南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Lab 5. Precautions for pointer , Memory Management(1)

王薇，于仕琪，廖琪梅, 王大兴

Topic

- Precautions for pointer
  - DON'TS
  - Suggestion
    - ✓Coding specification
    - ✓Tool: valgrind

- Memory Management(1)
  - Stack vs Heap
    - ✓compiler+system vs programmer
  - C/C++ vs Python
    - ✓compiler vs interpreter
    - ✓compiler+system+programmer vs interpreter+programmer

Precautions for Pointer

- DON'TS
  - 1. whild pointer
  - 2. memory leak
  - 3. free less or free more
  - 4. free stack
  - 5. dangling pointer

- Suggestion
  - Coding specification
  - Tools

DON'TS : 1. wild pointer

```c
#include<stdio.h>   //wild_pointer.c
#include<stdlib.h>
int main(intargc, char* argv[]){
    int*p1;
    *p1=0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return0;
}
```

```c
#include<stdio.h>   //wild_pointer.c
#include<stdlib.h>
int main(int argc, char* argv[]){
    int *p1=NULL;
    *p1=0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc wild_pointer.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
Segmentation fault (core dumped)
```

Wild pointers refer to pointers that have not been initialized or have been released but are still in use. The positions pointed to by these pointers are uncertain, random, and have no clear limitations.

Wild pointers may cause program crashes or unpredictable results, as the memory addresses they point to may already be occupied by other objects or programs, or reclaimed by the operating system

DON'TS : 2. memory leak

```c
#include<stdio.h>   //demo1.c
#include<stdlib.h>
int main(intargc, char* argv[]){
    int *p1=(int*)malloc(sizeof(int));
    *p1=0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return0;
}
```

```c
#include<stdio.h>   //demo2.c
#include<stdlib.h>
int main(intargc, char* argv[]){
    int d1=0x12345678;
    int *p1=&d1;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return0;
}
```

Memory leak refers to the waste of system memory **caused by dynamically allocated heap memory** in a program that is not released or cannot be released for some reason, resulting in **serious consequences such as slow program running speed or even system crashes**.

Q: Which piece(s) of code would lead to memory leak? demo1.c, demo2.c or both ?

DON'TS : 3. free more or free less

```c
#include <stdio.h>  //free less
#include <stdlib.h>
int main(int argc, char*argv[]){
    int *p1 = malloc(sizeof(int)*1);
    int *p2 = malloc(sizeof(int)*1);
    *p1=0x12345678;
    *p2=*p1;
    printf("p1:%p\tdata:0x%x\n
           p2:%p\tdata:0x%x\n",p1,*p1,p2,*p2);
    free(p1);
    return 0;
}
```

```c
#include <stdio.h>    //free more
#include <stdlib.h>
int main(int argc, char*argv[]){
    int *p1 = malloc(sizeof(int)*1);
    *p1=0x12345678;
    int *p2 = p1;
    printf("p1:%p\tdata:0x%x\n
           p2:%p\tdata:0x%x\n",p1,*p1,p2,*p2);
    free(p1);
    free(p2);
    return 0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
p1:0x55c8cd4132a0       data:0x12345678
p2:0x55c8cd4132c0       data:0x12345678
```

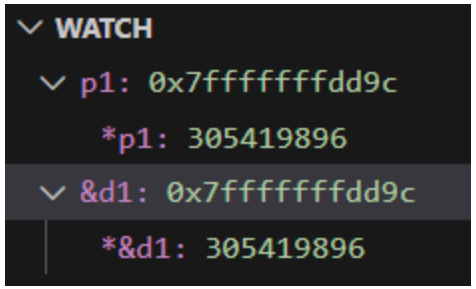
```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
p1:0x562895d2e2a0       data:0x12345678
p2:0x562895d2e2a0       data:0x12345678
free(): double free detected in tcache 2
Aborted (core dumped)
```

Q. Which piece of code would lead to memory leak, which piece of code would lead to program abort with error?

DON'TS : 4. free stack

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char*argv[]){
    int *p1=NULL;
    int d1=0x12345678;
    p1 = &d1;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    free(p1);
    return 0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
address: 0x7ffe8aa8477c data: 0x12345678
free(): invalid pointer
Aborted (core dumped)
```



Q1. What's the value of p1 after finish the assignment "`p1 = &d1;`"?
Q2. Is the address of P1 belongs to stack or heap?
Q3. While using free/del to release the space on stack, what would happen?

DON'TS : 5. dangling pointer

