CSC 411

Computer Organization (Spring 2024)
Lecture 5: Casting, Byte ordering, Pointers

Prof. Marco Alvarez, University of Rhode Island

Quick notes

- Getting access to a Linux machine (for windows users)
  - download your favorite OS and install
    - standalone or side-by-side with Windows
  - built-in Unix subsystem for Windows 10
    - https://docs.microsoft.com/en-us/windows/wsl/install-win10
  - use a virtual machine

```
$ gcc hello.c -o prog
$ gcc -S hello.c -o prog
$ hexdump prog
$ xxd prog
```

Casting in C

- Constants
  - considered as signed integers by default, unless the U suffix is included, e.g. `502123U`
- Casting
  - changes the way the data is interpreted, the bit sequence is **maintained**
    - this IS NOT the same as converting a positive value d into its negative $-d$
    - with two's complement, conversions between signed and unsigned basically add or subtract 2^w
  - **explicit casting**
    - requires the specification of the data type — parenthesized cast
  - **implicit casting**
    - occurs automatically in assignments and function calls
    - e.g. assigning an unsigned integer into a signed integer

Casting into “bigger” data types

- Sign extension
  - transform a w-bit integer into an integer with a **larger** bit-width $w + d$, preserving the same value
  - how? just make d copies of the MSB (extension)

				−8	4	2	1	
				0	1	1	0	6
−128	64	32	16	8	4	2	1	
0	0	0	0	0	1	1	0	6
				−8	4	2	1	
				1	1	1	0	−2
−128	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	0	−2

Practice

- What is the decimal value of this 64-bit number represented using two’s complement?

0xFFFFFFFFFFFFFFF8

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111000

Casting into “smaller” data types

- Truncation
  - transform a $w + d$-bit integer into an integer with a **smaller** bit-width w, preserving the rightmost w bits
  - how? just drop top d bits, $\mathrm{mod}\ 2^w$ for unsigned integers

−128	64	32	16	8	4	2	1	
0	1	0	1	0	1	1	0	86
				−8	4	2	1	
				0	1	1	0	6
−128	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	0	−2
				−8	4	2	1	
				1	1	1	0	−2

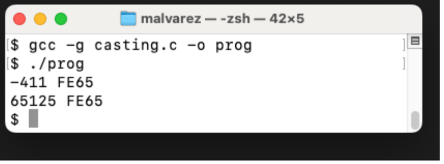
no sign change

−128	64	32	16	8	4	2	1	
1	0	0	1	0	1	1	0	−106
				−8	4	2	1	
				0	1	1	0	6
−128	64	32	16	8	4	2	1	
0	0	1	1	1	0	1	0	58
				−8	4	2	1	
				1	0	1	0	−6

sign change

What is the output?

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

int main() {
    // 411 is 0x019B
    // −411 is 0xFE65
    int16_t var1 = −411;
    uint16_t var2 = (uint16_t) var1;

    printf("%hd %x\n", var1, var1);
    printf(“%hu %x\n", var2, var2);

    return 0;
}
```



```
$ gcc -g casting.c -o prog
$ ./prog
-411 FE65
65125 FE65
$
```

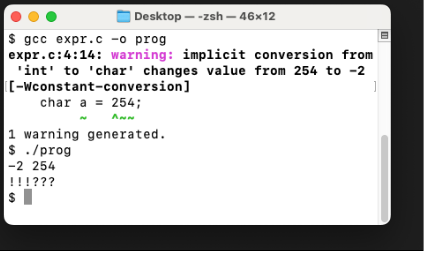
Casting in C

- Expressions (comparisons)
  - if an expression contains signed and unsigned integers, all signed values are **<u>implicitly casted</u>** to **unsigned**
    - **-2147483647 is the most negative number**

Expression	Type	Evaluation
0 == 0U	unsigned	true
-1 < 0	signed	true
-1 < 0U	unsigned	false
2147483647 > -2147483647 - 1	signed	true
2147483647U > -2147483647 - 1	unsigned	false
2147483647 > (int)2147483648U	signed	true
-1 > -2	signed	true
(unsigned)-1 > -2	unsigned	true

What is the output?

