CS 24000 - Programming In C

Week Two: Basic C Program Organization and Data Types

Zhiyuan Li
Department of Computer Science
Purdue University, USA

The Simplest C Program


```
int main() {
  return 0;
 }
```

- C programs must have a `main()` function
- `main()` is called first when the program is started by the OS
- `main()` returns an integer
- without a `return` statement, undefined value is returned

PURDUE
UNIVERSITY

Compiling C Programs

- The source code, i.e. the program listing, has a file extension ".c"
- The source code is often split into multiple ".c" files to make the program better organized logically
- We can use a C compiler, e.g. gcc, to
  - translate all *.c files into machine code before combining them into a single executable file (ready to be loaded for execution)

    gcc *.c (generate executable a.out)

    or gcc –o filename *.c (generate executable *filename*
- *Or* we can first store the machine code in multiple *object files* (*.o) by doing "gcc –c *.c"
- Next, we can "link" the object files into the executable, by
  - gcc *.o or gcc –o filename *.o

PURDUE UNIVERSITY

Separate Compilation

- One of the benefits of keeping *.o files is that

  after we modify a single .c file, we need to recompile only that file into a new .o file

  gcc –c file1.c

- Next we re-link all *.o together
- A .c file is also known as a *module* by some people.
- A .c file may contain
  - One or more functions (but exactly one .c file contains the main function)

  -- zero or more external declarations

- A .c file may also contain #include macros to import "header files", i.e. the .h files ("abc.h" or <stdio.h>)
  - Standard header files are quoted by < >

PURDUE UNIVERSITY

Definitions and Declarations

- For a function or a global variable, C makes a tricky distinction between its "definition" and their "declaration"
  - If not understood well, this can cause a lot of troubles for organizing the program files
- A function is *defined* by having both its header and its body written
  - The definition is **private** to the file in which it is written if the definition is labeled by a *static* keyword, e.g. **static int f(){}**
  - A private definition applies to the defining file only, and it does not conflict with functions with the same name defined in other files
  - Without the *static* keyword, the function definition applies to all files that have no private definition of a function of the same name
    - However, those files must *declare* the function, with the keyword *extern* before they can make a reference to it.

PURDUE UNIVERSITY

Global Variables

- A global variable can be referenced by more than one function.
  - It is not **defined** inside any function
  - Where it is defined, all information required for the compiler to allocate memory for it must be present
  - E.g. int A[1000];
- The definition of a global variable is private to its defining file if it is labeled by the *static* keyword,
    - e.g. **static int A [8];**
  - A private definition applies to the defining file only, and it does not conflict with global variables with the same name defined in other files

PURDUE
UNIVERSITY

– Without the static keyword, the definition applies to all files that have no private definition of a global variable of the same name

  • However, those files must *declare* the global variable, with the keyword *extern* before they can make a reference to it.

– Within the defining file, the definition of a global variable cannot be referenced by program statements prior to the definition, unless

  • There is a declaration of the variable with the *extern* key word.

– The extern declaration of a global variable only needs to have the type information that is necessary for compiler's *type checking*

  • E.g. extern int A[];

- In extern function declaration, both return type and the argument types must be present
- We now try to compile a few programs as examples

Local Variables

- Variables defined within a function are local to that function
  - A local variable is defined and declared at the same time.
  - If a local definition is labeled by the *static* keyword, then its memory location is unchanged throughout the program execution, and it keeps its updated value from one invocation of the function to the next
  - If a local definition is not *static*, then it is *"automatic"*
    - its memory location may change in different invocations of the defining function
    - its memory location is usually in the implicit calling stack of the program
      - Sometimes it is found in a register only, as a compiler's "optimization" result

Nested Blocks

- A function body may contain nested *blocks*
  - *A local variable definition applies to the innermost block that contains it*
- A block may contain zero or more local variable definitions, followed by zero or more *executable* statements.

```
main(){
 int a;
 float b=1.0;
 a=2; {
   float a;
   a =  1.0;
   b =   2*a;
   }
  printf("%f", b/a);
}
```

PURDUE
UNIVERSITY

Kind of Statements

- As in Java, C executable statements have different kinds:
  - Assignment statements
  - IF statements
  - Loops
    - while
    - for
    - do … while
  - Compound statements (i.e. an inner block)
  - Return statements
- Function calls
- ……

Expressions

- The distinction between an executable statement and an expression is somewhat fuzzy, you can write

```
int f(){
  int v;
  v;
  return;
}
```

and the program will still compile

We'll explain various expressions as we encounter them in examples

PURDUE UNIVERSITY

Quiz 0

- Warm up your clickers...

