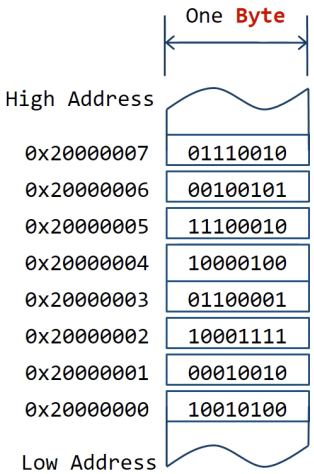
Práce s referencemi a ukazateli - C- pointer

Memory Address

- Memory is byte addressable
  - Memory is an array of bytes
  - Each byte has a memory address
  - Smallest data that can be addressed is a byte
  - Each address has 32 bits (ARM Cortex-M)
- An object may occupy multiple bytes
  - A word has four bytes
  - Word: Little endian *vs* Big endian



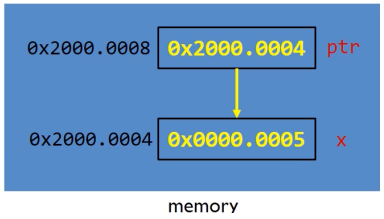


Reference and Dereference Operators

Reference Operator (&)
- Expression &x returns the address of variable x
- Ampersand (&) is called *reference operator or address-of operator*

Dereference Operator (*)
- Expression *p returns the value of the variable to which p points
- The asterisk (*) is called *dereference operator*

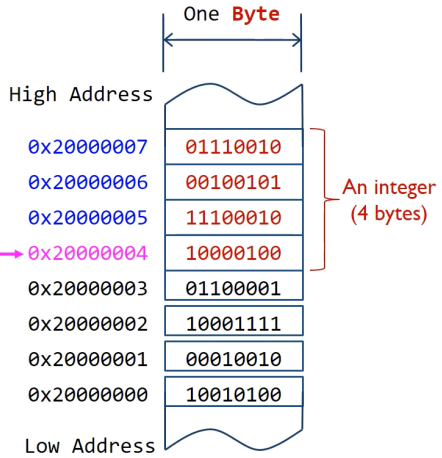
```
int x = 1;
int * ptr;  // Define a pointer that
            // can point to an integer
ptr = & x;  // Make the pointer points
            // to variable x
*ptr = 5;   // Dereferencing
```



Pointer

- The value of a *pointer* is simply the memory address of some variable
  - If a variable occupies multiple bytes in memory, its address is the lowest address of all bytes it occupies.
  - Address of the integer = 0x20000004, regardless of the endian
- Define pointers

```
int * ip;     // ip can hold the address of an integer
float * fp;   // fp can hold the address of a float
double * dp; // dp can hold the address of a double
char * cp;    // cp can hold the address of a character
```



Práce s referencemi a ukazateli - C- pointer

Reference and Dereference Operators

Reference Operator (&)
- Expression &x returns the address of variable x
- Ampersand (&) is called *reference operator* or *address-of operator*

Dereference Operator (*)
- Expression *p returns the value of the variable to which p points
- The asterisk (*) is called *dereference operator*

Summary:

```
& var:  address of var
* ptr:  value pointed by ptr
```

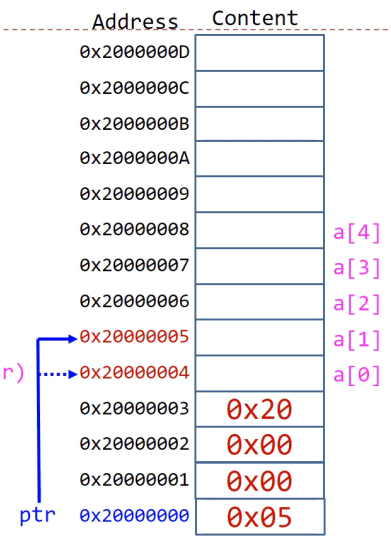
Pointer Arithmetic

- ptr is pointer, and stored at 0x20000000
- ptr = 0x20000004
- What happens if:

  ptr++;
- Is it true? ptr = 0x20000005

```
char * ptr;
char a[5];
ptr = & a[0];
ptr ++;
```

ptr = ptr + sizeof(char)



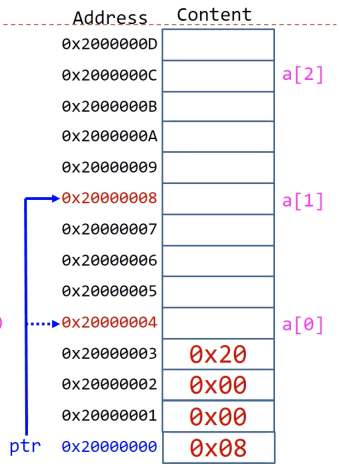
Pointer Arithmetic

- ptr is pointer, and stored at 0x20000000
- ptr = 0x20000004
- What happens if:

  ptr++;
- Is it true? ptr = 0x20000005

```
int * ptr;
int a[5];
ptr = & a[0];
ptr ++;
```

ptr = ptr + sizeof(int)

