内存溢出

内存溢出 OOM （out of memory），是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个int,但给它存了long才能存下的数，那就是内存溢出。

内存泄漏

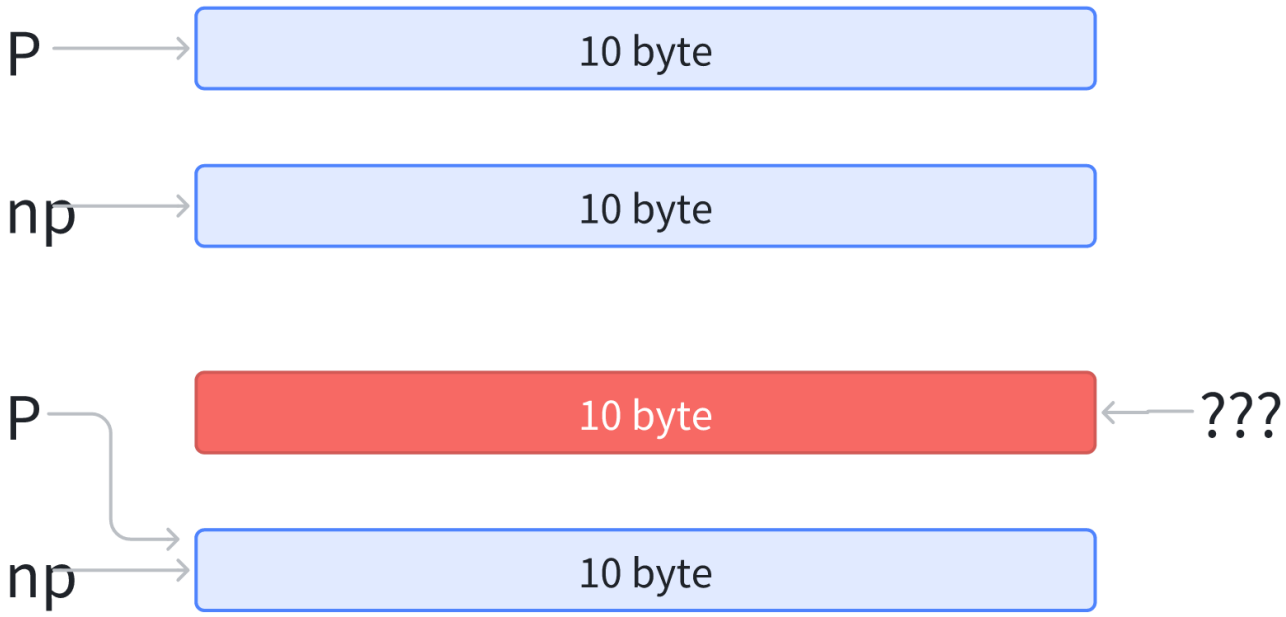
内存泄露 （memory leak），是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但**内存泄露堆积后果很严重**，无论多少内存,迟早会被占光。最终的结果就是导致**OOM**。 内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。

造成内存泄露的原因

- **指针重新赋值**

```
char* p = (char*)malloc(10);
char* np = (char*)malloc(10);
p = np;
```

其中，指针变量 p 和 np 分别被分配了 10 个字节的内存。如果程序执行p = np：



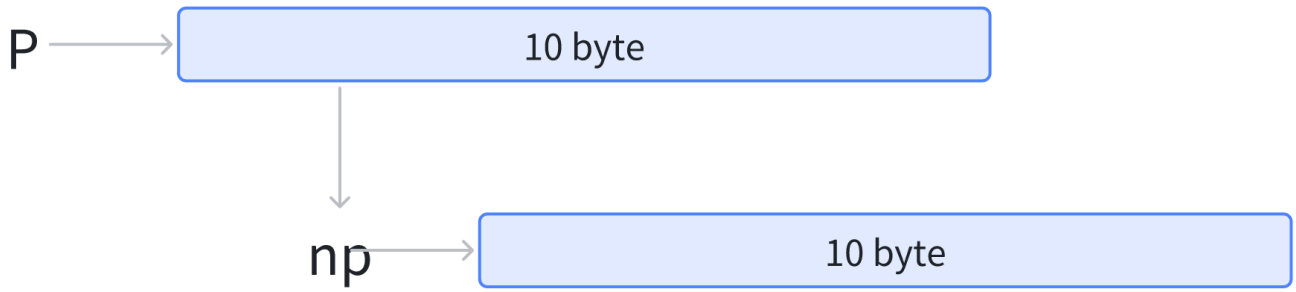
这时候，指针变量 p 被 np 指针重新赋值，其结果是 p 以前所指向的内存位置变成了孤立的内存。它无法释放，因为没有指向该位置的引用，从 而导致 10 字节的内存泄漏。

