COMP110: Principles of Computing

9: Primitive and Object Data

Learning outcomes

- **Explain** the representation of common "plain old data" types in memory
- **Distinguish** pass-by-reference and pass-by-value
- **Distinguish** allocation of memory on the stack and on the heap

Quiz C

- Dot and cross product
- Due **this time next week**

Maths quizzes — new rule

- Old quizzes are reopened!
- For the threshold criterion you must now
  - **Either** achieve a mark of 8/10 by the deadline
  - **Or** achieve a mark of 10/10 after the deadline

FALMOUTH UNIVERSITY

Pass by reference

References

References

- Our picture of a variable: a labelled box containing a value

References

- Our picture of a variable: a labelled box containing a value
- For “plain old data” (e.g. numbers), this is accurate

References

- Our picture of a variable: a labelled box containing a value
- For "plain old data" (e.g. numbers), this is accurate
- For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)

References

- Our picture of a variable: a labelled box containing a value
- For "plain old data" (e.g. numbers), this is accurate
- For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)
- It is possible (indeed common) to have **multiple references** to the same underlying object

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

FALMOUTH UNIVERSITY

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value
x	
y	
z	

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

<table>
<tr><th>Variable</th><th>Value</th></tr>
<tr><td>x</td><td><table><tr><td>a</td><td>30</td></tr><tr><td>b</td><td>40</td></tr></table></td></tr>
<tr><td>y</td><td></td></tr>
<tr><td>z</td><td></td></tr>
</table>

The wrong picture

```python
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z		

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z	a	50
	b	60

The right picture

```python
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

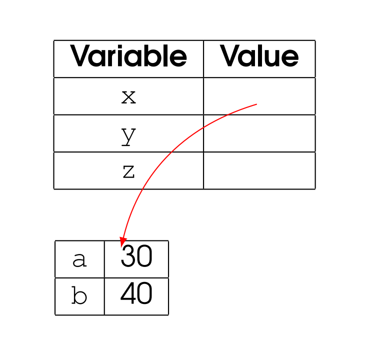
The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value
x	
y	
z	

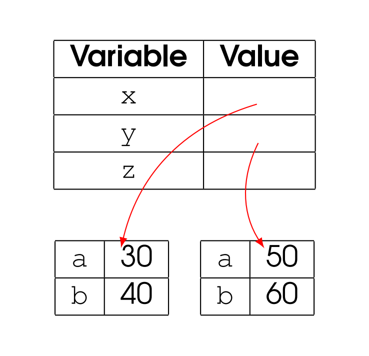
The right picture

```python
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



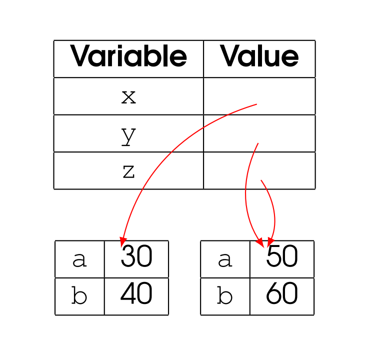
The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



The right picture

```python
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



Values and references

Socrative room code: `FALCOMPED`

```
a = 10
b = a
a = 20
print "a:", a
print "b:", b
```

Values and references

Socrative room code: `FALCOMPED`

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = a
a.value = 20
print "a:", a.value
print "b:", b.value
```

Values and references

Socrative room code: `FALCOMPED`

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = X(10)
a.value = 20
print "a:", a.value
print "b:", b.value
```

Pass by value

Pass by value

In **function parameters**, “plain old data” is passed by **value**

Pass by value

In **function parameters**, “plain old data” is passed by **value**

```
def double(x):
    x *= 2

a = 7
double(a)
print a
```

Pass by value

In **function parameters**, "plain old data" is passed by **value**

```
def double(x):
    x *= 2

a = 7
double(a)
print a
```

`double` does not actually do anything, as `x` is just a local copy of whatever is passed in!