```c
#include <stdio.h>  //dangling_pointer
#include <stdlib.h>
int main(int argc, char*argv[]){
    int *p1 = (int*) malloc(sizeof(int)*1);
    *p1 = 0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    free(p1);
    *p1 = 0x78563421;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return 0;
}
```

```c
#include <stdio.h>  //dangling_pointer
#include <stdlib.h>
int main(int argc, char*argv[]){
    int *p1 = (int*) malloc(sizeof(int)*1);
    *p1 = 0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    free(p1);
    p1=NULL;
    *p1 = 0x78563421;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return 0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc dangling_pointer.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
address: 0x55afc6f8a2a0 data: 0x12345678
address: 0x55afc6f8a2a0 data: 0x78563421
```

Seems Ok But Dangerous!!

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc dangling_pointer.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
address: 0x5630898572a0 data: 0x12345678
Segmentation fault (core dumped)
```

Tools： valgrind(1)

```c
#include<stdio.h>    //memory_leak.c
#include<stdlib.h>
int main(int argc, char*argv[]){
    int *p1=(int*)malloc(sizeof(int));
    *p1=0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return0;
}
```

step1. using “-g” option along with gcc/g++ to generate the executable file.
step2. invoke valgrind with “--leak-check=full” as option, the executable file as parameter to check the memory leak on the executable file.

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc -g memory_leak.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ valgrind --leak-check=full ./a.out
==9103== Memcheck, a memory error detector
==9103== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9103== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9103== Command: ./a.out
==9103==
address: 0x4a48040      data: 0x12345678
==9103==
==9103== HEAP SUMMARY:
==9103==     in use at exit: 4 bytes in 1 blocks
==9103==   total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==9103==
==9103== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9103==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==9103==    by 0x109185: main (memory_leak.c:4)
==9103==
==9103== LEAK SUMMARY:
==9103==    definitely lost: 4 bytes in 1 blocks
==9103==    indirectly lost: 0 bytes in 0 blocks
==9103==      possibly lost: 0 bytes in 0 blocks
==9103==    still reachable: 0 bytes in 0 blocks
==9103==         suppressed: 0 bytes in 0 blocks
==9103==
==9103== For lists of detected and suppressed errors, rerun with: -s
==9103== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

program running results

details about the memory leak

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can **automatically detect many memory management and threading bugs, and profile your programs in detail.** You can also use Valgrind to build new tools.

https://valgrind.org/

Tools： valgrind(2)

```c
#include <stdio.h>  //dangling_pointer
#include <stdlib.h>
int main(int argc, char*argv[]){
    int *p1 = (int*) malloc(sizeof(int)*1);
    *p1 = 0x12345678;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    free(p1);
    *p1 = 0x78563421;
    printf("address: %p\tdata: 0x%x\n",p1,*p1);
    return 0;
}
```

NOTES:
The program can be executed and the results appear correct,
but it brings greater risks!!!

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc -g dangling_pointer.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out
address: 0x56089bdb12a0 data: 0x12345678
address: 0x56089bdb12a0 data: 0x78563421
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ valgrind --tool=memcheck ./a.out
==37617== Memcheck, a memory error detector
==37617== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==37617== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==37617== Command: ./a.out
==37617==
address: 0x4a48040      data: 0x12345678
==37617== Invalid write of size 4
==37617==    at 0x1091E2: main (dangling_pointer.c:8)
==37617==  Address 0x4a48040 is 0 bytes inside a block of size 4 free'd
==37617==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==37617==    by 0x1091DD: main (dangling_pointer.c:7)
==37617==  Block was alloc'd at
==37617==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==37617==    by 0x1091A5: main (dangling_pointer.c:4)
==37617==
==37617== Invalid read of size 4
==37617==    at 0x1091EC: main (dangling_pointer.c:9)
==37617==  Address 0x4a48040 is 0 bytes inside a block of size 4 free'd
==37617==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==37617==    by 0x1091DD: main (dangling_pointer.c:7)
==37617==  Block was alloc'd at
==37617==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==37617==    by 0x1091A5: main (dangling_pointer.c:4)
==37617==
address: 0x4a48040      data: 0x78563421
==37617==
==37617== HEAP SUMMARY:
==37617==     in use at exit: 0 bytes in 0 blocks
==37617==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==37617==
==37617== All heap blocks were freed -- no leaks are possible
==37617==
==37617== For lists of detected and suppressed errors, rerun with: -s
==37617== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

program running results

invalid write/read on memory

program running results

Coding specification

- Tips1:
  - ✓ check if malloc/new is successful
- Tips2:
  - ✓ don't forget to free/del the space
- Tips3:
  - ✓ assign NULL to the pointer after del/free the related space

