文章目录

> 时隔一年, 重读C++ Primer这本圣经, 怀念去年这时基友们一起debug, 一起吃饭, 一起睡觉, 一起分享知识的那个夏天, 以这篇文章纪念我的好朋友, 希望有机会我们再聚在一起吃酒撸串夜灯下诉过去与未来, 同时以我微薄的知识向C++之父敬礼.
> 本文不会罗列C++基础语法, 只说明需要注意的地方, 所有需要注意的地方均为作者主观观点.

如有任何错误之处, 欢迎斧正.

编程风格

我认为学习一种语言, 一定要学习一种权威的编程风格指南, 就如python中的PEP8.
使用约定的风格, 可以减少协同工作的障碍, 建立程序猿之间的代码友谊. 虽然朋友说要对别人的代码宽容, 而我认为这就像纵容别人犯罪一样, 遵守一定的代码风格, 能瞬间拉近程序猿之间的距离, 增加代码可读性何乐而不为呢? 我喜欢看开源代码的原因之一就是很多著名的开源项目优雅的编程风格
简要的罗列一下建议遵守的编程风格(谷歌风格):

- 所有头文件都应该使用 #define 防止头文件被多重包含, 命名格式当是: _

- 当只有数据时使用 struct, 其它一概使用class
- 在类中使用特定的声明顺序: public: 在 private: 之前, 成员函数在数据成员 (变量) 前
- 整数用0, 实数用0.0, 指针用NULL, 字符(串)用 '\0'.
- 类, 结构体, 类型定义 (typedef), 枚举的每个单词首字母均大写, 不包含下划线(大驼峰命名)
- 变量名一律小写, 单词之间用下划线连接. 类的成员变量以下划线结尾
- 常量和枚举类型命名在名称前加k: kDaysInAWeek
- 常规函数使用大小写混合(大驼峰命名), get和set函数则要求与变量名匹配(set函数前加set前缀)
- 逗号后添加空格
- 不建议使用using namesapce std

更多细节参考Google 开源项目风格指南

基础

编译与执行:

5. 预处理阶段: 根据字符#开头的命令, 修改原始C程序
6. 编译阶段: 将文本文件翻译成汇编程序
7. 汇编阶段: 汇编器将编译程序翻译成机器语言指令(机器可识别), 并打包成可重定位目标程序
8. 链接阶段: 将调用函数目标文件合并到程序中, 形成可执行目标文件

1	//使用GUN编译器g++, -o选项将输入写入文件, 即用来存放可执行文件

2	$ g++ chapter1.cc -o chapter

变量:

变量的作用我认为有以下几点:

- 给一定大小的内存命名, 方便使用和增加可读性
- 通过变量的类型, 来决定到底访问几个字节长的内存
- 通过变量的来决定如何解释所读取的内存的数据(如有符号数和无符号数的解释不同)

左值与右值:

- 左值可以出现在赋值语句的左边
- 右值只能出现在赋值的右边, 不能出现在赋值语句的左边

const关键字:

6. 通过指定const变量为extern, 可以在整个程序中访问const对象
7. const引用是指向const对象的引用, 对象可读不可写
8. 指向const对象的指针, 定义时不需要初始化, 可以对指针重新赋值(修改其中保存的内存地址, 指向其他对象), 但所指向对象中的值不能修改(内存中保存的值不能修改)
9. const指针, const指针的值(保存的内存地址的值不能修改)不能修改, 也就是不能使const指针指向其他对象
10. 指向const对象的const指针, 既不能修改指针所指向的对象值(内存地址中保存的值), 也不能修改指针的指向(指针中保存的内存地址)


```
1  // file1.cc

2  extern const int buf_size =

3  10;
```

4	// file2.cc
5	extern const
6	int buf_size; //使用file1.cc中的buf_size

7		
8	//指向const对象的指针 const	
9	double *cptr;	

```
10
//const指针
        int err_numb = 2;
        int *
const
```

	cur_err = &err_numb;

预处理器:

- 使用预处理器变量避免头文件的多重包含

常用格式:

1	#ifndef _XXX_H_
2	#define _XXX_H_

```
3
        ...
4
#endif /* _XXX_H_ */
5
```

6	

指针:

- 指针保存的是对象的地址
- 数组名会自动转换为指向数组第一个元素的指针.
- 数组的下标访问数组时实际上是使用下标访问指针, 指针是数组的迭代器
- 当类中有指针数据成员, 不能使用系统自带的拷贝构造函数(系统默认导致浅拷贝, 两个对象指针指向同一块动态分配的内存)/赋值函数(=操作符重载), 请自定义(深拷贝)

