Chapter 2
Introduction to Computer Organization

Learning Objectives
The objectives of this chapter are to:

- Understand number systems and number base conversions
- Understand data representation in digital computers and learn how to examine these representations in the debugger
- Understand how memory works, including memory addresses, byte ordering

There has been a dramatically increase of cybercrime, so there is demand to analyze a computer system to gather evidence after it has been hacked, which helps solve a crime. Also we may have to gain information on systems for the purpose of debugging, performance optimization, or reverse-engineering. In doing so, it is important for digital investigators to understand raw data obtained from digital devices confiscated at the crime scene; thus, they must first grasp how data that composes of 1s and 0s are encoded into the computer and how data are stored. This chapter focuses on how computers store and process data. It serves as the foundation for digital investigation from a technological perspective.

2.1 Computer Organization

A modern computer is composed of software (e.g. operating system, application software) and hardware components. These hardware includes a Central Processing Unit (CPU), main memory, input/output devices, secondary memory, and a bus to communicate between different parts. A CPU, also known as processor, receives and decodes instructions from memory. Within CPU there are several special units. One

X. Lin, *Introductory Computer Forensics*,
https://doi.org/10.1007/978-3-030-00581-8_2

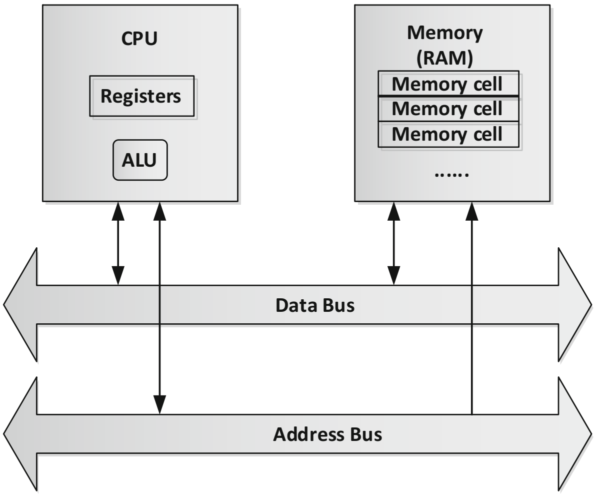

Fig. 2.1 Basic computer architecture [1]

is Arithmetic Logic Unit (ALU) that performs operations with numbers and handful of registers. Register allows quick access to data and instructions stored in memory. These registers are typically addressed from mechanisms that are different from the ones for the main memory (or RAM). Figure 2.1 depicts a basic computer architecture as was described.

Nevertheless, processor registers have limited size of storage spaces. Therefore, computer usually uses a storage hierarchy. It puts fast but expensive and small storage options close to the CPU while slower but larger and cheaper options are further away, as described in Fig. 2.2. Traditionally, computer storage is divided into primary, secondary, tertiary and off-line storage.

Primary storage in computer, also known as main memory, is RAM. It holds running programs and data that the programs use [2]. The main memory (or RAM) can be imaged as sequences of memory cells containing arrays of bits, where each bit represents a value. In actual hardware, data stored in the cells are represented as electrical impulses of on or off. Numerically, we denote these two states as 1 bit and 0 bit respectively, also known as binary numeric. These cells have a unique address (or array index) that a software instruction identifies and refers to. A sequence of 8 bits is a unit named byte. Since a typical cell stores one byte of memory, it is also known as byte-addressable memory. Each of these addresses refers to one byte of memory or points to one memory location in RAM.

PLEASE!! NOTE There are other types of addresses based on its size. These include the following: Nibble addressable memory (each address stores a nibble), bit-addressable memory (each address stores a bit), and word-addressable memory (each address stores a word).

