

CS202: Programming Systems

Week 10: Const-correctness

12/2016

Const-correctness

- When using the keyword **const** on a variable, it prevents this variable from being mutated.

Example: **const** int MAX = 100;

Const-correctness relates to type-safety

- When you declare `const` for parameters in a function, it makes the function safer by protecting arguments from being mutated unexpectedly.

E.g.:

```
void doSth(const string& s);
```

Understanding the const

What does it mean?

□ Case 1:

`const DataType* p;`

- p is a pointer to a const DataType
- p is pointing to an object of the class DataType. p could not be used to change the DataType object. But, p still can be NULL or point to somewhere else.
- If class DataType has a const function, say doSth(), it is ok to call p->doSth(). Otherwise, if doMutate() is not const, it is wrong to have p->doMutate()

Understanding the const

□ Case 2

`DataType* const p;`

- `p` is a const pointer to a `DataType` object.
- You could not change the pointer `p` but you can change the `DataType` data that `p` is pointing

□ Case 3

`const DataType* const p;`

- `p` is a const pointer pointing to a const object
- You could not change the pointer or the data

Understand the const

□ Case 4

const DataType& x;

- x is a reference to a constant DataType object.
- For example, calling x.getSth() is ok if getSth() is a constant function. Otherwise, x.doMutate() is not ok when doMutate() is not a constant function.

□ Case 5

DataType& const x;

- We don't have this due to the fact that reference is already constant!

Understanding the const

□ Case 6

`DataType const& x;`

- The same as `const DataType& x`

□ Case 7

`DataType const* x`

- The same as `const DataType* x`
- Don't mistype it as `DataType* const x`

Const member function

- It is a member function that inspects or reads the values rather than mutates its object.
- A **const** member function is known by a const suffix after the function's parameter list.
- E.g.:

```
void getSomething() const;
```


Example

Assuming that `getSth()` is a const function and `doMutate()` is a normal function.

```
void doSth(X& changeable, const X& unchangeable)
{
    changeable.getSth();    // OK: doesn't change a changeable obj
    changeable.doMutate();  // OK: changes a changeable obj

    unchangeable.getSth();  // OK: doesn't change an unchangeable obj
    unchangeable.doMutate(); //ERROR: attempt to change unchangeable obj
}
```

Return by reference in a const member function

- ❑ When you want to return a reference from a const member function: return reference-to-const

```
class Student {  
    public:  
        // Correct: the caller can't change the name  
        const string& getName() const;  
        // Wrong: the caller can change the name  
        string& getNameWrong() const;  
};
```

const function overloading

- ❑ You can have both a const member function and a mutator member function at the same time
- ❑ E.g.: The subscript operator often has both

```
const MyArr& operator[](unsigned index) const;  
MyArr& operator[](unsigned index);
```

Change inside a const member func

- ❑ When you want to change the members inside a constant member function, there are 2 ways:
- ❑ Keep the members as mutable by the keyword **mutable**
- ❑ Using `const_cast` for this
 - E.g.:
`MyClass* tmpPtr = const_cast<MyClass*>(this);`
 - `tmpPtr` will point to the same memory as of this pointer. It is a normal pointer rather than a `MyClass const *`