

# C++ Programming

Const Correctness, Part 2

# const and Functions

- const return values
- const parameters
- const functions

# const and Functions: Return Values

- Prevents the caller from modifying the data returned
- Useful for efficient encapsulation


```
const int sum(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    int i = sum(3,4);  
}
```

# const and Functions: Return Values

- Does not apply/make sense when returning by value
- const data returned by value can be assigned to non-const variables

```
class Person {  
    public:  
        const Person clone() {  
            return *this;  
        }  
};  
int main() {  
    Person p;  
    Person q = p.clone();  
}
```

*copy by default.*



- Works because we are not assigning the original const object – we are assigning a copy

# const and Functions: Return Values

- For data returned by pointer or reference, we almost always want to return it const

```
class Person {  
    public:  
        Person(string name) : _name(name) { }  
        string& name() { return _name; }  
    private:  
        string _name;  
};
```

```
int main() {  
    Person p("Jeff");  
    p.name() = "Joe"; // Bad for encapsulation!  
    cout << p.name() << endl;  
}
```

*name is return by reference.*

*and as it is passed by reference, anything inside would be changed without involving the class*

# const and Functions: Return Values

- Better:

```
class Person {  
    public:  
        Person(string name) : _name(name) { }  
        const string& name() { return _name; }  
        void name(const string& name) { _name = name; }  
    private:  
        string _name;  
};  
int main() {  
    Person p("Jeff");  
    p.name() = "Joe";    // Compilation error  
    p.name("Jim");       // Valid  
}
```

← So the string cannot be modified  
inside of the function.

update by passing  
parameters.

← in this case, you could only modify  
name using specific method.

# const and Functions: Parameters

Can pass by value, pointer, or reference

- Again, const makes a promise that the function will not modify the passed parameter *pass a copy of parameter instead.*
- const with pass by value not very useful/meaningful
- const with pointer passing also not very useful/meaningful
- Some people debate this, however ...

# const and Functions: Parameters

Pass by const reference

- Efficient – no copy needed, as with any pass by reference parameter
- Passing const means function cannot change the referenced object
  - Exactly the same as pass by value semantically
  - Cannot modify or reassign the variable/object
  - Can only call const functions if it is an object (coming up in a few slides)
- Generally, this is preferred



# const and Functions: Parameters

- Another advantage of accepting const reference parameters: we can accept temporaries
  - Temporary objects are always const
  - We cannot pass temporaries to a function that takes a pointer

```
void printName(string& name) {  
    cout << name << endl;  
}  
int main() {  
    printName("Joe"); // compilation error!  
}
```

*create a temp string - but fail since type mismatch.*

# const and Functions: Parameters

```
void printName(const string& name) {  
    cout << name << endl;  
}  
  
int main() {  
    printName("Joe"); // works!  
}
```

*as the temp is const type, it is fine now.*

# `const` and Functions: Functions

- A `const` member function does not modify the object it is called on
- In other words, it does not modify the `*this` object

# const and Functions: Functions

- As discussed earlier, when we create a function in a class, the compiler quietly inserts a `this` parameter as the first parameter in the function signature

```
class Person {  
    const string& name();  
};
```

essentially compiles to

```
class Person {  
    const string& name(Person *const this);  
};
```

*const pointer to person.*

*you cannot change where it's pointing at*

*But you can change the pointed obj.*

# const and Functions: Functions

- When we call a function on an object, the compiler silently passes a pointer to the object as the first parameter

```
Person p;  
cout << p.name();
```

essentially compiles to

```
Person p;  
cout << name(&p);
```

# const and Functions: Functions

- We can declare a function to be const, in which case \*this will also be made to be const

```
class Person {  
    const string& name() const;  
};
```

essentially compiles to

```
class Person {  
    const string& name(const Person* const this);  
    // Compared to the non-const version:  
    //  const string& name(Person* const this);  
};
```

# const and Functions: Functions

- Because `*this` is `const`, we cannot change/reassign the private data of the `*this` object

```
class Person {  
    const string& name() const  
    {  
        // compilation error -- '*this' is const  
        this->_name = "Joe";  
        return this->_name;  
    }  
};
```

# const and Functions: Functions

- Cannot call non-const functions within the function on `*this`
  - Even if those functions don't actually change the receiver
- Reference and pointer return types must be declared const if a function is declared const *i.e. for all getter methods*

```
class Person {  
    int age() {  
        return this->_age;  
    }  
    const string& name() const {  
        int age = this->age(); // compilation error  
        return this->_name;  
    }  
};
```



# const and Functions: Functions

- A `const` function offers a guarantee that `*this` object won't be changed by calling the function on it
  - You can always call a `const` function
  - You can only call a non-const function on non-const objects
- Generally, accessor functions should be declared `const`