```
#include <stdio.h>

int main() {
    char a = 254;
    unsigned char b = 254;
    unsigned int c = 0;

    printf("%d %d\n", a, b);

    if (-1 < c) {
        printf("yay\n");
    } else {
        printf("!!!???\n");
    }
}
```



```
$ gcc expr.c -o prog
expr.c:4:14: warning: implicit conversion from
 'int' to 'char' changes value from 254 to -2
[-Wconstant-conversion]
    char a = 254;
         ~   ^~~
1 warning generated.
$ ./prog
-2 254
!!!???
$ 
```

Memory organization

Memory organization

- Memory as a **byte array**
  - used to store **data and instructions** for computer programs
  - <u>contiguous sequence of bytes</u>
    - each byte can be individually accessed using its **unique address**
- Memory address
  - **<u>unique</u>** numerical identifier assigned to each byte in memory
  - a **pointer** variable stores a memory address, providing indirect access to the data stored at that location

Memory organization

- Data Representation
  - variables are stored in memory as sequences of bytes
  - interpretation depends on their type
    - integers, floating-point numbers, characters, etc.
- Operating system provides a private address space to each **"process"**
  - a process is a program being executed
  - an address space is one of those enormous arrays of bytes
  - each program can see only its own code and data within its enormous array

Machine words

- Computers have a **"word size"**
  - usually the size of integer-valued data and of memory addresses
    - word size = 32 provides an address range of $0 \ldots 2^{32} - 1$
      - 4294967295 bytes or 4GB
    - word size = 64 provides an address range of $0 \ldots 2^{64} - 1$
