Review

- What does the friend keyword in a class do?
- Is a friended function a public or private member?
- What is the difference between these two calls:

  f3 = f1.Add(f2)

  f3 = Add(f1, f2)
- Define conversion constructor.
- What does it do?
- How do we suppress what is does?

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 (t1.cpp):

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

- 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 (t2.cpp):

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

- 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

- There are two methods of passing arguments to functions in c++:
  - **Pass-by-value** – A copy of the argument is made and the function acts upon the copy
  - **Pass-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
```

- What is the main different between these two parameter passing mechanism (see t3.cpp)?

Review: const parameters

- One can add the ‘const’ key word to both parameter passing mechanisms

  void foo(const int x); //x is an input parameter (read only), cannot be modified inside foo

  void foo(const int &x); //x is an input parameter (read only), cannot be modified inside foo

- See t4.cpp
- What is the difference between these two parameter passing schemes?
  - Pass by value needs to make a copy
  - Pass by reference does not make a copy (just pass the reference of the actual parameter to the subroutine)
    - The reference of a variable is just the pointer to the variable (4 bytes mostly).
    - Pass by const reference is very useful when the parameter is a large data structure for reducing the overhead to make a copy.

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 function

- Any call to a member function has a "calling object"

  Fraction f1; /* 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.
- See t5.cpp on the right.

```
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
```

const objects

- Const variables are the ones that only have one value (initialized).

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

- Objects can be declared as const in 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 example

```
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.

const Summary

```
Class abc {
   public:
   abc();
   void show() const;  // const 1
   void what();
   private:
      void print(const abc & x);   // const 2
      int c;
      const int d;          // const 3
};

void abc:: show() const {
…
}
void abc::abc() : c(0), d(10) {}

Main()
{
    const int I = 10;  // const 4
    ……
    const abc xx;
    xx.show();
    xx.what();
}
```