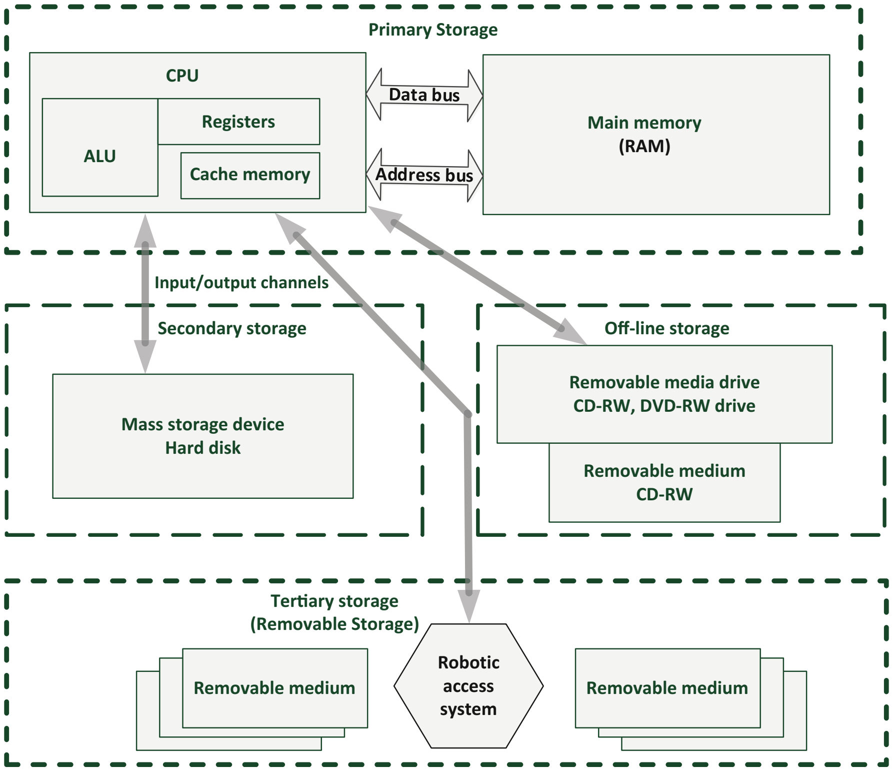

Fig. 2.2 Computer storage hierarchy according to their distance from the CPU [2]

RAM is not a secondary, but primary memory of the computer, and thus volatile. Although it is volatile storage space, it can access data hundreds of time faster than secondary memory like hard drive, which is why active programs are loaded into RAM in order to be processed seamlessly. All data in RAM are, in a sense, volatile because they degrade over time. Some are more volatile than others. For example, memory content is lost within nanoseconds as opposed to running processes which take few seconds. As a result, secondary storage is used to store programs and data when a computer is off. There are a variety of secondary storage devices, where hard disk is the most common one. Not all data from the hard disk are loaded into the system at one time. In fact, a computer typically has hard disk holding more space than the primary to compensate the amount of memory needed to keep the unused programs. For example, a computer bought from a retailer may come with a 2 TB hard drive, and only has a 16 GB of RAM.

As more and more types of removable media like backup tape drives, optical discs and flash memory drive have been introduced onto the market, there was an opportunity to create storage spaces of archiving data which is not accessed frequently. It is called tertiary storage which provides a third level of storage. It is primarily used for archiving rarely accessed information without human operators since it is much slower than secondary storage. The data is often copied to secondary storage before use. Typically, computer will first consult a catalog database to determine which tape or disc contains the information if it needs to read information from the tertiary storage. In order to fetch the medium, a robotic arm is usually involved to return the medium to its place in the library.

Unlike the other storages, Off-line storage cannot be accessed without human interaction. It is usually recorded in a secondary or tertiary storage device. It must be inserted or removed by a human operator. Since it can be easily physically transported, Off-line storage can be used to transfer information. Additionally, in the case of computer-based attack, Off-line storage increases information security because it is physically inaccessible for a computer. In modern personal computers (PCs), most secondary and tertiary storage media are also used for off-line storage.

