SUNDAY, JULY 6, 2014 AT 10:48

周末总结

过去的一周时间，真正做事的时间很少。一直再看美剧“Person of Interest”，这是一部不错的美剧，影射的是美国棱镜门事件。在这其中又发现了程序员的背影，they call programmer as code monkey!
这一周所谓没有真正做事情，是指自我学习的安排不是那么成功。
首先就是时间安排，以前是中午2点起床，然后3点半回实验室，但是这样就发现考勤，项目等等存在一些冲突和问题。
所以时间安排需要重新调整：
7:30 Get Up
|
8:30 Go and Work
|
11:10 Lunch
|
12:00 Siesta
…
|
13:15 Get Up
|
14:30 Go and Work
|
17:10 Dinner
|
18:00 Break
|
19:00 Work
|
21:30 Go Back & Trivia
|
22:30 Self-Study
|
00:00 Sleep

杭州西湖风景名胜区, 杭州市, 浙江省, 中国 • 26° MOSTLY CLOUDY

WEDNESDAY, JULY 23, 2014 AT 13:20

about the c++ coding summary

this time when I was previewing the cpp programming, I learned to use the point-to-function so that the other function could use the function name as a parameter, for instance:

```
void dis(double );//prototype dis
void dis2(double);//prototype dis2
double calculate(double,void
dis(double));//calling function

int main()
{
    ……
    void (*pf[2])(double);//declare the point-
to-function
    pf[0]=dis;
    pf[1]=dis2;
    for(int i=0;i<2;i++)
    {
        calculate(1.22,pf[i]);
    }
    ……
}
```

中国江苏省泰州市姜堰区华港镇, 泰州市, 江苏省, 中国 • 36° PARTLY CLOUDY

WEDNESDAY, JULY 23, 2014 AT 17:02

about the I/O&judgement summary

(1) I/O
c++:
cin
cout
cin:
a).character input:
char l;
cin>>l;//cin class can input the char type directly
b).string input:

```
    char *Str=new char[size];
    cin>>Str;//input the string, but can't get
in blank-space from keyboard with the method cin
                                ……
    delete [] Str;//after the operation, never
forget the delete method to release the memory


    e.g
    $>Alistair Dreeb
```

when you output the string, it only display the "Alistair" rather than "Alistair Dreeb" for it including the blank-space, the same with Tab and line-break(\t,\n) which are used as symbol of the end of the

string in ‘cin’.
Todo
to solve this problem, we could use several ideas.
1. cin.getline(const char *,int)
this function read a line per time, and this function does not leave the ‘\0’ in the stream
2. cin.get(const char *,int)
this function almost do the same as ‘getline’, but the only different is that it will leave the ‘\0’ in the stream
e.g.
cin.get(strname,size);
cin.get(strname1,size);//error for it inputing nothing as the first read was ‘\0’

fixed
cin.get(strname,size);
cin.get();//read newline;
cin.get(strname2,size);

```
    int year;
    cin>>year;
    char *strname=new char[89];
    cin.getline(strname,89);//error! cin>>year
then hit the ENTER and leave the ‘\n’ in the
stream that lead to a cin.getline() error.
```

fixed

```
    int year;
    cin>>year;
    cin.get();//read newline to get rid of the
‘\n’
    char *strname=new char[89];
    cin.getline(strname,89);
```

中国江苏省泰州市姜堰区华港镇, 泰州市, 江苏省, 中国 • 36° PARTLY CLOUDY

THURSDAY, JULY 24, 2014 AT 15:42

c).input the string and estimate to jump out of the circle

NO.1
when input one special character, jump out the loop.

```
    char a;
    cin>>a;
    while(‘q’!=a)
    {
        cout<<“enter the character:”<<endl;
        cin>>a
```

```
    }
    e.g.
    enter the character:
    see ken runqreally room
```

cin ignore the Space&Tab, which means there only 9 characters been counted.

TODO
to solve this, cin.get() can be used even the entered character was Space.

```
    char a;
    cin.get(ch);
    while('q'!=a)
    {
        cout<<"enter the character:"<<endl;
        cin.get(ch);
    }
```

NO.2
EOF(end of file) will set two bits, failbit and eofbit.
using function cin.eof() return bool type to ensure the end of file
using function cin.fail() return bool type to ensure the input type

```
    e.g.
    char ch;
    cin.get(ch);
    while(cin.fail()==false)//verify the cin was
right and don't get end of file
    {
        cin.get(ch);
    }
```

NO.3
the much more useful idea about test the entered character
while(cin.get(ch))
{
……
}

中国江苏省泰州市姜堰区华港镇, 泰州市, 江苏省, 中国 • 32° CLOUDY

SUNDAY, JULY 27, 2014 AT 17:47

default parameters of function and string assignment

（1）default parameters of function
assign the parameter of function in prototype as the default parameter value.

```
e.g.
void foo(const char * str,double x=3.14,int
u=2);//function prototype where we can assign
the default   value of the function parameter
int main()
{
    ……
}

void foo(const char *str,double x,int
u)//function definition
{
    ……
}
```

why the default parameter must be assigned in the declaration?

NO.1 the difference between function declaration and function definition
1).function declaration just tell the compiler that there is a function including some parameter which will be called
2).function definition will allocate details of the function method in the memory.

NO.2 actually we can assign the default parameter in both of declaration and definition, but that cause ambiguity. so the ANSCI recommend that assign in declaration.

NO.3 the default parameters should be located from right to left.

(2) string assignment
there is no better idea than strcp() for now.

