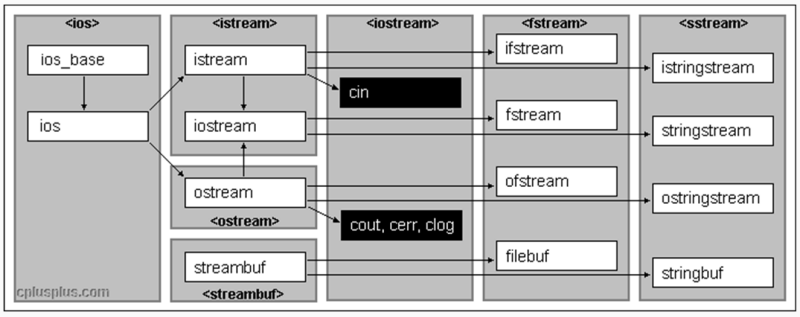
11. Classes III

17. Inheritance and Polymorphism

I/O Stream Inheritance

```
#include <iostream>
#include <fstream>
void print(std::ostream& os, int n) {
   os << n << '\n';
}
int main() {
   print(std::cout, 35);

   std::ofstream
      fout("temp.txt");
   if (fout.good()) {
      print(fout, 36);
      fout.close();
   }
}
```



출처] https://www.cplusplus.com/img/iostream.gif

```
// The std::ofstream class is derived from the ostream class.
// We say that ostream is the base class and std::ofstream is
// the derived class.
```

Inheritance Mechanics (1)

```
class B {
   // Details omitted
};

// To derive a new class D from B we use the following syntax:
class D: public B {
   // Details omitted
};

// Constructors, destructors, nonmember functions, assignment
// operators, and virtual methods(?) are not inherited
```


Inheritance Mechanics (2)

```
class B {
public:
   void f() {
      std::cout << "In function 'f'\n";
   }
};
class D: public B {
public:
   void g() {
      std::cout << "In function 'g'\n";
   }
};
…
B myB;
D myD;
myB.f();       // 'f'
myD.f();       // 'f'
myD.g();       // 'g'
```

Inheritance Mechanics (3)

```
class B {
};
class D1: public B {
};
class D2: public B {
};
class D3: public B {
};


class B1 {
};
class B2 {
};
class D: public B1, public B2 {
};
```


Inheritance Mechanics (4)

Inheritance type	Base access type	Derived access type
private (default)	private protected public	Inherited but inaccessible private private
protected	private protected public	Inherited but inaccessible protected protected
public	private protected public	Inherited but inaccessible protected public

Protected data are private to their own class

Simple Inheritance

```
#include <iostream>
class Base {
   int num1;
public:
   Base(int n) : num1(n) {}
   void print() const {      std::cout << num1 << " base\n";   }
   int Num1() const { return num1; }
};
class Derived : public Base {
   int num2;
public:
   Derived(int n1, int n2) : Base(n1), num2(n2) {}
   void print() const { std::cout << Num1() << num2 << "derived\n";
   } // overriding
};
int main(){
   Base b(2);                    Derived d(3, 5);
   b.print();                    d.print();      d.Base::print();
   b = d;                        // d = b;
}
```


Virtual Function (1)

```
#include <iostream>
class Base {
public:
   void f() {   std::cout << "base\n";          }
};
class Derived : public Base {
public:
   void f() {   std::cout << "derived\n";       }
};

int main() {
   Base b;
   Derived d;

   Base& br = b, &dr = d;
   br.f();      dr.f();

   Base* p1 = &b, * p2 = &d;
   p1->f();     p2->f();
}
```

Virtual Function (2)

```
#include <iostream>
class Base {
public:
   virtual void f() { // dynamic dispatch → polymorphism
      std::cout << "base\n";
   }
};

class Derived : public Base {
public:
   void f() override { // override: virtual overriding
                       // final: final virtual overriding
      std::cout << "derived\n";
   }
};
```


Virtual Function (3)

```
int main() {
   Base b;
   Derived d;

   Base& br = b;       // the type of br is Base&
   Base& dr = d;       // the type of dr is Base& as well
   br.f();             // prints "base"
   dr.f();             // prints "derived"

   Base* bp = &b;      // the type of bp is Base*
   Base* dp = &d;      // the type of dp is Base* as well
   bp->f();            // prints "base"
   dp->f();            // prints "derived"

   br.Base::f();       // prints "base"
   dr.Base::f();       // prints "base"
}
```

