The 'const' Keyword

Review: const Keyword

Generally, the keyword *const is* applied to an identifier (variable) by a programmer to express an intent that the identifier should not be used to alter data it refers to in some context (scope).

The compiler enforces this intent, for example:

```
int main()
{
  const int x=5;
  x=2;
}
```

```
small@shepherd:~/examples$ g++ test.cpp
test.cpp: In function ‘int main()’:
test.cpp:4: error: assignment of read-only variable ‘x’
```

In the above example, x is an identifier that refers to integer data in memory and the compiler enforces that x should not be used in a way that could result in a change to that data.

Another example:

```
int main()
{
  const int x=5;
  int *x_ptr;
  x_ptr = &x;
}
```

```
test.cpp: In function ‘int main()’:
test.cpp:5: error: invalid conversion from ‘const int*’ to ‘int*’
```

In this example, the compiler does not allow the 'int *' x_ptr to point to the address of x (which is of type 'const int *'). This is because x_ptr could then be used to change the data stored for x.

Review: const parameters

Recall that there are two methods of passing arguments to functions in c++:

By Value – A copy of the argument is made and the function acts upon the copy

By Reference – No copy of the argument is made, the function parameter is an identifier that refers to the same data as the argument (like an alias for the argument).

The method used to pass arguments is indicated by the function's parameters:

```
void foo(int x);  //x is passed by value
void foo(int &x); //x is passed by reference
```

and can add *const* to either:

```
void foo(const int x);  //the copy x cannot be altered (no use for this)
void foo(const int &x); //the data identifier x refers to cannot be altered by function foo
```

Passing by const reference can be very useful to the runtime overhead of copying data while still assuring that a function will not change the original.

Passing Objects By const Reference

Objects can also be passed by const reference to avoid copy overhead:

```
friend Fraction Add(const Fraction& f1, const Fraction& f2);
```

Just like with other types, the compiler will enforce that an object passed by const reference will not be used in a way that may change it's member data.

Const Member Functions

Remember that any call to a member function has a “calling object”

```
Fraction f1;     /* instantiate a fraction object */
f1.evaluate();  /* f1 is the calling object */
```

Since a member function has access to the calling objects data, we may want to make sure the calling object is never altered by a member function.

We call this a **const member function**, and it is indicated by using the *const* keyword after the member function declaration AND definition.

```
1  #include <iostream>
2
3  class IntHolder{
4  public:
5    IntHolder(int x);
6    void Illegal() const;
7  private:
8    int data;
9  };
10 IntHolder::IntHolder(int x) {
11   data = x;
12 }
13
14 void IntHolder::Illegal() const {
15   data = 1;
16 }
17
18 int main()
19 {
20   IntHolder myInt(5);
21 }
22
```

```
small@shepherd:~/examples$ g++ test.cpp
test.cpp: In member function ‘void IntHolder::Illegal() const’:
test.cpp:15: error: assignment of data-member ‘IntHolder::data’ in re
ad-only structure
```

Declaring const Objects

Declaring primitive type variables as *const* is easy. Remember that they must be initialized on the same line:

```
const int SIZE = 10;
const double PI = 3.1415;
```

Objects can be declared as const oin a similar fashion. The constructor will always run to intialize the object, but after that, the object's member data cannot be changed

```
const Fraction ZERO;        // this fraction is fixed at 0/1
const Fraction FIXED(3,4); // this fraction is fixed at 3/4
```

To ensure that a const object cannot be changed, the compiler enforces that **a const object may only call const member functions**.

See const_fraction example.

Const Member Data

Member data of a class can also be declared const. This is a little tricky, because of certain syntax rules.

Remember, when a variable is declared with const in a normal block of code, it must be initialized on the same line:

```
const int SIZE = 10;
```

However, it is NOT legal to initialize the member data variables on their declaration lines in a class declaration block:

```
class Thing
{
public:
   Thing();          /* constructor -- intialize member
                        data in here                  */
private:
  int x;             /* just declare here             */
  int y = 0;         /* this would be ILLEGAL         */
  const int Z = 10; /* would also be ILLEGAL          */
};
```

- But a const declaration cannot be split up into a regular code block. This attempt at a constructor definition would also not work, if Z were const:

```
Thing::Thing() {
  Z = 10;
}
```

Initialization List

We can use a special area of a constructor called an **initialization list** to overcome the problem of initializing const object members.

Initialization lists have the following format:

```
classname::classname(p1, p2) : member_var1(initial_val1), member_var2(p1) {
  // constructor body
}
```

The initialization list above will set member_var1 to 10 and member_var2 to the value passed as p1 to the constructor.

See init_list.cpp example.