OOP Midterm

1
```
#include <functional>
#include <algorithm>
sort(a,a+15,less<int>());
```

2
```
int h[9]={1,3,2,5,9,8,4,7,6};
```

3 This template can't be used for floating-point types.
For example, the call

```
multiply(2.3,4)
```

will result in a compile-time error, because within the generated instance

```
double multiply(double a,unsigned k)
{
    if (numeric_limits<double>::is_integer) return a<<k;
    else return a*(1<<k);
}
```

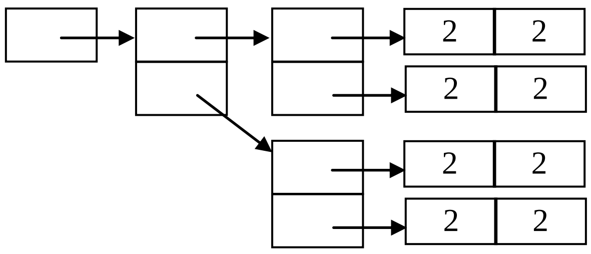
the red-colored code attempts to left-shift a floating-point number.

4
```
operator delete(*p);
delete p;
```

5
```
void* operator new[](size_t,void* buf)
{
    return buf;
}
```

6
```
#include<vector>
vector<vector<vector<int> > >
                    v(2,vector<vector<int> >(2,vector<int>(2,2)));
```

7 **v**

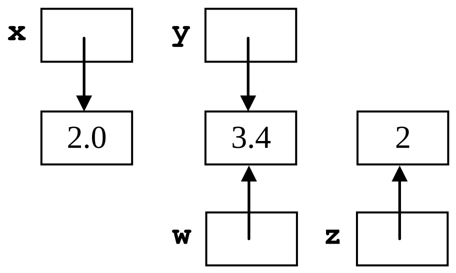


8 1) lvalue-2-rvalue, integral promotion ← best viable
2) lvalue-2-rvalue, integral conversion

9 1) Legal, since a function returning a reference yields an lvalue.
2) Legal, since a cast to a reference type yields an lvalue.

10 1) `int (*)[3];`
2) `int (*[2])[3];`

11



12 No
With
`using std::max;`
the global namespace *physically* contains
`template<typename T> const T& max(const T& x,const T& y);` // `std::max`
and
`template<typename T> T max(const T& x,const T& y);`
But this is illegal, since both have the same parameter list and so cannot be overloaded.
N.B. VC++ ↓, GNU C++ ↑

13 2)
∵ An explicit specialization must be defined in the namespace of which the template it specializes is a member.
Also, the return type of an explicit specialization needn't be an instance of the return type of the template it specializes.

14 2), 3) and 4) are legal

15
```
void (*set_unexpected(void(*)()))();
```
or
```
void (*set_unexpected(void()))();
```
or
```
typedef void(*pf)();
pf set_unexpected(pf);
```
or
```
typedef void f();
f* set_unexpected(f*);
```

16 1) Ambiguous
Call-by-value and call-by-reference are equally well.
2) Template A
∵ **(int)x** is an rvalue, so only template A that uses call-by-value is viable.

17
```
cout << a[0]+a[1]+a[2];
```

18
```
sum<U>(a[0])+sum<U>(reinterpret_cast<T(&)[n-1]>(a[1]))
```

19 The ctor isn’t an inline function whose definition can’t be placed in the class interface file.

20 The data members aren’t hidden.
There is no dtor to reclaim the storage allocated by the ctor.

21
```
const int& top(const stack* this)
{
    return this->stk[this->_top];
}
```

22
```
template<typename T,typename Ufn,typename Bfn>
T faccumulate(T* first,T* last,T init,Ufn p,Bfn f)
{
    T result=init;
    for (T* it=first;it!=last;++it)
        if (p(*it)) result=f(*it,result);
    return result;
}
```

23
```
template<typename T>
struct even {
   bool operator() (const T& x) const { return x%2==0; }
};
faccumulate(a,a+15,0,even<int>(),plus<int>());
```