Programming in Python

GAVRILUT DRAGOS

COURSE 12

Itegrating with C/C++

Python has several ways of integrating with C/C++ bindings. It can also integrate with low-level OS libraries regardless of the language they were written on.

There are several mechanisms used for binding:

- **struc** → to pack data in a C/C++ structure (this also include alignment)
- **ctype** module → to work with C/C++ primitive data types and libraries
- There is also the possibility of integrating C/C++ with Python (either write a Python library in C/C++ or use Python to execute code directly from C/C++)

Details about these modules can be found on:

- Python 2: https://docs.python.org/2/library/struct.html#
- Python 3: https://docs.python.org/3/library/struct.html#
- Python 2: https://docs.python.org/2/library/ctypes.html
- Python 3: https://docs.python.org/3/library/ctypes.html

struct

struct module provides a way of converting a list of bytes into a C/C++ structure (this also include alignment and padding bytes).

Functions:

- **struct**.*pack* (format, v_1,v_2,…v_n) ➔ returns a list of bytes organized in a C/C++ structure according to thee provided format
- **struct**.*unpack* (format, buffer) ➔ returns a tuple of values obtained from a buffer that was unpacked according to a specific format
- **struct**.*calcsize* (format) ➔ returns the size of the byte buffer that will be obtained using a specific format

struct

format field contains the following abbreviations with the following meaning:

- First character provides information about the struct data size and alignment, as follows:

Character	Endian	Alignment	Size
@	Native (the one used on current machine)	Native (the one used on C/C++ compiler)	Native (the one used on C/C++ compiler)
=	Native (the one used on current machine)	-	Standard
<	Little endian	-	Standard
>	Big endian	-	Standard
!	Big endian (for network)	-	Standard

- The default character if none is provided is **@**

struct

format field contains the following abbreviations with the following meaning:

- The rest of the characters describe a type as follows:

Character	C Type
c	char
b	signed char
B	unsigned char
?	bool
h	short
H	unsigned short
i	int
I	unsigned int

Character	C Type
l	long
L	unsigned long
q	long long
Q	unsigned long long
h	short
f	float
d	double
x	padding byte

struct

format field contains the following abbreviations with the following meaning:

- The following characters are used to describe pointer specific data:

Character	C Type
N	size_t
s	char[<number of characters>]
p	Pascal string <size><list of characters>
P	void*

- “**P**” and “**N**” are only valid for native sizes
- **struct** module is usually required if one interprets in Python a data buffer (network buffer, file content, etc) that was written in a binary mode from a C/C++ module.

OS Architecture (memory alignment)

```
struct Test
{
        int x;
        int y;
        int z;
};
```

```
sizeof(Test) = 12
```

x	x	x	x	y	y	y	y	z	z	z	z																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@iii",1,2,3)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
12
01 00 00 00 02 00 00 00 03 00 00 00
```

struct

pack function in Python 2.x returns a string (not a list of bytes).

Python 2.x

```
import struct

data = struct.pack("@iii",1,2,3)

print(len(data))
s = ""
for i in data:
       s += "%02X " % ord(i)
print(s)
```

Output

```
12
01 00 00 00 02 00 00 00 03 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        char y;
        int  z;
};
```

```
sizeof(Test) = 8
```

x	y	?	?	z	z	z	z																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@cci",b'A', b'B',3)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
8
41 42 00 00 03 00 00 00
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@cci",b'A', b'B' 3)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

The **"c"** specification expect a 1-byte value as a parameter. It is important to precede any character with **b** prefix

Output

```
8
41 42 00 00 03 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        char y;
        char z;
        int  t;
};
```

```
sizeof(Test) = 8
```

x	y	z	?	t	t	t	t																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@ccci",b'A', b'B', b'C',3)

print(len(data))
s = ""
for i in data:
       s += "%02X " % i
print(s)
```

Output

```
8
41 42 43 00 03 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        char y;
        char z;
        short s;
        int  t;
};
```

```
sizeof(Test) = 12
```

x	y	z	?	s	s	?	?	t	t	t	t																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@ccchi",b'A', b'B', b'C',3, 4)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
12
41 42 43 00 03 00 00 00 04 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        short y;
        char z;
        short s;
        int  t;
};
```

```
sizeof(Test) = 12
```

x	?	y	y	z	?	s	s	t	t	t	t																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@chchi",b'A',1, b'B', 2, 3)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
12
41 00 01 00 42 00 02 00 03 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        short y;
        double z;
        char s;
        short t;
        int u;
};
```

```
sizeof(Test) = 24
```

x	?	y	y	?	?	?	?	z	z	z	z	z	z	z	z	s	?	t	t	u	u	u	u								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@chdchi",b’A’,1, 1.0, b’B’, 2, 3)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
24
41 00 01 00 00 00 00 00 00 00 00 00 00 00 F0 3F 42 00 02 00 03 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        double y;
        int z;
};
```

```
sizeof(Test) = 24
```

x	?	?	?	?	?	?	?	y	y	y	y	y	y	y	y	z	z	z	z	?	?	?	?								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@cdi",b’A’, 1.0, 3)

print(len(data))
s = ""
for i in data:
       s += "%02X " % i
print(s)
```

Output

```
20
41 00 00 00 00 00 00 00 00 00 00 00 00 00 F0 3F 03 00 00 00
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@cdi",b'A', 1.0, 3)

print(len(data))
s = ""
for i in data:
        s += "%02X "
print(s)
```

```
struct Test
{
        char x;
        double y;
        int z;
;
```

```
sizeof(Test) = 24
```

To align a structure to a specific type (int/ double/ etc) add the number 0 followed by the letter that required for formatting at the end of the format string !

