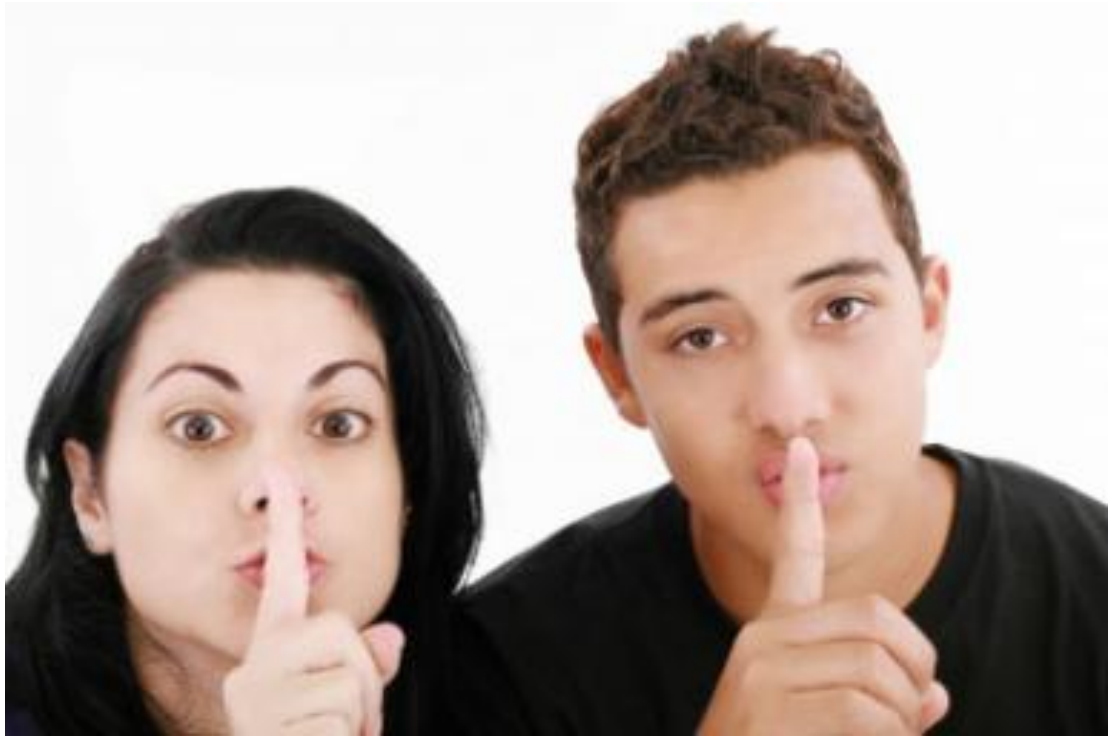
Friend Functions & Operator Overloading

Lecture 7

Who is a Friend?



A friend is one who has access to all your **`private`** stuff

What is Wrong with this Code?

```
#include <iostream>
using namespace std;

class Lonely
{
public:
     Lonely (int x)
     {
          this->x = x;
     }

private:
     int x;
};

void display(const Lonely& e)
{
     cout << e.x << endl;
}
```

```
int main()
{
     Lonely e1( 5 );
     display( e1 );

     return 0;
}
```

Compile Error

```
In function 'void display(const Lonely&)':
error: 'int Lonely::x' is private
  int x;
      ^
error: within this context
  cout << e.x << endl;
```

`friend` Function

- A `friend` function of a class is not a member function of the class
- BUT a `friend` function has access to the private members of that class, just as a member function does

Fixed!

```
#include <iostream>
using namespace std;

class Lonely
{
public:
     Lonely (int x)
     {
          this->x = x;
     }

     friend void display(const Lonely& e);

private:
     int x;
};

void display(const Lonely& e)
{
     cout << e.x << endl;
}
```

```
int main()
{
     Lonely e1( 5 );
     display( e1 );

     return 0;
}
```

Declares **friend**ship

Exercise

Extend the **`Lonely`** class with an **`equal`** function that takes two arguments of type **`Lonely`** (use **`const`** correctness!) – the function compares two objects and returns true if they have the same **`x`** values, otherwise false. Write a **`main()`** function to test it.

Answer

```
#include <iostream>
using namespace std;

class Lonely
{
public:
	Lonely (int x)
	{
		this->x = x;
	}

	friend void display(const Lonely& e);
	friend bool equal(const Lonely& e1,
			  const Lonely& e2);

private:
	int x;
};

void display(const Lonely& e)
{
	cout << e.x << endl;
}

bool equal(const Lonely& e1, const Lonely& e2)
{
	return ( e1.x == e2.x );
}
```

```
int main()
{
	Lonely e1( 5 ), e2( 3 ), e3( 5 );

	cout << "e1?=e1: " << equal( e1, e1 ) << endl;
	cout << "e1?=e2: " << equal( e1, e2 ) << endl;
	cout << "e2?=e2: " << equal( e2, e2 ) << endl;
	cout << "e1?=e3: " << equal( e1, e3 ) << endl;
	cout << "e2?=e3: " << equal( e2, e3 ) << endl;

	return 0;
}
```

```
e1?=e1: 1
e1?=e2: 0
e2?=e2: 1
e1?=e3: 1
e2?=e3: 0
```

When to Make `friend`s

- You can typically write friend functions as non-friend functions using normal accessor and mutator functions that should be part of the class
  - Less efficient, more complex

- In general, use a member function if the function involves one object, non-member [`friend`] if more than one

A Common Use for `friend`

- Have you ever wondered how to implement the following type of functionality?

```
string a("1"), b("2"), c("3");
string easyAs = a + b + c; // 123
```