Quiz #0 - 1

- Have you read any sections of the textbook?
- (a) **Yes**
- (b) **No**
- (c) What textbook?

Quiz #0 - 2

- Have you tried any program examples in the textbook?
- (a) **`Yes`**
- (b) **`No`**
- (c) There are program examples?

A note: Which C is for us?

- In this semester, to be consistent with the textbook
  - All exams assume ANSI C as explained in the textbook, i.e. C89
    - This is equivalent to ISO's C90
  - All labs and projects assume c89/c90

PURDUE
UNIVERSITY

Basic Data Types and Representation

- Why is it important to study how data (of various types) are represented on computers?
  - Data size may be different on different computers
    - To transfer data between different computers (e.g. with mobile computing), data may not be ready for use on the recipient without conversion (i.e. extension, truncation, and reordering)
  - Many computer applications must directly manipulate data at very low level
    - Images
    - Sound (music, …)
    - Data to be compressed/decompressed
    - Data to be encrypted/decrypted
    - Data to be camouflaged
    - … …

PURDUE UNIVERSITY

Primitive Data Types

- Integer
- Float
- Pointer (i.e. address)

PURDUE UNIVERSITY

Internal (machine) representation of data

- Data is stored in some memory components in computer hardware
  - Registers (explicit/implicit)
  - Caches
  - Main memory
  - Secondary storage
- The basic memory unit is a single *bit* that has two states: on/off or 1/0
  - A vector of n bits can represent 2^n states, or 2^n different values
  - If we assign the weight of 2^i to the i'th element of the vector, $0 \le i \le n-1$, then the n-bit vector can represent all integers in the range of $[0, 2^n - 1]$

PURDUE UNIVERSITY

Signed and Unsigned Integers

- We often need signed integers in our program
  - So, by default, an integer has a sign
- If we take the leading bit of the n-bit vector as the sign (1 for negative and 0 for non-negative)
  - this will half the magnitude of the absolute values of the representable integers
- If we just deal with nonnegative integers, then we can declare the integer variable to be *unsigned*, which doubles the representable magnitude.

Bytes

- As mentioned previously, the Unix system views a file to be a stream of bytes
- On most computers, the internal memory (i.e. the main memory) is also addressable in bytes
  – We access the next byte by incrementing, or decrementing, the address by 1.
- A byte contains 8 bits

Byte

- A byte = 8 bits
  - Decimal 0 to 255
  - Hexadecimal 00 to FF
  - Binary 00000000 to 11111111

- In C:
  - Decimal constant: `12`
  - Octal constant: `014`
  - Hexadecimal constant: `0xC`

0_{hex}	=	0_{dec}	=	0_{oct}	0	0	0	0
1_{hex}	=	1_{dec}	=	1_{oct}	0	0	0	1
2_{hex}	=	2_{dec}	=	2_{oct}	0	0	1	0
3_{hex}	=	3_{dec}	=	3_{oct}	0	0	1	1
4_{hex}	=	4_{dec}	=	4_{oct}	0	1	0	0
5_{hex}	=	5_{dec}	=	5_{oct}	0	1	0	1
6_{hex}	=	6_{dec}	=	6_{oct}	0	1	1	0
7_{hex}	=	7_{dec}	=	7_{oct}	0	1	1	1
8_{hex}	=	8_{dec}	=	10_{oct}	1	0	0	0
9_{hex}	=	9_{dec}	=	11_{oct}	1	0	0	1
A_{hex}	=	10_{dec}	=	12_{oct}	1	0	1	0
B_{hex}	=	11_{dec}	=	13_{oct}	1	0	1	1
C_{hex}	=	12_{dec}	=	14_{oct}	1	1	0	0
D_{hex}	=	13_{dec}	=	15_{oct}	1	1	0	1
E_{hex}	=	14_{dec}	=	16_{oct}	1	1	1	0
F_{hex}	=	15_{dec}	=	17_{oct}	1	1	1	1

