COMPSCI 210 S1T Computer Systems

Representation of Structured Data

Agenda & Reading

- Agenda:
  - Representation of Structured Data
    - Bits, Bytes, words, longwords and quadwords
    - Strings
    - Arrays
    - Jagged Arrays
    - Records
    - Pointers
    - Class Instances
- Example:
  - 06\ConsoleApplication1 (C#)

Bits, bytes, words, longwords & quadwords

- Bit
  - Binary digit: 0 or 1
- Byte
  - Consists of 8 bits
  - Represent 2^8 = 256 different values
- Word
  - A pair of adjacent bytes grouped together
  - 16 bits
- Longword
  - 4 adjacent bytes grouped together
  - 32 bits
- Quadword
  - 8 adjacent bytes grouped together
  - 64 bits
- Note: People would tend to write the more significant digits of a number to the left of the less significant digits:

Big Endian & Little Endian

- On modern machine
  - Refer to Individual bytes by an address
  - Values that take up several bytes (e.g. int) can be referred to by the address of the low byte.
- There are two different formats to store a number
  - Big Endian
    - Store the most significant byte first
  - Little Endian
    - Store the least significant byte first
- Example:
  - Store the longword 0x12345678 at address 0x10000000

Big Endian

0x10000000	12
0x10000001	34
0x10000002	56
0x10000003	78

Little Endian

0x10000000	78
0x10000001	56
0x10000002	34
0x10000003	12

Structure - Strings

Note: All examples below are written in C# using Visual Studio!

- Strings
  - Stored as a sequence of bytes, with one character per byte (, or two)
  - The end of the strings is indicated by a null (Zero) byte.
  - In Big endian format
  - Note: data composed of more than a byte has to be aligned according to its size

String format (.NET): 2 bytes Unicode characters

```
S = "hello";
```

```
S = "helloo";
```

h
e
l
l
o
NULL

Memory Address

System.String format

```
0x013C1D90  e0 a3 0f 79
0x013C1D94  06 00 00 00
0x013C1D98  05 00 00 00
0x013C1D9C  68 00 65 00  h.e.
0x013C1DA0  6c 00 6c 00  l.l.
0x013C1DA4  6f 00 00 00  o...
0x013C1DA8  00 00 00 00
```

Size: 5 characters

End byte

Content

End

```
0x013C1D90  e0 a3 0f 79
0x013C1D94  07 00 00 00
0x013C1D98  06 00 00 00
0x013C1D9C  68 00 65 00  h.e.
0x013C1DA0  6c 00 6c 00  l.l.
0x013C1DA4  6f 00 6f 00  o.o.
0x013C1DA8  00 00 00 00
0x01        00 00 00 00
```

End byte

Padded with extra bytes - Aligned

Structure - Arrays

- Arrays
  - Compound objects composed of elements of the same type
  - Note: size and type are stored in memory (Visual Studio)
  - Example: byte[]

```
byte[] arr = new byte[3] { 0xaa, 0xab, 0xac };
```

aa
ab
ac

System.Byte format

```
0x013C1D90  18 44 12 79
0x013C1D94  03 00 00 00
0x013C1D98  aa ab ac 00
0x013C1D9C  00 00 00 00
```

Size: 3 bytes

Padded with an extra byte - Aligned

End

```
byte[] arr = new byte[4] { 0xaa, 0xab, 0xac, 0xad };
```

aa
ab
ac
ad

```
0x013C1D90  18 44 12 79
0x013C1D94  04 00 00 00
0x013C1D98  aa ab ac ad
0x013C1D9C  00 00 00 00
```

Size: 4 bytes

End

Structure - Arrays

- Arrays
  - Compound objects composed of elements of the same type
  - Note: size and type are stored in memory (Visual Studio)
  - Example: byte[]

```
byte[] arr = new byte[3] { 0xaa, 0xab, 0xac };
```

aa
ab
ac

System.Byte format

```
0x013C1D90  18 44 12 79
0x013C1D94  03 00 00 00
0x013C1D98  aa ab ac 00
0x013C1D9C  00 00 00 00
```