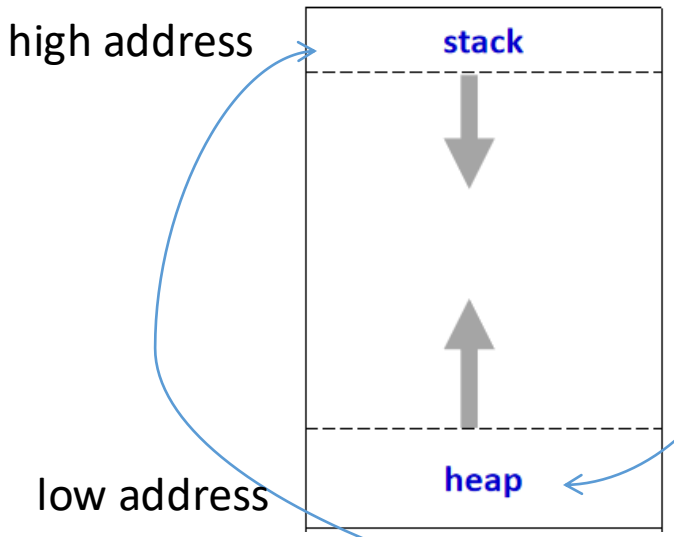
```c
#include<stdio.h>   //demo.c
#include<stdlib.h>
int main(int argc, char*argv[]){
    int *p1=(int*)malloc(sizeof(int));
    if(NULL!=p1){
        *p1=0x12345678;
        printf("address: %p\tdata: 0x%x\n",p1,*p1);
        free(p1);
        p1=NULL;
    }
    return 0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc -g demo.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ valgrind --leak-check=full ./a.out
==57150== Memcheck, a memory error detector
==57150== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==57150== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==57150== Command: ./a.out
==57150==
address: 0x4a48040      data: 0x12345678
==57150==
==57150== HEAP SUMMARY:
==57150==     in use at exit: 0 bytes in 0 blocks
==57150==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==57150==
==57150== All heap blocks were freed -- no leaks are possible
==57150==
==57150== For lists of detected and suppressed errors, rerun with: -s
==57150== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Memory Managment-Stack vs Heap(1)

Both Stack and heap belongs to dynamic memory area.

- Stack: LIFO, expand from high address to low address.
- heap: expand from low address to high address.



lab5 > C demo.c > main(int, char * [])

```
3  int main(int argc, char*argv[]){
4      int *p1=(int*)malloc(sizeof(int));
5      if(NULL!=p1){
6          *p1=0x12345678;
7          printf("address: %p\tdata: 0x%x\n",p1,*p1);
8          free(p1);
9          p1=NULL;
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS 1 MEMORY TERMINAL endian

```
-exec info proc mapping
process 69522
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5/demo
      0x555555555000     0x555555556000     0x1000     0x1000 /mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5/demo
      0x555555556000     0x555555557000     0x1000     0x2000 /mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5/demo
      0x555555557000     0x555555558000     0x1000     0x2000 /mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5/demo
      0x555555558000     0x555555559000     0x1000     0x3000 /mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5/demo
      0x555555559000     0x55555557a000    0x21000        0x0 [heap]
      0x7ffff7dcc000     0x7ffff7dee000    0x22000        0x0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7dee000     0x7ffff7f66000   0x178000    0x22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f66000     0x7ffff7fb4000    0x4e000   0x19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fb4000     0x7ffff7fb8000     0x4000   0x1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fb8000     0x7ffff7fba000     0x2000   0x1eb000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fba000     0x7ffff7fc0000     0x6000        0x0
      0x7ffff7fc9000     0x7ffff7fcd000     0x4000        0x0 [vvar]
      0x7ffff7fcd000     0x7ffff7fcf000     0x2000        0x0 [vdso]
      0x7ffff7fcf000     0x7ffff7fd0000     0x1000        0x0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7fd0000     0x7ffff7ff3000    0x23000     0x1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ff3000     0x7ffff7ffb000     0x8000    0x24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
```

Memory Managment-Stack vs Heap(2)

lab5 > C mm_stack_demo.c > main(int, char * [])

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char*argv[]){
4      char str[]="I'm here.";
5      char p[1024*1024*10] = {};
6      return 0;
7  }
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS 1 MEMORY TERMINAL