自增/自减操作符

- 前自增操作加1后返回加1的结果
- 后自增操作保存操作数原来的值, 返回未加1之前的值作为操作的结构
- 自减操作符类似与自增操作符

sizeof操作符

注意sizeof是操作符, 用于获得类型的长度

- 对数组做sizeof操作等效于将对其元素类型做sizeof操作的结果乘上数组元素的个数.
- 对指针做sizeof将返回存放指针所需的内存大小
- 对引用做sizeof将返回存放此引用类型对象所需所需的内存大小

switch语句执行匹配的case标号相关联的语句后, 会跨越case边界继续执行其他语句, 直到switch结束或者遇到break.

复制传参(pass by value)和引用/指针传参(pass by reference):

- 复制传参无法改变传入实参的值
- 复制传参增加了时间和存储空间的开销, 尽量减少pass by value
- 引用传参相当于传指针
- 返回值也要尽量传递引用(不要返回局部变量的引用)

static对象:

类内staitc数据成员不属于某个对象(类内声明, 类外定义). 类内static函数没有this指针(用于处理staitc数据成员)

static成员必须在class定义式之外被定义(除非他们是const并且是整型)

```
1  class Test {
2  public:
3  Test(
```

```
                const
        int
p):
price(p) {
4       std::
        cout
<<
"construc
tor Test."
<<
std::endl;
5   }
        void
6   setRate(
```

```
    const
    double r)
    {
7
        this->rate = r;
    }
8
    double getRate()
    {
    return rate; };
9
```

```
10  private:

11  static double rate;

12  //declare
```

```
        int price;
13
        };
14
double Test::rate =
15
        0.53;
        //define
```

```
16
                int
main(
17              int
argc,
                char
*argv[]) {
18
Test t(
                10);
```

```
19
Test t1(
20);
std::
cout
<<
t.getRate(
) <<
std::endl;
```

```
std::
        cout
<<
t1.getRat
e() <<
std::endl;
t.setRate(
2.22);
```

	std:: cout << t.getRate() << std::endl; }

一旦创建static对象被创建, 在程序结束前不会被销毁. 常用于生命周期跨越多个函数调用的对象

inline函数:

- 普通函数被调用: 调用前保存寄存器, 返回时恢复上下文, 复制实参, 程序还必须转向一个新的位置执行
- inline函数在编译时被展开, 从而消除额外的函数执行开销, 常用于小操作函数.内联函数要在头文件中定义

重载函数:

3. 确认候选函数(C++名字查找发生在类型检查之前)
4. 检查形参个数和形参类型匹配问题

函数指针:

重点理解: 函数指针是指向函数的指针

直接使用函数名等效于在函数名上取地址操作符

1	// pf是一个指针, *表明了pf的指针身份, pf的类型为bool (const string &, const string &)
2	bool (*pf)(

3	const string &, const string &);
4	
5	//类比与普通变量指针, *表示ps是指针, ps的类型为const string
6	

```
const
string *ps;
//typedef简化函数指针定义cmpFun等价于bool *(const string &, const string &)
```

```
10  typedef bool (*cmpFun)(const string &, const string &);
```

cmpFun
pf;

//typedef
简化普通
变量指针
pstr等价
于 const
string *

typedef

const

	string *p_str; pstr ps;

Stack和Heap

Stack是存在于某作用域的一块内存空间

Heap是由操作系统提供的一块全局内存空间(可动态分配获得此类空间)

namespace

标准库中所有文件被包裹在std命名空间中.

命名空间可以是不连续的

1	namespa

	ce myname {
	
	2
	
	…
	
	3
	
	} // 不以分号结束

异常

异常通过throw抛出对象引发的.异常可以传递给给非引用形参任意类型的对象

4. 通过栈展开(stack unwinding), 沿嵌套函数调用链继续向上, 直至为异常找到一个用于处理异常的catch语句
5. 捕获所有异常的catch子句形式为(. . .)
6. exception类型所定义的唯一操作是what虚函数