http://en.wikipedia.org/wiki/Hexadecimal

Words

- Most internal computer operations, such as additions, multiplications, comparisons, are performed on registers, which are the fastest memory devices
  - The number of bits (of an operand) to be operated upon by a single arithmetic operation usually matches the number of bits in a register
- For convenience, we can think of the number of bits in a register as the `word size`
- The size of a memory address is typically the same as the word size
  - The word size therefore also defines the maximum amount of memory that can be manipulated by a program

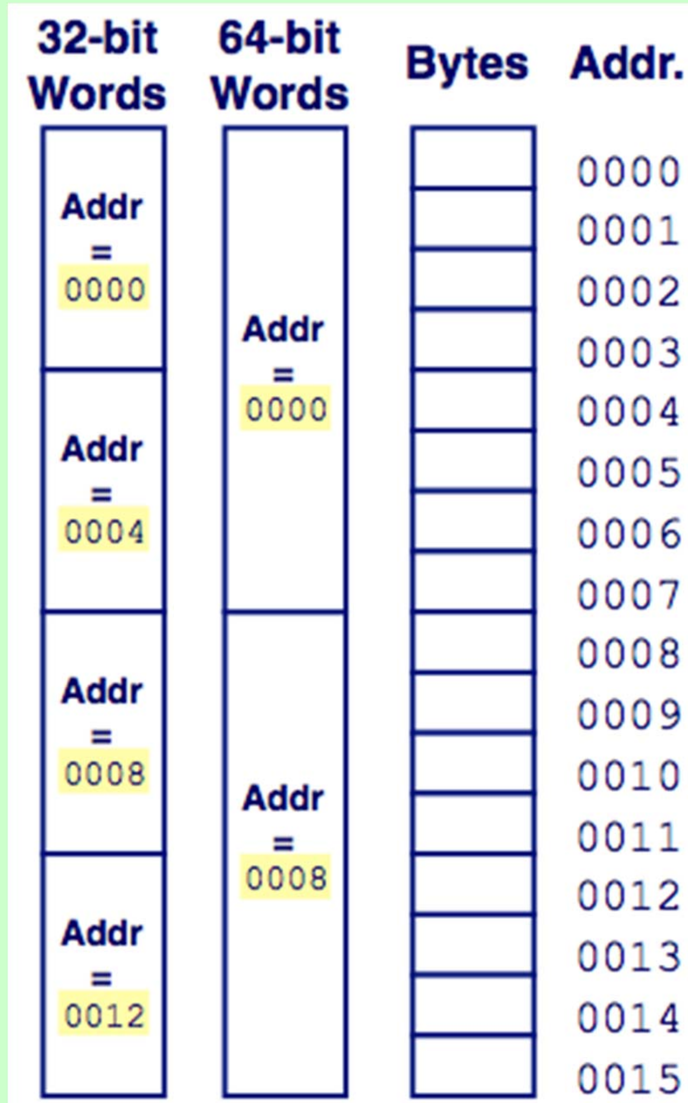
Today's general-purpose computers have either:

- 32-bit words => can address 4GB of data
- 64-bit words => 4G X 4GB

- If a number's magnitude is a multiple of the word size, then multiple hardware operations are needed to perform a single operation on the number
- If a number's magnitude is a fraction of the word size, then it is first stretched, or *extended,* to the word size before operated upon.
  - After the operation, the number of bits are reduced to restore the size of the number presentation, with the value remaining correct.