As capacity of hard disk and other storage devices increase dramatically in the past decade, it is tedious to use byte as measurement unit to represent computer memory capacity. For example, it is not unusual to see computer's or devices' disk size reaching up to 1 terabits. As a result, different digital units are used, such as kilo- (thousand bytes), mega- (million bytes), and giga- (billion bytes), to distinguish the size of a computer memory capacity [2]. Their relationships are shown in Fig. 2.3. These can be simply denoted through abbreviation such as KB for Kilobytes (not to be confused with low-case kb for Kilobits).

There are two conventions of kilo (KB) used in computer systems: 1000 in decimal systems (it is equivalent to 1000 bytes.) and 1024 in binary systems (it is equivalent to 1024 bytes.). Although these two different conventions are both referring to the same "KB", their hard drive sizes are different. This rule applies to all units of measurements (or prefix) for digital information.

Due to various hard disk manufactures opting between different conventions, confusion may arise with which convention it's referring. This is especially true in the case of a Seagate customer who was misled by a product label that advised the hard disk sizes to be larger than originally claimed; an approximately 7.4% gap in memory. The company reimbursed the customer after being sued [3]. As an effort to reduce confusion, IEEE proposes a guideline that helps distinguish between a

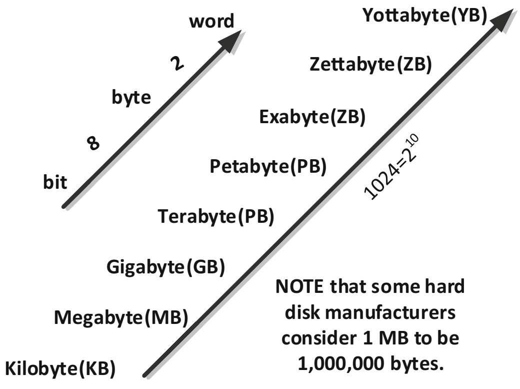

Fig. 2.3 Measurement units for digital information

decimal K (1000) and a binary K (1024). For instances, the conversion of low-case "k" refers to decimal kilo and upper-case "K" refers to binary kilo. However, not everyone strictly follows these conventions. Regardless, we use 1024 convention in this book.

2.2 Data Representation

Data is represented by *encoding*, which is the process of converting message or data into code. From code to local representation is decoding. There are many common ways that data is encoded in 1's and 0's. This is called data representation. In this book, we mainly focus on the following data representation:

- Unsigned integers (e.g. non-negative integers)
- Signed integers (e.g. negative, positive integers, and zero)
- Floating point numbers (e.g. approximations of real numbers)
- Characters (e.g. ASCII, Unicode)

There are many data types and complex data that are built on basic ones. You will be able to construct and deconstruct data by understanding some. Byte ordering is important in digital investigation, as well as byte alignment, which will be discussed in this section. But before we proceed into the lesson, we need to introduce how integers are encoded because it is the most common type of data, either unsigned integer or signed integer.

A typical memory cell (or one memory location) stores one byte of data, known as "byte-addressable" memory. It means that each address refers to one byte of memory. So, if we are to store a word into memory, we will have to split the 32 bits quantity into 4 bytes. With 8 switches of 1s and 0s, or 2^8, there would be 256 possibility for one byte. Each of these bytes corresponds to an integer between 0 and 255. Numbers we normally are familiar with like 5 and 8 are known as decimal notation. Each digit in a decimal formation corresponds to base 10. For example, 178 decimal digits can be written as $1 \times 10^2 + 7 \times 10^1 + 8 \times 10^0$. Similarly, a switch in byte represents a binary number where digits are represent as 1 and 0s instead of the 0–9 digits. To denote a binary number, we use "0b" or state that the digit is in base 2. For example, the data representation of decimal number 178 in binary is 0b10110010. We derive this binary number by converting the decimal to base 2. The simplest way is using a conversion table that listing the power of 2s from right to left starting at $2^0 = 1$, and incrementally, the exponent until $2^7 = 128$. Using the process of elimination, we figure out the greatest power that will fit into 178 is 128. Whenever a digit fits, it is a 1 bit. When it doesn't fit, it is a 0 bit. In this case, we note down 1 bit and subtract 178 from 128 to get 50 as our new digit to compare. We shift to the right and repeat the processes until there are no more base 2 digits to compare. To verify that we correctly convert decimal to binary, we can sum all base 2 numbers that are tick as 1 bits. For instance, $128 + 32 + 16 + 2 = 178$, shown in Table 2.1. Thus, we have correctly converted decimal to binary.

