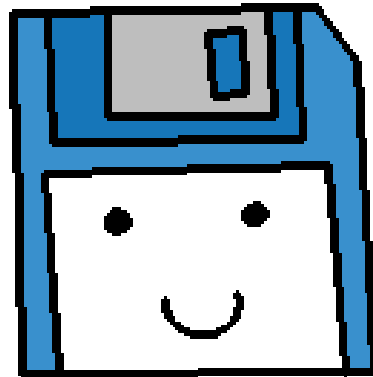


An abstract pattern of vertical bars and rectangles in various shades of green and teal, creating a textured, digital background at the top of the slide.

# Friends

# Friends of Classes

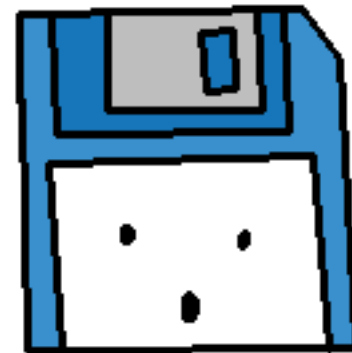
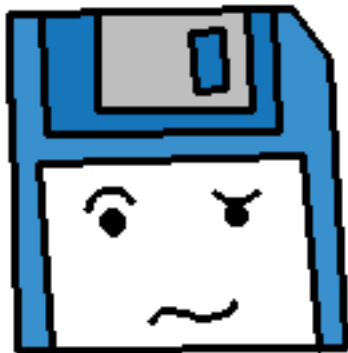


# Friends!

In C++, classes can have **friends**.

Friends do not belong to the class itself, but can access members of a class.

A class can have its private and protected members accessible to friend functions and friend classes.



# Why have friends?

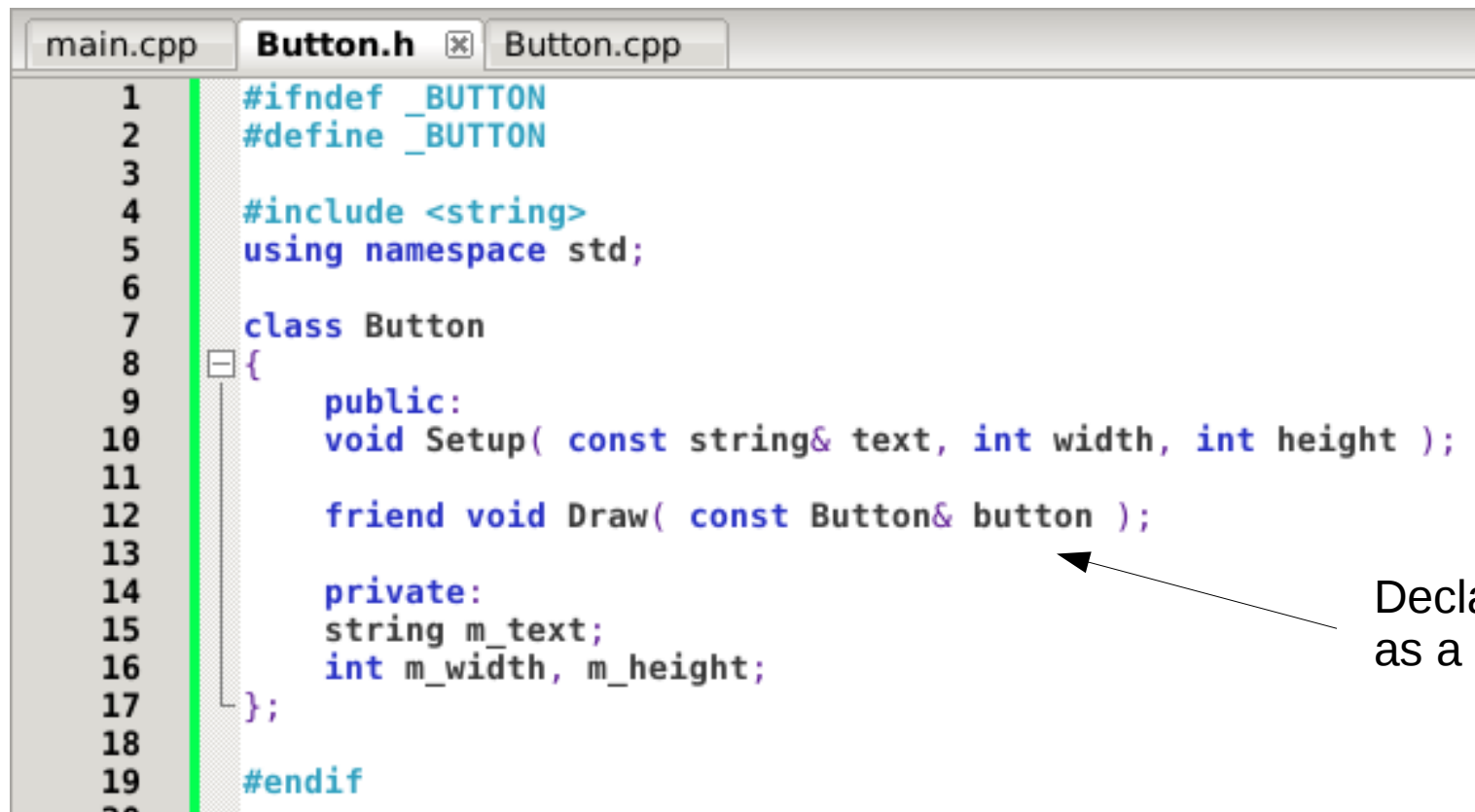
Maybe one class will work with another, and needs to access protected members.

