OOP MIDTERM SOLUTIONS

1 C-style: declare it static.

```
static auto hideme="Hide me!";
```

C++-style: declare it inside an anonymous namespace.

```
namespace { auto hideme="Hide me!"; }
```

2 1) Illegal.

`using std::sort;` introduces the STL function **`sort`** into the global namespace, making **`int sort=1`** illegal, since variable names can't be overloaded with function names in C++.

2) Legal.

In this case, the two **`sort`**'s are overloaded functions.

34) Legal

`using namespace std;` doesn't introduce any name into the scope.

3 **`p(x)`** is ok, due to ADL.

`p(x.y)` isn't, since ADL doesn't apply to built-in types.

4 Since **`-x`** is an rvalue, it can only be returned by value.

Corrected version

```
template<typename T>
T negate(T x) { return -x; }            // call by value
```

or

```
template<typename T>
T negate(const T& x) { return -x; }  // call by const reference
```

5 **`A::x`** isn't defined elsewhere in the program.

The ctor **`B::B()`** is private.

6 1) lvalue-to-rvalue, floating point promotion

2) Not viable – have to use call-by-const reference, since the actual parameter is of different type.

3) lvalue-to-rvalue, floating point promotion

1) and 3) are equally well, and the call is ambiguous.

7 a) `int (**)(int)`

b) `int (*)(int)`

8 a) doesn't match with **`std::max`**, i.e. it can't be obtained from **`std::max`** by substitution of **`T`**.

b) isn't declared inside the namespace **`std`**.

9 Only A ⇒ C It is a qualification conversion.

All the other five need **`const_cast`**. [Have to explain at least one of them.]

For example, B ⇒ C proceeds as follows

`int const** ⇒ int const*const* ⇒ int *const*`

Step 1 is a qualification conversion. Step 2 needs a **`const_cast`** to remove the red-colored **`const`** quailifier.

The remaining four are similar.

10
```
for_each(a,a+3,
[](int (&b)[4]){ for_each(b,b+4,[](int& x){ x++; }); }
);
```

or, a 3-point answer:

```
for_each(a,a+3,
[](int* b){ for_each(b,b+4,[](int& x){ x++; }); }
);
```

11 (1) `y%2==0? x+1: x`

(2) `reinterpret_cast<int*>(a)`

or

`reinterpret_cast<int(&)[12]>(a)`

(3) `reinterpret_cast<int*>(a)+12`

or

`reinterpret_cast<int(&)[12]>(a)+12`

(4) `add()`

12 (1) `typename iterator_traits<T>::value_type`

(2) `numeric_limits<U>::max()`

(3) `numeric_limits<U>::min()`

(4) `make_tuple(min,max,inmin,inmax)`

(5) `get<1>(minmax(a,6))`

13 120021

14
```
T** x=new (operator new(sizeof(T*),nothrow))
                T*(new (operator new(sizeof(T))) T(3));
```

15
```
bool(*)(int)
bool(&)(int)
decltype(prime)*          // watch the *
decltype(prime)&          // watch the &
function<bool(int)>
```
The last one may be replaced by
```
function<bool(const int&)>
function<const bool&(int)>
function<const bool&(const int&)>
```

16 (1) `s.push(atoi(p))`
(2) `bfn[index(p)](s.top(),v)`
(3) `strtok(nullptr," ")`
or
`strtok(NULL," ")`
(4) `s.top()`

17 (1) `stk[_top--]=n`
(2) `_top++`
(3) `stk[_top+1]`
(4) `const`

18 The ctor has to be redefined as
```
stack::stack() : _top(79),stk(*new int[1][80]) {}
```
or
```
stack::stack()
: _top(79),
  stk(reinterpret_cast<int(&)[80]>(*new int[80]))
{}
```
In either way, add the following dtor:
```
stack::~stack() { delete [] &stk; }
```