Since using binary seems long and cumbersome to write bytes to a computer, we use a more effective representation. By breaking the bytes into 2, we have 4 bits (also known as nybble). A nybble can take 16 configurations. These configurations can be represented by a single character called hexadecimal. A hexadecimal digit uses decimal digit to represent 0–9 and A to F to represent 10–15. Hexadecimal allows us to represent a byte as 2 digits instead of 8. To denote a hexadecimal number, we use "0x" or state that the digit is in base 16. For example, the data representation of decimal number 178 in hexadecimal is 0xB2, as shown in Table 2.2. We derive this hexadecimal number by dividing the 8 bits calculated from our binary conversion into two halves, referred to Table 2.3.

Generally speaking, a number can be represented in any base b, also known as radix. The format is shown as follows:

$$(d_{k-1}d_{k-2}d_{k-3}\ldots\ldots d_2d_1d_0)_b,$$

where the ds are digits, i.e., symbols for the integers between 0 and $b - 1$.This notation means a nonnegative integer written to the base b and is equal to

$$d_{k-1}b^{k-1} + d_{k-2}b^{k-2} + d_{k-3}b^{k-3} + \ldots\ldots + d_2b^2 + d_1b^1 + d_0.$$

Table 2.1 Binary of 178_{10}

Base 2	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Binary value	1	0	1	1	0	0	1	0

Table 2.2 Hexadecimal of 178_{10}

Binary	**1011**	**0010**
Hexadecimal value	0xB	0x2

Table 2.3 Conversion table

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Notably, sometime the parentheses are omitted, donated by $d_{k-1}d_{k-2}d_{k-3}\cdot\ldots\ldots d_2d_1d_{0b}$. For example, in the number $(1234567)_8$, 1234567 are digits and 8 is base.

Character popular encoding scheme like ASCII (American Standard Code for Information) maps numbers 0–9, letters a–z or A–Z, basic punctuation symbols, and blank spaces to binary integers. Let's use 'A' as our example, which is valued at 65 when encoded to decimal (base 10). If converted into hexadecimal (base 16), its value is 0x41.

ASCII is often denoted as 7-bit, but it's not unusually to see 8-bit, which extends the original character-mapping by adding additional characters on top of the first 128 characters made from 7-bits.

ASCII is nice and simple if you use American English, but it is quite limited for the rest of the world because their native symbols like Greek letters and Chinese characters cannot be represented. Unicode helps solve this problem by using more than 1 byte to store the numerical version of a symbol. Unicode offers a better scheme as it has a wider array of characters than ASCII by using multiple-bytes. It can be encoded in 8-, 16- or 32-bit binary formats, called UTF-8, UTF-16 and UTF-32. But how these multiple-byte quantities are arranged in memory? We will talk about this topic in the next section.

2.3 Memory Alignment and Byte Ordering

One important aspect we need to cover is Memory Alignment (or sometime referred to as Data Alignment). It dictates how data is arranged and accessed in computer memory. Data is distributed in 2-, 4-, 8-, 16-, or 32-byte chunks at a time, where the larger a byte that the CPU distributes means faster a computer can access data from memory. For example, a 32-bit processor requires a 4-byte integer to reside at a memory address that is evenly divisible by 4. This requirement is called "memory alignment". Padding unused byte between the end of the last chunk and next chunk correctly aligns the data structure in order to optimize the transfer of data more efficiently; Less cache miss and less bus transactions. In other words, accessing a misaligned structure will yield a lower overall performance. Let's take a memory that uses byte address as our example. The computer would break up the 32-bits into 4-bytes chunk and arrange them in consecutive orders. This means the next 3 bytes are stored at offset of 0x1001, 0x1002, and 0x1003 if the first byte is stored at 0x1000. Since the data is chunked into 4-bytes at a time, memory address cannot start with a memory address of 0x1001, 0x1003, or 0x1005 as they are not divisible by 4. Note that storing data in sequences may not apply to all cases because there are two ways to distribute and store multiple-byte in memory. This is called Endianness, which will be covered in this section. But first we must understand why a modern processor is restricted to access memory at granularity.