Size: 3 bytes

Padded with an extra byte - Aligned

End

```
byte[] arr = new byte[4] { 0xaa, 0xab, 0xac, 0xad };
```

aa
ab
ac
ad

```
0x013C1D90  18 44 12 79
0x013C1D94  04 00 00 00
0x013C1D98  aa ab ac ad
0x013C1D9C  00 00 00 00
```

Size: 4 bytes

End

Jagged Arrays

aa	ab	
ac	ad	ae

- Jagged Arrays
  - An array of which each element is itself an array is called an array of arrays. The elements of a jagged array can be of different sizes.

```
byte[][] b2 = new byte[][] { new byte[] { 0xaa, 0xab },
   new byte[] { 0xac, 0xad, 0xae } };
```

In memory

aa
ab
ac
ad
ae

System.Byte[][]

```
0x013C1D90  c8 ad 19 79
0x013C1D94  02 00 00 00
0x013C1D98  5a 98 15 79
0x013C1D9C  a8 1d 3c 01
0x013C1DA0  b8 1d 3c 01
0x013C1DA4  00 00 00 00
0x013C1DA8  18 44 12 79
0x013C1DAC  02 00 00 00
0x013C1DB0  aa ab 00 00
0x013C1DB4  00 00 00 00
0x013C1DB8  18 44 12 79
0x013C1DBC  03 00 00 00
0x013C1DC0  ac ad ae 00
0x013C1DC4  00 00 00 00
```

Size: 2 elements

System.Byte[]

Padded with an extra byte - Aligned

End

Jagged Arrays

- int[][]

8	7	
6	5	4

```
int[][] i2 = new int[][] { new int[] { 8, 7 },
    new int[] { 6, 5, 4 } };
```

```
0x013C1D90  54 c0 12 79
0x013C1D94  02 00 00 00
0x013C1D98  2a 98 15 79
0x013C1D9C  a8 1d 3c 01
0x013C1DA0  bc 1d 3c 01
0x013C1DA4  00 00 00 00
0x013C1DA8  f0 40 12 79
0x013C1DAC  02 00 00 00
0x013C1DB0  08 00 00 00
0x013C1DB4  07 00 00 00
0x013C1DB8  00 00 00 00
0x013C1DBC  f0 40 12 79
0x013C1DC0  03 00 00 00
0x013C1DC4  06 00 00 00
0x013C1DC8  05 00 00 00
0x013C1DCC  04 00 00 00
0x013C1DD0  00 00 00 00
```

System.Int32[][]

Size: 2 elements

In memory

8
7
6
5
4

Records

- A record is compound object composed of fields of possibly different types (are specified at compile-time)
  - Similar to struct (in C) / class with only instance fields (in Java)
  - Fields may be of different sizes. They are stored at successive addresses.

```
class MyPoint {
  public int x;
  public int y;
  public MyPoint(int x, int y){
      this.x = x;
      this.y = y;
  }
}
```

```
class BinaryNode {
  BinaryNode left;
  BinaryNode right;
  char name;
  public BinaryNode(char n) {
    name = n;
  }
}
```

```
MyPoint pt1 = new MyPoint(9,8);
```

9
8

```
BinaryNode a = new BinaryNode('a');
```

```
0x013C1D90  a0 30 a2 00
0x013C1D94  09 00 00 00
0x013C1D98  08 00 00 00
0x013C1D9C  00 00 00 00
```

MyPoint Type

x

y

0x013C1D90	left
0x013C1D94	right
0x013C1D98	name

Example

- BinaryNode

```
BinaryNode a = new BinaryNode('a');
BinaryNode b = new BinaryNode('b');
BinaryNode n = new BinaryNode('*', a, b);
```

BinaryNode Type

```
0x013C1D90  c4 30 a2 00  ....
0x013C1D94  00 00 00 00  ....
0x013C1D98  00 00 00 00  ....
0x013C1D9C  61 00 00 00  a...
0x013C1DA0  00 00 00 00  ....
0x013C1DA4  c4 30 a2 00  ....
0x013C1DA8  00 00 00 00  ....
0x013C1DAC  00 00 00 00  ....
0x013C1DB0  62 00 00 00  b...
0x013C1DB4  00 00 00 00  ....
0x013C1DB8  c4 30 a2 00  ....
0x013C1DBC  90 1d 3c 01  ..<.
0x013C1DC0  a4 1d 3c 01  ..<.
0x013C1DC4  2a 00 00 00  *...
0x013C1DC8  00 00 00 00  ....
```