Output

```
20
41 00 00 00 00 00 00 00 00 00 00 00 00 00 F0 3F 03 00 00 00
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@cdi0d",b'A', 1.0, 3)

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

To align a structure to a specific type (int/ double/ etc) add the number 0 followed by the letter that required for formatting at the end of the format string ! **If you don't specify "0d" format (if you use another number that 0) an error will occur.**

Output

```
24
41 00 00 00 00 00 00 00 00 00 00 00 00 00 F0 3F 03 00 00 00 00 00 00 00
```

OS Architecture (memory alignment)

```
struct Test
{
        char x;
        short y;
        int z;
        char t;
};
```

```
sizeof(Test) = 12
```

x	?	y	y	z	z	z	z	t	?	?	?																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@chic0i",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
12
41 00 01 00 02 00 00 00 42 00 00 00
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@chic0i",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

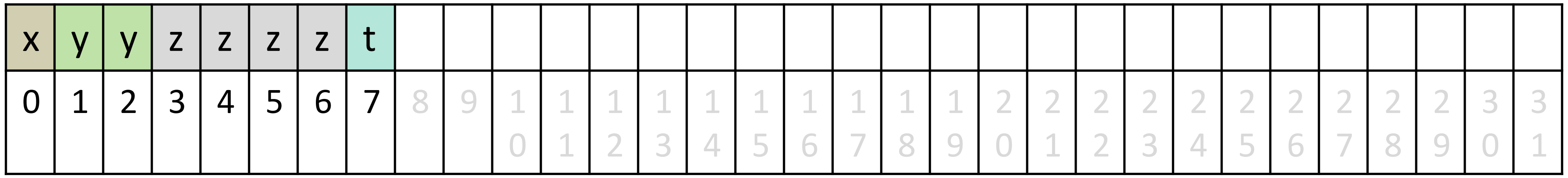
Output

```
12
41 00 01 00 02 00 00 00 42 00 00 00
```

OS Architecture (memory alignment)

```
#pragma pack(1)
struct Test
{
            char x;
            short y;
            int z;
            char t;
};
```

```
sizeof(Test) = 8
```



struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("=chic",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
8
41 01 00 02 00 00 00 42
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("=chic",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Use '=' character to disable alignments and padding for any type

Output

```
8
41 01 00 02 00 00 00 42
```

OS Architecture (memory alignment)

```
#pragma pack(2)
struct Test
{
		char x;
		short y;
		int z;
		char t;
};
```

```
sizeof(Test) = 10
```

x	?	y	y	z	z	z	z	t	?																						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("@chic0h",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
10
41 00 01 00 02 00 00 00 42 00
```

OS Architecture (memory alignment)

```
#pragma pack(1)
_declspec(align(16)) struct Test
{
		char x;
		short y;
		int z;
		char t;
};
```

```
sizeof(Test) = 16
```

x	y	y	z	z	z	z	t	?	?	?	?	?	?	?	?																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("=chic0d",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Output

```
8
41 01 00 02 00 00 00 42
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("=chic0d",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

Structure padding only works with **@** character at the beginning

Output

```
8
41 01 00 02 00 00 00 42
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("=chicxxxxxxxx",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

The solution in this case is to add extra padding with **x** character manually

Output

```
16
41 01 00 02 00 00 00 42 00 00 00 00 00 00 00 00
```

struct

The previous example would be packed in a Python list of bytes as follows:

Python 3.x

```
import struct

data = struct.pack("=chic8x",b'A', 1, 2, b'B')

print(len(data))
s = ""
for i in data:
        s += "%02X " % i
print(s)
```

The same can be achieved by adding a **number** in front of a character (specifies that that character should be counter for multiple times)

Output

```
16
41 01 00 02 00 00 00 42 00 00 00 00 00 00 00 00
```

struct

Pack and unpack can be used together to convert a set of values into a byte buffer. In this case, the structure is form out of 3 integers, a string of 10 characters and one float value in this order.

Python 3.x

```
import struct

data = struct.pack ("@3i10sf",1,2,3,b"Python",1.5)
print(len(data))
print (struct.unpack("@3i10sf",data))
```

Output

```
28
(1, 2, 3, b'Python\x00\x00\x00\x00', 1.5)
```

struct

Not specifying the number of characters in a string means only one character. In the previous example only the letter '**P**' will be added.

Python 3.x

```
import struct

data = struct.pack ("@3isf",1,2,3,b"Python",1.5)
print(len(data))
print (struct.unpack("@3isf",data))
```

The null terminated character is not added !!!

Output

```
20
(1, 2, 3, b'P', 1.5)
```

struct

Packing also support pascal style string (first characters represents the length)

Python 3.x

```
import struct

data = struct.pack("10p",b’Python’)

print(len(data))
s = ""
for i in data:
       s += "%02X " % i
print(s)
```

Output

```
10
06 50 79 74 68 6F 6E 00 00 00
```

struct

Using the pascal style strings allows one to truncate a string to its original size when unpacking.

Python 2.x/3.x

```
import struct

result = struct.unpack("10p",struct.pack("10p",b"Python"))
print (result)
result = struct.unpack("10s",struct.pack("10s",b"Python"))
print (result)
```

In the second case (using “s” instead of “p”) the string has extra 0 (zeros) padded at its end.

Output

```
('Python',)
('Python\x00\x00\x00\x00',)
```