**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:
  
  ```c
  int* const myVariable;
  ```

- To allow the stored address to be replaced, but have the referenced value be otherwise unchangeable:
  
  ```c
  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.
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:

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

- Example:

```cpp
int* initArray = new int[6];
const int* myVariable = initArray;
myVariable = new int[3];

// Above: Legal, not initialized
- myVariable[2] = 3;  // Illegal!
```

...
const and Parameters

• Suppose a method is defined as follows:

```cpp
void someMethod(const Object* obj)
```

• What implications would this have?
void someMethod(const Object* obj)

• What implications would this have?
  – As obj is defined const Object*, we would get a pointer to an unmodifiable instance of Object.
  – What are we able to pass in as an argument to obj?
void someMethod(const Object* obj)

• Which of these would be proper calls?
  – const Object obj();
    someMethod(&obj);
  – Object* obj = new Object();
    someMethod(obj);
const and Parameters

void someMethod(const Object* obj)

• Which of these would be proper calls?
  – const Object obj;
  someMethod(obj);
  – Object* obj = new Object();
  someMethod(obj);

Trick Question! Both!
const and Parameters

```c
void someMethod(const Object* obj)
```

- While the original argument to methods of this form do not have to be `const`, they become `const` within the method.
const and Parameters

```c
void someMethod(Object* obj)
```

- Which of these would be proper calls?

  - `const Object obj();
    someMethod(&obj);`

  - `Object* obj = new Object();
    someMethod(obj);`
void someMethod(Object* obj)

• Which of these would be proper calls?
  - const Object obj();
    someMethod(&obj);
  - Object* obj = new Object();
    someMethod(obj);
**const** and Parameters

- **const** objects cannot be passed by-reference to non-**const** function parameters.
  - As there is no guarantee that the referenced object will not be modified when passed to a non-**const** parameter, the compiler blocks this.
  - For value types, since a separate value is created, that separate copy is safe for the called function to edit.
const and Parameters

• const objects cannot be passed by-reference to non-const function parameters.
  – An interesting consequence of this:

```cpp
void someMethod(string &str);  
someMethod(string(“Hello World”));  
// Will be a compile-time error  
// due to the compile-time constant.
```
**const and Parameters**

- `const` objects cannot be passed by-reference to non-`const` function parameters.
  - An interesting consequence of this:

```cpp
void someMethod(const string &str);

someMethod(string("Hello World"));
// Will work without issue!
```
**const and Parameters**

- A signature of the latter type – 
  ```
  void someMethod(const string &str)
  ```
  has one additional benefit.
  - Since `str` is passed by reference here, the system doesn’t have to copy its value...
  - And since `str` is declared `const`, its value cannot be changed.
const and Parameters

• A signature of the latter type –

```c
void someMethod(const string &str)
```

has one additional benefit.

– Consider if this were a very large string.
– Or, just some very large object.
– This makes the program more efficient in terms of run-time and in terms of memory use.
**const and Return Values**

- **const** may also be applied to return values!
  - Consider if we were to return a reference to an object’s internal field.
  - Rather than copy the internal object, we may wish to return it while blocking write access within the object.

Example: `const Object* gimmeObject();`