# const and Functions: Functions

```
class Person {  
    public:  
        int age() {  
            return this->_age;  
        }  
        const string& name() const {  
            return this->_name;  
        }  
    private:  
        string _name;  
        int _age;  
};
```

```
int main() {  
    Person p;  
    const Person q;  
    p.name();           // valid  
    p.age();            // valid  
    q.name();           // valid  
    q.age();            // compilation error  
}
```

# const and Functions: Functions

- What does it mean for a member function to be const?
- Two camps:

1. Bitwise constness: *strict vs default* Does not modify any of the bits in the object (this is the C++ compiler's definition)
2. Conceptual constness: Can modify the object, but only in ways that are undetectable to clients

# const and Functions: Functions

- Will it compile? *yes -*

*we're not changing the pointer*

*we're changing the pointed object.*

```
class Person {  
public:  
    Person(const string& name, Person* bff) : _name(name), _bff(bff) {  
    }  
    void name(const string& name) {  
        this->_name = name;  
    }  
    void changeBFFName(const string& name) const {  
        _bff->name(name);  
    }  
private:  
    string _name;  
    Person* _bff;  
};
```

*what we're actually  
change is what bff  
pointing at.*

```
int main() {  
    Person p("Joe", NULL);  
    Person q("Jen", &p);  
  
    q.changeBFFName("Bob");  
}
```

# `const` and Functions: Functions

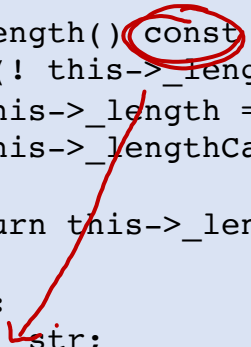
- This leads to the notion of *conceptual constness*
- Adherents to this philosophy argue that a `const` function should be able to modify the bits of the receiving object, but only in ways undetectable by clients

# `const` and Functions: Functions

- Example: the `length ( )` function of a `String` class
  - We want it to be `const` so that it can be called on both `const` and non-`const` `String` objects
  - We also want it to be `const` because, semantically, a `length ( )` function does not change the object on which it is invoked
  - However, we might want to cache the length to improve efficiency on later calls to `length ( )` ...

# const and Functions: Functions

```
class String {  
    public:  
        String(const char* s) : _lengthCached(false) {  
            // ...  
        }  
        int length() const {  
            if (! this->_lengthCached) {  
                this->_length = strlen(_str);    // compilation error!  
                this->_lengthCached = true;        // compilation error!  
            }  
            return this->_length;  
        }  
    private:  
        char* _str;  
        bool _lengthCached;  
        int _length;  
};
```



# `const` and Functions: Functions

- What can we do?
  - We want the function to be `const` , but we also want to be able to cache the length
  - That is, we want *conceptual constness* – we argue that the function should be able to change some bits of the object, but only in ways that are imperceptible to the client
- Fortunately, the `mutable` keyword was added to the C++ standard for just this purpose
- `mutable` members can be modified in `const` functions



# const and Functions: Functions

```
class String {  
    public:  
        String(const char* s) : _lengthCached(false) {  
            // ...  
        }  
        int length() const {  
            if (! this->_lengthCached) {  
                this->_length = strlen(_str);    // hurray!  
                this->_lengthCached = true;        // hurray!  
            }  
            return this->_length;  
        }  
    private:  
        char* _str;  
        mutable bool _lengthCached;  
        mutable int _length;  
};
```

*it works since  
these variables are  
mutable.*


# `const` and Functions: Functions

- `const` member functions specify which member functions may be invoked on `const` objects
- Member functions differing only in their constness can be overloaded.

# const and Functions: Functions

```
class String {  
    public:  
        // operator[] for non-const objects  
        char& operator[](int position) {  
            return this->_str[position];  
        }  
        // operator[] for const objects  
        const char& operator[](int position) const {  
            return this->_str[position];  
        }  
    private:  
        char* _str;  
};
```

```
String s1 = "Hello";  
const String s2 = "World";  
  
cout << s1[0]; // calls non-const String::operator[]  
cout << s2[0]; // calls const String::operator[]
```



# const and Functions: Functions

- By overloading operator[ ], we can have const and non-const Strings handled differently:

```
String s = "Hello";

cout << s[0];      // ok, reading a non-const String
s[0] = 'x';        // ok, writing to a non-const String

const String cs = "World";

cout << cs[0];     // ok, reading a const String
cs[0] = 'x';       // compilation error!
```

# const Correctness

- If
  - a pointer or reference parameter is not modified,
  - a pointer or reference member is returned,
  - a member function does not change member data,
- then it should be declared `const`

# const Correctness

- Rule of thumb for const functions:

- Start by making your functions const

- Remove constness as needed

*try to making as const-friendly  
as possible.*

# const Correctness

- Widely used in libraries
- Documents the properties of your code
  - Compiler enforces them
- Assists in system robustness
- Can allow the compiler to optimize in some instances