We don't want to add any public functions to access these because only one other class should have access!

# Friend functions

A Friend function can access **private members of a class** directly.

The Friend function can be part of another class, or just be a function independent of any objects.

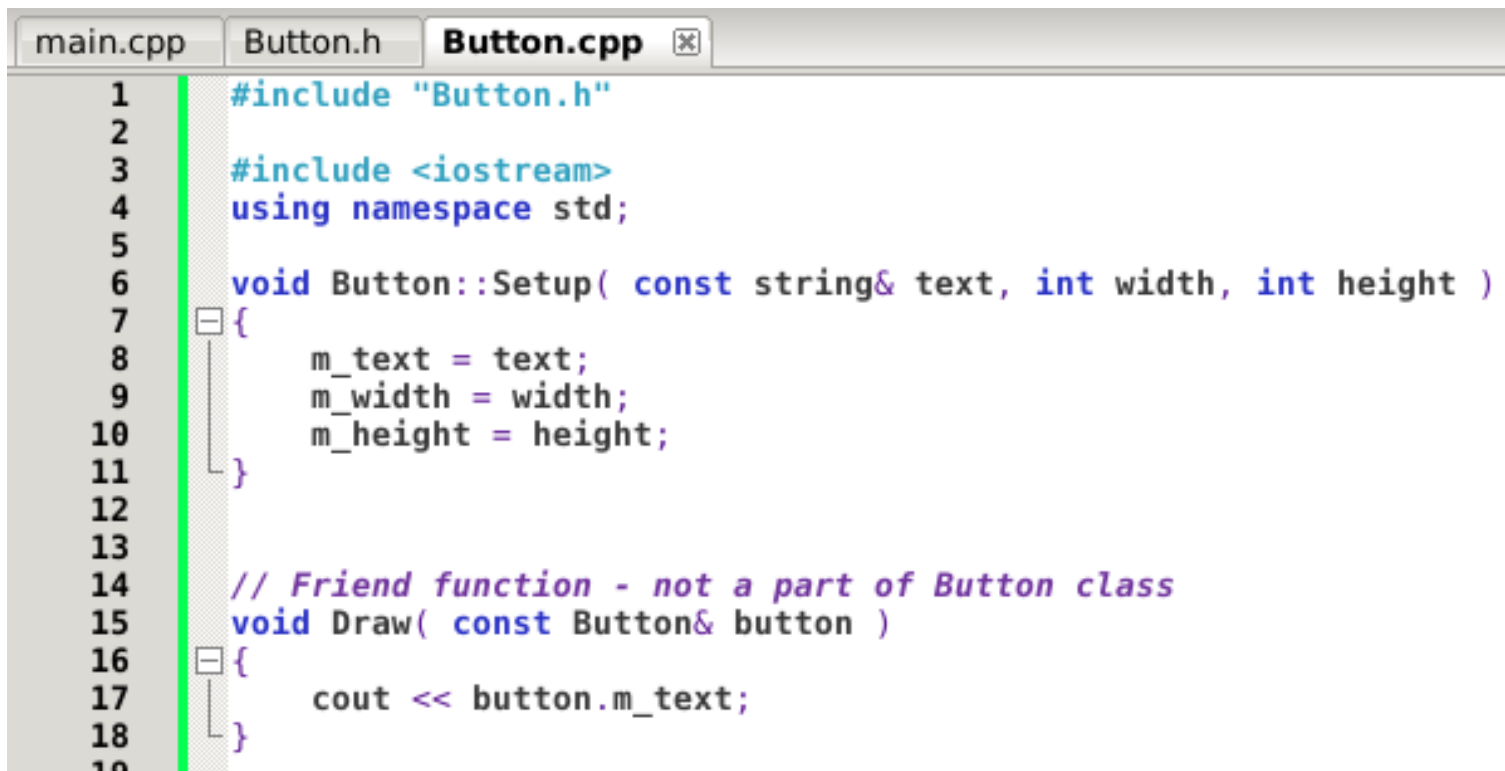


```
1  #ifndef _BUTTON
2  #define _BUTTON
3
4  #include <string>
5  using namespace std;
6
7  class Button
8  {
9      public:
10     void Setup( const string& text, int width, int height );
11
12     friend void Draw( const Button& button );
13
14     private:
15     string m_text;
16     int m_width, m_height;
17 };
18
19 #endif
```

Declaring a function  
as a friend

# Friend functions

A friend function can be just a normal function that is not a class member



The screenshot shows a code editor with three tabs: main.cpp, Button.h, and Button.cpp. The Button.cpp tab is active, displaying the following code:

```
1  #include "Button.h"
2
3  #include <iostream>
4  using namespace std;
5
6  void Button::Setup( const string& text, int width, int height )
7  {
8      m_text = text;
9      m_width = width;
10     m_height = height;
11 }
12
13
14 // Friend function - not a part of Button class
15 void Draw( const Button& button )
16 {
17     cout << button.m_text;
18 }
19
```

An arrow points from the text below to the `Draw` function definition in the code.

