Reference and Pointer in C++

Let's code for a pointer

下边是 C++ 代码：

```
int main() {
    int A = 0;
    auto *ptr_A = &A;
    *ptr_A = 1;
    return 0;
}
```

好，我们来看看上边这段代码的反汇编代码（为了你更好的理解，我直接使用了 visual studio 的反汇编代码）：

```
int main() {
00DA1680  push        ebp
00DA1681  mov         ebp,esp
00DA1683  sub         esp,0DCh
00DA1689  push        ebx
00DA168A  push        esi
00DA168B  push        edi
00DA168C  lea         edi,[ebp-0DCh]
00DA1692  mov         ecx,37h
00DA1697  mov         eax,0CCCCCCCCh
00DA169C  rep stos    dword ptr es:[edi]
00DA169E  mov         eax,dword ptr [__security_cookie (0DA8004h)]
00DA16A3  xor         eax,ebp
00DA16A5  mov         dword ptr [ebp-4],eax
    int A = 0;
00DA16A8  mov         dword ptr [A],0 ;这里取A的地址并且给A赋值为0，完成变量的声明与初始化操作
    auto *ptr_A = &A;
00DA16AF  lea         eax,[A]  ;将A的地址取出并且将该地址放在 eax(累加器) 寄存器中
00DA16B2  mov         dword ptr [ptr_A],eax  ;将刚才存在 eax 中的地址赋值给 ptr_A
    *ptr_A = 1;
00DA16B5  mov         eax,dword ptr [ptr_A]  ;将 ptr_A 存储的地址赋值给 eax 寄存器
00DA16B8  mov         dword ptr [eax],1  ; 向 eax 中存储的地址中写入立即数 1

    return 0;
00DA16BE  xor         eax,eax
}
```

在汇编语言中，可以总结出来一个流程：

1. 声明并初始化 A 变量。
2. 将 A 变量的地址存储在 ptr_A引用中。
3. 取出 ptr_A 中存储的地址。
4. 向刚才取出的地址中写入立即数 1.

Let's code for a reference

下边是 C++ 代码：

```
int main() {
    int A = 0;
    int &ref_A = A;
    ref_A = 1;

    return 0;
}
```

好，我们来看看上边这段代码的反汇编代码（为了你更好的理解，我直接使用了 visual studio 的反汇编代码）：

```
int main() {
00611680  push        ebp
00611681  mov         ebp,esp
00611683  sub         esp,0DCh
00611689  push        ebx
0061168A  push        esi
0061168B  push        edi
0061168C  lea         edi,[ebp-0DCh]
00611692  mov         ecx,37h
00611697  mov         eax,0CCCCCCCCh
0061169C  rep stos    dword ptr es:[edi]
0061169E  mov         eax,dword ptr [__security_cookie (0618004h)]
006116A3  xor         eax,ebp
006116A5  mov         dword ptr [ebp-4],eax

    int A = 0;
006116A8  mov         dword ptr [A],0 ;这里取A的地址并且给A赋值为0，完成变量的声明与初始化操作
    auto &ref_A = A;
006116AF  lea         eax,[A] ;将A的地址取出并且将该地址放在 eax(累加器) 寄存器中
006116B2  mov         dword ptr [ref_A],eax ; 将刚才存在 eax 中的地址赋值给 ref_A

    ref_A = 1;
006116B5  mov         eax,dword ptr [ref_A] ; 向 ref_A 引用的地址写入 eax 寄存器中
006116B8  mov         dword ptr [eax],1 ; 向 eax 中存储的地址中写入立即数 1

    return 0;
006116BE  xor         eax,eax
}
```

在汇编语言中，可以总结出来一个流程：

1. 声明并初始化 A 变量。
2. 将 A 变量的地址存储在 ref_A 引用中。
3. 向 ref_A 中存储的地址中写入数 1。

分析

首先我们来看着俩程序的流程，基本上是一模一样的，但是好像有点儿不对。pointer 比 reference 好像多了一个步骤。对，就是多了个步骤。那么为什么会这样呢?

我们看看 pointer 和 reference 的概念:

1. pointer 中存储的是指向的变量的地址，**注意**：**“存储”**二字说明 pointer 是不能直接用的。如果想用它，那么就必须先将其存储的地址取出来才能用，也就是使用操作符 `*` 将其地址取出才能继续使用。
2. reference 是变量的别名，**注意**：**“别名”**二字说明它其实就是它指向的变量本身。使用变量本身和使用别名其实都是一样的。

为什么会是这样?

