CV-QUALIFIED TYPES

Good interface design

Florian Warg, Max Staff

May 2, 2017

MOTIVATION 1

```c
int min(int* arr, size_t len);
```

- think of a function that takes an array by pointer
- since it is called by reference, the function might alter the array
- how can we guarantee the user that we do not alter data?

MOTIVATION 2: COPY OPERATIONS

```
class Person {
private:
    string name;
    int age;
public:
    /* ... */
    string getName() { return name; }
};
```

- getters (accessors) should allow to read values of private members
- imagine ”name” to be a huge object (really long string)
- every time we check the value we would have to copy and return the whole string
- how can we reduce the overhead?

MOTIVATION 2: REDUCING OVERHEAD?

```
class Person {
private:
    string name;
    int age;
public:
    /* ... */
    string& getName() { return name; }
};
```

- we choose to return the private variable by reference
- overhead is reduced from copying the whole object to copying a constant size reference
- have we introduced a new problem?

MOTIVATION 2: VIOLATION OF OOP RULES

```
Person p("Alice");
cout << p.getName(); // print "Alice"
p.getName() = "";
cout << p.getName(); // print ""
```

- why do we use classes instead of structs?
- we use getters and setters so the object can validate changes to private members
- this example behaves like a struct with public members
- can we reduce overhead and still conform to oop rules?

THE CONST QUALIFIER

- we can mark variables as constant (const)
- const variables can not be modified
- trying to modify const vars will cause a compiler error

```
const T name; // name is const obj of type T
T const name; // name is const obj of type T

const T* name;  // ptr to const T
T const* name;  // ptr to const T

T* const name;  // const ptr to T

const T* const name;  // const ptr to const T
T const* const name;  // const ptr to const T
```

- references have the same rules as pointers

- we can modify example 1 to make strong guarantees about the behavior of the function without exposing implemetation

```
/* old */
int min(int* arr, size_t len);
/* better */
int min(const int* arr, size_t len);
```

- what is const here? do we need more consts?

- we use const in example 2 to prevent modification of private variables through getter methods

```
/* old */
string& Person::getName() { return name; }
/* better */
const string& Person::getName() { return name; }
```

DECLARING CONSTANT VALUES

- const can also be used to declare constants in code

```c
/* bad because not type-safe */
#define PI 3.1415926
/* better */
const double PI = 3.1415926;
```

- those constants can also be used for array sizes

```c
size_t n = 32;
int arr[n]; // only works with VLA

const size_t N = 32;
int arr2[N]; // works without VLA
```

MAKING NON-CONST THINGS CONST

- the compiler can implicitly cast objects to const objects
- although it does not work the other way around
- the compiler can always make things "more const"

```
void f(const int n) { /* ... */ }
void g(int n) { /* ... */ }
int x = 42;
const int y = 42;
f(x); // int -> const int
f(y); // const int -> const int
g(x); // ERROR - cannot treat const as non-const
g(y); // const int -> const int
```

- `const_cast<T>()` can be used to remove const (please do not use this, yet)

CONST IN CLASSES - ATTRIBUTES

- remember, const values cannot be changed
- remember, attributes have to be constructed before we enter the ctor

```
struct X {
    const int N;
};
/* error because no default ctor */
```

CONST IN CLASSES - ATTRIBUTES

- use initializer list to set constants
- default values in the class definition are also possible

```
struct X {
    const int N;
    X() { N = 42; }
};
/* compiler error */

struct X {
    const int N;
    X() : N(42) {}
};
/* OK */
```

- we can make methods in classes const, too
- const methods can not modify state of the object
- const methods can be called on const objects

```
class Rectangle {
private:
    /* ... */
    void setWidth(double width);
    double getWidth() const { return width; }
};
Rectangle r1;
const Rectangle r2(2, 4);
r1.setWidth(2); // OK
r2.setWidth(2); // error - r2 is const
r1.getWidth();  // OK
r2.getWidth();  // OK - function is const
```

GENERAL RULES FOR CONST

- use const as much as possible where it makes sense
- use const to define type-safe constant values
- getters for primitive types return by value
- getters for non-primitive types return by const reference
- accessor methods can be constant functions
- express read-only intent of functions with const reference arguments

VOLATILE QUALIFIER

- prevent optimization for variable
- usually only used for low-level access (OS) or embedded systems

```cpp
volatile int x = 42; // prevent optimization
cout << x;
// cannot optimize to cout << 42
// because x might have been changed
```

MUTABLE QUALIFIER

- mutable attributes of a const object can still be changed
- it is generally a bad idea to mess with constness
- mutable has almost no applications besides debugging

```
struct ShopItem {
    int id;
    string name;
    mutable int priceCents;
};

const ShopItem item = { 0, "Club-Mate", 120 };
item.id = 1; // ERROR
item.priceCents = 140; // OK
```