中国江苏省泰州市姜堰区华港镇, 泰州市, 江苏省, 中国 • 24° SHOWERS

MONDAY, JULY 28, 2014 AT 18:28

string I/O and member function of class string

string I/O is almost the same as type char, just use cin>>(string)… but when we should include the blank-space into the string object with cin, something special changed.

```
e.g.
string str;
getline(cin,str);
>$ go away
>$echo str
>$go away
```

class string
the definition was included in the file
meanwhile the method operat with the string object was included in

the file string.h(old-c standard),now we use cstring file

```
e.g.
#include <string>       //make the class string
available
#include <cstring>  //make the method operate
with string object available
using std::string       //to available the
class, we must ensure the scope std
```

some methods:
lenghth() //return the integer about the string character's numbers, including the '\0'
size() //the same with the method length(), except for it does not including the member '\0'
c_str() //return the char * type(transform the string into char)

中国江苏省泰州市姜堰区华港镇, 泰州市, 江苏省, 中国 • 30° MOSTLY SUNNY

FRIDAY, AUGUST 1, 2014 AT 21:20

function reload and function template

technically, template function serves for function reload.
first of all, let us know the compiler's work about how to recognize which function the code hoping to call.

(1)create the candidate function list that contains all the function and template function which have the same function name being called.

(2)create the feasible function list that contains the function whose parameters are absolutely matched.

(3)check if the function calling is all right.

after the match operate, the compiler will judge the function signature is fully-matched or promote-transformation or standard-transformation or custom-transformation.

NOTE:
priority of the function matching

#1 conventional function
#2 explicit specialization of template function
#3 conventional template function

西浙大路 38号, 杭州市, 浙江省, 中国 • 27° MOSTLY CLOUDY

FRIDAY, AUGUST 1, 2014 AT 23:20

C++中用new动态创建二维数组的格式一般是这样：

TYPE (* p)[N] = new TYPE [][N];
其中，TYPE是某种类型，N是二维数组的列数。采用这种格式，列数必须指出，而行数无需指定。在这里，p的类型是TYPE*[N]，即是指向一个有N列元素数组的指针。
还有一种方法，可以不指定数组的列数：

```
int * *p;
p = new int *[10]; //注意，int *[10]表示一个有10个元素的指针数组
for (int i = 0; i != 10; ++i)
{
p[i] = new int[5];
}
```

这里是将p作为一个指向指针的指针，它指向一个包含10个元素的指针数组，并且每个元素指向一个有5个元素的数组，这样就构建了一个10行5列的数组。
当数组使用完毕，释放空间的代码是：

```
for(int i = 0; i != 5; i++)
{
delete[] p[i];
}
delete[] p;
```

处理二维数组，可以用降维或是二维法。
降维法是用一位数组来接受二维数组，将二维元素的首地址&a[0][0]作为参数，传递给函数，函数用int *接受。
二维法就直接用二维数组来接受，但是需要指定列数。

杭州西湖风景名胜区, 杭州市, 浙江省, 中国 • 27° MOSTLY CLOUDY

MONDAY, AUGUST 11, 2014 AT 21:18

the procedure of program compiling

1 –>source code
2 –>compile preprocess
3 –>compile the code itself
4 –>optimize the code
5 –>assemble the code
6 –>link the program
7 –>generate the executable file

2 replace all the preprocess key word with the file or variable.
for instance, #define #undef
#include

#ifdef #ifndef #else #elif #endif
LINE FILE (special word)
3 there only are key words, constant variable,variable and so on after preprocess. compiling the code itself just check the correct of the lexical, syntax according to the rules. And then generate the intermediate code.

4 the place where the optimization happens is uncertain. Totally, the compiler will optimize the intermediate code and the object code. on intermediate code, it will delete the public expression, optimize the loops, delete the unused assignment and so on.
on object code, it will decrease the volume of visit of the register and, meanwhile, adapt the code into the shortest one.

5 assembly process will translate the assembly code into machine code and get the object code. the object code include three types, relocatable file, share file, executable file. the assembly process will get the relocatable file.

6 link program process will solve the problem of object code file, which needs to call other source code or library, can not be executed. there are two ways having it solved, the first one is that called runtime link and the other called static link.

source code—-[compile S2-S3-S4-S5]—->object file—-[link S6]—->executable
||
||
\/
＊.obj for Win
＊.o for Unix
package the .obj files into library file for Win
package the .o files into archive file for Unix

杭州西湖风景名胜区, 杭州市, 浙江省, 中国 • 27° MOSTLY CLEAR

TUESDAY, AUGUST 12, 2014 AT 22:15

placement new

1.经典例子
#include
#include
const intchunk = 16;
class Foo
{
public :
int val( ) { return _val; }
Foo( ) { _val = 0; }

```
private :
int_val;
};
//预分配内存，但没有Foo对象
char*buf = new char[ sizeof(Foo) * chunk ];
int
main( void )
{
//在buf中创建一个Foo对象
Foo*pb = new (buf) Foo;
//检查一个对象是否被放在buf中
if ( pb->val() == 0 )
{
cout <<“new expressio worked!” <<endl;
}
//到这里不能再使用pb
delete[] buf;
return 0;
}
```

但我不了解这种“定位new表达式”相对于一般的new的优点是什么？它的用法比一般的new对象要复杂！

2.使用定位new表达式作用

1）placement new的作用就是：创建对象但是不分配内存，而是在已有的内存块上面创建对象。用于需要反复 创建并删除的对象上，可以降低分配释放内存的性能消耗。

2）定位new表达式（placement new expression，C++ primer P347），允许程序员将对象创建在已经被分配好的内存中，new表的式的形式如下：

```
new (place_address) type
new (palce_address) type (initializer-list)
```

place_address必须是个指针，指向已经分配好的内存。为了使用这种形式的new表达式，必须包含头文件。

定位new表达式不能调用delete删除 placement new的对象，需要人为的调用对象的析构函数，并且人为的释放掉占用的内存。

```
#include
#include

using namespace std;

const int chunk = 16;

class Foo
{
public:
int val(){return _val;}
Foo(){_val=0;}
private:
int _val;
```

};

int main()
{
// 预分配内存buf

char *buf = new char[sizeof(Foo) * chunk];

