C/C++ 踩过的坑和防御式编程

相信你或多或少地用过或者了解过 C/C++，尽管今天越来越少地人直接使用它，但今天软件世界大多数软件都构筑于它，包括编译器和操作系统。因此掌握一些 C/C++ 技能的重要性不言而喻。

这场 Chat 本人将从小处入手，以亲身踩过的坑作为示例，讲述一下 C++ 的常见的坑，以及其防御方法——防御式编程。主要内容包括：

- C/C++ 基础知识简介
- C/C++ 常见问题复现示例
- 内存泄露问题排查
- 防御式编程理论
- 防御式编程实践

大家好，我是林奇思妙想。很高兴能够和各位在 Gitchat 上交流一些平常用C/C++, 写C/C++ 的经验分享。

为了表明这是一场严肃的、有深度的交流，我们先不走偏，先回顾一下经典教材上关于C/C++的基础知识。尽量保持简单，点到为止，希望你没有被吓走。作为一枚典型猿族，我就话不多说，直接带领大家入坑（笑~）。

注：所有的程序示例在MS VS2017下调试的。

C/C++ 基础知识简介

我们说到 C/C++ 一般指两部分:

- C++兼容 C 的部分
- C++独有部分

先看一下C/C++共有部分（有基础的同学可以略过这一部分, 或者可以跟我一起温习）。

C/C++ 共有部分

常量

常量是指运行时值不能改动的一类“变量”，他们的值是编译进目标程序中的。

1）立即数

如字面常量 12, 123.5f, “abc”.

2）常量对象

```
const int SIZE_A = 11;
const Mat MAT_A(12,22,-1);
```

变量在运行时占有内存地址空间，且它的值可以在运行时被更改

变量

1）普通值变量

```
float a = 9.1f;
Mat mat_a(1,2,-1);
```

2）指针变量 p_a

```
int *p_a = &a;
```


表达式

表达式是指能被编译器编译为指令的语句，通常以“；”结束。

表达式分以下几种:

1）赋值

```
int a = 4;
```

2）逗号

```
a = 9,  b = 11;
```

3）判断

```
if (a > b)
{
```

```
do_something();
}
```

4）循环

```
for(int i=0; i<MAX; i++)
{
do_something();
}
```

5）函数调用

```
do_something();
```

其它数据结构

结构体等

```
struct st_a
{
int x;
int y;
};
```


通常是 public 的，即没有封装性。

C++独有

类是C++所独有，也称对象。通常我们用一个类来表示一类客观事物的层次、继承关系。

如下图所示：

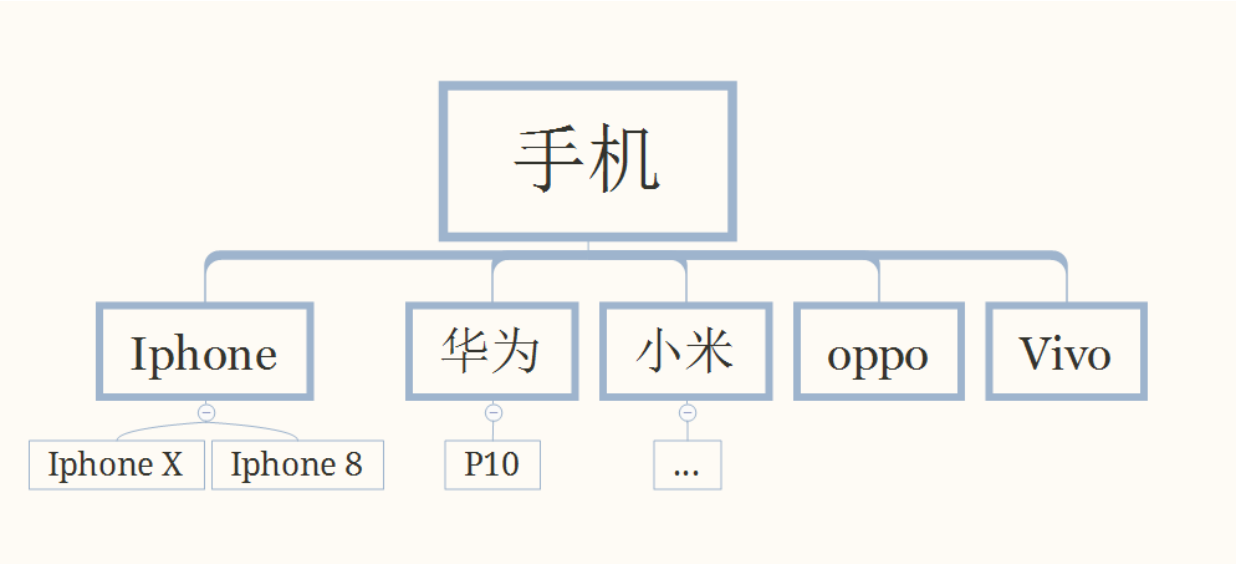

图 0.1 类的继承关系

类

1）继承

继承就如0.1所示，从父类到子类，越来越具体。

2）封装

封装是指对于某一子类，可以控制哪些信息能被外部看到，如控制我们能获取手机的大小，颜色等属性，对用户隐藏手机串号等信息。

3）多态

多态是C++精华所在，但也是C++的难点所在。一些教材常常用高大上的描述把人搞昏。譬如我手边这本书里面是这样讲多态的（C++程序设计教程——钱能版）：

> “多态是基于类的层次结构的，当指针飘忽不定地可能指向类层次中的上下不同对象时，以指针间访的形式实施的操作便是表现多态的条件。”

这段文字真是一骑绝尘，不食人间烟火，难免把人带到小黑屋子——关着。

用通俗的话说，多态是指多个子类有一个共有操作，我们在父类中定义一个统一的抽象虚接口，然后各个子类分别实现。这样子，运行时，依据子类是什么，动态选择子类的方法。 这样子描述，我们又不可避免地走入了经典教材的巨梗——不懂啊！ 不如直接看如下代码吧。

如下代码所示：

```
// code start
 /*
 B is base class,
```

```
A -- C is sub-class
*/
class B
{
public:
    virtual void do_sth() = 0;
};

class A : public B
{
public:
    void do_sth()
    {
        cout << "- A do_sth()\n";
    }
};

class C : public B
{
public:
    void do_sth()
    {
        cout << "- C do_sth()\n";
    }
};

void do_sth(B *id_b)
{
    id_b->do_sth();
}

int main()
{

    A* id_a = new A();
    C* id_c = new C();

    do_sth(id_a);
    do_sth(id_c);

    return 0;
}

// code end
```


函数运行输出结果是：

```
A do_sth()
C do_sth()
```

注意其中的粗体代码。

看“ void do_sth(B *id_b)” , 我们用的基类指针作为函数的接口参数，但是“do_sth(id_a); ”传递参数时，我们传的是A 或者 C 对象的指针！ 多态使得调用的接口一致，更利于抽象和简化。

C/C++ 常见问题复现示例

如果算上最新接触C到现在，已经有9年。写过无数的C/C++代码。有些坑是自己挖的，有些则是语言层面上的“陷阱” 。

坑**1**

立即数左移越出范围。先看一段代码：

```
assert((1 <<  3) == pow(2,  3));
assert((1 << 30) == pow(2, 30));
assert((1 << 62) == pow(2, 62));
```

先不运行，猜测一下问题在哪一行？

运行结果如下：

```
Assertion failed: (1 << 62) == pow(2, 62), file
```

我们再进一步看一下他们的值：

```
cout << pow(2, 62) << endl;
cout << (1 << 62) << endl;
```

```
4.61169e+18
0
```

可知前者是对的，后者是错的，在C语言中，左移结果最大是 32 位。

为了验证，我们再看一下：

```
cout << pow(2, 62) << endl;
cout << (1 << 62) << endl;
cout << (4.61169e+18) << endl;
```

运行结果是：

```
4.61169e+18
0
4.61169e+18
```

符合我们的预期。

坑**2**

sprintf()越界问题。

```
char buf[10];
float x = 1/3.0f;
sprintf(buf, "cols = %f", x);
printf(buf);
```

运行后，buf会越界，出现地址异常！正确的做法是给 buf 一个更大的地址。 但是这类栈溢出在大型的工程中，防不胜防。其实可以在 C++ 中，考虑用更一种更安全的方式。

```
float f = 1 / 3.0f;
ostringstream ss;
ss << "num is " << f << endl;
cout << ss.str();
```


坑 **3**

case语句，不打括号。

// 示例代码K_0:

```
int num = 2;
switch (num)
{
case 0:
    do_0();
    break;
case 1:
    do_1();
    do_xx();
    do_xxx();
    do_xxxx();
    break;
default:
    do_default();
}
// code end
```

这段代码运行起来没有问题，

逻辑没有问题，还不至于成坑，但是在实际的大型项目中，一个项目因需求变化可能需要频繁改动。如上述代码，有人在”do_xxxx（）”后加上另外一个分支。变成：

```
        do_xx();
        do_xxx();
        do_xxxx();
    case 2:
        do_2();
        do_2_x();
        do_2_xx();
        break;
```

这个时候逻辑就会出问题。而且这种错误很难调试，最好的办法是预防。
预防的方法也很简单，给case分支加上大括号。如下：

```
switch (num)
{
    case 0:
        {
            do_0();
            break;
        }
    case 1:
        {
            do_1();
            do_xx();
            do_xxx();
            do_xxxx();
            break;
        }
    default:
        {
            do_default();
        }
}
```


