OOP Midterm solution

1 a) function-to-pointer conversion, pointer conversion, qualification conversion

```
void(const void*)
→ void(*)(const void*)      // function-to-pointer conversion
→ void*                     // pointer conversion
→ const void*               // qualification conversion
```

b) `void (*(&g())[1])()` // the name `g` may be omitted

c) The call `operator new[](3*sizeof(int))` allocates more storage than the call `operator new(3*sizeof(int))`. The extra storage is used to store the number of elements in the array by a `new` expression. However, the extra storage is useless in 2), as there is no `new` expression in 2).

d) The function returns a reference to a local temporary location that is no longer available after the function returns.
More precisely, the compiled code looks like

```
double const& f(int x) { const double tmp=x; return tmp; }
```

2 a) Template 2, since it is a specialist in `T*`.

b) Ambiguous
Template 1 uses call-by-value, and template 2 uses call-by-reference.
Both parameter-passing methods are equally well for the call `p(x)`.

c) All of 1), 2) and 3)
A class template explicit specialization may have a different set of class memebers from the generic class template.

d) Only 1)
A function template explicit specialization for `T` = `int` is obtained by substituting `int` for every occurrence of `T` in the signature of the function template.
For example, by substituting `int` for `T` in the underlined siganature

```
template<typename T> T f(T& x) { return x; }
```

we obtain

```
template<> int f<int>(int& x) { return x; }
```

In conclusion, a function template explicit specialization may not have a differenct signature format, but may have a different body, i.e. a different implementation (algorithm) of the function. For example,

```
template<> char f<char>(char& x)  // same signature format
{
    // do whatever you want
}
```

2 e) Only 3)

`template<typename T> void p(const T&) {}`

The declaration `const T` declares the type `T` to be a constant type.

For `T` = `const int*`, `T` is a pointer type. Thus, `const T` means the pointer type is a constant type.

Thus, the explicit specialization for `T` = `const int*` is

```
template<> void p(const int*const&) {}
```

3 a) The viable functions are

```
float A::f(float x);
double A::f(double x);          // 2
double f<double>(double x);     // 3
```

Since the ordinary function 2 is better than the instantiated function 3, the best viable function is

`double A::f(double x);`

b) Using the same argument as a), the compiler first resolves the call `f(7)` to a call to the instantiated function:

```
int f(int x)
{
    return numeric_limits<T>::is_integer? x: f(x); // 1
}
```

In a similar manner, the compiler then resolves the call `f(x)` in line one to be a recursive call. (Note that this recursive call will never be executed.)

Were `f(x)` replaced by `A::f(x)`, an ambiguity would occur. This is because the instantiated function is no longer a candidate, and both

```
float A::f(float x);
double A::f(double x);
```

are viable, but neither is better.

4 a) (1) `!comp(value,*it)&&!comp(*it,value)`

(2) `comp(value,*it)`

b) Given

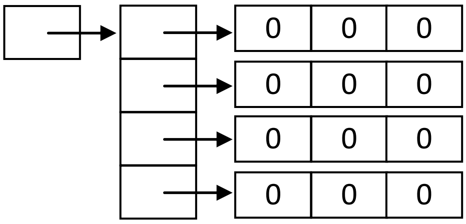
```
binary_search(c,c+6,"Yankees",less<const char*>())
bool binary_search(T*,T*,const T&,Compare)
```

the compiler first determines that `T` = `char*` from `c` and `c+6`, but then finds out `T` = `const char*` from `"Yankees"`.

This ambiguity can be easily resolved by type conversion

`const_cast<char*>("Yankees")` or `(char*)"Yankees"`

5 a)



b) It still works, but the benefit of memoization disappears. To see why, observe that each time the function **`c`** is called, it receives a zero-initialized vector. This means two things: firstly, the function works as if there is no **`cache`**; and secondly, since it is as if there is no **`cache`**, the problem of recomputation remains.

c) See lecture note

6 (1) `m!=n && n!=0`

(2) `s.push(m-1); s.push(n); m--; n--;`

or

`s.push(m-1); s.push(n-1); m--;`

(3) `n=s.top(); s.pop(); m=s.top(); s.pop();`

7 a) (1) `new int*[80]`

(2) `while (!empty()) pop(); delete [] stk;`

(3) `stk[++_top]=new int(n);`

or

`++_top; stk[_top]=new int(n);`

(4) `delete stk[_top--];`

or

`delete stk[_top]; --_top;`

(5) `return *stk[_top];`

or

`return stk[_top][0];`

b)
```
bool empty(const stack* this)
{
    return this->_top==-1;
}
```

8 a)
```
template<typename T>
T max(T* a,int n)
{
    return accumulate(a,a+n,numeric_limits<T>::min(),
                                        std::max<T>);
}
```
Note that the qualified name **std::max<T>** is necessary.

b)
```
char* h(char* r,char* s)
{
    if (strlen(s)%2==0) itoa(atoi(r)+1,r,10);
    return r;
}
```

9 a) **77 2 3**

b)
```
void* operator new(size_t sz,int* pool)
{
    static int offset=0;
    int old_offset=offset;
    offset+=sz;
    return reinterpret_cast<char*>(pool)+old_offset;
}
```
or
```
void* operator new(size_t sz,int* pool)
{
    static char* available=reinterpret_cast<char*>(pool);
    char* old_available=available;
    available+=sz;
    return old_available;
}
```
or,
```
void* operator new(size_t,int* pool)
{
    static int next=0;
    return pool+next++;
}
```
This last version isn't recommended, for it doesn't make use of the 1st parameter.