// 在buf中创建一个Foo对象

Foo *pb=new (buf) Foo;
// 检查一个对象是否被放在buf中

if(pb->val()==0) cout<<“new expression worked!”<<endl;
// 这里不存在与定位new表达式匹配的delete表达式，即：delete pb,其实只是为了
// 释放内存的话，我们不需要这样的表达式，因为定位new表达式并不分配内存。
// 如果在析构函数中要做一些其他的操作呢？就要显示的调用析构函数。
// 当程序不再需要buf时，buf指向的内存被删除，它所包含的任何对象的生命期也就
// 都结束了。

delete[] buf;
return 0;
}
定位new表达式的应用：

在我们的程序中，首先使用malloc为对象分配空间，然后通过定位new表达式将对象创建在已经被分配好的内存中，完成构造函数的调用。
3)
“new (start) Screen;”
这是定位new操作
Screen *ps = new (start) Screen;
表示在已经开辟好的内存区start中为堆对象Screen申请一个内存
也就是说，如果start占１００个字节
你现在从它的第 1 个字节开始，往里面写入Screen
这样做今后要释放内存的时候有麻烦，如果Screen里面有指针，并开辟了空间，那么delete ps会调用析构函 数，然而在释放指针变量所占的内存空间时也释放了为对象所分配的内存空间，所以start中的那块内存也丢失了，因为ps也指向了start.所以要避免这种情况，就要先ps->~Screen()这样显示的调用Screen的析构函数，再delete []start。

杭州西湖风景名胜区, 杭州市, 浙江省, 中国 • 23° MIST AND FOG

FRIDAY, AUGUST 22, 2014 AT 23:43

char * initialization

C's rules for string literals are different from C++'s rules. In C, a string literal is an array of char, not an array of const char* – but attempting to modify it has undefined behavior. This means that in C you can legally write char *s = “hello”; s[0] = 'H';, and the compiler won't necessarily complain – but the program is likely to die with a segmentation fault when you run it. This was done to maintain backward compatibility with C code written before the const keyword was introduced. C++ had const from the very beginning, so this particular compromise wasn't necessary.

30.2681° N, 120.118° E • 27° MOSTLY CLEAR

MONDAY, AUGUST 25, 2014 AT 23:29

Templates and Template Classes in C++ (coding/C++/chapter10/8.cpp#t8.h#t8.cpp)

What's better than having several classes that do the same thing to different datatypes? One class that lets you choose which datatype it acts on.
Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class. In essence, this is what is known as generic programming; this term is a useful way to think about templates because it helps remind the programmer that a templated class does not depend on the datatype (or types) it deals with. To a large degree, a templated class is more focused on the algorithmic thought rather than the specific nuances of a single datatype. Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data. For example, you could make a templated stack class that can handle a stack of any datatype, rather than having to create a stack class for every different datatype for which you want the stack to function. The ability to have a single class that can handle several different datatypes means the code is easier to maintain, and it makes classes more reusable.The basic syntax for declaring a templated class is as follows:

```
template <class a_type> class a_class {...};
```

The keyword 'class' above simply means that the identifier a_type will stand for a datatype. NB: a_type is not a keyword; it is an identifier that during the execution of the program will represent a single datatype. For example, you could, when defining variables in

the class, use the following line:
a_type a_var;

and when the programmer defines which datatype 'a_type' is to be when the program instantiates a particular instance of a_class, a_var will be of that type. When defining a function as a member of a templated class, it is necessary to define it as a templated function:

```
template<class a_type> void
a_class<a_type>::a_function(){...}
```

When declaring an instance of a templated class, the syntax is as follows:

```
a_class<int> an_example_class;
```

An instantiated object of a templated class is called a specialization; the term specialization is useful to remember because it reminds us that the original class is a generic class, whereas a specific instantiation of a class is specialized for a single datatype (although it is possible to template multiple types). Usually when writing code it is easiest to precede from concrete to abstract; therefore, it is easier to write a class for a specific datatype and then proceed to a templated - generic - class. For that brevity is the soul of wit, this example will be brief and therefore of little practical application. We will define the first class to act only on integers.

```
class calc
{
    public:
    int multiply(int x, int y);
    int add(int x, int y);
};
int calc::multiply(int x, int y)
{
    return x*y;
}
int calc::add(int x, int y)
{
    return x+y;
}
```

We now have a perfectly harmless little class that functions perfectly well for integers; but what if we decided we wanted a generic class that would work equally well for floating point numbers? We would use a template.

```
template <class A_Type> class calc
{
    public:
    A_Type multiply(A_Type x, A_Type y);
    A_Type add(A_Type x, A_Type y);
};
template <class A_Type> A_Type
calc<A_Type>::multiply(A_Type x,A_Type y)
```

```
{
    return x*y;
}
template <class A_Type> A_Type
calc<A_Type>::add(A_Type x, A_Type y)
{
    return x+y;
}
```

To understand the templated class, just think about replacing the identifier A_Type everywhere it appears, except as part of the template or class definition, with the keyword int. It would be the same as the above class; now when you instantiate an object of class calc you can choose which datatype the class will handle.

```
calc <double> a_calc_class;
```

Templates are handy for making your programs more generic and allowing your code to be reused later.

杭州西湖风景名胜区, 杭州市, 浙江省, 中国 • 27° MOSTLY CLEAR

WEDNESDAY, AUGUST 27, 2014 AT 21:45

pointer function and its usage(coding/C++/chapter10/8.cpp#t8.h#t8.cpp)

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. This is useful because functions encapsulate behavior. For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function. But sometimes you would like to choose different behaviors at different times in essentially the same piece of code. Read on for concrete examples.

Example Uses of Function Pointers
Functions as Arguments to Other Functions
If you were to write a sort routine, you might want to allow the function's caller to choose the order in which the data is sorted; some programmers might need to sort the data in ascending order, others might prefer descending order while still others may want something similar to but not quite like one of those choices. One way to let your user specify what to do is to provide a flag as an argument to the function, but this is inflexible; the sort function allows only a fixed set of comparison types (e.g., ascending and descending).

A much nicer way of allowing the user to choose how to sort the data is simply to let the user pass in a function to the sort function.

This function might take two pieces of data and perform a comparison on them. We'll look at the syntax for this in a bit.

Callback Functions

Another use for function pointers is setting up "listener" or "callback" functions that are invoked when a particular event happens. The function is called, and this notifies your code that something of interest has taken place.