Addresses

- Addresses specify byte location in computer memory
  - address of first byte in word
  - address of following words differ by 4 (32-bit) and 8 (64-bit)



Endians

- For an integer of the size of a single byte, conceptually we can think it to take the format:

  $d_7d_6d_5d_4d_3d_2d_1d_0$

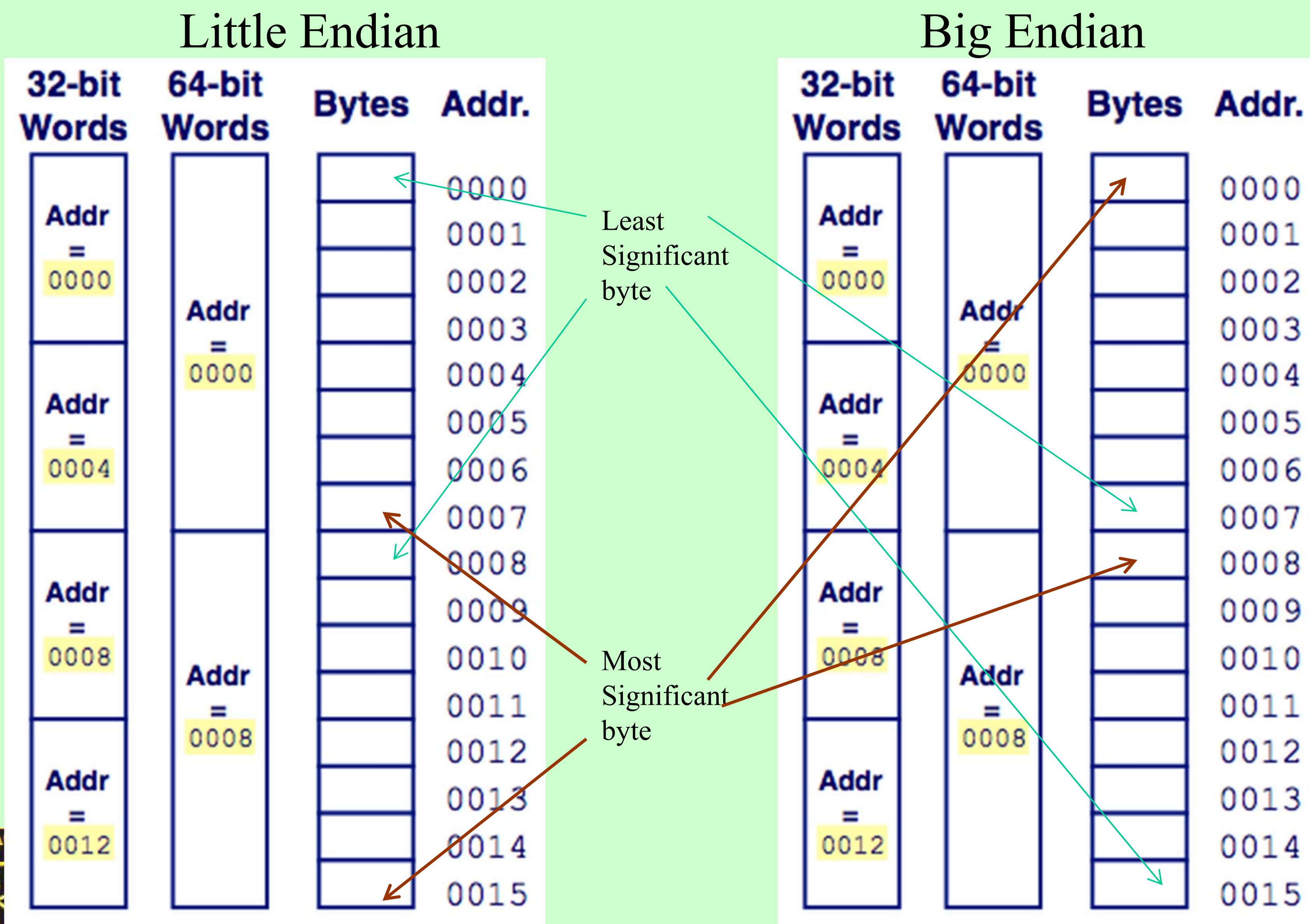
  with d_0 being the least significant bit

