【博客藏经阁丛书】

C++进阶心法

吕吕 王琥 编著

北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

【博客藏经阁丛书】

C++进阶心法

吕王琥 编著

北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

内 容 简 介

本书共10章，其中，第1章介绍了C、C++的基础知识，包括关键字volatile、数组与指针、编译模式等；第2～9章介绍了C++基础与进阶语法，包括数据类型、引用、名字空间、左值与右值，以及内存管理，面向对象的封装、继承与多态，异常处理，C++ 0x新标准等内容；第10章给出了业界常见的编码规范与建议。本书不仅介绍了C++的传统语法，而且还融入了C++最新的变革内容，旨在帮助读者对C++有一个更加全面的了解，快速掌握C++编程技巧，并将其应用于工程实践中。

本书既可作为C++编程人员以及相关专业技术人员的参考用书，也可作为高等院校、高职高专院校程序设计相关课程的教学用书。

图书在版编目(CIP)数据

C++进阶心法 / 吕吕，王琥编著. -- 北京 ：北京航空航天大学出版社，2018.12

ISBN 978-7-5124-2240-7

Ⅰ. ①C… Ⅱ. ①吕… ②王… Ⅲ. ①C++语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第265079号

C++进阶心法

吕 吕 王 琥 编著

责任编辑 孙兴芳

*

北京航空航天大学出版社出版发行

北京市海淀区学院路37号(邮编100191) http://www.buaapress.com.cn

发行部电话：(010)82317024 传真：(010)82328026

读者信箱：copyrights@buaacm.com.cn 邮购电话：(010)82316936

艺堂印刷（天津）有限公司印装 各地书店经销

*

开本：710×1 000 1/16 印张：32.75 字数：698千字

2019年4月第1版 2019年4月第1次印刷 印数：2 000册

ISBN 978-7-5124-2240-7 定价：89.00元

前　言

C++既是一门特性丰富、应用广泛、富有挑战、值得深入学习的面向对象的编程语言，也是计算机相关专业必学的基础课程之一。C++以C语言为基础，继承了C语言高效、跨平台的优良特性，同时又做出了极大扩展，引入了面向对象、模板泛型、函数式编程、模板元编程等高级特性，使之成为一门与时俱进的现代型高级编程语言，能够应对各种复杂的应用场景，例如操作系统、高并发服务框架与后台、桌面应用、移动开发、嵌入式开发等。当然，正因为C++具有诸多复杂的功能特性，从而增加了其学习成本。

C语言是C++的基础，是C++的子集，因此C++中的很多知识点都可归于C语言，在学习C++的同时，也是在学习C语言。本书开始介绍了部分C语言的基础内容，用于辅佐C++的学习。比如不太常见的关键字volatile、重要而易出错的野指针、基础的分离编译模式等，这些都是C++中最为基础的知识，每一名合格的程序员都应该掌握。

本书主体内容是围绕C++编程语法展开的，对C++知识点的讲解深度会略高于基础教材，因此初学者在阅读本书时要有耐心，并需结合文中代码示例好好揣摩思考。对于有疑问的知识点，一定要动手实践，将自己的思考和疑问通过代码的形式表达出来，只有这样，才有助于加深对C++晦涩知识点的理解。除了C++的基础内容外，本书还涉及了C++ 0x新标准提出的常用特性，让读者在学习传统C++的同时，对C++有一个与时俱进的了解。比如，C++ 11中的关键字auto、就地初始化与列表初始化、Lambda表达式等都是值得读者去学习和掌握的，并可将其应用于工程实践中。

掌握编程语言的语法知识只能保证编写的代码能够编译运行，但是，一名成熟的C++开发人员心中必须有一把标尺，这把标尺就是编码规范。初具规模的项目代码应不仅能编译和运行，而且能长久地迭代变更、维护交接。所以，为了能够编写出整洁、规范、优雅的代码，我们应该遵循必要的编码规范和风格，力争让自己写出的代码不被他人诟病。本书在参考了《Google C++编程风格指南》并结合个人经验的基础