Why would you ever write code with callback functions? You often see it when writing code using someone's library. One example is when you're writing code for a graphical user interface (GUI). Most of the time, the user will interact with a loop that allows the mouse pointer to move and that redraws the interface. Sometimes, however, the user will click on a button or enter text into a field. These operations are "events" that may require a response that your program needs to handle. How can your code know what's happening? Using Callback functions! The user's click should cause the interface to call a function that you wrote to handle the event.

To get a sense for when you might do this, consider what might happen if you were using a GUI library that had a "create_button" function. It might take the location where a button should appear on the screen, the text of the button, and a function to call when the button is clicked. Assuming for the moment that C (and C++) had a generic "function pointer" type called function, this might look like this:

```
void create_button( int x, int y, const char
*text, function callback_func );
```

Whenever the button is clicked, callback_func will be invoked. Exactly what callback_func does depends on the button; this is why allowing the create_button function to take a function pointer is useful.

Function Pointer Syntax

The syntax for declaring a function pointer might seem messy at first, but in most cases it's really quite straight-forward once you understand what's going on. Let's look at a simple example:

```
void (*foo)(int);
```

In this example, foo is a pointer to a function taking one argument, an integer, and that returns void. It's as if you're declaring a function called "*foo", which takes an int and returns void; now, if *foo is a function, then foo must be a pointer to a function. (Similarly, a declaration like int *x can be read as *x is an int, so x must be a pointer to an int.)

The key to writing the declaration for a function pointer is that you're just writing out the declaration of a function but with (*func_name)

where you'd normally just put func_name.

Reading Function Pointer Declarations
Sometimes people get confused when more stars are thrown in:

```
void *(*foo)(int *);
```

Here, the key is to read inside-out; notice that the innermost element of the expression is *foo, and that otherwise it looks like a normal function declaration. *foo should refer to a function that returns a void * and takes an int *. Consequently, foo is a pointer to just such a function.

Initializing Function Pointers
To initialize a function pointer, you must give it the address of a function in your program. The syntax is like any other variable:

```
#include <stdio.h>
void my_int_func(int x)
{
    printf( "%d\n", x );
}

int main()
{
    void (*foo)(int);
    /* the ampersand is actually optional */
    foo = &my_int_func;

    return 0;
}
```

(Note: all examples are written to be compatible with both C and C++.)

Using a Function Pointer
To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call. The act of calling it performs the dereference; there's no need to do it yourself:

```
#include <stdio.h>
void my_int_func(int x)
{
    printf( "%d\n", x );
}


int main()
{
    void (*foo)(int);
    foo = &my_int_func;

    /* call my_int_func (note that you do not
need to write (*foo)(2) ) */
    foo( 2 );
    /* but if you want to, you may */
    (*foo)( 2 );
```

```
    return 0;
}
```

Note that function pointer syntax is flexible; it can either look like most other uses of pointers, with & and *, or you may omit that part of syntax. This is similar to how arrays are treated, where a bare array decays to a pointer, but you may also prefix the array with & to request its address.

Function Pointers in the Wild

Let's go back to the sorting example where I suggested using a function pointer to write a generic sorting routine where the exact order could be specified by the programmer calling the sorting function. It turns out that the C function qsort does just that.

From the Linux man pages, we have the following declaration for qsort (from stdlib.h):

```
void qsort(void *base, size_t nmemb, size_t
size,
        int(*compar)(const void *, const void
*));
```

Note the use of void*s to allow qsort to operate on any kind of data (in C++, you'd normally use templates for this task, but C++ also allows the use of void* pointers) because void* pointers can point to anything. Because we don't know the size of the individual elements in a void* array, we must give qsort the number of elements, nmemb, of the array to be sorted, base, in addition to the standard requirement of giving the length, size, of the input.

But what we're really interested in is the compar argument to qsort: it's a function pointer that takes two void *s and returns an int. This allows anyone to specify how to sort the elements of the array base without having to write a specialized sorting algorithm. Note, also, that compar returns an int; the function pointed to should return -1 if the first argument is less than the second, 0 if they are equal, or 1 if the second is less than the first.

For instance, to sort an array of numbers in ascending order, we could write code like this:

```
#include <stdlib.h>

int int_sorter( const void *first_arg, const
void *second_arg )
{
int first = *(int*)first_arg;
int second = *(int*)second_arg;
if ( first < second )
{
    return -1;
}
else if ( first == second )
```

```
{
    return 0;
}
else
{
    return 1;
}
}

int main()
{
int array[10];
int i;
/* fill array */
for ( i = 0; i < 10; ++i )
{
    array[ i ] = 10 - i;
}
qsort( array, 10 , sizeof( int ), int_sorter );
for ( i = 0; i < 10; ++i )
{
    printf ( "%d\n" ,array[ i ] );
}

}
```

Using Polymorphism and Virtual Functions Instead of Function Pointers (C++)
You can often avoid the need for explicit function pointers by using virtual functions. For instance, you could write a sorting routine that takes a pointer to a class that provides a virtual function called compare:

```
class Sorter
{
public:
virtual int compare (const void *first, const
void *second);
};

// cpp_qsort, a qsort using C++ features like
virtual functions
void cpp_qsort(void *base, size_t nmemb, size_t
size, Sorter *compar);
```

inside cpp_qsort, whenever a comparison is needed, compar->compare should be called. For classes that override this virtual function, the sort routine will get the new behavior of that function. For instance:

```
class AscendSorter : public Sorter
{

virtual int compare (const void*, const void*)
{
    int first = *(int*)first_arg;
    int second = *(int*)second_arg;
    if ( first < second )
    {
```

```
        return -1;
    }
    else if ( first == second )
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
};
```

and then you could pass in a pointer to an instance of the AscendSorter to cpp_qsort to sort integers in ascending order.
But Are You Really Not Using Function Pointers?
Virtual functions are implemented behind the scenes using function pointers, so you really are using function pointers–it just happens that the compiler makes the work easier for you. Using polymorphism can be an appropriate strategy (for instance, it's used by Java), but it does lead to the overhead of having to create an object rather than simply pass in a function pointer.