0x013C1D90		
	left	null
	right	null
	name	a
0x013C1DA4		
	left	null
	right	null
	name	b
0x013C1DB8		
	left	
	right	
	name	*

Pointers

- A pointer variable holds values that are addresses of objects in memory.
- Storage allocated a pointer is the size necessary to hold a memory address.
- Every pointer is associated with a data type.
- The data type specifies how the contents and size of the memory address should be interpreted
- Example: int

```
int i = 9;
int* ipt = &i;
```

0x0012f440	9

Address

```
Console.WriteLine((int)ipt);
Console.WriteLine(*ipt);
```

Print Address

Print value

- Example: int[]

```
int[] iArray = { 4, 5, 6 };
```

Address

0x013C1D98	4
0x013C1D9C	5
0x013C1DA0	6

Manipulating Pointers

- Addition/Subtraction
  - You can add/subtract a value n of type int, uint, long, or ulong to a pointer, p. The result p+n is the pointer resulting from adding (n * the size of the pointer) to the address of p
  - Example: int[]

```
int[] iArray = { 4, 5, 6 };
...
```

```
for (int i = 0; i < 3; i++) {
  Console.WriteLine(*(p2 + i));
}
```

4
5
6

0x013C1D98	4
0x013C1D9C	5
0x013C1DA0	6

- Accessing elements
  - To access a member of a pointer type, use the member access expression that takes the form:
    - expression -> identifier

```
struct sPoint {
  public int x;
  public int y;
  ...
}
```

x	9
y	8

```
sPoint s1 = new sPoint(9, 8);
sPoint* sptr = &s1;
Console.WriteLine((int)sptr);
Console.WriteLine(sptr->x);
Console.WriteLine(sptr->y);
```

Print Address

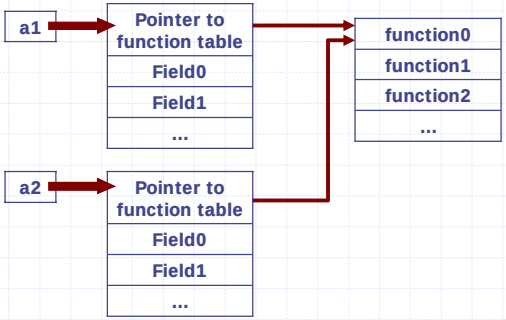
```
0x013C1D90  a0 30 a2 00
0x013C1D94  09 00 00 00
0x013C1D98  08 00 00 00
0x013C1D9C  00 00 00 00
```


Class Instances

- Function Table = a table of the instance functions of the Class
  - The function address is obtained from the function table.
  - The function table never changes. So objects of the same type can share the function table

```
public class A {
  public int field0;
  ...
  public void function0() {
  ...
}
```

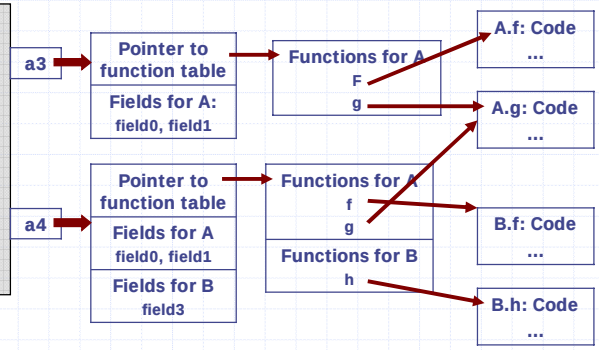
```
A a1 = new A();
A a2 = new A();
```



Extends

- When a class extends another class, the layout of the first part of the field and function table of the subclass is the same as that of the superclass

```
public class A {
  public int field0;
  public int field1;
  public void f() {...}
  public void g() {...}
}
public class B extends A {
  public int field3;
  ...
  public void f() {...}
  public void h() {...}
  ...
}
```




```
A a3 = new A();
B a4 = new B();
```