Virtual Function (4)

```
class Base {
public:
// A destructor is a special member function that is called
// when the lifetime of an object ends.
    virtual ~Base() {}
};
class Derived : public Base {
public:
    ~Derived() {}
};

int main() {
    Base* b = new Derived;
    delete b;  // Makes a virtual function call to Base::~Base()
               // since it is virtual, it calls Derived::~Derived()
               // which can release resources of the derived class,
               // and then calls Base::~Base() following the usual
               // order of destruction
}
```


Uses of Inheritance (1)

```
#ifndef TEXT_H_INCLUDED
#define TEXT_H_INCLUDED
#include <string>

class Text {
   std::string text;
public:
   Text(const std::string& t);
   virtual std::string get() const;
   virtual void append(const std::string& extra);
};
#endif
```

Uses of Inheritance (2)

```
#include "text.h"
Text::Text(const std::string& t): text(t) {
}

std::string Text::get() const {
   return text;
}

void Text::append(const std::string& extra) {
   text += extra;
}
```


Uses of Inheritance (3)

```
#ifndef FANCYTEXT_H_INCLUDED
#define FANCYTEXT_H_INCLUDED
#include "text.h"
class FancyText: public Text {
   std::string left_bracket;
   std::string right_bracket;
   std::string connector;
public:
   FancyText(const std::string& t, const std::string& left,
      const std::string& right, const std::string& conn);
   std::string get() const override;
   void append(const std::string& extra) override;
};
#endif
```

Uses of Inheritance (4)

```
#include "fancytext.h"
FancyText::FancyText(const std::string& t, const std::string& left,
   const std::string& right, const std::string& conn):
   Text(t), left_bracket(left),
   right_bracket(right), connector(conn) {
}

std::string FancyText::get() const {
   return left_bracket + Text::get() + right_bracket;
}

void FancyText::append(const std::string& extra) {
   Text::append(connector + extra);
}
```


Uses of Inheritance (5)

```
#ifndef FIXEDTEXT_H_INCLUDED
#define FIXEDTEXT_H_INCLUDED
#include "text.h"
class FixedText: public Text {
public:
   FixedText();
   void append(const std::string&) override;
};
#endif
```

Uses of Inheritance (6)

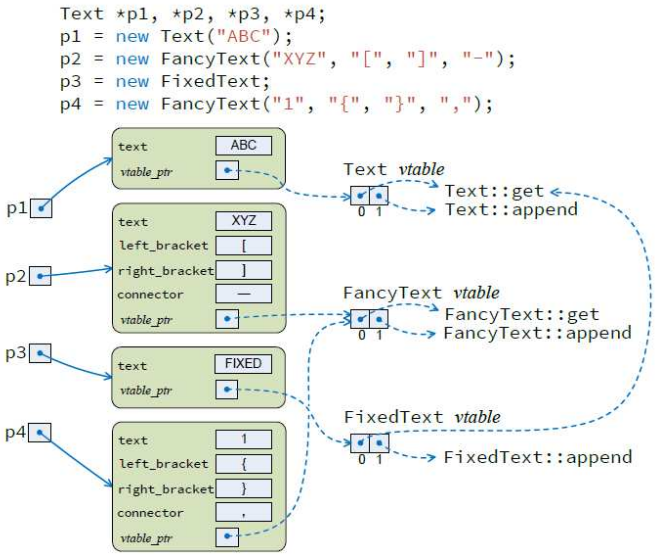
```
#include "fixedtext.h"
FixedText::FixedText(): Text("FIXED") {
}

void FixedText::append(const std::string&) {
}
```


Uses of Inheritance (7)

```
#include <iostream>
#include "text.h"
#include "fancytext.h"
#include "fixedtext.h"
int main() {
   Text t1("plain");
   FancyText t2("fancy", "<<", ">>", "***");   FixedText t3;
   std::cout << t1.get() << '\n';
   std::cout << t2.get() << '\n';
   std::cout << t3.get() << '\n';
   t1.append("A");   t2.append("A");   t3.append("A");
   std::cout << t1.get() << '\n';
   std::cout << t2.get() << '\n';
   std::cout << t3.get() << '\n';
   t1.append("B");   t2.append("B");   t3.append("B");
   std::cout << t1.get() << '\n';
   std::cout << t2.get() << '\n';
   std::cout << t3.get() << '\n';
}
```

