YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 223b: Data Structures and Programming Techniques *Handout #13*
Professor M. J. Fischer *March 2, 2008*

Using `const` in C

1 Type qualifiers

A type specifier such as `int` can be qualified with one or more type qualifiers: `const`, `volatile`, `restrict`. The type specifier and qualifiers can be written in any order, so

```
const int volatile
int volatile const
volatile const int
```

all mean the same thing. `const` makes a variable read-only. `volatile` and `restrict` help the compiler generate correct and/or more efficient code in special circumstances. We focus on `const` in the remainder of this document, although much of what we say holds for the other qualifiers as well.

The type qualifiers all refer to lvalues. That is, they affect the variables that are used to hold values, not the values themselves. A variable declared `const` is read-only, i.e., values cannot be assigned to it after initialization. This makes the variable into a constant; hence the name `const`. For example, after writing

```
const double pi = 3.14159265358979323846;
```

one can use `pi` in expressions but not assign to it.

One could achieve the same effect in this simple case by writing

```
#define PI 3.14159265358979323846
```

However, these two methods of defining constants are not at all the same. In the first case, `pi` is like a normal variable except that it is read-only. In the second case, `PI` is replaced by the actual numeric literal. One can construct a reference to `pi` but not to `PI`. Thus, `&pi` is a reference to `pi`, whereas `&PI` results in a syntax error.

The type of `&pi` is "reference to a variable of type const double". A pointer variable `q` suitable for storing this reference could be declared

```
const double * q;
```

As usual with pointer variable declarations, the `*` between a type specification T and a variable name v means that v is a variable that can store references to other variables having declared type T. After having declared q in this way, one could store the reference `&pi` into `q` with the assignment statement

```
q = &pi;
```

Note that this assignment is legal even though `q` has type `const double *`. The `const` qualifier in the declaration of `q` applies to the thing that `q` points at, not to `q` itself. The effect of that `const` is to prevent assignments such as

```
*q = 2.7182818284590452354;
```

which otherwise would happily store the value of the mathematical constant e into `pi`.

Suppose we have another pointer variable like `q` but without the `const`:

```
double * r;
```

The compiler will complain if you write

```
r = &pi;
```

by giving a warning, "assignment discards qualifiers from pointer target type". Although nothing has been done yet to violate the integrity of the value of `pi`, `r` does not promise that the pointers stored into it necessarily point to read-only objects, so assignments to those objects must be considered legal by the compiler, e.g.,

```
*r = 2.7182818284590452354;
```

This of course would change (or attempt to change) the value of `pi`, violating the intended restriction on `pi`.

Suppose one wants a read-only variable s that points to a constant `pi`. Here's how it could be declared:

```
const double * const s = &pi;
```

The `const` to the left of the `*` applies to the target of the pointer, whereas the one to the right refers to the pointer variable itself. The equal sign introduces the initializer and makes `s` initially point to `pi`. Since `s` itself is read-only and contains a pointer to a read-only variable, neither of the following assignments are legal:

```
s = q;
*s = 2.7182818284590452354;
```

2 Type compatibility

We said that `const` only affects lvalues. The `const` qualifier is removed from the type when fetching a value from a variable of a `const` type. Thus, the value fetched from `pi` has type `double`, *not* `const double`, and can therefore be stored in an ordinary variable of type `double`, e.g.,

```
double x;
x = pi;
```

stores 3.14159... into `x`. `x` itself is not read-only, so `x++`, for example, changes the value in `x` to 4.14159....

The other place where `const` and non-`const` types mix is with pointers. It is always allowed to store a pointer to a non-`const` type into a pointer variable of the corresponding `const` type. For example, using the definitions above,

```
const double * q;
double * r;
```

the assignment

```
    q = r;
```

is legal, but

```
    r = q;
```

is not, giving the same "assignment discards qualifiers" warning that we saw above. `q` promises not to tamper with the target of any pointer that is put there, whereas `r` has made no such promise. Hence, if `q` gives its value to `r`, it is implicitly violating its no-tamper promise since `r` has made no such guarantee.

One often sees function parameters that are pointers to `const` objects. It tells both the compiler and the programmer that those objects are not affected by the function call, enabling better optimization, and giving the programmer further assistance at avoiding coding errors. For example, the standard function `fopen()` has the prototype

```
    FILE *fopen(const char *path, const char *mode);
```

It promises not to change either of the strings pointed at by its two arguments. However, it is perfectly all right to call it with actual arguments that are not declared `const`, e.g.,

```
    FILE* in;
    char buf[100];
    // ... code to store a string into buf goes here  ...
    in = fopen( buf, "r" );
```

3 Using `const` with `typedef`

The use of `const` with type names defined by `typedef` can be confusing if one forgets that `typedef` defines a complete type, not a type fragment. Here's an example.

```
    typedef double * Dptr;
```

defines a new type `Dptr` whose values are references to doubles. It can be used just like any other type name to declare a variable `r`, e.g.,

```
    Dptr r;
```

The resulting type of `r` is `double *` as in the examples above. Thus, `Dptr` is just a name for `double *` and is not a distinct type.

However,

```
    const Dptr q;
```

does *not* give `q` the type `const double *`. The reason is that the definition of `Dptr` is not simply substituted for the name. Rather, the rules for types are applied to `Dptr` just as they are for any other type. We saw before that the qualifier `const` in

```
    const double pi;
```

applies to the variable `pi`, making it read-only. The same thing happens with

```
    const Dptr q;
```

The variable `q` is made read-only, and it takes values of type `Dptr`, which are references to read/write doubles. Moving the qualifier to the other side of the type name does not change the meaning, so

```
Dptr const q;
```

still means exactly the same thing.

How can we say that we want the thing that a `Dptr` value references to be read-only? We must define a new type name:

```
typedef const double * const_Dptr;
```

Now, we can get the effect we want for `q` by writing

```
const_Dptr q;
```

Similarly, using these new type names, we could declare and initialize our variable `s` above by writing

```
const const_Dptr s = &pi;
```

In this case, one might prefer to write the `const` and the type name in the opposite order, viz.

```
const_Dptr const s = &pi;
```

This still means the same thing—that `s` is read-only, and the value that it stores is a pointer to a read-only `double`.

Because this situation occurs frequently in the paradigm for defining abstract data types in C using opaque types, you will frequently see definitions such as

```
// Opaque type definitions
typedef struct player* Player;
typedef const struct player* const_Player;
```

Here, the base type is `struct player` (corresponding to `double` in the above examples), `Player` is a reference to a read/write variable of the base type, and `const_Player` is a reference to a read-only variable of the base type.