类

类定义了一个新的类型和新的作用域, 切记类定义以分号结束

struct和class的唯一差别在于默认访问级别上, struct的成员默认为public, class成员默认为private

- 类静态成员static: 静态数据成员被类的所有对象所共享, 包括该类派生类的对象, 也就是说, 静态数据成员属于类, 而不属于某个对象. static成员函数没有this指针, 不能被声明为虚函数.
- 隐式形参this: 类中每个成员函数都有一个额外的、隐含的形参(this, const成员函数时, this的类型为const class_type *const this)将该成员函数与调用该函数的类对象绑定在一起. 形参this初始化为调用函数的对象的地址
- 常量成员函数: 在成员函数形参表后声明const, 用来表明隐式形参this的类型为const class_type *
- 构造函数: 与类同名且没有返回值, 一个类可以有多个构造函数, 构造函数不能声明为const. 注意:其中的构造函数的初始化列表有顺序
- const成员函数: 对于一个不会改变数据的函数, 果断加上const关键字, 表示该函数不允许修改类的数据成员
- 虚函数: 虚函数希望派生类对此函数进行override, 纯虚函数要求派生类必须override这个函数. 注意: 通过基类的引用或指针调用虚函数才能引发动态绑定

理解初始化列表, 初始化列表初始化数据成员(注意const对象或引用类型只能初始化不能赋值), 没有初始化列表的的构造函数在函数体中对数据成员赋值. 成员被初始化的顺序就是定义成员的顺序

1	// 链式编程

```
2
    Screen &
    Screen::m
    ove(
            char
    c) {
3
    contents[
    cursor] =
    c;
4
```

5	return *this;
	//this是一个指针, 解引用后是一个类类型Screen
	}

构造函数和析构函数的顺序

直接上代码会比较清晰


```
1  class Foo {
2  public:
3      Foo() { std::
4          cout << "Foo default constructor." <<
```

```
    std::endl;
}

5

Foo(

const Foo &foo) {
std::

6       cout <<

        "Foo copy constructor." << std::endl;
}

7
```

```
~Foo() {
std::
8
        cout
<<
        "Foo
deconstr
uctor."
<<
std::endl;}
9
        };
10
```

```
        class Bar {

11

        public:

12

        Bar() {
        std::

13

                cout << 
```

```
        "Bar default constructor." << std::endl;
}
14
Bar(
15
const Bar &bar) {
std::
        cout <<
16
        "Bar copy constructor." << std::endl;
}
```

```
17
~Bar() {
std::
        cout
<<
18
        "Bar
deconstr
uctor."
<<
std::endl;}
19
        };
```

```
20
        class Yes {
21
public:
22
Yes() {
std::
```

```
        cout <<
23
        "Yes default construct
        or." << std::endl;
}
24
~Yes() {
std::
        cout <<
25
        "Yes deconstr uctor." <<
std::endl;
}
```

```
26
};
27
class Base {
28
public:
29
```

```
Base() {
std::
30      cout
<<
"Base
construct
or." <<
std::endl;
}
31
~Base() {
std::
32      cout
```

```
                << "Base deconstructor." << std::endl;
        }
33
    private:
34
        Foo foo_;
35
```

```
};
36
class Derived : public Base {
37
38
```

```
public:

39

Derived()
{ std::

      cout
<<

40

"Derived
construct
or." <<
std::endl;
}

41

Derived(
```

```
Derived(
    const Bar &bar,
42
    const Yes &yes);
43
Derived(
    const Yes &yes,
44
    const Bar &bar);
```

```
45
~Derived(
) { std::
        cout
<<
46
"Derived
deconstr
uctor."
<<
std::endl;}
47
private:
```

```
48
    Bar bar_;
49
    Yes yes_;
50
};
```

```
51
                Derived::
                Derived(
52              const Bar
                &bar,
                const Yes
                &yes) {
53
                std::
                    cout
                <<
```

```
54
                "Derived argument: (bar, yes) construct
                or." << std::endl;
55
        }
56
        Derived::Derived(
57
            const Yes
```

```
                &yes,

                const Bar
                &bar) {
58

                std::

                      cout
                <<
59

                "Derived
                argument
                (yes, bar)
                construct
                or." <<
                std::endl;

60
```

```
}

61

int
main(

int
argc,

char
*argv[]) {
```

```
std::
      cout
<<
"create
simple
obejct
foo and
bar." <<
std::endl;
Foo foo;
```

```
Bar bar;

Yes yes;

std::

      cout
<<

"create
Base class
" <<
std::endl;
```

```
Base base;
std::cout << "case 1 : (default constructor) " << std::endl;
```

```
Derived derived1;
std::cout << "case 2 : (argument bar, yes)" << std::endl;
```

```
Derived derived2(bar, yes);
std::cout << "case 3 : (argument yes, bar)" <<
```

	std::endl; Derived derived3(yes, bar); }

运行结果:

1	// 创建三个简单的对象
2	create simple obejct foo
3	and bar.
4	Foo

5	default constructor.	Bar
6	default constructor.	
7		

	Yes
	default
8	constructor.
9	// 创建基类对象
10	
	create

Base class Foo default constructor. //先对成员变量初始化

	Base	
14		
	constructor. // 调用基类构造函数	
15		
	//	
	Derived调用默认构造函数	
16		

	case 1 : (
17	
	default
	constructor)
18	
	Foo
	default
19	
	constructor. //先调用基类构造函数

```
20
        Base
constructor.
21
        Bar
22
default
constructor. //自身
```

23	成员变量初始化
24	Yes default
25	construct or. Derived

```
26  constructor.  //调用默认构造函数
27
        // 调用以
        bar,
        yes
28  为参数的构造函数
```

```
29          case 2 : (argument bar, yes)
30          Foo
31          default constructor.
32          Base
```

33	constructor. Bar default
34	constructor. //注意 bar, yes 构造的顺序
35	

36	Yes default constructor.	
37	Derived	
38	argument: (bar, yes)	

```
                constructor.
39
                //调用以
40              yes,
                bar为参数的构造函数
41
                case 3 :
```

```
                (argument yes, bar)
42
                    Foo
                default
43
                constructor.
44
                    Base
```

45	constructor. Bar
46	default constructor. //注意
47	bar, yes 构造的顺序

48	Yes
	default
49	constructor.
50	Derived
	argument
	(yes, bar)

51 constructor.

52 //析构过程

(构造过程的逆)

53 Derived

```
54  deconstructor. //

    case3的析构

55

        Yes

    deconstructor.

56

        Bar

57
```

```
                deconstr
                uctor.
58
                    Base
                deconstr
                uctor.
59
                    Foo
60              deconstr
                uctor.
```

61

Derived

deconstr
uctor. //

62 case2的析
构

Yes

deconstructor.

Bar

deconstructor.

Base

deconstructor.

	Foo deconstr uctor.	
	Derived deconstr uctor. //	
	case1的析	

	构 Yes deconstructor. Bar deconstructor.	

Base

deconstr
uctor.

Foo

deconstr
uctor.

	// Base 的析构	
	Base deconstructor.	

Foo

deconstructor.

//三个简单对象的析构

Yes

deconstructor.

	Bar deconstructor. Foo deconstructor.

- 类先初始化成员变量, 然后执行构造函数内部逻辑

- 成员对象初始化的次序完全不受它们在初始化表中次序的影响, 只由成员对象在类中声明的次序决定
- 析构函数的调用顺序和构造函数调用顺序相反

参考浅出C++对象模型——理解构造函数、析构函数执行顺序

复制控制

具有指针成员的类一般需要定义自己的复制控制(防止浅拷贝), 复制构造或赋值操作符应该显式使用基类的复制构造或赋值操作符

- 复制构造函数: 形参通常为const引用, 一般不设置为explicit. 禁用复制需要将复制构造函数声明为private. (含有指针数据成员时, 避免浅拷贝应该定义复制构造函数)
- 赋值操作符: 类内赋值操作符重载, 包含隐式this形参, 右操作数一般为const引用, 返回值一般应该为引用.
- 析构函数: 资源回收, 尤其是指针所指向的动态分配的内存, 析构函数不可重载

1	拷贝赋值函数(赋值操作符重载):
2	1. 检测值是否

3	自我赋值(self assignment, 地址比较 this == &object)
4	2. 左值内存 delete清空
5	

6	3. 分配与右值相同大小内存 4. 赋值到左值

操作符重载原则

- 赋值(返回对*this的引用), 下标, 调用([])和成员访问箭头必须定义为成员函数
- 符合操作符通常定义为成员函数
- 自增, 自减, 解引用操作符通常定义为类成员函数(改变对象状态或与类型紧密相关)
- 算术, 相等, 关系, 位, 流操作符一般定义为普通非成员函数(设置为友元)

对象的创建

C++提供两种方法分配和释放未构造的原始内存:

3. allocator类, 提供可感知类型的内存分配
4. 标准库中的operator new和operator delete分配和释放需要的大小的原始的、未类型化的内存

1	1. ClassName object(para);
2	
3	// 使用new(先分配内存 malloc, 指针类型转换 然后调用构造函数)后, 需要 delete(先调用析构函数, 再释放内存 free)掉分配的堆内存, 防止内存泄漏
4	

5	// 该表达式调用名为operator new的标准库函数分配足够大的原始的未类型化的内存
6	2. ClassName *object = new ClassName(param);
7	

	// 当使用delete表示删除动态分配内存时, 首先对object指向的对
8	象析构, 然后调用operator delete的标准库函数释放该对象所用的内存, operator delete不会调用析构函数
9	
	3.
10	delete object;