Polymorphism (1)

- Virtual table (vtable)



Polymorphism (2)

```
#include <iostream>
class NoVTable {
   int data;
public:
   void set(int d) { data = d; }
   int get() { return data; }
};
class HasVTable {
   int data;
public:
   virtual void set(int d) { data = d; }
   virtual int get() { return data; }
};
int main() {
   NoVTable no_vtable;        HasVTable has_vtable;
   no_vtable.set(10);         has_vtable.set(10);
   std::cout << "no_vtable size = " << sizeof(no_vtable) << '\n';
   std::cout << "has_vtable size = " << sizeof(has_vtable) << '\n';
}
```

Abstract Class

```
class Shape { // abstract class
public:
   virtual double span() const = 0; // pure virtual function (method)
   virtual double area() const = 0;
};
Shape myShape; // illegal – abstract class

class Ellipse: public Shape {
protected:
   double major_radius;
   double minor_radius;
public:
   Ellipse(double major, double minor);
   double span() const override;
   double area() const override;
};
class Circle: public Ellipse {
public:
   Circle(double radius) : Ellipse(radius, radius) {};
};
```


Copy Constructor

```
className(const className &)
className(const className &) = default;
className(const className &) = delete;
#include <iostream>
class Ex {
   int x;
public:
   Ex(int x = 0) : x(x) {}
   Ex(const Ex& e) : x(e.x) {}
   void print() const {
      std::cout << x << std::endl;
   }
};
Ex Copy(Ex e) {
   e.print();
   return e;
}
int main() {
   Ex e1(100);   Ex e2(e1);

   e1.print();      e2.print();
   Copy(e1).print();
}
```

Copy Assignment Operator

```
className &operator= (className)
className &operator= (const className &)
className &operator= (const className &) = default;
className &operator= (const className &) = delete;
#include <iostream>
class Ex {
   int x;
public:
   Ex(int x = 0) : x(x) {}
   //Ex& operator= (Ex e) { x = e.x;       return *this; }
   Ex& operator= (const Ex& e) {
      x = e.x;
      return *this;
   }
   void print() const {
      std::cout << x << std::endl;
   }
};
int main() {
   Ex e1(100), e2;

   e2 = e1;
   e2.print();
}
```


Increment Operator/Casting (1)

```
const className &operator++()
const className &operator++(int)
operator double() const
#include <iostream>
class Rational {
public:
   int numerator;   int denominator;
   Rational(int n = 0, int d = 1) : numerator(n), denominator(d) {}
   const Rational& operator++() {
      numerator += denominator;
      return *this;
   }
   const Rational &operator++(int) {
      const Rational save(*this);
      numerator += denominator;
      return save;
   }
   operator double() const {
      return (double)numerator / denominator;
   }
};
```

Increment Operator/Casting (2)

```
std::ostream& operator<<(std::ostream& os, const Rational& r) {
   os << r.numerator << "/" << r.denominator;
   return os;
}

int main() {
   Rational r1(1, 5), r2, r3;
   r2 = r1++;
   std::cout << r2 << std::endl;
   r3 = ++r1;
   std::cout << r3 << std::endl;
   std::cout << (double)r1 << std::endl;
   std::cout << static_cast<double>(r1) << std::endl;
}
```


Dynamic Cast

```
dynamic_cast<type>(expression)
// pointer or reference type conversion (up, down, sideways)
// fail: pointer-nullptr, reference-exception
```

```
#include <iostream>
class B {
public:
    int b;
    B() : b(10) {}
    virtual ~B() {}
};

class D : public B{
public:
    int d;
    D() : d(20) {}
    ~D() override {}
};
```

```
int main() {
    B* p1 = new B;
    D* p2 = dynamic_cast<D*>(p1);
    if (p2) std::cout << "1:" << p2->d << "\n";
    D* p3 = new D;
    B* p4 = dynamic_cast<B*>(p3);
    if (p4) std::cout << "2:" << p4->b << "\n";
    B* p5 = new D;
    D* p6 = dynamic_cast<D*>(p5);
    if (p6) std::cout << "3:" << p6->d << "\n";
}
```