上，给出了一些规范和建议，比如命名方式、头文件使用规范与包含顺序、编码格式等。当然，这些只是一家之言，仅供参考。

本书记录的关于C++的点滴，实则是自己和身边一同求学的小伙伴对C++的学习认知过程，在这里分享给每一位C++从业者，希望能够用自己的绵薄之力帮助到需要帮助的人。我相信，只要读者潜心钻研，多读多练，就肯定能从本书中学有所得。当然，由于个人水平有限，书中难免存在不足甚至错误的地方，欢迎大家在CSDN博客留言指正，共同探讨。联系方式见笔者博客主页：https://blog.csdn.net/K346K346。

本书从编写到出版得到了北京航空航天大学出版社的大力支持，在此深表感谢。另外，还要感谢本书的另一位作者——我的大学舍友王琥，他参与了本书的编写工作；感谢身边的同学和同事在工作和生活上给予的无私帮助。最后，要感谢爱人cat在身后的默默支持与理解，以及家人的辛劳付出和母亲对我的人生教诲。

学习的道路并不寂寞，因为有知识相伴；学习的道路也不会平坦，但逆风的地方更适合飞翔。

吕　吕

2018年12月于寿县君子镇

目 录

第1章

C++中的C

1.1 认识 volatile

volatile 是“易变的”“不稳定的”意思，是 C 语言中一个较为少用的关键字，它用来解决变量在“共享”环境下容易出现读取错误的问题。

1. volatile 的作用

若变量被定义为 volatile，则该变量可能会被意想不到地改变，即在程序运行过程中一直变，若希望这个值被正确处理，则每次都要从内存中读取该值，而不会因编译器优化从缓存的地方读取（比如读取缓存在寄存器中的数值），从而保证 volatile 变量被正确读取。

在单任务环境中，如果一个函数体内部在两次读取变量值之间的语句没有对变量值进行修改，那么编译器就会设法对可执行代码进行优化。由于访问寄存器的速度要比 RAM（从 RAM 中读取变量的值到寄存器中）快，所以以后只要变量的值没有改变，就一直从寄存器中读取变量的值，而不对 RAM 进行访问。

而在多任务环境中，虽然一个函数体内部在两次读取变量值之间没有对变量的值进行修改，但是该变量仍然有可能被其他的程序（如中断程序、另外的线程等）修改，如果这时还是从寄存器而不是从 RAM 中读取，就会出现被修改的变量值不能及时得到反映的问题。下面的程序就对这一现象进行了模拟。

```
#include <iostream>
using namespace std;
int main(int argc,char * argv[])
{
    int i = 10;
    int a = i;
    cout << a << endl;
    _asm
    {
        mov dword ptr [ebp - 4],80
    }
```

```
    int b = i;
    cout << b << endl;
    getchar();
}
```

程序在 VS 2017 环境下生成 Release 版本，输出结果为

```
10
10
```

阅读以上程序时需注意以下几个要点：

① 以上代码必须在 Release 模式下考察，因为只有在 Release 模式下才会对程序代码进行优化，而这种优化在变量共享的环境下容易引发问题。

② 在语句“b=i;”之前，已经通过内联汇编代码修改了 i 的值，但是 i 的变化却没有反映到 b 中，如果 i 是一个被多个任务共享的变量，那么这种优化所带来的错误很可能是致命的。

③ 汇编代码“[ebp - 4]”表示变量 i 的存储单元，因为 ebp 是扩展基址指针寄存器，存放函数所属栈的栈底地址，所以先入栈，占用 4 字节。随着函数内声明的局部变量增多，esp(栈顶指针寄存器)就会相应减小，因为栈的生长方向由高地址向低地址生长。i 为第一个变量，栈空间已被 ebp 入栈占用了 4 字节，所以 i 的地址为 ebp－i，而[ebp－i]表示变量 i 的存储单元。

那么如何抑制编译器对读取变量的这种优化，以防止错误读取呢？volatile 可以轻松解决这个问题。将上面的程序稍作修改，把变量 i 声明为 volatile 即可，观察如下程序：

```
#include <iostream>
using namespace std;
int main(int argc,char * argv[])
{
    volatile int i = 10;
    int a = i;
    cout << a << endl;
    _asm
    {
        mov dword ptr [ebp - 4],80
    }
    int b = i;
    cout << b << endl;
}
```