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc -g -O0 -o mm_stack_demo mm_stack_demo.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ valgrind --tool=memcheck ./mm_stack_demo
==91562== Memcheck, a memory error detector
==91562== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==91562== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==91562== Command: ./mm_stack_demo
==91562==
==91562== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==91562==
==91562== Process terminating with default action of signal 11 (SIGSEGV)
==91562==  Access not within mapped region at address 0x1FFE801D30
==91562== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==91562==    at 0x109180: main (mm_stack_demo.c:3)
==91562==  If you believe this happened as a result of a stack
==91562==  overflow in your program's main thread (unlikely but
==91562==  possible), you can try to increase the size of the
==91562==  main thread stack using the --main-stacksize= flag.
==91562==  The main thread stack size used in this run was 8388608.
==91562== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==91562==
==91562== Process terminating with default action of signal 11 (SIGSEGV)
==91562==  Access not within mapped region at address 0x1FFE801D28
==91562== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==91562==    at 0x4831134: _vgnU_freeres (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_core-amd64-linux.so)
==91562==  If you believe this happened as a result of a stack
==91562==  overflow in your program's main thread (unlikely but
==91562==  possible), you can try to increase the size of the
==91562==  main thread stack using the --main-stacksize= flag.
==91562==  The main thread stack size used in this run was 8388608.
==91562==
==91562== HEAP SUMMARY:
==91562==     in use at exit: 0 bytes in 0 blocks
==91562==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
```

stack overflow

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char*argv[]){
   char str[]="I'm here.";
   char p[1024*1024*10] = {};
   return 0;
}
```

Requesting a large space in the **stack** space may lead to stack overflow.

In this demo, the size of the space is 1024*1024*10 sizeof char.

The space on stack for C/C++ is managed by Compiler and the system.

Q. Smaller the size of char array “p”, such as 1024*10, generate the executable file and use valgind again, what’s the result?

Memory Managment-Stack vs Heap(3)

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char*argv[]){
    char str[]="I'm here.";
    char *p = (char*)malloc(sizeof(char)*1024*1024*10);
    if(p!=NULL){
        free(p);
        p=NULL;
    }
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS 1 MEMORY TERMINAL

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ gcc -g -O0 -o mm_heap_demo mm_heap_demo.c
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ valgrind --tool=memcheck ./mm_heap_demo
==91012== Memcheck, a memory error detector
==91012== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==91012== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==91012== Command: ./mm_heap_demo
==91012==
==91012==
==91012== HEAP SUMMARY:
==91012==     in use at exit: 0 bytes in 0 blocks
==91012==   total heap usage: 1 allocs, 1 frees, 10,485,760 bytes allocated
==91012==
==91012== All heap blocks were freed -- no leaks are possible
==91012==
==91012== For lists of detected and suppressed errors, rerun with: -s
==91012== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$
```

it's ok to apply for a large space on heap.
DO remember to free it.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char*argv[]){
   char str[]="I'm here.";
   char *p =
(char*)malloc(sizeof(char)*1024*1024*10);
   if(p!=NULL){
      free(p);
      p=NULL;
   }
   return 0;
}
```

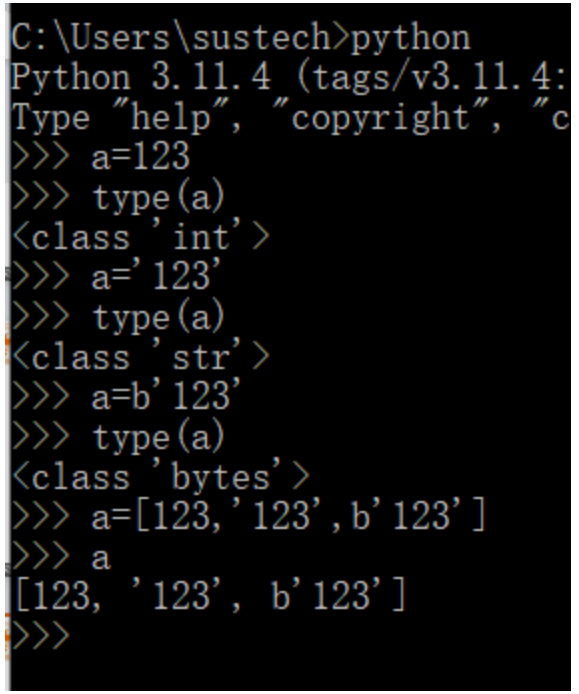
Requesting a large space in the **heap** space would not lead to heap overflow.