Misalignment causes hardware complications, since the memory is typically aligned on a word boundary. A misaligned memory access will introduce multiple aligned memory references [7]. The following diagram illustrates two situations, and in both cases the CPU accesses 4-byte chuck of data. Also, memory is accessed in 4-bytes. In summary, data is arranged (or 4 bytes aligned in our example below) and accessed in computer memory.

The situation on the left in Fig. 2.4 shows a case where data is aligned and situation located on the right shows a case where the data is misaligned. For the second situation, CPU has to perform extra work to access the data: Load 2 chucks of data, shift out unwanted bytes then combine them together, as shown in Fig. 2.5. Thus, the CPU data access performance suffers and CPU cycle is wasted for misaligned data in order to correctly retrieve the data from memory.

Generally, an access to an object of size s bytes at byte address A is aligned if A mod $s = 0$. Let's use the following as our example:

In Table 2.5, line 3 occupies 4 bytes of memory, line 4 occupies 3 bytes of memory, and line 5 occupies 4 bytes of memory. These numbers of bytes are not random but derived by type of data, which can be referred to the list in Table 2.4. It can thus be concluded that the struct object Student occupies 11 bytes (4 + 3 + 4) in memory. However, 11 isn't divisible of 4 (assuming that this case uses the common 32-bit x86 processor.). Therefore, the compiler will add an unused padding byte, making the struct Student allocate 12 bytes instead of 11 bytes of memory. This is why the size that a struct object occupies is often larger than the total storage required for the structure (Table 2.5).

PLEASE! NOTE You can use the expression "sizeof(struct Student)" to see for yourself. It gets the actual size of its occupied memory space.

System with Intel processors can still perform with a misaligned data structure; however, in some Unix systems it results in bus errors. Most compilers compensate the problem by automatically aligning data variables according to their types and the

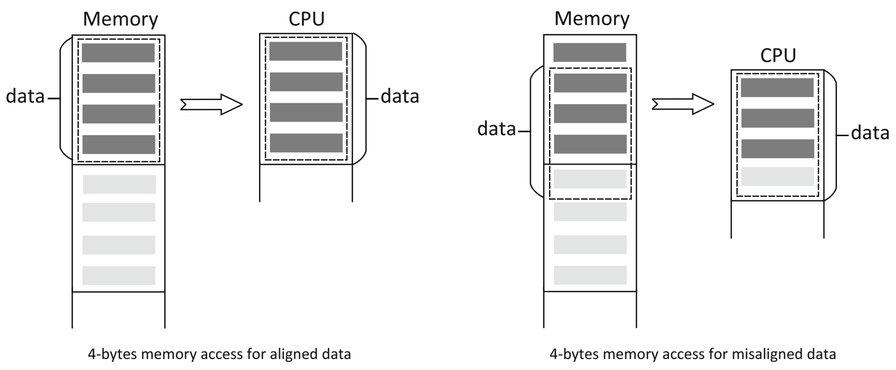
Memory
CPU
data
data
Memory
CPU
data
data
4-bytes memory access for aligned data
4-bytes memory access for misaligned data

Fig. 2.4 Memory mapping from memory to CPU cache [4]