程序输出结果为

```
10
80
```

当第二次读取变量 i 的值时，就已经获得了变化之后的值。跟踪汇编代码可知，凡是声明为 volatile 的变量，每次都是从内存中读取变量的值，而不是在某些情况下直接从寄存器中取值。

2. volatile 的应用场景

(1) 并行设备的硬件寄存器(如状态寄存器)

假设要对一个设备进行初始化，此设备的某一个寄存器为 0xff800000。观察如下程序：

```
int   * output = (unsigned   int * )0xff800000;   //定义一个 I/O 端口;
int init()
{
    int i;
    for(i = 0;i < 10;i ++ )
    {
        * output = i;
    }
}
```

编译器经过优化后认为前面的循环都是无用的，对最后的结果毫无影响，因为最终只是将 output 这个指针赋值为 9，所以最后编译器编译的代码结果相当于：

```
int init(void)
{
    * output = 9;
}
```

如果对此外部设备进行初始化的过程必须像上述代码一样顺序地对其赋值，则优化过程显然不能达到目的；反之，如果不是对此端口进行反复写操作，而是进行反复读操作，则其结果是一样的，编译器在优化后，也许代码对此地址的读操作只做了一次。然而，从代码的角度来看是没有任何问题的。这时就应使用 volatile 通知编译器这个变量是不稳定的，在遇到此变量时不要优化。

(2) 一个中断服务子程序中会访问到的变量

观察如下程序：

```
static int i = 0;
int main()
{
    while(1)
    {
        if(i) dosomething();
    }
}
/ * Interrupt service routine * /
```

```
void IRS()
{
    i = 1;
}
```

上面示例程序的本意是产生中断时，由中断服务子程序 IRS 响应中断，变更程序变量 i，在 main 函数中调用 dosomething 函数。但是，由于编译器判断在 main 函数中没有修改过 i，因此可能只执行一次从 i 到某寄存器的读操作，然后每次 if 判断都只使用这个寄存器中的“i 副本”，导致 dosomething()函数永远不会被调用。如果在变量 i 前加上 volatile 修饰，编译器就会保证对变量 i 的读/写操作都不会被优化，从而保证变量 i 被外部程序更改后能及时在原程序中得到感知。

(3) 多线程应用中被几个任务共享的变量

当多个线程共享某一个变量，且该变量的值会被某一个线程更改时，应用 volatile 声明。其作用是：防止编译器优化把变量从内存装入 CPU 寄存器中，当一个线程更改变量后，未及时同步到其他线程中而导致程序出错。volatile 的意思是让编译器每次操作该变量时都一定要从内存中真正读取，而不是使用已经存在于寄存器中的值。示例如下：

```
volatile  bool bStop = false;  //bStop 为共享全局变量
//第一个线程
void threadFunc1()
{
    while(!bStop){...}
}
//第二个线程终止上面的线程循环
void threadFunc2()
{
    bStop = true;
}
```

如果 bStop 不使用 volatile 定义，那么上述循环将是一个死循环，因为 bStop 已经被读到了寄存器中，寄存器中 bStop 的值永远不会变成 FALSE；而用 volatile 定义后，程序在执行时每次均从内存中读取 bStop 的值，这样就不会造成死循环了。

是否了解 volatile 的应用场景是区分 C/C++程序员和嵌入式开发程序员的有效办法，进行嵌入式开发的伙伴们经常同硬件、中断、RTOS 等打交道，这些都要求用到 volatile 变量，不懂得 volatile 将会给程序设计带来灾难。

3. volatile 的常见问题

下面的问题可以测试面试者是不是真正了解 volatile。

① 一个参数既可以是 const 也可以是 volatile 吗？为什么？

② 一个指针可以是 volatile 吗？为什么？

③ 下面的函数有什么错误？

```
int square(volatile int * ptr)
{
    return * ptr *  * ptr;
}
```