这种方法看起来很笨，但比后期要发布前出bug,把班加个昏天黑地要强。

坑**4**

sizeof() 用于一个结构体时其值不是绝对的。与平台相关，也与编译指令相关。看例子：

```
struct pack_struct
{
```

```
    char t;
    uint32_t x;
};
cout << sizeof(pack_struct);
```

输出是什么？

有很多同学一看，说这个容易， char 占用一个字节， uint32_t 占用 4 个字节，所以共占用 5 个字节。还有同学想到了4字节对齐，说应该是4的倍数，所以应该是8。那正确的结果又是什么呢？

其实答案是：两者都有可能。与编译控制有关。要想结果得到“5”， 用下面的：

```
#pragma pack(1)
struct pack_struct
{
    char t;
    uint32_t x;
};
#pragma pack()
```

要想得到“8”， 把 pack(1) 改成 pack(4) 。

一般默认的编译参数，不同的平台是不一样的。所以要求我们不能写硬编码的代码。例如下面的代码将得不到我们想要的结果：

```
pack_struct arr_pack[8];
char *p_tmp = (char*)arr_pack;
pack_struct *pack_idx_1 = (pack_struct*) (p_tmp + 5);
```

示例有点儿绕，我们j是想要通过指针的方式得到数组arr_pack的第一个指针。这里用了硬编码，写成了5。这里硬编码的问题可能是，在其它的平台上不一定能正确工作！

为了方便移植，应该改写成：

```
pack_struct arr_pack[8];
char *p_tmp = (char*)arr_pack;
pack_struct *pack_idx_1 = (pack_struct*) (p_tmp +
sizeof(pack_struct));
```

坑5

浮点数的比较。某君写了如下代码：

```
    float x = 1.333 - 1;
    cout << x << endl;

    if (x == 0.333)
    {
        cout << "x = " << 0.333  << endl;
    }
    else
    {
        cout << "x != " << 0.333 << endl;
    }
```

请问一下输出是什么？

不用想，笔者在这里把它们写出来，肯定是有坑的，所以输出结果也是“惊世骇俗”的，输出为：

```
0.333
x != 0.333
```

看起来不可思议！！！问题来了，那怎样才能正确地作浮点数相等判断呢？也很简单，一般用偏离一个中心的距离小于某个精度来判断相等：


```
if (fabs(f1 - f2) < 预先指定的精度)
{
    //...
}
```

实际一般推荐用 1e-5 作为精度（看各项目的精度要求啰）。

坑**6**：模版的使用

首先来看一下一个正常的模版类使用方法：

```
`// ### main.cpp  ###`

    #include <iostream>
    using namespace std;

    template<class T> void disp(T t);

    template<class T>
    void disp(T t)
    {
        cout << t << endl;
    }
```

```
int main()
{
    disp(8);
    disp<float>(8.8f);
    return 0;
}
```

这个能正常编译运行，但是把这些模版细节都放到 main.cpp 中，看得很昏。于是我们想要把他们移出 main.cpp 。

假如我们有一个头文件 test.h 和一个实现文件 test.cpp 。

我们把模版的声明移到 test.h ， 模版的实现移至 main.cpp ， 此时三个文件内容分别如下：

```
// ### test.h ###


template<class T> void disp(T t);

// ### test.cpp ###
template<class T>
void disp(T t)
{
    cout << t << endl;
}
```



```
/ ### main.cpp ###


#include <iostream>
using namespace std;

int main()
{
    disp(8);
    disp<float>(8.8f);
    return 0;
}
```

编译运行，发现Linker出错：

```
error LNK2019: unresolved external symbol "void __cdecl disp<int>(int)"
(??$disp@H@@YAXH@Z) referenced in function main
error LNK2019: unresolved external symbol "void __cdecl disp<float>
(float)" (??$disp@M@@YAXM@Z) referenced in function main
fatal error LNK1120: 2 unresolved externals
-- FAILED.
```

错误出现了，我们想知道“为什么”？

这是因为目前 C++ 还不支持模版分开编译。分开会导致模版函数在具化过程中找不到外部符号。通常地做法是把声明和实现全部放在头文件中。下面是正确的版本：

```
// ### test.h ###


        template<class T> void disp(T t);

        template<class T>
        void disp(T t)
        {
            cout << t << endl;
        }


// ### test.cpp ###


        // test.cpp file content...


// ### main.cpp ###


        #include <iostream>
        using namespace std;

        int main()
        {
            disp(8);
            disp<float>(8.8f);
            return 0;
        }
```


这回终于对了！

坑**7**：栈被意外修改

看下面这段代码。你觉得 Line 28 会异常吗？ 实际结果是”会“！

```
18  int main()
19  {
20
21      FILE *fp = fopen("main.cpp", "rb");
22      char a ;
23      char *buf = &a;
24      int N = 99;
25
26      fread(buf, 1, 1024, fp);
27
28      assert(N == 99);
29
30      fclose(fp);
31
```