我们之前在学习 C 的时候，老师肯定对我们说过，指针非常危险，一定要小心使用，一不留神就会造成系统崩溃。根据 pointer 的特性来分析，pointer 可以随意指向。单线程的 C 语言如果造成崩溃，可能还比较容易分析出来崩溃点在哪，但是 C++ 程序那么复杂，多线程、多进程，假如因为一个指针造成系统崩溃，那将是一个噩梦。（现在我们的程序开机就上了 67 个线程，一次崩溃，我光找崩溃点找了两天 o(╥﹏╥)o）既然这样，引用就诞生了。

引用其实就是给已经有的变量取个别名，既然是别名，就要求这个变量必须存在，也就是那个变量必须是初始化过的。就像下边这些代码:

```
int &a = 0; // 错误，0 不是变量，本身就没有名字，通过名字压根儿就找不到这玩意儿，咋给人家取别名嘛
int &a; // 错误，取别名，你得有对象才能取别名啊，没对象你取个啥。
int b = 0;
int &a = b; // 正确
```

引用对于指针最大的好处就是资源释放不需要手动管理啦。因为引用和被引用的对象本身就是同一个东西，一损俱损，一荣俱荣。这样对于编程来说难度大大就降低了。

引申

假如我有这样的一个类:

```
class eventLoop {
public:
    double &age() {return age_;}

private:
    double age_;
}

int main() {
    eventLoop eventLoop_;
    auto &ref_age = eventLoop_.age();
    auto age = eventLoop_.age();

    return 0;
}
```

同样的，我们也从汇编语言开始分析:

```
int main() {
00081720  push        ebp
```

```
00081721  mov         ebp,esp
00081723  sub         esp,0F0h
00081729  push        ebx
0008172A  push        esi
0008172B  push        edi
0008172C  lea         edi,[ebp-0F0h]
00081732  mov         ecx,3Ch
00081737  mov         eax,0CCCCCCCCh
0008173C  rep stos    dword ptr es:[edi]
0008173E  mov         eax,dword ptr [__security_cookie (088004h)]
00081743  xor         eax,ebp
00081745  mov         dword ptr [ebp-4],eax
    eventLoop eventLoop_;
00081748  lea         ecx,[eventLoop_]
0008174B  call        eventLoop::eventLoop (08118Bh)
    auto &ref_age = eventLoop_.age();
00081750  lea         ecx,[eventLoop_]
00081753  call        eventLoop::age (08102Dh)
00081758  mov         dword ptr [ref_age],eax
    auto age = eventLoop_.age();
0008175B  lea         ecx,[eventLoop_]
0008175E  call        eventLoop::age (08102Dh)
00081763  movsd       xmm0,mmword ptr [eax] ; movsd 是字符串传送指令
00081767  movsd       mmword ptr [age],xmm0

    return 0;
0008176C  xor         eax,eax
}
```

我们主要看**第 19~21 行**和**第 23~26 行。**我们会发现，函数调用过程都是一样的，但是函数返回之后发生了变化。

我们之前说过，引用只是取别名，并且 `age()` 函数返回的是 `eventLoop_.age_` 这个参数的引用，所以只需要一步就可以取得想要的年龄参数。

但是如果不使用引用来接收返回值呢？我们直接从汇编中可以看出，当函数返回之后，产生了一个临时赋值变量，这个临时赋值变量其实就是 `eventLoop_.age_` 的别名，再取出来这个临时赋值变量中存储的值赋值给 `age` 这个参数。

如果单纯使用第 22 行代码来说，有几个点需要注意：

1. 如果函数返回的对象非常非常大，这样会可能会产生类拷贝，这样非常非常的浪费时间，也非常非常浪费资源。
2. 如果你是想实现一个这样的操作，就是在**对象**中取出来某个成员变量，并且修改这个成员变量的某些属性，就会出现错误。因为你取出来的是**拷贝**的成员变量，不是**对象**中存储的那个对象。所以结果肯定是错的。

题外话

我们都知道 C++ 中有很多跟数字相关的基础类型，比如 int、double、long 等等，考虑一个问题，这么多的类型，假如我要让这些数据进行某种数学运算，那我是不是为了让这些类型全部都支持，程序就应该写成这样：

```cpp
int sum(int a, int b) {return a + b;}
int sum(double a, int b) {return a + b;}
int sum(long a, long b) {return a + b;}
```

那我要是天天都这样写程序，我人要累死了。既然我都不想写这么蠢的代码，C++ 语言的制定者们肯定也不会这样。他们发明了 `template` 类模板，函数模板。写法如下：

```
// 这里写一个函数模板
template <typename T>
T sum(T a, T b) { return a + b; }

int main() {
    int a = 1;
    int b = 2;
    int c = sum(a, b);
}
```

上边的程序中 `sum` 函数就是个函数模板，当在 `main` 函数中使用它时，它会根据语境自动生成对应的函数代码。上边它生成的函数就是 `int sum(int a, int b) { return a + b; }`。终于不需要那么费劲写傻 fu fu 的代码了 (*￣◡￣)。