The `const` Keyword

Extreme Encapsulation

Humble Beginnings

- There are often cases in coding where it is helpful to use a `const` variable in a method or program.
  - Even when working with fixed values, it is best to abstract them with variable names.
  - It's more helpful (debugging-wise) to see "array_length" than tons of copies of the same number everywhere.

Humble Beginnings

- With object-orientation, many classes may take permanent values which are unique to each instance of the class, specified during initialization.
  - As these should not change at any point in the object's lifetime, `const` makes sense.

Humble Beginnings

- The use of `const` is fairly straightforward for the primitive data types – the basic building blocks of the language.
- Things get more complicated when we use `const` with pointers and with objects.

`const` and Objects

- What would it mean for an object to be `const`?

`const` and Objects

- What would it mean for an object to be `const`?
  - If declared `const`, an object should not be modifiable.
  - Problem: how can we use its methods while being sure not to modify it?

`const` and Objects

- In C++, whenever a variable is declared `const`, no modifications are allowed to it, *in a by-value manner*.
  - As the compiler is not powerful enough to ensure that its methods do not modify it, by default C++ blocks all use of *any* class methods.
  - This would be a **huge** problem for encapsulation.

`const` and Objects

- The C++ solution to the problem this poses: *functions* can be declared `const`.
  - Appending the `const` keyword to a function signifies that the method is not allowed to alter the class in any manner.
  - Inside that method, *all* fields of the class will be treated as if they were declared `const`.

`const` and Objects

- Let us now examine how this would look in code, through our frequent `Person` class example.

A First Object

```
public class Person
{
    private:
    const string name;
    int age;

    public:
    Person(string name, int age)
    string getName() const;
    int getAge() const;
    void haveABirthday();
}
```

A First Object

```
string Person::getName() const
{
   return this->name;
}

int Person::getAge() const
{
   return this->age;
}
```

A First Object

```
public void haveABirthday()
{
   this->age++;
}
```

- Note: declaring this method as `const` would result in a compile-time error, as `age` would be treated as `const` within the method.

`const` and Objects

- Which of the following code lines is invalid?

```
const Person p(“Harrison Ford”, 73);

string name = p.getName();
int age = p.getAge();
p.haveABirthday();
```

`const` and Objects

- Which of the following code lines is invalid?

```
p.haveABirthday();
```

- As this method is not declared `const`, a compile-time error would result from this method being called upon `const p`.

`const` and Pointers

- When we add pointers into the mix, things get even more interesting.
  - What might we wish to be constant?
  - The stored address / pointer
  - The referenced value

`const` and Pointers

- In order to have a `const` *pointer* to a changeable value, use the following syntax:
  - `int* const myVariable;`
- To allow the stored address to be replaced, but have the referenced *value* be otherwise unchangeable:
  - `const int* myVariable;`

`const` and Pointers

- Using the syntax below, while `obj` is declared by-reference, the compiler will block any attempts to modify its contents:
  - `const Object* obj;`
  - The referenced object `obj` is considered constant.

`const` and Pointers

- The simplest way to think of it – read `const` definitions from right to left.
  - `int* const myVariable;`
  - `const int* myVariable;`
  - When `const` is fully on the left, it modifies the direct right instead.
  - `int const* myVariable;`
    - Is the same definition, with different ordering.

`const` and Pointers

- While very powerful, const syntax can get rather crazy:
  - `const Object* const obj;`
  - Translation:
  - `const Object* const obj;`
  - A const reference…
  - `const Object* const obj;`
  - to a const Object.

`const` and Pointers

- Similar rules apply to arrays.
  - The following may store a constant reference to an array with changeable values:
  - `int* const myVariable;`
  - The following may store a replaceable reference to arrays whose values are treated as `const`:
  - `const int* myVariable;`

`const` and Pointers

- Example:

```
int* initArray = new int[6];
int* const myVariable = initArray;
myVariable = new int[3];
  //Above: Not legal
  - myVariable[2] = 3; // Legal!

  - …
```

`const` and Pointers

- Example:

```
int* initArray = new int[6];
const int* myVariable = initArray;
myVariable = new int[3];
   //Above: Legal, not initialized
   - myVariable[2] = 3; //Illegal!
```

…