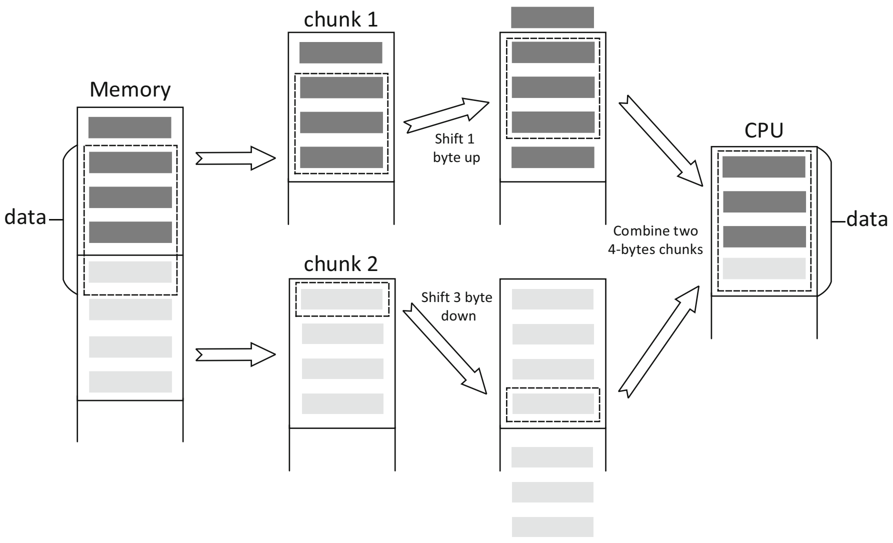

Fig. 2.5 Misaligned data slows down data access performance [4]

Table 2.4 Data alignment for each type

Data type	Alignment (bytes)
Char	1
Short	2
INT	4
Float	4
Double	4 or 8

Table 2.5 Example C

1	struct Student
2	{
3	int id;
4	char province[3]; //ON, BC etc. + terminating null
5	int age;
6	};

particular processor being used as such with the struct Student example explained. This also applies to union or classes objects.

In systems, byte of multiple-byte data element can be arranged depending on its order in which byte addressable memory is stored. This is known as Endianness (or Byte ordering). There are two popular types of endian:

- little-endian
- big-endian

The name endianness derived from Jonathan Swift's book "Gulliver's Travels". In the book, he talks about how a vigorous war starts over a silly debate that composes of people who prefer cracking boil eggs from the little end ("little-endian") and people who prefer creaking from the bigger end ("big-endian").

It is such a trivial thing, yet people argue about the ordering scheme in computer architecture too. Traditionally, things are done in big-endian, but then Microsoft came and decided to do little-endian. All the architecture architectures for Windows x86, x64, x32 operation systems are done in little-endian. However, both bit-ordering schemes are very much in use. For instances, processors like IBM 360/370, Motorola, Suns, 68 k, MIPS, Sparc, and HP PA use big-endian. These are general mainframe and big computers. Little-endian processes are Intel 80 × 86, DEC Vax, DEC Alpha, and SuperH. However, there are architectures like MIPS and Intel 64 IA-64 that operate in either big-endian or little-endian [5].

Either way, it is important to know little-endian and big-endian because one incorrect byte can throw off the whole machine. This is especially true when sending data over the network between two different machines that use opposite endianness because data would be read in the wrong order.

Within memory there is a high address and low address. Little endian stores with the least significant byte first (low address) in smallest address; read from right to left. Big endian stores the most significant byte first (high address) in the smallest address; read from left to right.

To fully understand these two different byte-ordering concepts, we will use 4 bytes number "90, AB, 12, CD" as our example, where each byte requires 2 hex digits, referred to Table 2.6.

Obviously, if we read the little endian version from low to higher memory addresses, we obtain CD 12 AB 90. We have to flip over to get the actual value of 0x90AB12CD. It means that the lower-order byte or least significant byte of a number is stored at the lowest address, and the high-order byte or most significant byte at the highest address in memory.