In this demo, the size of the space is 1024*1024*10 sizeof char.

The space on heap for C/C++ is managed by programmer.

Memory Managment-C/C++ VS Python(1)

	C/C++	Python
Language Type	Compiled	Interpreted
Operating efficiency	Faster, real time	Slower
Static/dynamic type	Static type languages determine variable types at compile time	Dynamic type languages determine variable types at runtime.
Memory Mangment	1. Compiler + System for non-heap space 2. programmer for heap	1. interpreter for most situation 2. garbage collector used by programmer for very few situation
others	...	...



https://docs.python.org/3/c-api/memory.html

Exercise 1

```c
#include<stdio.h>
int main()
{
   int numbers1[] = {2,4,6,8,10};
   int sum = 0;
   int *p1 = &numbers1[1];
   printf("numbers1 = %p\n", numbers1);
   printf("p1       = %p\n", p1);
   for(int i = 0; i < 3; i++)
      sum += *(p1+i);
   printf("sum      = %d\n",sum);

   int numbers2[5]={1,2,3,4,5};
   int *p2 = (int*)(&numbers2 + 1);
   printf("numbers2     = %p\n", numbers2);
   printf("numbers2 + 4 = %p\n", numbers2 + 4);
   printf("p2           = %p\n", p2);
   printf("*(numbers2+1)= %d\n",*(numbers2+1));
   printf("(p2-1)       = %d\n",*(p2-1));
   return 0;
}
```

Run the program and explain the result to SA.

Exercise 2

```
#include <iostream>
using namespace std;
int main()
{
   int matrix[][4] = {1,3,5,7,9,11,13,15,17,19};
   int *p = *(matrix + 1);
   p += 3;
   cout << "*p++ = " << *p++ << endl;

   const char *str = "Welcome to programming.";
   long *q = (long *)str;
   q++;
   char *r = (char *)q;
   cout << r << endl;

   unsigned int num = 0x3E56AF67;
   unsigned short *pshort = (unsigned short *) &num;
   cout << "*pshort = 0x" <<  hex << *pshort << endl;
   return 0;
}
```

Run the program and explain the result to SA.

Exercises 3

- 3-1. Complete the code on the right to finish the following task:
  - 1. Determine whether the current system is in big-endian(BE) or little-endian(LE) based on the storage location of byte0 in `numA`.
  - 2. Store each byte in `numA` to a new space (`numB` or pointed by `pnumB`) in reverse order.
    - ✓ If the command-line parameter of the program is 'H', use heap mode to implement swapping.
    - ✓ If the command-line parameter of the program is 'S', use stack mode to implement swapping.
    - ✓ Print out the value of `numB` (or pointed by `pnumB`) in hexadecimal.
- 3-2. Use the tool valgrind to check if there is memory problem on the code.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
   int numA = 0x11223344;
   if(argc == 2){
      if(argv[1][0]=='H'){
         int *pnumB = (int*)malloc(sizeof(int));
         if(pnumB!=NULL){
            /*complete code here*/
         }
      }
      else if(argv[1][0]=='S'){
         /*complete code here*/
      }
   }
   return 0;
}
```

```
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out H
Data A_addr: 0x0x7ffed47fb3f8, A_data: 0x11223344, This is LE
Data B_addr: 0x0x562d6f40a2a0, B_data: 0x44332211
ww2@DESKTOP-4NIH4UK:/mnt/c/Users/sustech/Desktop/C_CPP_CODE/lab5$ ./a.out S
Data A_addr: 0x0x7ffdd4954cc8, A_data: 0x11223344, This is LE
Data B_addr: 0x0x7ffdd4954ccc, B_data: 0x44332211
```

Tips on Big-Endian and Little-Endian

BE stores the big-end first, the lowest memory address is the biggest.
LE stores the little-end first, the lowest memory address is the littlest.

Big-Endian

2003	44
2002	33
2001	22
2000	11

Little-Endian

2003	11
2002	22
2001	33
2000	44

```
#include<stdio.h>
union data
{
   int a;
   char c;
};

int main()
{
   union data endian;
   endian.a = 0x11223344;

   if(endian.c == 0x11)
      printf("Big-Endian\n");
   else if(endian.c == 0x44)
      printf("Little-Endian\n");

   return 0;
}
```

Q: Run the demo on your system, is your system Big-Endian or Little-Endian?