Object-Oriented Programming

Lesson 3

Keywords: this, friend, and const

Keyword this

```
CBook a = CBook("C++", 2014);
CBook b = CBook("Physics", 1960);
```

```
a.Display();
b.Display();
```

```
void CBook::Display()
{
   cout << "Title: " << m_sTitle <<
           " year: " << m_nYear << endl;
}
```

How does the first call to the function know that the title and year for the object a are to be displayed and the second call to the function know that the title and year for the object b are to be shown?

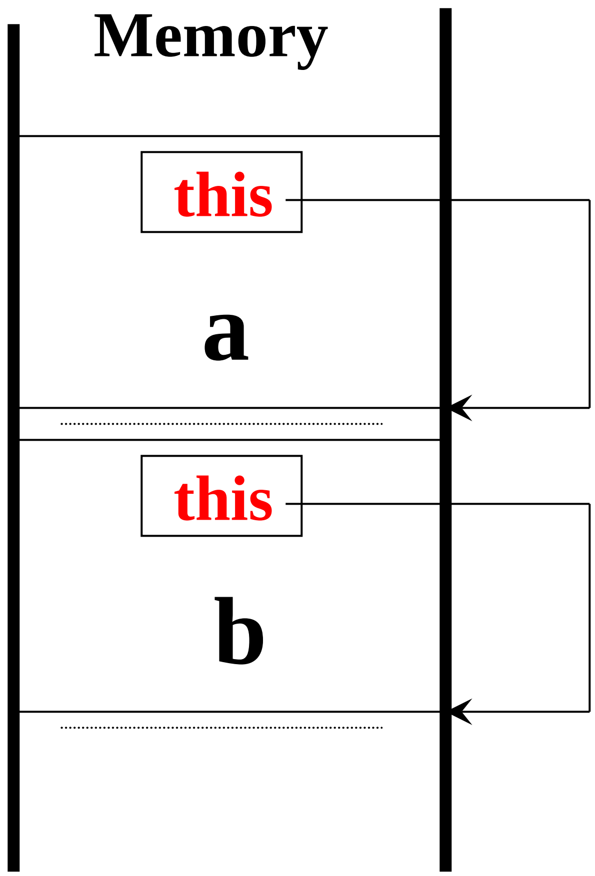
Each object has a pointer that identifies that object.

Keyword this

When an object is created, the compiler allocates storage for it and calls the constructor.

The allocated storage has an extra field which holds the address of the area reserved for the object. The value stored in this field can be accessed through the keyword (pointer) this. **Every** (non static) **member function always gets this pointer as an additional hidden argument.**

```
my_class* const this;
```



⇒ it is not possible to change value stored in this pointer;
⇒ it is not possible to access the address of this pointer.

Keyword this

Usually this pointer is used implicitly:

```
void CBook::Display()
{   cout << "Title: " << m_sTitle <<
              " year: " << m_nYear << endl;
}
```

It can however be used explicitly:

```
void CBook::Display()
{
   cout << "Title: " << this->m_sTitle <<
              " year: " << this->m_nYear << endl;
}
```

```
void CBook::func()
{      CBook q;
       this = &q;
       &this;
}
```

not possíble

Keyword this

You should **not** use this keyword everywhere because it might not add anything to the meaning of the code and often indicates an inexperienced programmer.

But when you do actually need it, it's there.

```
CBook CBook::Clone()
{
        return *this;
}
```

```
bool CBook::Compare(CBook* par)
{
        return (this == par);
}
```

Access control with **friend** keyword

Access specifiers control access to class members.

What if you want to explicitly grant access to a function that isn't a member of the current class?

This is accomplished by declaring that function a friend *inside* the class definition.

You can declare a global function as a friend, and you can also declare a member function of another class, or even an entire class, as a friend.

The friend declaration can be accomplished in any part of the class definition (i.e. either public, protected or private).

```cpp
class X;
class Y
{ public:
        void f1(X* x);
        void f2(X* x);
};
```

```cpp
class X
{
        int i;
public:
        X() { i = 0; };
        friend void Y::f1(X*);
        friend class Z;
        friend void h();
};
```

```cpp
class Z
{
        int j;
public:
        Z() { X x; j = x.i; };
        void g(X* x) { x->i = j; };
};
```

```cpp
void Y::f2(X* x)
{
        x->i = 33;
}
```

error

```cpp
void Y::f1(X* x)
{
        x->i = 22;
}
```

```cpp
void h()
{
   X x;
   x.i = 100; //access to a private member
}
```

```cpp
class my_class2;     // forward declaration
class my_class1
{public:
       friend void compare(my_class1&, my_class2&);
       my_class1(int A) { a = A}
private:      int a;
};
```

