const Correctness in C^{++}

Partly taken from Items #3, #21, & #28
Effective $C^{++}(3^{rd}$ edition), by Scott Meyers

September 5, 2017

Brian A. Malloy

Go Back
Full Screen
Quit

1. Motivation

- Permits specification of semantic constraints
- Enlists aid of compiler to enforce constraint
- Enables communication with compiler & other programmers

1.1. Usage: Classes

- Can be used in classes for:
  1. pointers
  2. static or non-static data
  3. Function declarations: return value, params & whole fn
- Item 29: using const with a return value can make it possible to improve the safety and efficiency of a function that would otherwise be problematic.

1.2. Usage: Outside Classes

- global or namespace constants
- static objects




Go Back
Full Screen
Quit

2. const & Pointers

```
#include <iostream>

class A{
public:
  A(int n) : number(n) {}
  int getNumber() const { return number; }
  void setNumber(int n) { number = n; }
private:
  int number;
};
```

```
int main() {
  // non-const pointer, non-const data
  A *a = new A(17);

  // non-const pointer, const data
  const A *b = new A(18);
  b = a; // error: b->setNumber(17);

  // const pointer, non-const data
  A * const c = new A(19);
  c->setNumber(99); // error: c = b;

  // const pointer, const data;
  // can only call const functions
  const A * const d = new A(20);
  return 0;
}
```

3. const & Functions

- All functions can have const parameters
- All functions can return a const value
- For member functions, the whole function can be const

Go Back
Full Screen
Quit

4. const Parameters

The operator<< for string won't compile; but it's not because of operator<<

```
class string {
public:
  char* getBuf() { return buf; }
private:
  char *buf;
};
ostream&
operator<<(ostream& out, const string& s) {
  return out << s.getBuf();
}
```

5. const Return Value

Assignment to result of binary operators is not permitted for built-in types:

```
int i, j, k;
(i+j)=k; // this won't compile!
```

User-defined types s/ behave the same as built-in types. We can make this happen by making operator+ return const:

```
const string operator+(const char*) const;
const string operator+(const string&) const;
string a, b, c;
(a+b)=c;
```

5.1. const Return: Assignment

The following works for built-in types:

```
int i, j, k;
(i=j)=k; // this compiles fine
```

So we permit it for user defined types;

```
string& operator=(const string& rhs);
string a, b, c;
(a=b)=c;
```

5.2. Return value for []s

We can use *const* to preserve const-ness:

```
const char& operator[](int index) const;
char& operator[](int index);

string s1 = "Hello";
cout << s1[0]; // calls non-const []
s1[0] = 'x';   // writing non-const string

const string s2 = "World";
cout << s2[0]; // calls const []
s2[0] = 'x';   // error! Writing const string
```

5.3. By-value Return

It's never legal to modify the return value of a function that returns a built-in type. Thus, the following won't compile:

```
char operator[](int index) const;
s[0] = 'x';  // This won't compile!
```




Go Back
Full Screen
Quit

6. Need 2 []s

```
The following [] won't preserve const-ness:
char& operator[](int index);

string s = "I'm not constant";
s[0] = 'x';  // this is okay:s isn't const

const string cs = "I'm constant";
cs[0] = 'x'; // modifies the const string,
             // but compilers won't notice
```

6.1. Cannot distinguish r/l-value

```
const char& operator[](int index) const;
char& operator[](int index);
```

Why would you want to distinguish?
It's useful in reference counting because reads can be much less expensive to implement than writes.

To distinguish: use a proxy class.