11	//placement new 不分配内存, 而是使用已分配但未构造内存的初始化一个对象(接受一个指针)	4. new (place_address) type // place_address为指

	针

关于C++内存分配的new, operator new, placement new

- C++创建对象时仅分配用于保存数据成员的堆空间, 成员函数没有单独的空间
- C++用new创建对象时返回一个对象指针, object指向一个ClassName的对象, C++分配给object的仅仅是存放指针值的空间, 并且用new 动态创建的对象必须用delete来撤销该对象(只有delete对象才会调用其析构函数)
- 相同class的各个对象互为友元(friend)

类模版

泛型编程是以独立于任何特定类型的方式编写代码

模板形参可以是类型形参, 也可以是非类型形参

模板本身并不是一种类型, 当编译器看到模板定义的时候, 不立即产生代码, 只有在看到用到模板时, 编译器才会对模板进行实例化

- 在类本身的作用域内, 可以使用类模板的非限定名
- 类外定义的类模板成员函数, 必须以关键字template开头, 后接类的模板形参表, 必须指出是那个类的成员并包含模板形参
- 类可以拥有本身为类末班或者函数模板的成员(成员模板). 另外, 此类成员模板定义在类外部时, 需要包含两个模板的形参表
- 模板特化, 我认为是对一些特殊模板形参进行实现(比如模板形参中包含指针的时候)

1	template <

2	class Type>	
3	class Queue {	
4	public:	

	Queue() {};
5	// 可以使用 Queue ，由编译器推断
6	
	private:
7	
	QueueItem

```
8       *head;
            //非类作用域空间必须显式使用模板形参
9
            void destory();
10
        };
```

```
11
template
<
        class
Type>
12
        void
Queue
13
::destory()
{
```

```
        //
somethin
g
}
```

STL

输出/输出流:

- std::cout, 结果是左操作数的值, 输出操作返回的是输出流本身
- std::cin, 类似于`std::cout`, 输入操作符返回器做操作数作为结果

顺序容器

顺序容器：将单一类型元素聚集起来成为容器, 然后根据位置来存储和访问这些元素. 顺序容器包括`vector`, `list`, `deque`, 顺序容器的适配器`stack`(基于`deque`), `queue`(基于`deque`), `priority_queue`(基于`vector`)

vector对象动态增长:

vector的元素连续存储. 其中size()函数统计vector已有元素个数, capacity()指在vector必须重新分配存储空间之前可存储的元素个数.可见vector分配存储空间的策略是增幅小于1时取1，之后每达到2的次方时倍增(变为原来容量的2倍)

```
1	// 测试程序
2	void TestIncrease(std::vector<std::string>&vec) {
3
```

```
std::
    cout << "size: " << vec.size() << std::endl;

std::
    cout <<
```

```
"capacity:
" <<
vec.capac
ity() <<
std::endl;
7
}
8
void
TestVecto
r() {
9
```

```
std::
10
vector<std::string> vec;
11
TestIncrease(vec);
12
for(std::
```

```
13  vector<std::string>::size_type ix = 0;
14
    ix != 24; ++ix) {
15
```

```
        vec.push_back(
16
        "hello");
17
        TestIncrease(vec);
18
    }
```

```
19
        }
20
        // 测试结果, 只保留重要部分
21
        size:
        0
```

```
22
                capacity:
23              0
24              size:
                1
25
```

```
                capacity:
                    1
26
                size:
                    2
27
28              capacity:
```

	2
29	
	...
30	
	size:
	8
31	

	capacity:
32	8
33	size:
	9
34	
	capacity:

	16	
35		
	...	
	size:	
	16	

capacity:

16

size:

17

capacity:

32

...

size:

24

	capacity: 32

const_iterator和const的iterator:

迭代器可以理解为指针.

- const_iterator创建的对象，自身的值可以改变，但不能改变其指向的元素的值，对对象解引用返回的是一个const值
- 声明const的迭代器时, 必须初始化, 初始化后, 迭代器自身不可再变, 迭代所指向的元素值可以改变(这里const修饰的是迭代器)

关联容器

关联容器和顺序容器的本质区别: 关联容器通过key存储和读取元素, 顺序容器通过位置存储和访问容器

- map通过key的小于关系排序, 自定义数据结构应该重载<操作符. map迭代器解引用产生pair对象

关联容器map，set

参考链接及书目

- 第四版
- What is the use of “using namespace std”?
- C/C++ 预处理器参考
- Super fast C++ logging library
- C++预处理和预定义
- 预处理指令 (Preprocessor Directives)