相对应的答案如下：

① 是的。例如只读的状态寄存器，它是 volatile，因为它可能会被意想不到地改变；它是 const，因为程序不应该试图去修改它。

② 是的，尽管这并不很常见。例如，当一个中断服务子程序修改一个指向一个 buffer 的指针时。

③ 上述代码有点变态，其目的是用来返回指针 * ptr 指向值的平方，但是，由于 * ptr 指向一个 volatile 型参数，所以编译器将产生类似下面的代码：

```
int square(volatile int * ptr)
{
    int a,b;
    a = * ptr;
    b = * ptr;
    return a * b;
}
```

由于 * ptr 的值可能被意想不到地改变，因此 a 和 b 可能是不同的，导致这段代码返回的可能不是所期望的平方值。正确的代码如下：

```
long square(volatile int * ptr)
{
    int a;
    a = * ptr;
    return a * a;
}
```

4. 嵌入式编程中 volatile 的作用

嵌入式编程中经常用到 volatile 这个关键字，常见用法可以归纳为以下两点：

① 告诉 compiler 不能做任何优化。比如要往某一地址送两个指令：

```
int * ip = ...;          //设备地址
 * ip = 1;               //第一个指令
 * ip = 2;               //第二个指令
```

对于上述程序，compiler 可能优化为

```
int * ip = ...;
* ip = 2;
```

结果第一个指令丢失。如果用 volatile，则 compiler 不允许做任何优化，从而保证程序的原意：

```
volatile int * ip = ...;
* ip = 1;
* ip = 2;
```

② 表示用 volatile 定义的变量会在程序外被改变，每次都必须从内存中读取，而不能把其放在 cache 或寄存器中重复使用。例如：

```
volatile char a;
a = 0;
while(!a)
{
    //do some things;
}
doother();
```

如果没有 volatile，doother()就不会被执行。volatile 能够避免编译器优化带来的错误，但使用 volatile 的同时还需要注意，频繁地使用 volatile 很可能会增加代码尺寸和降低性能，因此要合理地使用 volatile。

1.2 数组与指针详解

1.2.1 数 组

数组的大小(元素个数)一般在编译时决定，也有少部分编译器可以在运行时动态地决定数组的大小，比如 icpc(Intel C++编译器)。

1. 数组名的意义

数组名的本质是一个文字常量，代表数组第一个元素的地址和数组的首地址。数组名本身不是一个变量，不可以寻址，且不允许为数组名赋值。假设定义数组：

```
int A[10];
```

再定义一个引用：

```
int * &r = A;
```

这是错误的写法，因为变量 A 是一个文字常量，不可寻址。如果要建立数组 A 的引用，则应该定义如下：

```
int * const &r = A;
```

此时，在数据区开辟一个无名临时变量，将数组 A 代表的地址常量复制到该变量中，再将常引用 r 与此变量进行绑定。此外，若定义一个数组 A，则 A、&A[0]、A+0 是等价的。

在 sizeof()运算中，数组名代表的是全体数组元素，而不是某个元素。例如，定义 int A[5]，生成 Win 32 的程序，则“sizeof(A)”就等于“5\ * sizeof(int)=5\ * 4=20”。示例程序如下：

```
#include <iostream>
using namespace std;
int main()
{
    int A[4] = {1,2,3,4};
    int B[4] = {5,6,7,8};
    int (&rA)[4] = A;      //建立数组 A 的引用
    cout << "A:" << A << endl;
    cout << "&A:" << &A << endl;
    cout << "A+1:" << A+1 << endl;
    cout << "&A+1:" << &A+1 << endl;
    cout << "B:" << B << endl;
    cout << "rA:" << rA << endl;
    cout << "&rA:" << &rA << endl;
}
```

运行结果：

```
A:0013F76C
&A:0013F76C
A+1:0013F770
&A+1:0013F77C
B:0013F754
rA:0013F76C
&rA:0013F76C
```

阅读以上程序时需注意如下几点：

① A 与 &A 的结果在数值上是一样的，但是 A 与 &A 的数据类型不同。A 的类型是 int[4]，而 &A 的类型是 int(*)[4]，它们在概念上是不一样的，这就直接导致 A+1 与 &A+1 的结果完全不一样。

② 为变量建立引用的语法格式是 type& ref，因为数组 A 的类型是 int[4]，因此为 A 建立引用的语法格式是“int (&rA)[4]=A;”。