Similarly, the solution to the data sent over the network problem is to use a network byte order to rearrange the bytes stored at consecutive memory location when it detects the byte order scheme of that machine. A similar problem also exists

Table 2.6 Little-Endian vs. Big-Endian

Address	Value
Little-Endian	
1000	CD
1001	12
1002	AB
1003	90
Big-Endian	
1000	90
1001	AB
1002	12
1003	CD

when data exchange between computers over network. This is where XDR (External Data Representation) comes into play. XDR is a standard data serialization format, which allows data to be transferred between different kinds of computer systems [6]. It is independent of the transport layer. Converting from the local representation to XDR is called encoding. Converting from XDR to the local representation is called decoding. It uses a base of 4 bytes and order by big-endian. Variables would be padded by a divisible of four bytes.

Review Questions

1. Convert the following decimal numbers to binary numbers
   (a) 102
   (b) 18
   (c) 7
2. Solve for x in the following equations.
   (a) $x_{10} = 1001010_2$
   (b) $FCB8_{16} = x_2$
3. Suppose a computer with Intel processor has memory locations from 0x0000 to 0x0003, each storing 1 byte. What is the actual value stored there? (in decimal)

Address	Hex contents
0x0000	10
0x0001	23
0x0002	01
0x0003	A1

4. What is Byte-addressable memory?
5. How many bits are there in a nybble? How many bits are there in a byte?

2.4 Practice Exercise

The objective of this exercise is to give you a better understanding of how a computer stores and processes data.

2.4.1 Setting Up the Exercise Environment

For this exercise, assume that you have a physical or virtual Linux system with a C compiler installed, for example Linux GNU GCC.

2.4.2 *Exercises*

Consider the following C program

```
#include <stdio.h>
struct Student
{ int id;
char province[3]; //ON, BC etc. + terminating null
int age;
};
int main( ){
struct Student student1;
// Assign values to structure variables
student1.id = 100364168;
strncpy(student1.province, "ON\0", sizeof(student1.province));
student1.age= 18;
printf("The size of struct member id is %d bytes\n", sizeof(student1.id));
printf("The size of struct member province is %d bytes\n", sizeof(student1.
province));
printf("The size of struct member age is %d bytes\n", sizeof(student1.age));
printf("The size of struct Student is %d bytes\n", sizeof(struct Student));
return 0;
}
```

Part A: Data Alignment
Answer the following questions based on the above C code, by filling in all of the blanks where indicated based on the output of the program (Table 2.7).

Note that the size of a variable or data type is evaluated using sizeof operator in C Programming.

Table 2.7 Storage sizes of variables or data types in the above C code

Variable or data type		Size in bytes
student1.id	**100**	______________________________
student1.province	**101**	______________________________
student1.age	**102**	______________________________
Add lines **100**, **101**, and **102**	**103**	______________________________
Struct student	**104**	______________________________

Q1. Which is bigger? ____________

1. The size of struct Student (**Line 104**).
2. Subtotal of the sizes of all the members of struct Student (**Line 103**).

Part B: Get the Representation of Data Using GDB

In the following exercises we will look into how computers represent data based on the above code. Specially, you will use GDB commands to look into the memory to get the representations of all the members of struct Student. For example, if we declare an int variable,

```
int a = 16;
```

Then we can use the x command in GDB

```
(gdb) x/4bt &a
0xbffff56c:   00010000   00000000   00000000   00000000
```

where the first item is memory address where an integer value (in 4 bytes) is stored, the second item is the content stored in the memory. If you want to learn more about GDB & how to use it, you can refer to Appendix A, “How to use GDB to debug C programs”, as a reference.

Answer the following questions after debugging the above C program, by filling in all of the blanks with the output of GDB commands (Table 2.8):

Table 2.8 Data presentation of variables in the above C code

Variables	Representation of values (in hexadecimal) (after a value is assigned to the variable)
student1.id	____
student1.province	____
student1.age	____
Student1	____

Part C: Examining Endianness

Consider the following C program

```
#include <stdio.h>
#include <stdlib.h>
int main(){
unsigned char digits[4];
digits[0] = 0x12;
digits[1] = 0x34;
digits[2] = 0x56;
```

(continued)

```
    digits[3] = 0x78;
    int * ptr = (int * )digits;
    return 0;
    }
```

You can use the following GDB command to print the content at the memory address specified by a pointer, assuming p is an integer pointer.

```
(gdb) x/x p
0xbffff570:   0x78563412
```

where the first item is memory address that the integer pointer p contains, the second item is the content stored in the memory specified by the pointer.