类似的情况，连续重复new的情况也是类似：

```
int* p = new int;
p = new int ...;  //错误
```

- **错误的内存释放**

假设有一个指针变量 p，它指向一个 10 字节的内存位置。该内存位置的第三个字节又指向某个动态分配的 10 字节的内存位置。



如果程序要执行以下赋值语句时：

```
free(p);
```

很显然，如果通过调用 free 来释放指针 p，则 np 指针也会因此而变得无效。np 以前所指向的内存位置也无法释放，因为已经没有指向该位置的 指针。换句话说，np 所指向的内存位置变为孤立的，从而导致内存泄漏。 因此，每当释放结构化的元素，而该元素又包含指向动态分配的内存位置的指针时，应首先遍历子内存位置（如本示例中的 np），并从那里开始 释放，然后再遍历回父节点，如下面的代码所示：

```
free(p->np);
free(p);
```

- **返回值不正确处理**

有时候，某些函数会返回对动态分配的内存的引用，如下面的示例代码所示：

```
char *f(){
 return (char *)malloc(10);
}
void f1(){
 f();
}
```

函数 f1 中对 f 函数的调用并未处理该内存位置的返回地址（就是没接收返回的指针地址），其结果将导致 f 函数所分配的 10 个字节的块丢失，并导致内存泄漏。

- 在内存分配后忘记使用 free 进行释放，这个很容易理解

如何避免内存泄漏

- **确保没有在访问空指针。 **
- **每个内存分配函数都应该有一个 free 函数与之对应，alloca 函数除外。 **
- **每次分配内存之后都应该及时进行初始化，可以结合 memset 函数进行初始化，calloc 函数除外。 **
- **每当向指针写入值时，都要确保对可用字节数和所写入的字节数进行交叉核对。 **
- **在对指针赋值前，一定要确保没有内存位置会变为孤立的。 **
- **每当释放结构化的元素（而该元素又包含指向动态分配的内存位置的指针）时，都应先遍历子内存位置并从那里开始释放，然后再遍历回父节点。 **
- **始终正确处理返回动态分配的内存引用的函数返回值。**

内存泄露工具

这是一个linux下的内存泄露检测工具，具体内存泄露情况如下：

test.cpp

```
#include <iostream>
using namespace std;

int main(){
    int* a = new int(5);
    cout<<*a<<endl;
    return 0;
}
```

问题 输出 调试控制台 终端 1: bash

```
kuanyew@DESKTOP-H1Q5E6E:/mnt/g/ubuntu$ valgrind ./a.out
==2557== Memcheck, a memory error detector
==2557== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2557== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2557== Command: ./a.out
==2557==
==2557== error calling PR_SET_PTRACER, vgdb might block
5
==2557==
==2557== HEAP SUMMARY:
==2557==     in use at exit: 4 bytes in 1 blocks
==2557==   total heap usage: 3 allocs, 2 frees, 73,732 bytes allocated
==2557==
==2557== LEAK SUMMARY:
==2557==    definitely lost: 4 bytes in 1 blocks
==2557==    indirectly lost: 0 bytes in 0 blocks
==2557==      possibly lost: 0 bytes in 0 blocks
==2557==    still reachable: 0 bytes in 0 blocks
==2557==         suppressed: 0 bytes in 0 blocks
==2557== Rerun with --leak-check=full to see details of leaked memory
==2557==
==2557== For lists of detected and suppressed errors, rerun with: -s
==2557== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```


无内存泄露情况：

test.cpp

```cpp
#include <iostream>
using namespace std;


int main(){
    int* a = new int(5);
    cout<<*a<<endl;
    delete a;
    return 0;
}
```

问题 输出 调试控制台 终端 1: bash

```
kuanyew@DESKTOP-H1Q5E6E:/mnt/g/ubuntu$ g++ test.cpp
kuanyew@DESKTOP-H1Q5E6E:/mnt/g/ubuntu$ ./a.out
5
kuanyew@DESKTOP-H1Q5E6E:/mnt/g/ubuntu$ valgrind ./a.out
==2564== Memcheck, a memory error detector
==2564== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2564== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2564== Command: ./a.out
==2564==
==2564== error calling PR_SET_PTRACER, vgdb might block
5
==2564==
==2564== HEAP SUMMARY:
==2564==     in use at exit: 0 bytes in 0 blocks
==2564==   total heap usage: 3 allocs, 3 frees, 73,732 bytes allocated
==2564==
==2564== All heap blocks were freed -- no leaks are possible
==2564==
==2564== For lists of detected and suppressed errors, rerun with: -s
==2564== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
kuanyew@DESKTOP-H1Q5E6E:/mnt/g/ubuntu$
```