这个就是我在工作中遇到的一个实际问题，当时一直监控一个变量，这个变量总是莫名其妙被更改了， 最后挖出来罪魁祸首就是一个读把栈破坏了。结果我的 “N” 成了一个无意义的超大数。

所以一定要严格控制好指针的越界行为，编译器无法知道你的意图，这个只有靠自己来把控，但也有一些方法来排查和防卫这种问题的发生。这是接下来要讲的内容。

内存泄露问题排查

内存泄露问题是大型 C++项目中最棘手的问题之一。有人会想，嘿，反正我内存多，不用担心用完，让它去泄露吧， 但我只能告诉你， too young , too simple, sometimes naive。

内存只有有哪怕一个地址的泄露，都可能导致严重的宕机事故，特别是在一些重要的嵌入式领域，如医疗，核电，飞行控制器软件中。

有很多工具可以帮忙排查内存泄露问题。从怀疑内存泄露到证实一般要经过以下的步骤：

怀疑

开机后一切正常，系统运行一段时间后，越来越卡。此时需要排除是不是 CPU 温度过高，软件处理数据量是不是变大了。如果都正常，那就着手怀疑是内存泄露问题。

如果在Linux上，可以通过 free －m 命令看到剩余的内存。如果运行长时间后，剩余内存变得越来越小。则大概率是内存泄露。

验证

下一步可以通过一些工具去协助监控内存的分配和释放， 如：coverity (力荐，因为是我的前东家新思科技产的，当然也是最好用的，Valgrind(开源，免费）等。

防御式编程理论

讲了这么多 坑，终于要讲一些理论总结了，应该松口气还是憋口气呢（笑）？

我们编程最终的目标是让产品能平稳运行，而且要”不以人的意志为转移“式地平稳运行。

这其中包括两点：

- 产品代码要逻辑正确、完备；
- 代码能够让人读而知其义，能尽量避免他人犯错；

防御式编程就是在做到 1 后，再把第 2 条做好。

在这里，一定要看清楚，做到1后再做2。 有些软件管理可能过分强调防御，在功能，逻辑都还没有完备时，大搞特搞防御，其实并不可取。其实，本人倾向于当功能雏形做好后，再防御。毕竟过早防御会使得代码过分臃肿。

说了这么久”防御“，读者可能已经昏了。那到底什么是防御？ 别急，让我们先来看一个例子。

如取上面的例子N, 防御式编程应该是：

```
FILE *fp = fopen("main.cpp", "rb");

const int MAX_BUF = 8;
char *buf = new char[MAX_BUF];
int N = 99;
int i_to_read = 1024;

assert(MAX_BUF - i_to_read > 0);
fread(buf, 1, i_to_read, fp);

assert(N == 99);

fclose(fp);
delete[] buf;
```

assert 那一行就是防止别人犯错，并把错误尽早地暴露出来的防御代码。

防御式编程实践

其实防御式编程理论远比这个要深。在实践中，各聪明的大师们总结了一套规律来严防死守内存越界，数组越界，值为负，除数为零， 野指针等。

下面分别说明。

防内存越界。我是懒人，还是取上面的例子：）

```
assert(MAX_BUF - i_to_read > 0);
fread(buf, 1, i_to_read, fp);
```

防数组越界

如下，假如我们要写一个函数，返回数组的索引为 idx 的元素值。

```
int get_arr_by_idx(int *arr, int len, int idx);

int get_arr_by_idx(int *arr, int len, int idx)
{
    assert(idx <= len - 1);
    return arr[idx];
}
```


防值为负

```
char buf[9] = 0;
int sz_to_set = 4;
assert(sz_to_set >= 0);
memset(buf, 0xAF, sz_to_set);
```

防被除数为零

```
double t = 9.9f;
for (int i = -2; i < 2; i++)
{
 assert(i != 0);
 cout << t / i << endl;
}
```

野指针

```
char *buf = NULL;
assert(NULL != buf);
printf(buf);
```

当然，说了这么多的防御技巧，其实最好的防御是命名。（惊讶的表情）。名命得好，可以让人”望文生义“。笔者就常常用下面这几个命名：

- idx：表元素索引， 是大于等于 0 的;
- sz，len：表元素长度;
- p_xx：表示 xx 的一个指针;
- 全大写：表示常量;
- m_xx ：表示成员变量;

当然，关于命名上笔者不是能手，想成为能手，就去读一些成熟的优秀的开源框架中的命名。 相信我，你会受益良多的。推荐 boost，Linux Kernel，Google Protobuf.

上面说了这么多， 如果你有幸从中学到一些，一定要告诉笔者，让笔者欣慰欣慰。

最后， 谢谢你的阅读。

林奇思妙想 于 深圳

2017/11/24