Also, the following GDB command can be used to print the content of a specific array item. We'll be discussing an array of 4 unsigned chars as an example for simplicity.

```
(gdb) x/1bx &digits[0]
0xbffff570:   0x12
```

where the first item is memory address where the 1st array item (in 1 byte) is stored, the second item is the content stored in the 1st array item.

Answer the following questions based on the above C code, by filling in all of the blanks where indicated with the most appropriate response (Table 2.9):

Table 2.9 Memory adresses and stored values of char array elements in the above C code

Item of array *digits*	First item	Second item	Third item	Fourth item
Memory address	____________	____________	____________	____________
Stored value	____________	____________	____________	____________

Q2. What is the value in hexadecimal format at the memory address specified by the integer pointer ptr? ____________

Q3. According to your debug output, which endianness (big or little endian) is used in your system? Briefly explain your rationale for your conclusion. If necessary, give a diagram to help in your explanation.

Appendix A: How to Use GDB to Debug C Programs

In order to use gdb to debug a C program, you should compile your C program with -g option. It allows the compiler to collect the debugging information. For example,

```
gcc –g –o test test.c
```

where gcc is the compiler, test.c is the C program and test is the executable file.

Suppose we need to debug the executable file, the followings are basic steps for debugging a c program using gdb debugger:

Step 1: start gdb

```
gdb ./test
```

Step 2: Set up a break point inside C program

Syntax: break <line_number>

Note that since now on, you execute the commands in the gdb command line, not in the bash command line.

Step 3: Execute the C program in gdb debugger

```
run [args]
```

where args is the command line arguments you pass to the program.

Afterwards, you can use various gdb commands to examine executing Code. Example options of examining executing Code include:

- p or print: Print the content of variables or parameters.
- x or examine: Examine memory contents in different forms, including binary and hexadecimal forms. It uses the following syntax:

  ```
  x/[NUM][SIZE][FORMAT] [Address]
  ```

  where NUM is the number of objects to display, SIZE is the size of each object (b = byte, h = half-word, w = word, g = giant word (eight bytes)), FORMAT indicates how to display each object (d = decimal, x = hex, o = octal, t = binary), and [Address] is the memory address. For example,

  the following x command will display a program's variable a's actual value in hex form when given the argument &a. 4 is the repeat count or the number of units whose size is specified by argument *b*, which stands for byte as the unit size. ‘x’ means that you want to display or output the value in hexadecimal form, which is the default display format for the x command.

  ```
  (gdb) x/4bx &a
  0xbffff56c: 0x10 0x00 0x00 0x00
  ```

Step 4: Continue, stepping over and in after a breakpoint

There are three kinds of gdp operations after a breakpoint:

- c or continue: Execution will continue until the next breakpoint in your code.
- n or next: Executing the next line of code after the current breakpoint.
- s or step: The s command is very similar to the n command, except for that the s command steps into a function and executes it line by line, whereas the n command simply treats a function call as one line of code.

Step 5: Quit from the gdb debugger
Syntax: quit

References

1. What is the difference between memory and hard disk space? http://pc.net/helpcenter/answers/memory_and_hard_disk_space
2. https://en.wikipedia.org/wiki/Computer_data_storage
3. Seagate customers eligible for manufacturer refunds, free software Seagate is offering a settlement agreement to Sara Cho, the woman who sued the . . . http://arstechnica.com/gadgets/2007/10/seagate-customers-eligible-for-manufacturer-refunds-free-software/
4. Data Alignment. http://www.songho.ca/misc/alignment/dataalign.html
5. kilobyte. http://www.webopedia.com/TERM/K/kilobyte.html
6. External Data Representation (XDR) http://en.wikipedia.org/wiki/External_Data_Representation
7. Aligning Addresses http://www.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/addressAlign.html