- C++ supports *operator overloading*, which means you can write special-purpose code for how operators interact with classes
  - In the case above, the `string` library overloaded the **+** operator for the **`string`** class

Operator Overloading

- An operator is really just a function, with special syntax

  a + b ≈ add(a, b)

- So operator overloading builds on functions and function overloading
  - Language extension to include user-defined types

- Leads to more readable code
  - Operators become sensitive to context

Example

```
#include <iostream>
using namespace std;

class Lonely
{
public:
	Lonely (int x)
	{
		this->x = x;
	}

	friend void display(const Lonely& e);
	friend bool equal(const Lonely& e1,
			  const Lonely& e2);
	friend Lonely operator +(const Lonely& e1,
				 const Lonely& e2);
private:
	int x;
};

void display(const Lonely& e)
{
	cout << e.x << endl;
}

bool equal(const Lonely& e1, const Lonely& e2)
{
	return ( e1.x == e2.x );
}
```

```
Lonely operator +(const Lonely& e1, const Lonely& e2)
{
	Lonely r( e1.x + e2.x );
	return r;
}

int main()
{
	Lonely e1( 1 ), e2( 2 ), e3( -1 );

	display( e1 ); // 1
	display( e2 ); // 2
	display( e3 ); // -1

	e3 = e1 + e2;
	display( e3 ); // 3

	return 0;
}
```

General Format

```
<return type> operator <symbol>(<args>);
```

- At least one argument must be of a class type
- May be, but does not have to be, a `friend` of a class
- You cannot create a new operator; you cannot overload dot (`.`), scope resolution (`::`), `.*` and `?:`
- You cannot change the number of arguments
- Some operators (`=`, `[]`, and `->`) require special handling

Exercise

Convert the **`equal`** function (**`friend`** of the Lonely class) to overloading the `==` operator. Alter the **`main`** function to reflect the change.

Answer

```
#include <iostream>
using namespace std;

class Lonely
{
public:
	Lonely (int x)
	{
		this->x = x;
	}

	friend void display(const Lonely& e);
	friend bool operator ==(const Lonely& e1,
				const Lonely& e2);
	friend Lonely operator +(const Lonely& e1,
				 const Lonely& e2);
private:
	int x;
};

void display(const Lonely& e)
{
	cout << e.x << endl;
}

bool operator ==(const Lonely& e1, const Lonely& e2)
{
	return ( e1.x == e2.x );
}
```

```
Lonely operator +(const Lonely& e1, const Lonely& e2)
{
	Lonely r( e1.x + e2.x );
	return r;
}

int main()
{
	Lonely e1( 5 ), e2( 3 ), e3( 5 );

	cout << "e1?=e1: " << ( e1 == e1 ) << endl;
	cout << "e1?=e2: " << ( e1 == e2 ) << endl;
	cout << "e2?=e2: " << ( e2 == e2 ) << endl;
	cout << "e1?=e3: " << ( e1 == e3 ) << endl;
	cout << "e2?=e3: " << ( e2 == e3 ) << endl;


	return 0;
}
```

```
e1?=e1: 1
e1?=e2: 0
e2?=e2: 1
e1?=e3: 1
e2?=e3: 0
```

Automatic Type Conversion

- With the right constructors, C++ can do type conversions for your classes

```
Lonely e1( 5 )
display( e1 + 7 ); // 12
```

- When the appropriate version of `operator +` is not found, the compiler looks for a constructor that takes a single integer
  - The constructor `Lonely(int x)` converts 7 to a Lonely object so the two values can be added!

Unary Operators

- The operator - can be either binary (a-b) or unary (-a)
- Both/either can be overloaded, depending on the argument(s)

```
Lonely operator -(const Lonely& e1, const Lonely& e2);
Lonely operator -(const Lonely& e1);
```

- Implement these!

Answer

```
Lonely operator -(const Lonely& e1, const Lonely& e2)
{
   return ( e1.x - e2.x );
}

Lonely operator -(const Lonely& e1)
{
   return -e1.x;
}
```

Let's Replace the `display` Function!

```
Lonely e1( 5 ), e2( 3 );
display( e1 - e2 );
display( -e1 );
```

- Overload the `<<` operator!

```
cout << e1-e2 << endl << -e1 << endl;
```

Overloading <<

(very similar for >>)

- The << operator is a binary operator
  - The first operand is the output stream
  - The second operand is the value following <<

1 2

```
cout << "Hello world";
cout << endl;
```

- It returns a **reference** to the output stream so that chaining works

```
( cout << "Hello world" ) << endl;
cout << endl;
```

- General form:

```
ostream& operator <<(ostream& outs, const ClassName& v)
{
      outs << v.whatever();
      …
      return outs;
};
```

Exercise

Convert the **`display`** function to overload the `<<` operator. Change your **`main`** function to take advantage.

Answer

```
#include <iostream>
using namespace std;

class Lonely
{
public:
	Lonely (int x)
	{
		this->x = x;
	}

	friend ostream& operator <<(ostream& os,
				const Lonely& e);

private:
	int x;
};

ostream& operator <<(ostream& os, const Lonely& e)
{
	os << e.x;
	return os;
}
```

```
int main()
{

	Lonely e1( 5 ), e2( 3 ), e3( 5 );

	cout << e1 << endl
	     << e2 << endl
	     << e3 << endl;

	return 0;
}
```

Wrap Up

- Using the friend modifier allows you to specify non-member functions that have access to private class variables
  - Typically used for functions that involve multiple classes
  - Simplifies code, may improve performance
- Operator overloading is a common usage of the friend modifier and allows you to specialize operators for your classes
  - Often makes code more readable