Tips in C++

Tongust

April 23, 2017

1 Static Member

Static member of class should be defined outside the class body, it's prohibitted that both of the decleartion and definitions happened in the class body. Usually, the *static* is defined in the class x.cpp file. Such as: const int mclass::statmem = 10;

2 #ifundef #undef Directive

The #undef directive undefines a constant or preprocessor macro defined previously using #define. For exemple

#define E 2.718
int e_sq = E * E;
#undef E
int E = 1.0;E wiil not be substitued by macro

3 strcmp vs strncmp vs memcmp

```
strcmp    compares null-terminated C strings
strncmp   compares at most N characters of null-terminated C strings
memcmp    compares binary byte buffers of N bytes
So if you have these strings:
  const char s1[] = "12345\0\0\0\0"; // extra null bytes at end
  const char s2[] = "12345\0abc"; // embdded null byte
  const char s3[] = "123456789";
Then these results holds true:
  strcmp(s1,s2) == 0;
  strncmp(s1,s3,5) == 0;
  strncmp(s1,s2,5) == 0;
  strncmp(s1,s2,8) == 0;// string s1 and s2 are the
                           same up through the null terminator
  memcmp(s1,s2,8) != 0;
```

4 dynamic memory and pointer

```
1. apply new memory (sigle):
int *ip = new int(12);
delete ip;
2. apply new memory (1D array)
int *iap = new int[4];
delete []iap;
3. apply new memory (2D array)
int N = 100, M = 100;
int **iapp = new int[N][M];
We need two steps
a)
for (int i = 0; i != N; ++i)
  delete []iapp[i];
or:
int **e = iapp+N;
for (int **t = iapp; t != e; ++t)
  delete []*t;
b)
delete []iapp;
4. apply 2+ order pointer, i.e., a pointer to a pointer that point a new int
4.1 the 2 order pointer points a static pointer
int **ipp = new int*(nullptr);
int d = 0;
*ipp = &d;
delete ipp;
4.2 the second order pointer points to a dynamic pointer
4.2.1
int **ipp = new int*(new int (12) );
delete *ipp;
delete ipp;
4.2.2
int **ipp = new int *(new int (12));
int *ip = new int(13);
delete *ipp;
*ipp = ip;
// NOT ipp = &ip; or *ipp = &ip;
delete ip;
delete ipp;
```

5 non-Virtual Constructor and Virtual Desconstructor

5.1 Why dont we have virtual Constructor?

ref: http://www.stroustrup.com/bs_faq2.html#virtual-ctor
A virtual call is a mechanism to get work done given partial information. In particular, "virtual" allows us to call a function knowing only an interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

5.2 Virtual Desconstructor

A base class generally should define a virtual destructor. When we *delete* a pointer to a dynamically allocated object, it may occur that the static type of the pointer differs from the dynamic one of the object being destroyed. **Executing *delete* on a pointer to base that points to a derived object has undefine behavior if the base's destructor is not virtual.**

```
1  class B {/* Base class */
2  public:
3  /* Need to be virtual.
4   * dynamic binding for the
5   * destructor when we delete the pointer.
6   */
7  virtual ~B() = default;
8  }
```

6 Memory fragmentation

Memory Fragmentation: The classic symptom of memory fragmentation is that you try to allocate a large block and you can't, even though you appear to have enough memory free.
Memory pool is used for solving Memory fragmentation problem. Pool allocation is a memory allocation scheme that is very fast, but limited in its usage. For more information on pool allocation (also called simple segregated storage, see concepts concepts and Simple Segregated Storage). Memory pools are basically just memory you've allocated in advance (and typically in big blocks).

6.1 Placement new

There are many uses of placement new. The simplest use is to place an object at a particular location in memory. This is done by supplying the place as a pointer parameter to the new part of a new expression:

```
1          #include <new>        // Must #include this to use "placement
2          #include "Fred.h"     // Declaration of class Fred
3          void someCode()
4          {
5            char memory[sizeof(Fred)];     // Line #1
6            void* place = memory;          // Line #2
7            Fred* f = new(place) Fred();   // Line #3 (see "DANGER" belo
8            // The pointers f and place will be equal
9            // ...
10       // destroy
11           char memory[sizeof(Fred)];
12           void* p = memory;
13           Fred* f = new(p) Fred();
14           // ...
15           f->~Fred();   // Explicitly call the destructor for the plac
16         }
```

DANGER: You are taking sole responsibility that the pointer you pass to the placement new operator points to a region of memory that is big enough and is properly aligned for the object type that youre creating. Neither the compiler nor the run-time system make any attempt to check whether you did this right. If your Fred class needs to be aligned on a 4 byte boundary but you supplied a location that isnt properly aligned, you can have a serious disaster on your hands.

7 Global var and local var

```
1      /*main.c*/
2      /*-------------------------*/
3      /* global variable */
4      #include <unistd.h>
5      #include <stdio.h>
6      char *glo_ppa[4];
7      char **glo_pp = glo_ppa;
8      int main() {
9          /* local variable */
10         char *loc_ppa[4];
11         printf("%d\n", glo_pp[4]);
```

```
12          /*
13           * echo 0
14           */
15          printf("%d\n", loc_pp[4]);
16          /*
17           * echo non-zero
18           */
19          exit(0);
20      }
```