Function Pointers Summary
Syntax
Declaring
Declare a function pointer as though you were declaring a function, except with a name like *foo instead of just foo:

```
void (*foo)(int);
```

Initializing
You can get the address of a function simply by naming it:

```
void foo();
func_pointer = foo;
```

or by prefixing the name of the function with an ampersand:

```
void foo();
func_pointer = &foo;
```

Invoking
Invoke the function pointed to just as if you were calling a function.

```
func_pointer( arg1, arg2 );
```

or you may optionally dereference the function pointer before calling the function it points to:

```
(*func_pointer)( arg1, arg2 );
```

Benefits of Function Pointers
1, Function pointers provide a way of passing around instructions

for how to do something
2，You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments
3，This flexibility can also be achieved by using classes with virtual function

---

杭州西湖风景名胜区, 杭州市, 浙江省, 中国 • 26° MOSTLY CLOUDY

WEDNESDAY, SEPTEMBER 3, 2014 AT 23:23

operator overloading in c++

In C++ the overloading principle applies not only to functions, but to operators too. That is, of operators can be extended to work not just with built-in types but also classes. A programmer can provide his or her own operator to a class by overloading the built-in operator to perform some specific computation when the operator is used on objects of that class. Is operator overloading really useful in real world implementations? It certainlly can be, making it very easy to write code that feels natural (we'll see some examples soon). On the other hand, operator overloading, like any advanced C++ feature, makes the language more complicated. In addition, operators tend to have very specific meaning, and most programmers don't expect operators to do a lot of work, so overloading operators can be abused to make code unreadable. But we won't do that.

An Example of Operator Overloading

```
Complex a(1.2,1.3);     //this class is used to
represent complex numbers
Complex b(2.1,3);       //notice the
construction taking 2 parameters for the real
and imaginary part
Complex c = a+b;        //for this to work the
addition operator must be overloaded
```

The addition without having overloaded operator + could look like this:

```
Complex c = a.Add(b);
```

This piece of code is not as readable as the first example though– we're dealing with numbers, so doing addition should be natural. (In contrast to cases when programmers abuse this technique, when the concept represented by the class is not related to the operator–ike using + and - to add and remove elements from a data structure. In this cases operator overloading is a bad idea, creating confusion.)

In order to allow operations like Complex c = a+b, in above code we overload the "+" operator. The overloading syntax is quite simple, similar to function overloading, the keyword operator must be followed by the operator we want to overload:

```
class Complex
{
public:
    Complex(double re,double im)
            :real(re),imag(im)
            {};
    Complex operator+(const Complex& other);
    Complex operator=(const Complex& other);
private:
    double real;
    double imag;
};
Complex Complex::operator+(const Complex&
other)
{
double result_real = real + other.real;
double result_imaginary = imag + other.imag;
return Complex( result_real, result_imaginary );
}
```

The assignment operator can be overloaded similarly. Notice that we did not have to call any accessor functions in order to get the real and imaginary parts from the parameter other since the overloaded operator is a member of the class and has full access to all private data. Alternatively, we could have defined the addition operator globally and called a member to do the actual work. In that case, we'd also have to make the method a friend of the class, or use an accessor method to get at the private data:

```
friend Complex operator+(Complex);
Complex operator+(const Complex  &num1, const
Complex &num2)
{
double result_real = num1.real + num2.real;
double result_imaginary = num1.imag + num2.imag;
return Complex( result_real, result_imaginary );
}
```

Why would you do this? when the operator is a class member, the first object in the expression must be of that particular type. It's as if you were writing:

```
Complex a( 1, 2 );
Complex a( 2, 2 );
Complex c = a.operator=( b );
```

when it's a global function, the implicit or user-defined conversion can allow the operator to act even if the first operand is not exactly of the same type:

```
Complex c = 2+b;        //if the integer 2  can
```

```
be converted by the Complex class, this
expression is valid
```

By the way, the number of operands to a function is fixed; that is, a binary operator takes two operands, a unary only one, and you can't change it. The same is true for the precedence of operators too; for example the multiplication operator is called before addition. There are some operators that need the first operand to be assignable, such as : operator=, operator(), operator[] and operator->, so their use is restricted just as member functions(non-static), they can't be overloaded globally. The operator=, operator& and operator, (sequencing) have already defined meanings by default for all objects, but their meanings can be changed by overloading or erased by making them private.

Another intuitive meaning of the "+" operator from the STL string class which is overloaded to do concatenation:

```
string prefix("de");
string word("composed");
string composed = prefix+word;
```

Using "+" to concatenate is also allowed in Java, but note that this is not extensible to other classes, and it's not a user defined behavior. Almost all operators can be overloaded in C++:

```
    +       -       *       /       %       ^
&       |
    ~       !       ,       =               =
    ++      --      <<      >>      ==      !=
&&      ||
    +=      -=      /=      %=      ^=      & =
|=      *=
    <<=     >>=     [ ]     ( )     ->      ->*
new     delete
```

The only operators that can't be overloaded are the operators for scope resolution (::), member selection (.), and member selection through a pointer to a function(.*). Overloading assumes you specify a behavior for an operator that acts on a user defined type and it can't be used just with general pointers. The standard behavior of operators for built-in (primitive) types cannot be changed by overloading, that is, you can't overload operator+(int,int).

The logic(boolean) operators have by the default a short-circuiting way of acting in expressions with multiple boolean operations. This means that the expression:

```
if(a && b && c)
```

will not evaluate all three operations and will stop after a false one is found. This behavior does not apply to operators that are overloaded by the programmer.