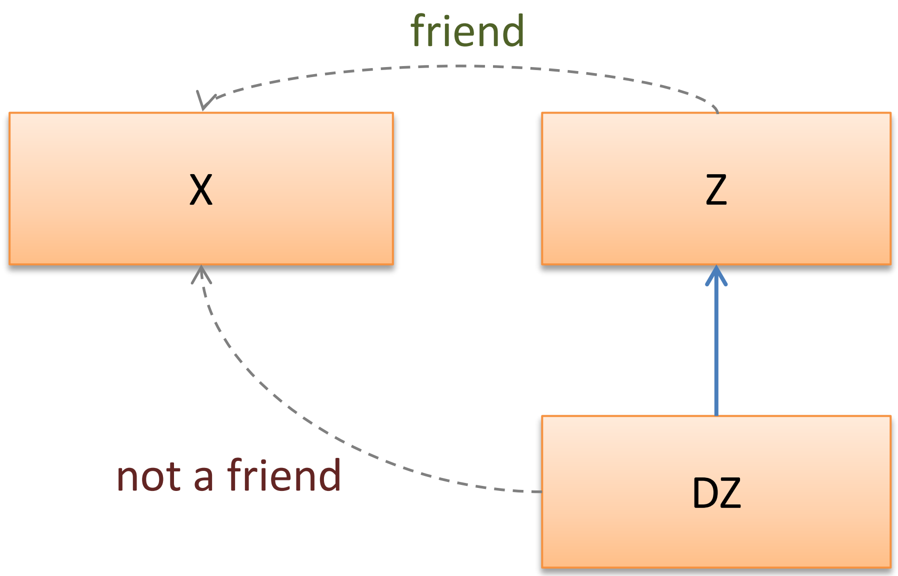
```cpp
class my_class2
{public:
       friend void compare(my_class1&, my_class2&);
       my_class2(int A): { a = A}
private:      int a;
};
```

```cpp
void compare(my_class1 &cl1, my_class2 &cl2)
{
       if(cl1.a==cl2.a) cout << "equal\n";
       else cout << " not equal\n";
}
```

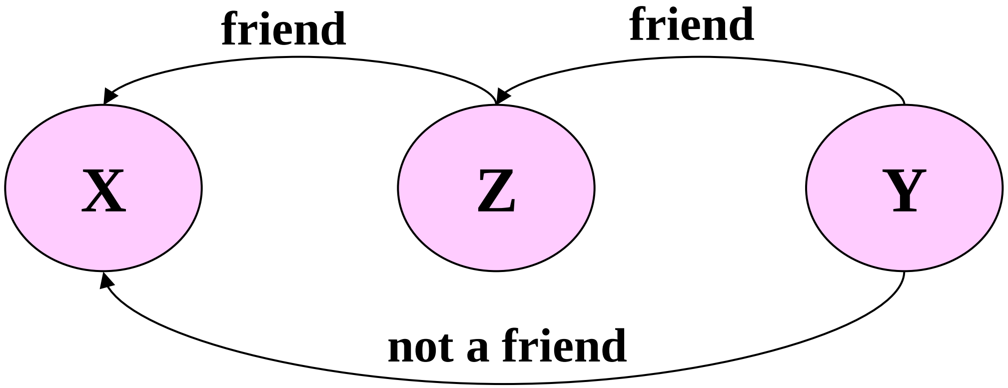
Access control with **friend** keyword

Friendship is not inherited.

class Z is a friend of class X



class DZ is **not** a friend of class X!




```
class Z
{
     int j;
public:
     Z()
    { X x; j = x.i; }
     friend class Y;
};
```

OK

class Y is a friend of class Z

```
class Y
{
public:
     Y () { X x; x.i; }
};
```

error

```
class X
{
     int i;
public:
     X() { i = 0; }
     friend class Z;
};
```

class Z is a friend of class X

class Y is **not** a friend of class X!

Keyword const

Since its origin, const has taken on a number of different purposes:

- value substitution (to eliminate the use of #define)
- objects
- pointers
- function arguments
- return types
- class objects and member functions

- value substitution (to eliminate the of #define)

```
#define    NUM_STUDENTS    150
```

(preprocessor, simple text replacement, no type checking)

```
const int num_students = 150;
```

(compiler, *constant folding* (reducing complicated constant expressions to simple ones by performing the necessary calculations at compile time, type checking)

Keyword const - objects

If you have an objects whose value will not change, you should declare this object as const.

```
int main(int argc, char* argv[])
{
        cout << "Insert a letter..." << endl;
        const char a = cin.get();
        cout << "ASCII of the letter " << a << " is "
             << static_cast<int>(a);
        return 0;
}
```

```
Insert a letter...
A
ASCII of the letter A is 65
```

Keyword const - pointers

When using const with pointers, you have two options: const can be applied to what the pointer is pointing to (i.e. to the object), or the const can be applied to the address stored in the pointer itself (i.e. to the object's address).

```
char c1 = 'a', c2 = 'b';
const char* letter = &c1;

letter = &c2;
*letter = 'c';    //error
*letter = c2;     //error
```

