Introduction to "const"

Here we describe how to use the `const` keyword to prevent unwanted changes to variables or objects.

The `const` keyword is used to inform the compiler that something (a variable or object) should not change once it has been defined. Since the compiler enforces this *constant* behavior, checks to make sure constant entities are not changed are made at compile-time, rather than when the program runs.

Constant Variables

Sometimes it is useful to mark variables as being constants.

For example, some named values are always meant to be fixed:

```
const float PI = 3.14159;
```

and some are used in contexts that require that they are fixed:

```
const int NUM_SCORES = 10;
double scores[NUM_SCORES];  // array size must be known
                            // (i.e., fixed) at compile-time!
```

Constant Pointers

Note that with pointers, the `const` keyword can go in two places. Either before the type:

```
int i = 7;
int j = 8;
const int *ip = &i;
```

meaning that we are not allowed to change the value of the pointed-to item:

```
*ip = 10;  // Wrong!
```

but that we can change *what* the pointer points to:

```
ip = &j;   // Ok!
```

Or, the `const` keyword may go after the `*`:

```
int i = 7;
int j = 8;
int * const ip = &i;
```

in which it means we can change the value of the pointed to item:

```
*ip = 10;  // Ok!
```

but cannot change the address the pointer holds:

```
ip = &j;   // Wrong!
```

Moreover, we can use `const` in both places at the same time:

```
int i = 7;
int j = 8;
const int * const ip = &i;
*ip = 10;  // Wrong!
ip = &j;   // Wrong!
```

Constant References

As with pointers, we can use the `const` keyword with references:

```
int i = 7;
int j = 8;
const int &ir = i;
```

Aside: Remember, a reference is really just a pointer with which you don't use `&` and `*`.

---

meaning that we cannot change the thing it refers to:

```
ir = 10;  // Wrong!
```

It is pointless to put `const` after the `&` since references can't refer to other items after they are defined:

```
ir = &j;  // Wrong--tries to assign an address
          // to the int ir refers to.
```

Constant Arrays

An array may also be made `const`:

```
const int a[] = { 1, 2, 3, 4 };
```

which means that the elements cannot be changed:

```
a[0] = 6;  // Wrong!
```

With arrays, the `const` keyword may also be placed after the type of elements:

```
int const a[] = { 1, 2, 3, 4 };
```

although it means the same thing:

```
a[0] = 6;  // Wrong!
```

Aside: We will prefer to put `const` before the type name.

Constant Parameters

Often times, functions (or methods) will take an argument via a pointer or reference:

```
foo(char *s);
void String::append(String &str);
```

Doing so is more efficient than *pass by value*. This is because the address passed is often smaller than the whole object. (In fact, pass by value isn't even available for arrays.)

Nonetheless, pointers or references allow a function to change the argument passed to it, whether the function should do so or not.

To prevent changing arguments that should not be changed, these pointers or references should be declared `const`:

```
foo(const char *s);
void String::append(const String &str);
```

By doing so, we retain the *efficiency* of pointers and references, and prevent the *error* of accidentally changing arguments.

Constant Methods

Those methods of a class that do not change the object should be marked as `const`:

```
class Point
{
public:
  Point();
  Point(int xval, int yval);
  void move(int dx, int dy);
  int get_x() const;
  int get_y() const;
  int foo() const;

private:
  int x, y;
};

int Point::get_x() const
{
  ...
}
```

which must be done both in the class definition and the method definition(s).

Aside: We will refer to methods that are not marked as `const`, like `Point::move()`, as *non-constant* methods.

Marking methods as `const` allows the compiler to check whether calling a particular method on a *constant* object is a valid thing to do. For example,

```
Point pt1(1, 2);
const Point pt2(3, 4);
const Point &pt3 = pt1;
const Point *pp = &pt1;
int x, y;

x = pt1.get_x();  // All fine, pt1 is not constant.
y = pt1.get_y();
pt1.move(4, -8);

x = pt2.get_x();  // Fine, doesn't change object
y = pt2.get_y();  // Fine, doesn't change object
pt2.move(4, -8);  // Won't work!  Can't change
                  // (i.e., move) constant

x = pt3.get_x();  // Fine, doesn't change object
```

```
y = pt3.get_y();  // Fine, doesn't change object
pt3.move(4, -8);  // Won't work!  Can't change
                  // (i.e., move) constant

x = pp->get_x();  // Fine, doesn't change object
y = pp->get_y();  // Fine, doesn't change object
pp->move(4, -8);  // Won't work!  Can't change
                  // (i.e., move) constant
```

Marking a method as `const` also tells the compiler the method cannot change the object's data members and cannot call non-constant methods on the object:

```
int Point::foo() const
{
  x++;     // Wrong!
  move();  // Wrong!
}
```