Even the simplest C++ application, like a “hello world” program, is using overloaded operators. This is due to the use of this technique almost everywhere in the standard library (STL). Actually the most basic operations in C++ are done with overloaded operators, the IO(input/output) operators are overloaded versions of shift operators(<<, >>). Their use comes naturally to many beginning programmers, but their implementation is not straightforward. However a general format for overloading the input/output operators must be known by any C++ developer. We will apply this general form to manage the input/output for our Complex class:

```
friend ostream &operator<<(ostream &out, Complex
c)     //output
{
    out<<"real part: "<<c.real<<"\n";
    out<<"imag part: "<<c.imag<<"\n";
    return out;
}
friend istream &operator>>(istream &in, Complex
&c)     //input
{
    cout<<"enter real part:\n";
    in>>c.real;
    cout<<"enter imag part: \n";
    in>>c.imag;
    return in;
}
```

Notice the use of the friend keyword in order to access the private members in the above implementations. The main distinction between them is that the operator>> may encounter unexpected errors for incorrect input, which will make it fail sometimes because we haven't handled the errors correctly. A important trick that can be seen in this general way of overloading IO is the returning reference for istream/ostream which is needed in order to use them in a recursive manner:

```
Complex a(2,3);
Complex b(5.3,6);
cout<<a<<b;
```

HANGZHOU WEST LAKE SCENERY SPOT, HANGZHOU, ZHEJIANG, CHINA • 24° MOSTLY CLEAR

WEDNESDAY, SEPTEMBER 17, 2014 AT 23:07

Inheritance in C++

The ability to use the object-oriented programming is an important feature of C++. Lesson 12: classes in C++ introduced the idea of the class; if you have not read it and do not know the basic details of

classes, you should read it before continuing this tutorial.

Inheritance is an important feature of classes; in fact, it is integral to the idea of object oriented programming. Inheritance allows you to create a hierarchy of classes, with various classes of more specific natures inheriting the general aspects of more generalized classes. In this way, it is possible to structure a program starting with abstract ideas that are then implemented by specific classes. For example, you might have a class Animal from which class dog and cat inherent the traits that are general to all animals; at the same time, each of those classes will have attributes specific to the animal dog or cat.

Inheritance offers many useful features to programmers. The ability, for example, of a variable of a more general class to function as any of the more specific classes which inherit from it, called polymorphism, is handy. For now, we will concentrate on the basic syntax of inheritance. Polymorphism will be covered in its own tutorial.

Any class can inherit from any other class, but it is not necessarily good practice to use inheritance (put it in the bank rather than go on a vacation). Inheritance should be used when you have a more general class of objects that describes a set of objects. The features of every element of that set (of every object that is also of the more general type) should be reflected in the more general class. This class is called the base class. base classes usually contain functions that all the classes inheriting from it, known as derived classes, will need. base classes should also have all the variables that every derived class would otherwise contain.

Let us look at an example of how to structure a program with several classes. Take a program used to simulate the interaction between types of organisms, trees, birds, bears, and other creatures coinhabiting a forest. There would likely be several base classes that would then have derived classes specific to individual animal types. In fact, if you know anything about biology, you might wish to structure your classes to take advantage of the biological classification from Kingdom to species, although it would probably be overly complex. Instead, you might have base classes for the animals and the plants. If you wanted to use more base classes (a class can be both a derived of one class and a base of another), you might have classes for flying animals and land animals, and perhaps trees and scrub. Then you would want classes for specific types of animals: pigeons and vultures, bears and lions, and specific types of plants: oak and pine, grass and flower. These are unlikely to live together in the same area, but the idea is essentially there: more specific classes ought to inherit from less specific classes.

Classes, of course, share data. A derived class has access to most of the functions and variables of the base class. There are, however, ways to keep a derived class from accessing some attributes of its

base class. The keywords public, protected, and private are used to control access to information within a class. It is important to remember that public, protected, and private control information both for specific instances of classes and for classes as general data types. Variables and functions designated public are both inheritable by derived classes and accessible to outside functions and code when they are elements of a specific instance of a class. Protected variables are not accessible by functions and code outside the class, but derived classes inherit these functions and variables as part of their own class. Private variables are neither accessible outside the class when it is a specific class nor are available to derived classes. Private variables are useful when you have variables that make sense in the context of large idea.

HANGZHOU WEST LAKE SCENERY SPOT, HANGZHOU, ZHEJIANG, CHINA • 25° MOSTLY CLEAR

WEDNESDAY, SEPTEMBER 17, 2014 AT 23:19

C++ Inheritance - Syntax

Before beginning this lesson, you should have an understanding of the idea of inheritance. If you do not, please read lesson 19. This lesson will consist of an overview of the syntax of inheritance, the use of the keywords public, private, and protected, and then an example program following to demonstrate each.

The syntax to denote one class as inheriting from another is simple. It looks like the following: class Bear : public Animal, in place of simply the keyword class and then the class name. The ": public base_class_name" is the essential syntax of inheritance; the function of this syntax is that the class will contain all public and protected variables of the base class. Do not confuse the idea of a derived class having access to data members of a base class and specific instances of the derived class possessing data. The data members - variables and functions - possessed by the derived class are specific to the type of class, not to each individual object of that type. So, two different Bear objects, while having the same member variables and functions, may have different information stored in their variables; furthermore, if there is a class Animal with an object, say object BigAnimal, of that type, and not of a more specific type inherited from that class, those two bears will not have access to the data within BigAnimal. They will simply possess variables and functions with the same name and of the same type.

A quick example of inheritance:

```
class Animal
{
```

```
    public:
    Animal();
    ~Animal();
    void eat();
    void sleep();
    void drink();

    private:
    int legs;
    int arms;
    int age;
};
//The class Animal contains information and
functions
//related to all animals (at least, all animals
this lesson uses)
class Cat : public Animal
{
    public:
    int fur_color;
    void purr();
    void fish();
    void markTerritory();
};
//each of the above operations is unique
//to your friendly furry friends
//(or enemies, as the case may be)
```

A discussion of the keywords public, private, and protected is useful when discussing inheritance. The three keywords are used to control access to functions and variables stored within a class.

```
public:
```

The most open level of data hiding is public. Anything that is public is available to all derived classes of a base class, and the public variables and data for each object of both the base and derived class is accessible by code outside the class. Functions marked public are generally those the class uses to give information to and take information from the outside world; they are typically the interface with the class. The rest of the class should be hidden from the user using private or protected data (This hidden nature and the highly focused nature of classes is known collectively as encapsulation). The syntax for public is:
public:
Everything following is public until the end of the class or another data hiding keyword is used.