      - 18446744073709551615 bytes or 16EB
- Machines support multiple data formats
  - fractions or multiples of word size

Byte ordering

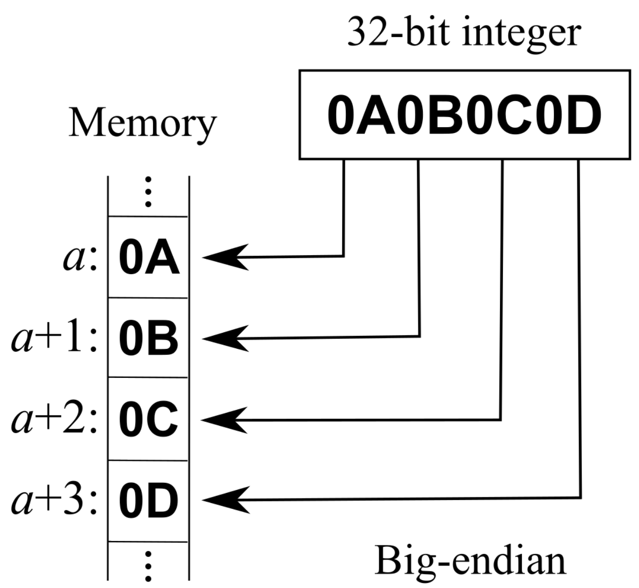
```
x = 0x1A2B3C4D
assume &x is 0x010
```

- Big endian
  - stores the most significant byte at the lowest memory address
  - IBM PowerPC, Motorola 68000, SPARC, network byte order

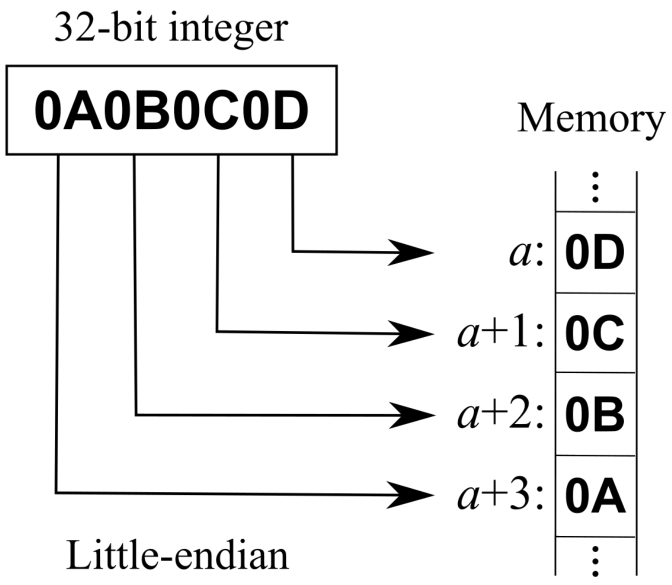
0x00D	0x00E	0x00F	0x010	0x011	0x012	0x013	0x014	0x015	0x016
			1A	2B	3C	4D			

- Little endian
  - stores the least significant byte at the lowest memory address
  - intel x86, ARM, RISC-V, MIPS

0x00D	0x00E	0x00F	0x010	0x011	0x012	0x013	0x014	0x015	0x016
			4D	3C	2B	1A			



https://en.wikipedia.org/wiki/Endianness



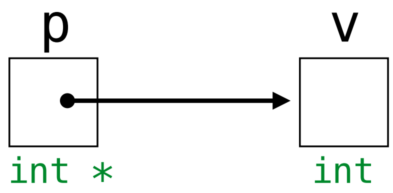
https://en.wikipedia.org/wiki/Endianness

Pointers

Variables and pointers

- Every variable (regardless of **scope**) exists at some **memory address**
  - a memory address corresponds to a unique location
- The compiler translates names into memory addresses when generating machine level code
  - C allows programmers to manipulate variables and their memory addresses directly

A **pointer** is a variable that stores the address of another variable



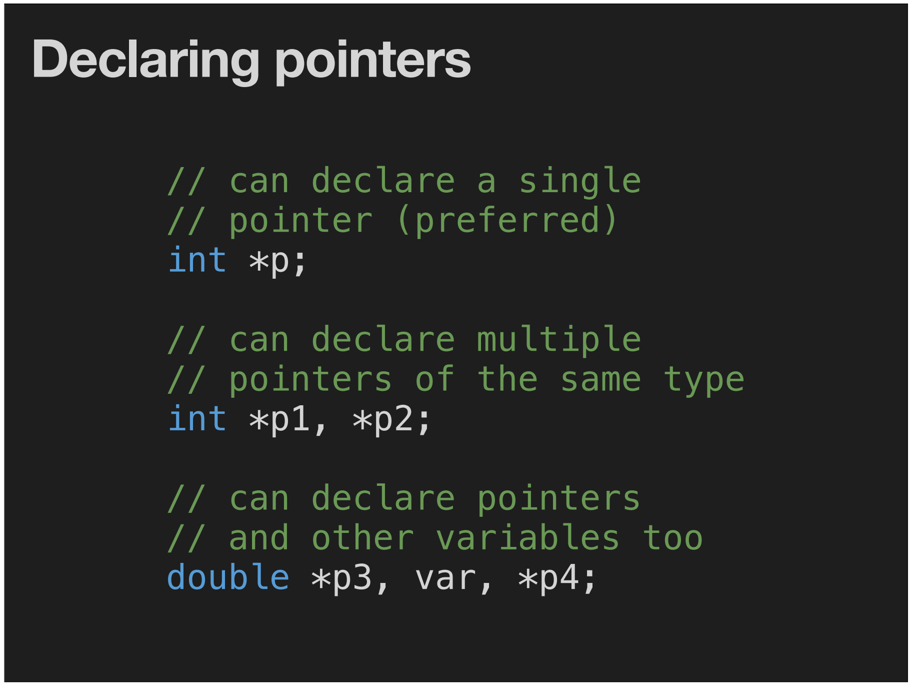
Pointers in C

- Pointers must be declared before use
  - **pointer type** must be specified
- Pointer operators
  - **address-of operator**: used to get the memory address of another variable/object

&

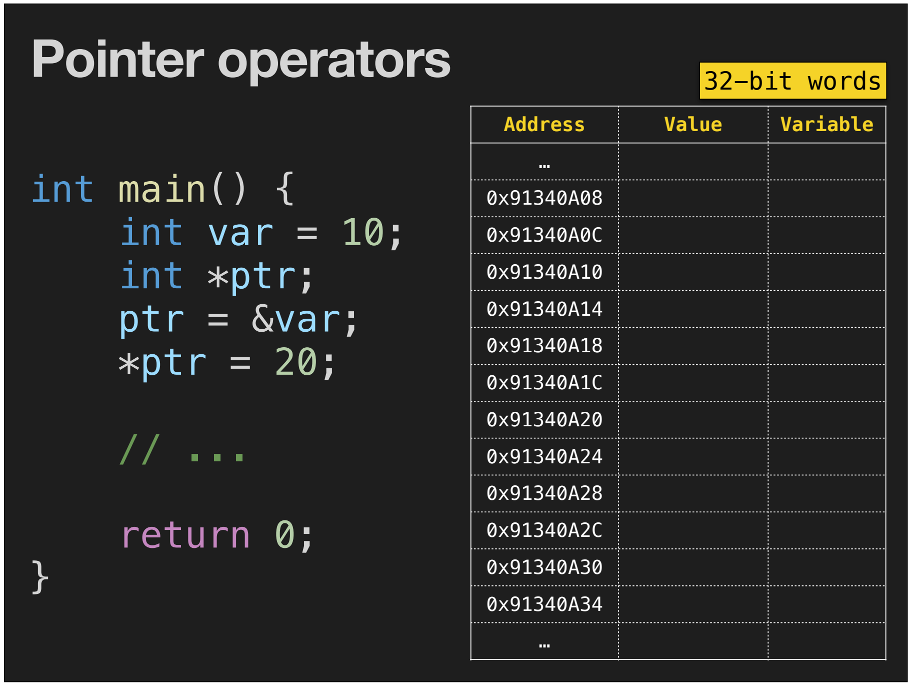
  - **dereference operator**: used to get the actual value of a given memory address (dereferencing a pointer)

*



Declaring pointers