- For an integer that takes the word size, e.g. four bytes, the issue becomes more complicated
  - To store the number on a register, it is simple:

    $d_{31}\ldots d_{25}d_{24}\ \ d_{23},,, d_{17}d_{16}\ \ d_{15}\ldots d_9d_8\ d_7\ldots d_1d_0$

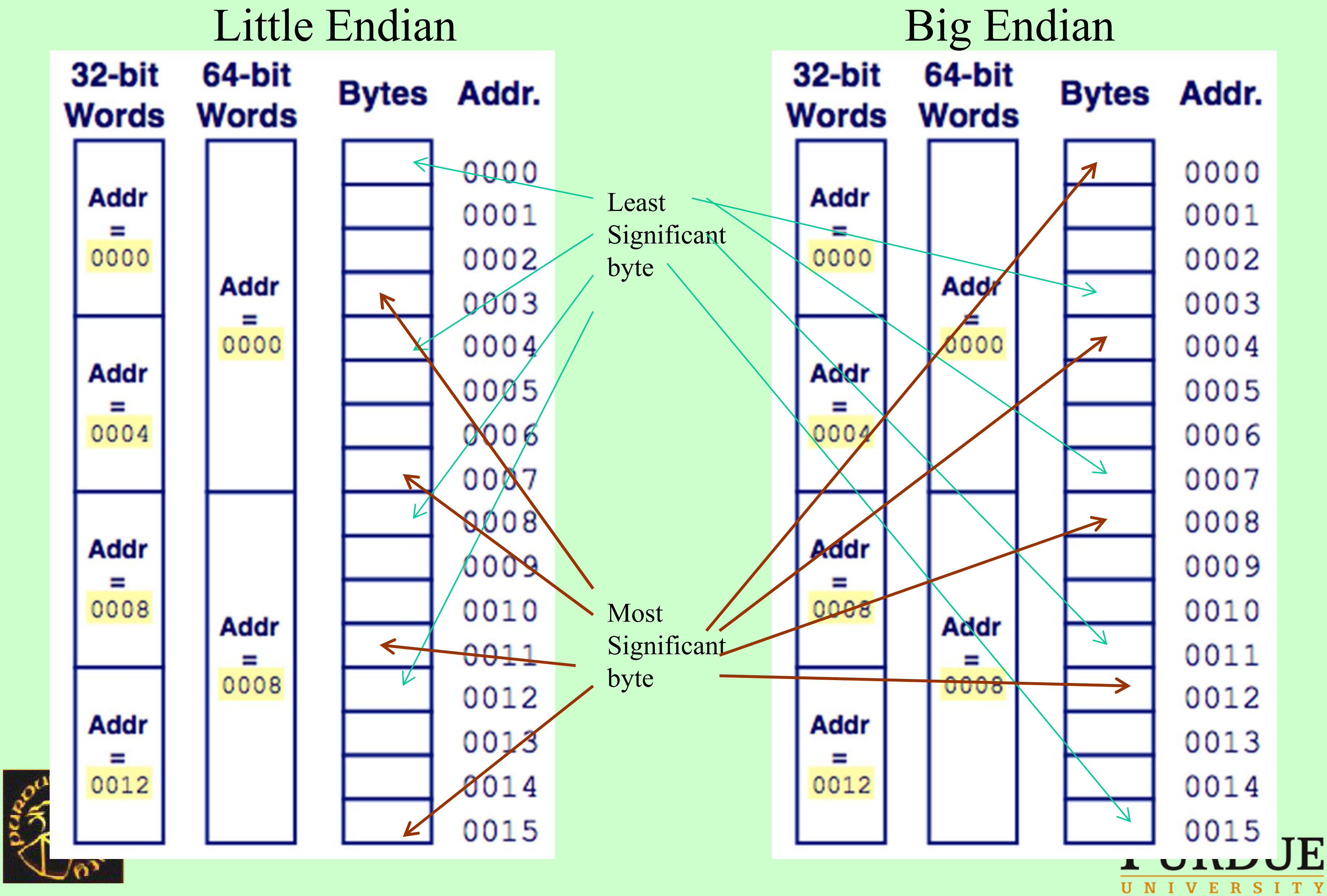
  - However, to it on the main memory, the question of byte ordering arises