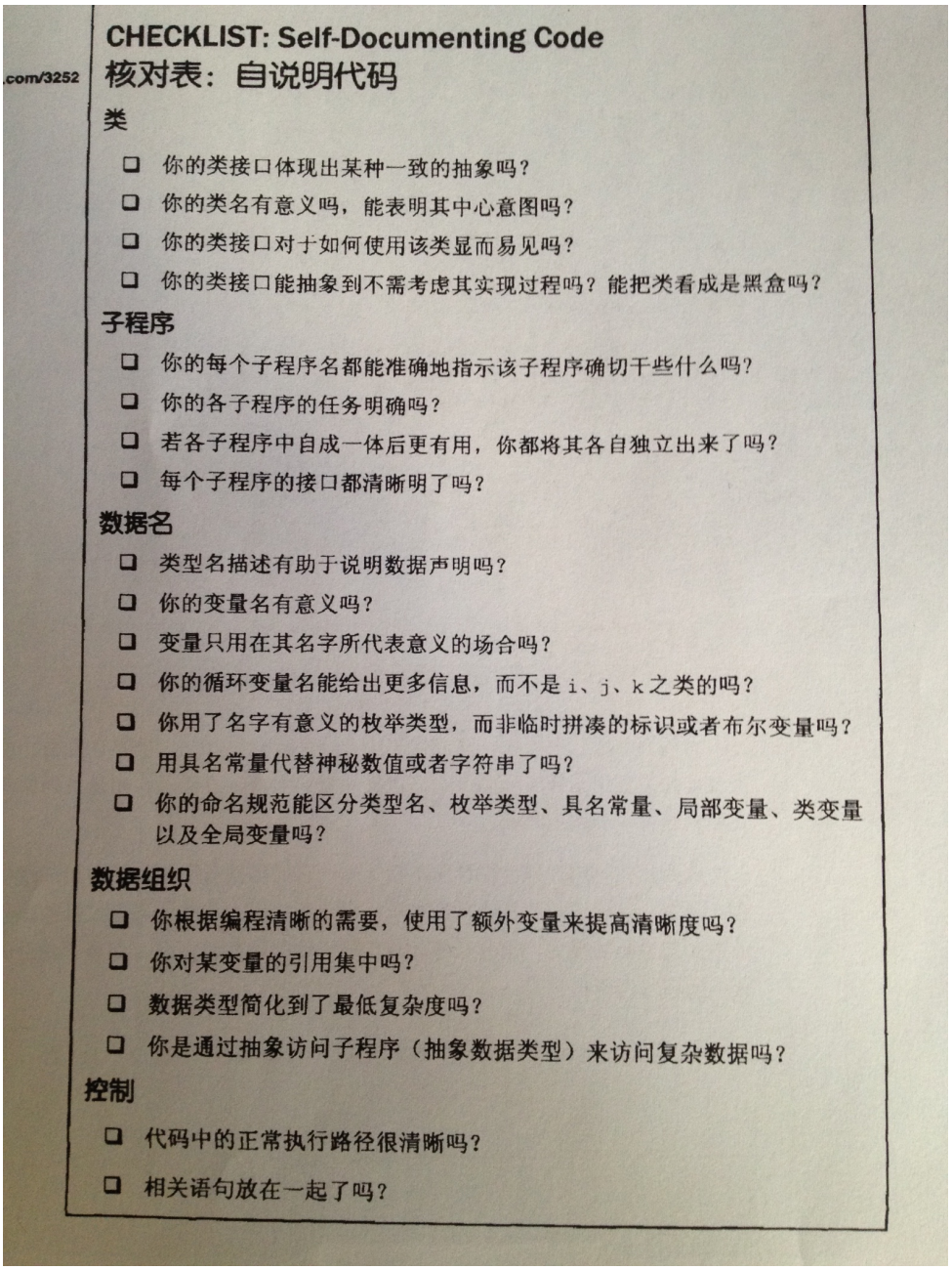
In general, a well-designed class will have no public fields– everything should go through the class's functions. Functions that retrieve variables are known as 'getters' and those that change values are known as 'setters'. Since the public part of the class is intended for use by others, it is often sensible to put the public section at the top of the class.
protected:

Variables and functions marked protected are inherited by derived classes; however, these derived classes hide the data from code outside of any instance of the object. Keep in mind, even if you have another object of the same type as your first object, the second object cannot access a protected variable in the first object. Instead, the second object will have its own variable with the same name - but not necessarily the same data. Protected is a useful level of access control for important aspects to a class that must be passed on without allowing it to be accessed. The syntax is the same as that of public. specifically,
protected:
private:

Private is the highest level of data-hiding. Not only are the functions and variables marked private not accessible by code outside the specific object in which that data appears, but private variables and functions are not inherited (in the sense that the derived class cannot directly access these variables or functions). The level of data protection afforded by protected is generally more flexible than that of the private level. On the other hand, if you do not wish derived classes to access a method, declaring it private is sensible.
private:

HANGZHOU WEST LAKE SCENERY SPOT, HANGZHOU, ZHEJIANG, CHINA • 25° MOSTLY CLEAR

SUNDAY, JANUARY 4, 2015 AT 14:19



.com/3252

CHECKLIST: Self-Documenting Code
核对表：自说明代码

类

- ❑ 你的类接口体现出某种一致的抽象吗？
- ❑ 你的类名有意义吗，能表明其中心意图吗？
- ❑ 你的类接口对于如何使用该类显而易见吗？
- ❑ 你的类接口能抽象到不需考虑其实现过程吗？能把类看成是黑盒吗？

子程序

- ❑ 你的每个子程序名都能准确地指示该子程序确切干些什么吗？
- ❑ 你的各子程序的任务明确吗？
- ❑ 若各子程序中自成一体后更有用，你都将其各自独立出来了吗？
- ❑ 每个子程序的接口都清晰明了吗？