```
// can declare a single
// pointer (preferred)
int *p;

// can declare multiple
// pointers of the same type
int *p1, *p2;

// can declare pointers
// and other variables too
double *p3, var, *p4;
```

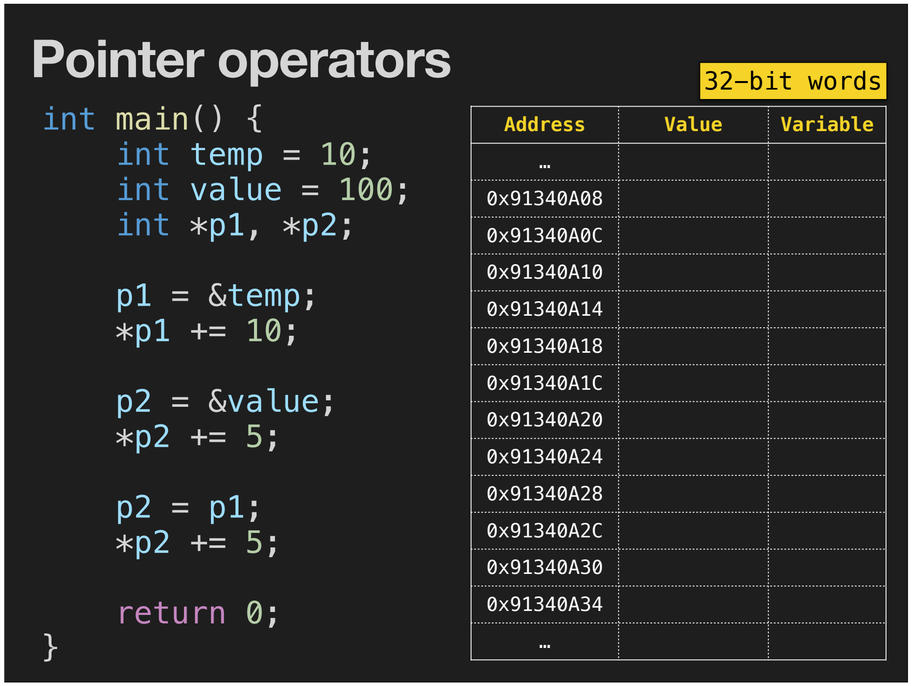


Pointer operators