**Defining** the Friend function inside of the Button.cpp file. Can access private member `m_text`!



# Friend functions

```
main.cpp * Button.h Button.cpp
1  #include <iostream>
2  using namespace std;
3
4  #include "Button.h"
5
6  int main()
7  {
8      Button buttons[5];
9      buttons[0].Setup( "New",    15, 5 );
10     buttons[1].Setup( "Save",   15, 3 );
11     buttons[2].Setup( "Save",   15, 3 );
12     buttons[3].Setup( "Undo",   10, 5 );
13     buttons[4].Setup( "Exit",   12, 3 );
14
15     for ( int i = 0; i < 5; i++ )
16     {
17         Draw( buttons[i] );
18     }
19
20     return 0;
21 }
```

Call the Friend function normally, it will have access to the **argument's** private members.

# Friend functions

Say we have a Fraction class. Normally when we write Fractions in math, we have something like:

$$\frac{1}{2} + \frac{4}{5} = \frac{a}{b}$$



# Friend functions

If we have a Fraction class in C++, and want to multiply a second Fraction, this *could* be a function that belongs to the Fraction class, but it looks awkward:

```
Fraction.sum = fraction1.Multiply( fraction2 );
```

It looks a little better to have the fractions both be arguments:

```
Fraction.sum = Multiply( fraction1, fraction2 );
```

# Friend functions

But if we create an external function to multiply two fractions

```
Fraction.sum = Multiply( fraction1, fraction2 );
```

then we'd need to add  
Getters & Setters for the  
fraction...

- Set Numerator
- Set Denominator
- Get Numerator
- Get Denominator

# Friend functions

Instead of adding these  
getters/setters,

- Set Numerator
- Set Denominator
- Get Numerator
- Get Denominator

We could instead make the **Multiply** function a friend of the Fraction class, so **Multiply** can access the numerators and denominators directly.

# Friend functions

Within the class, we **declare** the function as a **friend**.

```
class Fraction
{
    public:
        void Setup( int num, int denom );
        void Display();

        friend Fraction Multiply( const Fraction& one, const Fraction& two );

    private:
        int m_num, m_denom;
};
```

Outside of the class, we **define** the function.

```
Fraction Multiply( const Fraction& one, const Fraction& two )
{
    Fraction result;
    result.Setup(
        one.m_num * two.m_num,
        one.m_denom * two.m_denom
    );
    return result;
}
```

Note that it is not a member of Fraction.

# Friend functions

Now we can call it in main:

```
Fraction f1, f2;  
f1.Setup( 1, 2 );  
f2.Setup( 3, 4 );  
  
Fraction product = Multiply( f1, f2 );
```

And once we get to **operator overloading**, we can do this instead:

```
Fraction product = f1 * f2;
```



# Friend classes

Besides just functions, we can make Classes friends with each other.

The **Friend Class** has access to the other's private members, but not vice versa.

```
4  class Number
5  {
6
7      public:
8      friend class NumberContainer;
9
10     private:
11     int m_number;
12 };
13
14 class NumberContainer
15 {
16     public:
17     void Setup()
18     {
19         for ( int i = 0; i < 10; i++ )
20         {
21             m_lstNumbers[i].m_number = i / 2;
22         }
23     }
24     private:
25     Number m_lstNumbers[10];
26 };
27
```

The diagram illustrates the relationship between two classes, `Number` and `NumberContainer`. The `Number` class (lines 4-12) has a private member `m_number` (line 11). The `NumberContainer` class (lines 13-26) has a public method `Setup()` (line 17) that iterates over an array of `Number` objects (`m_lstNumbers`, line 25) and assigns values to their `m_number` members (line 21). An orange arrow points from the `friend class NumberContainer;` declaration in the `Number` class to the `Setup()` method in the `NumberContainer` class, indicating that `NumberContainer` is a friend of `Number`. A blue arrow points from the `m_number` member in the `Number` class to the `m_number` member access in the `Setup()` method, showing that `NumberContainer` has access to the private member of its friend class.



# Notes about friend classes

Friendships go one-way;  
if class A is a friend of class B,  
it doesn't automatically mean that:  
class B is a friend of class A.

Friendships are not transitive  
if class A is a friend of class B,  
and class B is a friend of class C  
it doesn't automatically mean that:  
class C is a friend of class A.

Friendships are not inherited;  
a child class does not  
inherit its parents' friends.

Access due to friendship *is* inherited:  
If Class “Email” is a friend of “Attachment”  
then Email can access Attachment's  
members, including the members inherited  
by its children, through the children.

# Friends!

We will see in-code examples of  
using friend functions in the  
lecture on  
**operator overloading**