=

```
char c1 = 'a', c2 = 'b';
char const* letter = &c1;

letter = &c2;
*letter = 'c';    //error
*letter = c2;     //error
```

constant object

```
char c1 = 'a', c2 = 'b';
char* const letter = &c1;

letter = &c2;     //error
*letter = 'c';
*letter = c2;
```

constant address

Keyword const - pointers

You can also make a const pointer to a const object using either of two legal forms:

```
char c1 = 'a', c2 = 'b';
const char* const letter = &c1; //1 or
char const* const letter = &c1; //2

letter = &c2;    //error
*letter = c2;    //error
```

You **can assign** the address of a non-const object to a const pointer because you're simply promising not to change something that is OK to change.

```
char c1 = 'a';
const char* const s = &c1;
```

You **can't assign** the address of a const object to a non-const pointer because then you're saying you might change the object via the pointer.

```
const char c1 = 'a';
char* s = &c1;    // error
```

Keyword const - pointers

The place where strict constness is **not** enforced is with character array literals (because there's so much existing C code that relies on this).

The following code will be accepted by the compiler without complaint.
This is technically an error because a character array literal (**"hello"** in this case) is created by the compiler as a constant character array, and the result of the quoted character array is its starting address in memory. Modifying any of the characters in the array is a runtime error, although not all compilers enforce this correctly.

```
char* str = "hello";
str[1] = 'a';
```

← error

```
char str [] = "hello";
str[1] = 'a';
```

← Ok

Keyword const - functions

```
void f1 (const int a)
{
	a++;   //error
}
```

```
const int f2 (int a)
{
	return ++a;
}
```

```
int main(int argc, char* argv[])
{
		f1(7);

		const int i1 = f2(6);
		int i2 = f2(6);
}
```

If you are passing objects by value, specifying const has no meaning to the client (it means that the passed argument cannot be modified inside the function).
If you are returning an object of a user-defined type by value as a const, it means the returned value cannot be modified.
If you are passing and returning addresses, const is a promise that the destination of the address will not be changed.

Keyword const - functions

For built-in types, it doesn't matter whether you return by value as a const, so you should avoid confusing the client programmer and leave off the const when returning a built-in type by value.
Returning by value as a const becomes important when you're dealing with user-defined types. If a function returns a class object by value as a const, the return value of that function cannot be assigned to or otherwise modified.

```
class my_class
{     int i;
public:
      my_class (int ii) : i(ii) {}
      void inc () { i++; }
};
```

```
my_class f3 (my_class mc)
{
      mc.inc();
      return mc;
}
```

```
int main(int argc, char* argv[])
{
      my_class obj1(3);
      f3(obj1).inc();
}
```

cannot be modified

```
const my_class f3 (my_class mc)
{
      mc.inc();
      return mc;
}
```

Keyword const - functions

If you pass or return an address (either a pointer or a reference), it's possible for the client programmer to take it and modify the original value. If you make the pointer or reference a const, you prevent this from happening.
Whenever you're passing an address into a function, you should make it a const if at all possible.

```
void f4 (my_class*) {}
void f5 (const my_class*) {}

int main(int argc, char* argv[])
{
       my_class obj(1);
        my_class* p1 = &obj;
        const my_class* p2 = &obj;

        f4(p1);
        f4(p2);   //error

        f5(p1);
        f5(p2);
}
```

a function that takes a const pointer is more general than one that does not

Standard argument passing

Your first choice when passing an argument is to pass by reference, and by const reference at that.

To the client programmer, the syntax is identical to that of passing by value, so there's no confusion about pointers – they don't even have to think about pointers.

For the creator of the function, passing an address is virtually always more efficient than passing an entire class object, and if you pass by const reference it means your function will not change the destination of that address, so the effect from the client programmer's point of view is exactly the same as pass-by-value (only more efficient).

Keyword const - classes

The use of const inside a class means "this is constant for the lifetime of the object." However, each different object may contain a different value for that constant.

It is not possible to initialize the const in the class definition.
The special initialization point is called the constructor initializer list.

```
class my_class
{
        int i;
        const int max;
public:
        my_class (int ii, int m) : max(m) { i = ii;}
        void inc () { if (i < max) i++; }
};
```

```
my_class obj1(1, 50);
my_class obj2(2, 10);
```

Keyword const – member functions

Class member functions can be made const.

If you declare a member function const, you tell the compiler the function can be called for a const object. A member function that is not specifically declared const is treated as one that will modify data members in an object, and the compiler will not allow you to call it for a const object.

```
class my_class
{
   int i;
   const int max;
public:
   my_class (int ii, int m) : i(ii), max(m) {}
   void inc () { if (i < max) i++; }
   void Display () const { cout << i }
};
```

```
my_class obj1(1, 50);
const my_class obj2(2, 10);

obj1.Display();
obj2.Display();

obj1.inc();
obj2.inc();    //error
```

Every member function that does not modify the state of the object should be declared as const!!!

Keyword const – member functions

Neither constructors nor destructors can be const member functions because they virtually always perform some modification on the object during initialization and cleanup.

When a const member function is defined in *.cpp file, its signature must include the const suffix:

```
void my_class::Display () const
{
	cout << i << endl;
};
```

In const member functions the pointer this is defined as:

```
const my_class *const this;
```

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapters 4, 5, 8