```
int main() {
    int var = 10;
    int *ptr;
    ptr = &var;
    *ptr = 20;

    // ...

    return 0;
}
```

32-bit words

Address	Value	Variable
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		

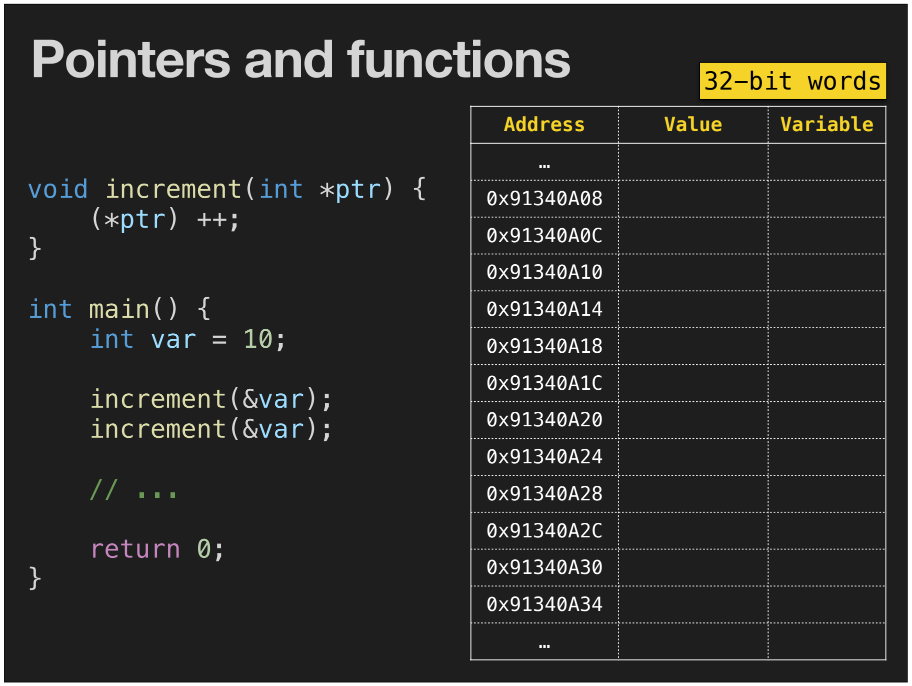


Pointer operators

```
int main() {
    int temp = 10;
    int value = 100;
    int *p1, *p2;

    p1 = &temp;
    *p1 += 10;

    p2 = &value;
    *p2 += 5;

    p2 = p1;
    *p2 += 5;

    return 0;
}
```

32-bit words

Address	Value	Variable
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		



Pointers and functions

```
void increment(int *ptr) {
    (*ptr) ++;
}

int main() {
    int var = 10;

    increment(&var);
    increment(&var);

    // ...

    return 0;
}
```

32-bit words

Address	Value	Variable
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		

Pointers in C

- Declaring a pointer just allocates space for the pointer
  - 4 (32-bit architecture) or 8 bytes (64-bit architecture)
  - it DOES NOT allocate additional memory
- Generic pointers (`void *`)
  - use properly to avoid bugs and vulnerabilities
- Pointer to functions