Pass by reference

Pass by reference

However, instances are passed by **reference**

```
class Box:
    def __init__(self, v):
        self.value = v

def double(x):
    x.value *= 2

a = Box(7)
double(a)
print a.value
```

Pass by reference

However, instances are passed by **reference**

```
class Box:
    def __init__(self, v):
        self.value = v

def double(x):
    x.value *= 2

a = Box(7)
double(a)
print a.value
```

`double` now has an effect, as `x` gets a reference to the `Box` instance

Lists are objects too

Lists are objects too

```
a = ["Hello"]
b = a
b.append("world")
print a  # ["Hello", "world"]
```

Lists are objects too

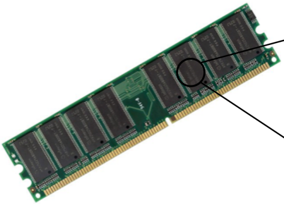
```
a = ["Hello"]
b = a
b.append("world")
print a   # ["Hello", "world"]
```

... which means you should be careful when passing lists into functions, because the function might actually change the list!

Data representation

Memory

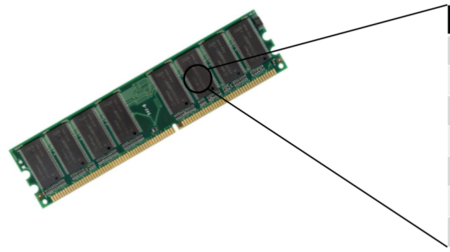
Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**

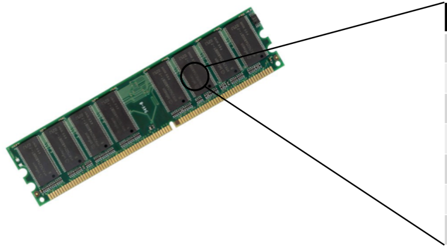
Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**
- Each box has a number, its **address**

Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**
- Each box has a number, its **address**
- Each box contains a **byte** (8 bits)

Data representation

Data representation

- All data is stored as **sequences of bytes**

Data representation

- All data is stored as **sequences of bytes**
  - Sequence of bits, in multiples of 8

Data representation

- All data is stored as **sequences of bytes**
  - Sequence of bits, in multiples of 8
  - Sequence of numbers between 0–255

Number bases

Number bases

- Decimal = base 10

Number bases

- Decimal = base 10
- Binary = base 2

Number bases

- Decimal = base 10
- Binary = base 2
- Octal = base 8

Number bases

- Decimal = base 10
- Binary = base 2
- Octal = base 8
- Hexadecimal = base 16

Number bases

- Decimal = base 10
- Binary = base 2
- Octal = base 8
- Hexadecimal = base 16
    - Uses `A`–`F` as extra "digits"

Number bases

- Decimal = base 10
- Binary = base 2
- Octal = base 8
- Hexadecimal = base 16
  - Uses `A`–`F` as extra “digits”
- We sometimes write the base as a **subscript**

Number bases

- Decimal = base 10
- Binary = base 2
- Octal = base 8
- Hexadecimal = base 16
  - Uses `A`–`F` as extra “digits”
- We sometimes write the base as a **subscript**
  - $123_{10} = 7B_{16} = 173_8 = 01111011_2$

Number bases in code

Number bases in code

- In most languages (including C, C++, C#, Python):

Number bases in code

- In most languages (including C, C++, C#, Python):
  - Decimal: `123`

Number bases in code

- In most languages (including C, C++, C#, Python):
  - Decimal: `123`
  - Binary: `0b1111011`

Number bases in code

- In most languages (including C, C++, C#, Python):
  - Decimal: `123`
  - Binary: `0b1111011`
  - Hexadecimal: `0x7B`

Number bases in code

- In most languages (including C, C++, C#, Python):
  - Decimal: `123`
  - Binary: `0b1111011`
  - Hexadecimal: `0x7B`
- In some languages (including C, C++, Python 2.x):

Number bases in code

- In most languages (including C, C++, C#, Python):
  - Decimal: `123`
  - Binary: `0b1111011`
  - Hexadecimal: `0x7B`
- In some languages (including C, C++, Python 2.x):
  - Octal: `0173`

Number bases in code

- In most languages (including C, C++, C#, Python):
  - Decimal: `123`
  - Binary: `0b1111011`
  - Hexadecimal: `0x7B`
- In some languages (including C, C++, Python 2.x):
  - Octal: `0173`
- So beware of leading zeroes!

```
>>> print 77
77
>>> print 077
63
```

Why is hexadecimal useful?

Why is hexadecimal useful?

- It is easy to convert between binary and hexadecimal

Why is hexadecimal useful?

- It is easy to convert between binary and hexadecimal
- One hex digit = 4 bits

Why is hexadecimal useful?