数据名

- ❑ 类型名描述有助于说明数据声明吗？
- ❑ 你的变量名有意义吗？
- ❑ 变量只用在其名字所代表意义的场合吗？
- ❑ 你的循环变量名能给出更多信息，而不是 i、j、k 之类的吗？
- ❑ 你用了名字有意义的枚举类型，而非临时拼凑的标识或者布尔变量吗？
- ❑ 用具名常量代替神秘数值或者字符串了吗？
- ❑ 你的命名规范能区分类型名、枚举类型、具名常量、局部变量、类变量以及全局变量吗？

数据组织

- ❑ 你根据编程清晰的需要，使用了额外变量来提高清晰度吗？
- ❑ 你对某变量的引用集中吗？
- ❑ 数据类型简化到了最低复杂度吗？
- ❑ 你是通过抽象访问子程序（抽象数据类型）来访问复杂数据吗？

控制

- ❑ 代码中的正常执行路径很清晰吗？
- ❑ 相关语句放在一起了吗？

Self-Documenting Code

- programming style
- comments

TIANMUSHAN ROAD, HANGZHOU, ZHEJIANG, CHINA • 19° MOSTLY SUNNY

SUNDAY, JANUARY 4, 2015 AT 15:48

Key of Effective Comments

- Repeat of the Code

```
1. repeat the code that provide any other
information
```

- Explanation of the Code

```
1. explain the complex, important and skilful
code
2. when use this comments, check the code if it
is optimized whose complexity can be reduced
```

- Marker in the Code

```
1. remind the programmer of something to be done
```

- Summary of the Code

```
1. explain dozen of the code to help someone
else to quick go through it
```

- Description of the Code's Intent

```
1. identify the intention of the code
```

- Information That Cannot Possibly Be Expressed by the Code Itself

HANGZHOU WEST LAKE SCENERY SPOT, HANGZHOU, ZHEJIANG, CHINA • 18° MOSTLY SUNNY

MONDAY, JANUARY 12, 2015 AT 23:18

Comments

ENDLINE COMMENTS

endline comments are not recommended except the following circumstance.

explain the variable

note about the block of the code

COMMENT PARAGRAPHS OF CODE

- comment should explain the purpose of the paragraph of code
- code itself should try to do some explanations witn a good program style
- focus the comments on 'why' other than 'how', which means that explaining the intention of the paragraph code is the purpose of comment
- delete the useless comments
- use comment to note of some abnormal usage of code that is helpful for improving the performance

---

HANGZHOU WEST LAKE SCENERY SPOT, HANGZHOU, ZHEJIANG, CHINA • 7° MOSTLY CLEAR

MONDAY, JANUARY 12, 2015 AT 23:21

Comment Details

COMMENT DATA DECLARATIONS

- note about the units of the variable (personally add this info into the name of the variable)
- note about the range of the variable if possible
- explain the meaning of the variable

COMMENT CONTROL STRUCTURE

- note about the intention of this control structure
- note about the nested structure(if…else,while,for,switch…case)

COMMENT ROUTINES

- commenting of routines had better be close to the routines
- one or two sentence to explain the intention of the routines, if you can not do this,please review the design of this routine
- note about the variables and remind of the input and output
- note about the options of the variables
- note about the limitation of the routines

COMMENT CLASS, FILES, AND PROGRAMS

- about the comment of the class:

- explain the design of this class
- remind of the limitation of the class and the usage of the interface of the class
- explain the interface of the class
- never provide the details of the class
- about the comment of the file:
- comment at the begin of the file
- explain the intention of the file and the summary of the content
- note about the coder information
- about the book paradigm for program documentation: `to be done`

GET HELP FROM IEEE STD

HANGZHOU WEST LAKE SCENERY SPOT, HANGZHOU, ZHEJIANG, CHINA • 7° MOSTLY CLEAR

FRIDAY, JUNE 19, 2015 AT 11:13

记出差

见鬼了…去出了趟差，车票钱没拿到自己那一份，还把洗簌用的落在了宾馆…更重要的是东西找不到了……这都是什么鬼！

XIXI ROAD, HANGZHOU, ZHEJIANG, CHINA • 26° MOSTLY CLOUDY

SATURDAY, JULY 11, 2015 AT 13:20

算法工程师复习笔记

1.有两个 int 型的变量 a 和 n(0 <= n <= 31),要求写一个表达式,使该表达式的值和 a 的第 n 位相同。

```
int i=1
i << n;
a & i;
a & (1 << n)
```

2.int MyItoa(char * s)其功能是将s中以字符串形式存放的非负整数,转换成相应整数返回。例如,如果 s 中存放字符串 “1234”,则该函数的返回值就是 1234。假设s中的字符全是数字,且不考虑s是空串或s太长的情况。

```
int results = 0;
```

```
for (int i=0; i < strlen(s); ++i) {
    result = result*10 + int(s[0] - '0');
}
```

3.

```
void * MyMemcpy( void * dest , const void * src,
int n) {
    char * pDest = (char * )dest;
    char * pSrc = ( char * ) src;
    for( int i = 0; i < n; i ++ ) { //逐个字节拷贝
源块的内容到目的块
    * (pDest + i) = * ( pSrc + i );
    }
    return dest;
}
```

上面的 MyMemcpy 函数是有缺陷的,在某些情况下不能得到正确结果,缺陷在哪里?如何改进?

空指针拷贝有问题，加一个判断即可

LAOHE MOUNTAIN, HANGZHOU, ZHEJIANG, CHINA • 25° RAIN SHOWERS

MONDAY, FEBRUARY 8, 2016 AT 00:44

新年

这是我的本命年之后的第一年，也是开始工作的第一年，漫漫前途不知在何方，心有戚戚唯有踌躇前行了。
新年愿父母身体健康，少病少灾，顺顺利利
愿爱人身体健康，开开心心，万事如意

JIANGYAN, TAIZHOU, JIANGSU, CHINA • -1° MOSTLY CLEAR

Created in Day One