```
// declare and initialize
int (*func) (int, int) = &my_func;
// use
c = (*func)(a, b);
```

Pointers in C

- The `null` pointer
  - reads/writes with a null pointer can generate a **segmentation fault** signal
  - used to represent the absence of a value, is a pointer with all zeros `0x00000000`
- Pointers and arrays
  - when declaring an array, the array is treated as a “constant pointer” to the first element of the array
  - array names are NOT variables

Pointer arithmetic

- As pointers hold memory addresses, we can add values to it
  - can think about addresses as unsigned integers
- Must be careful !
  - p+1 does NOT add 1 “byte” to the memory address, it adds the size of the variable pointed by p
- Can use pointer arithmetic to work with arrays

`a[i]` is equivalent to `*(a+i)`

Changing a pointer inside a function

```
#include <stdio.h>

void seek(int *array, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (*array == key) {
            return;
        }
        array ++;
    }
}

int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(data, 3, 5);
    printf("%d\n", *p);

    return 0;
}
```

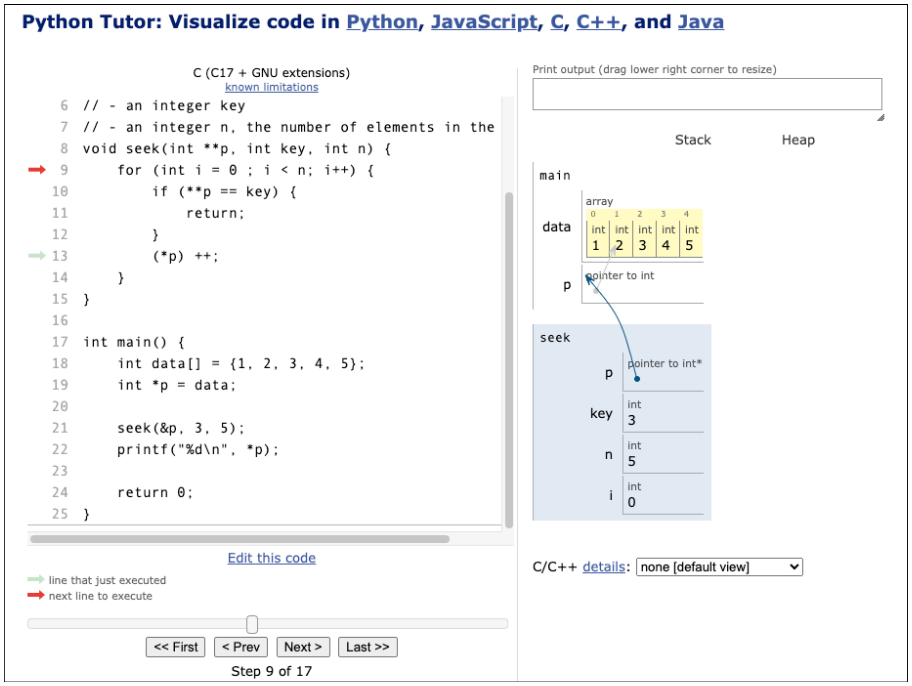
does it work?

Using double pointers

```c
// function to search for a key in an array
// - pointer to an array of integers
// - an integer key
// - an integer n, the number of elements

void seek(int **p, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (**p == key) {
            return;
        }
        (*p) ++;
    }
}
```

Using double pointers

```c
int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(&p, 3, 5);
    printf("%d\n", *p);

    return 0;
}
```



Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

C (C17 + GNU extensions)
known limitations

```c
// - an integer key
// - an integer n, the number of elements in the
void seek(int **p, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (**p == key) {
            return;
        }
        (*p) ++;
    }
}

int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(&p, 3, 5);
    printf("%d\n", *p);

    return 0;
}
```

Edit this code

line that just executed
next line to execute

Step 9 of 17

Print output (drag lower right corner to resize)

C/C++ details: none [default view]