- It is easy to convert between binary and hexadecimal
- One hex digit = 4 bits

$$\underbrace{0001}_{1}\,\underbrace{1110}_{E}\,\underbrace{0010}_{2}\,\underbrace{0100}_{4}\,\underbrace{0000}_{0}$$

Why is hexadecimal useful?

- It is easy to convert between binary and hexadecimal
- One hex digit = 4 bits

$$\underbrace{0001}_{1}\,\underbrace{1110}_{E}\,\underbrace{0010}_{2}\,\underbrace{0100}_{4}\,\underbrace{0000}_{0}$$

So $1\,1110\,0010\,0100\,0000_2 = 1E240_{16}$

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Integers

Integers

- Represented in **binary**

Integers

- Represented in **binary**
- **Unsigned** integers:

Integers

- Represented in **binary**
- **Unsigned** integers:
  - n bits $\implies$ numbers from 0 to $2^n - 1$ inclusive

Integers

- Represented in **binary**
- **Unsigned** integers:
  - n bits $\implies$ numbers from 0 to $2^n - 1$ inclusive
- **Signed** integers:

Integers

- Represented in **binary**
- **Unsigned** integers:
  - n bits $\implies$ numbers from 0 to $2^n - 1$ inclusive
- **Signed** integers:
  - Store negative number $-a$ as $2^n - a$

Integers

- Represented in **binary**
- **Unsigned** integers:
  - n bits $\implies$ numbers from 0 to $2^n - 1$ inclusive
- **Signed** integers:
  - Store negative number $-a$ as $2^n - a$
  - **2's complement**

Integers

- Represented in **binary**
- **Unsigned** integers:
  - n bits $\implies$ numbers from 0 to $2^n - 1$ inclusive
- **Signed** integers:
  - Store negative number $-a$ as $2^n - a$
  - **2's complement**
  - n bits $\implies$ numbers from -2^{n-1} to $2^{n-1} - 1$ inclusive

2's complement uses modular arithmetic

Endian-ness

Endian-ness

- Intel x86/x64 architecture is **little endian**

Endian-ness

- Intel x86/x64 architecture is **little endian**
- Bytes are in “reverse” order (least significant first)

Endian-ness

- Intel x86/x64 architecture is **little endian**
- Bytes are in "reverse" order (least significant first)
- E.g. $123456_{10} = 1E240_{16}$, represented as `40 E2 01 00`

Fractional numbers

Fractional numbers

- Generally stored using **floating point**

Fractional numbers

- Generally stored using **floating point**
- More on this another time

Strings

Strings

- Stored as sequences of **characters** encoded as **integers**

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string
- **ASCII** encodes characters as 8-bit integers

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string
- **ASCII** encodes characters as 8-bit integers
  - 255 characters: Latin alphabet, numerals, punctuation

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string
- **ASCII** encodes characters as 8-bit integers
  - 255 characters: Latin alphabet, numerals, punctuation
- **UTF-32** encodes characters as 32-bit integers

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string
- **ASCII** encodes characters as 8-bit integers
  - 255 characters: Latin alphabet, numerals, punctuation
- **UTF-32** encodes characters as 32-bit integers
  - **Unicode** characters: ASCII + several other alphabets, Asian languages, symbols, emoji, ...

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string
- **ASCII** encodes characters as 8-bit integers
  - 255 characters: Latin alphabet, numerals, punctuation
- **UTF-32** encodes characters as 32-bit integers
  - **Unicode** characters: ASCII + several other alphabets, Asian languages, symbols, emoji, ...
- **UTF-8** encodes characters as 8, 16, 24 or 32-bit integers

Strings

- Stored as sequences of **characters** encoded as **integers**
- Often **null-terminated**
  - Zero character signifies the end of the string
- **ASCII** encodes characters as 8-bit integers
  - 255 characters: Latin alphabet, numerals, punctuation
- **UTF-32** encodes characters as 32-bit integers
  - **Unicode** characters: ASCII + several other alphabets, Asian languages, symbols, emoji, ...
- **UTF-8** encodes characters as 8, 16, 24 or 32-bit integers
  - More common Unicode characters are smaller $\Longrightarrow$ more efficient than UTF-32

Object data

Object data

- Fields are fixed size

Object data

- Fields are fixed size
- Stored consecutively in memory

Object data

- Fields are fixed size
- Stored consecutively in memory
- Variable-size data may be stored indirectly, using pointers