64-bit word



Little Endian
Big Endian
32-bit Words
64-bit Words
Bytes
Addr.
Addr = 0000
Addr = 0004
Addr = 0008
Addr = 0012
Addr = 0000
Addr = 0008
0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
0013
0014
0015
Least Significant byte
Most Significant byte

32-bit word



Little Endian
Big Endian
32-bit Words
64-bit Words
Bytes
Addr.
Addr = 0000
Addr = 0004
Addr = 0008
Addr = 0012
Addr = 0000
Addr = 0008
0000
0001
0002
0003
0004
0005
0006
0007
0008
0009
0010
0011
0012
0013
0014
0015
Least Significant byte
Most Significant byte

Data Types

- The base data type in C
  - **`int`** - used for integer numbers
  - **`float`** - used for floating point numbers
  - **`double`** - used for large floating point numbers
  - **`char`** - used for characters
  - **`void`** - used for functions without parameters or return value
  - **`enum`** - used for enumerations
- The composite types are
  - pointers to other types
  - functions with arguments types and a return type
  - arrays of other types
  - **`struct`**s with fields of other types
  - **`union`**s of several types

Qualifiers, Modifiers & Storage

- Type qualifiers
  - **`short`** - decrease storage size
  - **`long`** - increase storage size
  - **`signed`** - request signed representation
  - **`unsigned`** - request unsigned representation

- Type modifiers
  - **`volatile`** - value may change without being written to by the program
  - **`const`** - value not expected to change

Sizes

Type	Range (32-bits)	Size in bytes
`signed char`	−128 to +127	1
`unsigned char`	0 to +255	1
`signed short int`	−32768 to +32767	2
`unsigned short int`	0 to +65535	2
`signed int`	−2147483648 to +2147483647	4
`unsigned int`	0 to +4294967295	4
`signed long int`	−2147483648 to +2147483647	4 or 8
`unsigned long int`	0 to +4294967295	4 or 8
`signed long long int`	−9223372036854775808 to +9223372036854775807	8
`unsigned long long int`	0 to +18446744073709551615	8
`float`	1×10^{-37} to 1×10^{37}	4
`double`	1×10^{-308} to 1×10^{308}	8
`long double`	1×10^{-308} to 1×10^{308}	8, 12, or 16

Character representation

- ASCII code (American Standard Code for Information Interchange): defines 128 character codes (from 0 to 127),
- In addition to the 128 standard ASCII codes there are other 128 that are known as extended ASCII, and that are platform- dependent.
- Examples:
  - The code for ‘A’ is 65
  - The code for ‘a’ is 97
  - The code for ‘b’ is 98
  - The code for ‘0’ is 48
  - The code for ‘1’ is 49

Understanding types matter…

- Some data types are not interpreted the same on different platforms, they are machine-dependent
  - **`sizeof( x )`** returns the size in bytes of the object **`x`** (either a variable or a type) on the current architecture

Declarations

- The declaration of a variable allocates storage for that variable and can initialize it

```
 int lower = 3, upper = 5;
char c = ‘\\’, line[10], he[3] = “he”;
float eps = 1.0e-5;
char arrdarr[10][10];
unsigned int x = 42U;
char* ardar[10];
char* a;
void* v;
void foo(const char[]);
```

- Without an explicit initializer local variables may contain random values (static & extern are zero initialized)

- We can call printf or use a debugger to see the exact bit patterns of a piece of data

Conversions

- What is the meaning of an operation with operands of different types?
- char c; int i; … i + c …
- The compiler will attempt to convert data types without losing information; if not possible emit a warning and convert anyway
- Conversions happen for operands, function arguments, return values and right-hand side of assignments.

- Reading assignments
  - K&R Chapters 1 to 4