Early Binding vs Late Binding

- The process of converting identifiers (such as variable and function names) => addresses.

Early binding (static binding)

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

class Base
{
public:
    void show()
    {
        cout << "This is Base class";
    }
};

class Derived: public Base
{
public:
    void show()
    {
        cout<<"This is Derived class";
    }
};

int main(void)
{
    Base *bp = new Derived;
    /* The function call decided at compile time (compiler sees type
    of pointer and calls base class function. */
    bp->show();
    return 0;
}
```

```
This is Base class[Finished in 1.4s]
```

- As the name indicates, the compiler (or linker) directly associates an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.
- In this case, at the compile time, the compiler detects pointer bp* and type of the pointer is Base so it calls out the show() function in Base class

Late Binding (dynamic binding)

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

int add(int x, int y)
{
    return x + y;
}

int main()
{
    // Create a function pointer and make it point to the add function
    int (*pFcn)(int, int) = add;
    cout << pFcn(5, 3) << '\n'; // add 5 + 3

    return 0;
}
```

```
8
[Finished in 1.2s]
```

- declaring a virtual function.
- adds code that identifies the kind of object at runtime then matches the call with the right function definition
- In this case, the compiler detects type of function show(), through virtual function, is Derived so it calls out the function in the Derived class

Friend class & Friend Function
Template class

Friend class

- access private and protected members of other class in which it is declared as friend

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

class A {
private:
    int a;

public:
    A() { a = 0; }
    friend class B;
    // Friend class
    // Grant B access to private members of A

};

class B {
private:
    int b;

public:
    void showA(A& x)
    {
        /* Since B is friend of A, it can access
        private members of A */
        cout << "A::a=" << x.a;
    }
};

int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

```
A::a=0[Finished in 1.4s]
```

Friend function

- Like friend class, a friend function can be given a special grant to access private and protected members.
- Friend function can be:
    a. A member of another class
    b. A global function

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

class B;

class A {
public:
    void showB(B&);
};

class B {
private:
    int b;

public:
    B() { b = 0; }
    friend void A::showB(B& x);
    // Friend function
    // Grant B access to privates members of A
};

void A::showB(B& x)
{
    /* Since showB() is friend of B, it can
    access private members of B*/
    cout << "B::b = " << x.b;
}

int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}

```

```
B::b = 0[Finished in 1.3s]
```

Conclusion

- Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
- Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- Friendship is not inherited.

Templates

- create a single function or a class to work with different data types using (to pass data type as a parameter)
- often used in larger codebase for the purpose of code reusability and flexibility of the programs.

Function Templates

Operation	A function template	A normal function
Data types	works with different data types at once	only works with one set of data types
Identical operations (with 2 or more data types)	can perform the same task writing less and maintainable code.	use function overloading to create two functions with the required function declaration.

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

// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T sum(T num)
{
   return num+num;
}

int main()
{
    cout << sum<int>(1) << endl;
    cout << sum<double>(1.8);
  return 0;
}
```

```
2
3.6[Finished in 1.4s]
```

- Now I can call the function once and use it with whatever type of input data.

Class Templates

- a class implementation that is the same for all classes, only the data types used are different.
- make it easy to reuse the same code for all data types.
- use when a class defines something that is independent of the data type.

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

template <class T>
class Calculator
{
private:
    T num1, num2;

public:
    Calculator(T n1, T n2)
    {
        num1 = n1;
        num2 = n2;
    }

    T add()
    {
        return num1 + num2;
    }
};

int main()
{
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);
    cout << intCalc.add() << endl;
    cout << floatCalc.add();

    return 0;
}
```

```
3
3.6[Finished in 1.1s]
```

- Now I just need to define the class once, and then we can pass any data based on the data type we input.