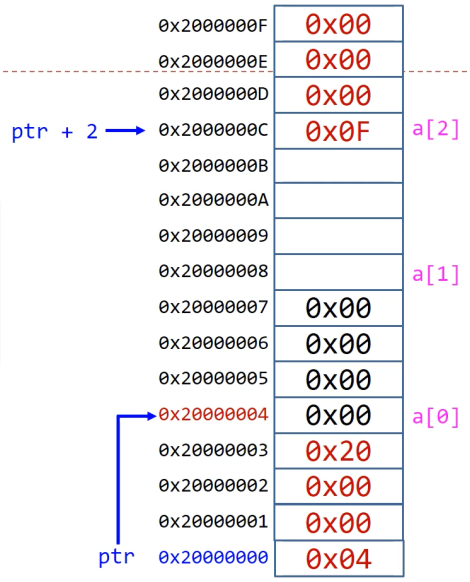


Práce s referencemi a ukazateli - C- pointer

Pointer and Array

```
int *ptr;  // Assume stored at 0x20000000
int a[5];  // Assume started at 0x20000004
*a = 7;    // array[0] = 7;
ptr = a;   // ptr points to array[0]
*(ptr + 2) = 15; // array[2] = 15
```

```
ptr = ptr + 2 * sizeof (int)
```



Operator Precedence

- ++/-- have greater precedence than *
  - *ptr++
    - Equivalent to *(ptr++)
    - write/read the value pointed, and then increment the pointer
    - x = *ptr++; is equivalent to:

```
x = *ptr;
ptr++;
```

  - (*ptr)++
    - The value pointed by ptr is incremented by 1.
    - x = (*ptr)++; is equivalent to

```
x = *ptr
*ptr = * ptr + 1;
```

Hard-coded Pointer: Set pin PA.5 to high

- Example 1: Define an address

```
#define GPIOA_ODR   0x48000014
*((uint32_t *) GPIOA_ODR) |= 1<<5;
```

- Example 2: Define a pointer

```
#define GPIOA_ODR   ((uint32_t *) 0x48000014)
*GPIOA_ODR |= 1<<5;
```

- Example 3: Define a de-referenced pointer

```
#define GPIOA_ODR   (*((uint32_t *) 0x48000014))
GPIOA_ODR |= 1<<5;
```

Práce s referencemi a ukazateli - C- pointer

Hard-coded Pointer: Read GPIO pins

- Example 1: Define an address

```
#define GPIOA_IDR   0x48000010
data = *((volatile uint32_t *) GPIOA_IDR);
```

Add `volatile` to force compiler to read a new value each time

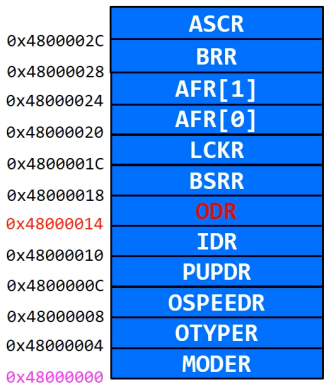
- Example 2: Define a pointer

```
#define GPIOA_IDR   ((volatile uint32_t *) 0x48000010)
data = * GPIOA_IDR;
```

- Example 3: Define a de-referenced pointer

```
#define GPIOA_IDR   (*((volatile uint32_t *) 0x48000010))
data = GPIOA_IDR;
```

A More Convenient Approach: Define a pointer pointing to a structure




```
typedef struct {
  volatile uint32_t MODER;    // Mode register
  volatile uint32_t OTYPER;   // Output type register
  volatile uint32_t OSPEEDR;  // Output speed register
  volatile uint32_t PUPDR;    // Pull-up/pull-down register
  volatile uint32_t IDR;      // Input data register
  volatile uint32_t ODR;      // Output data register
  volatile uint32_t BSRR;     // Bit set/reset register
  volatile uint32_t LCKR;     // Configuration lock register
  volatile uint32_t AFR[2];   // Alternate function registers
  volatile uint32_t BRR;      // Bit Reset register
  volatile uint32_t ASCR;     // Analog switch control register
} GPIO_TypeDef;

// Casting memory address to a pointer
#define GPIOA ((GPIO_TypeDef *) 0x48000000)
```

```
GPIOA->ODR |= 1UL<<5;
```