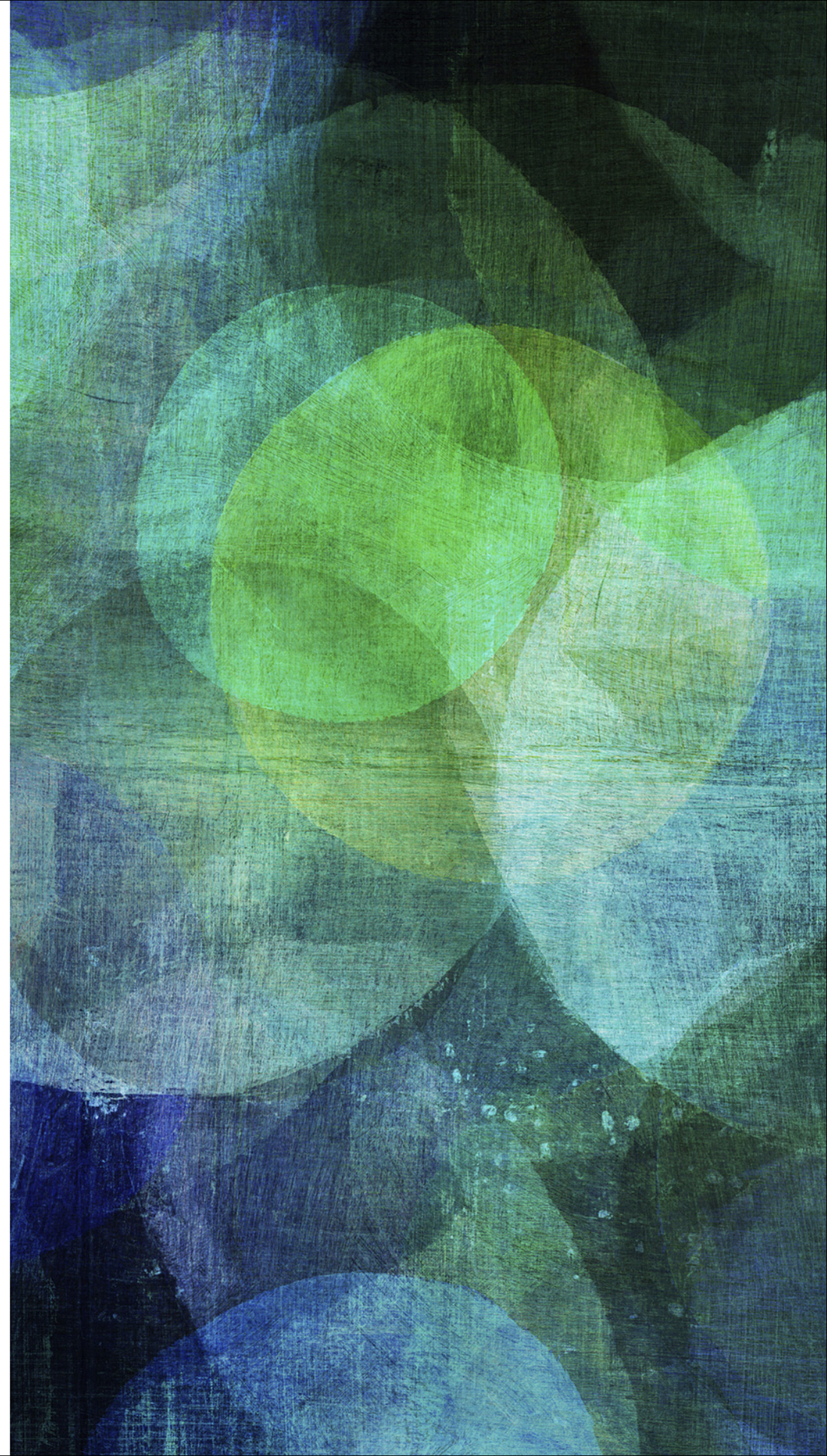
C++入门

类

class



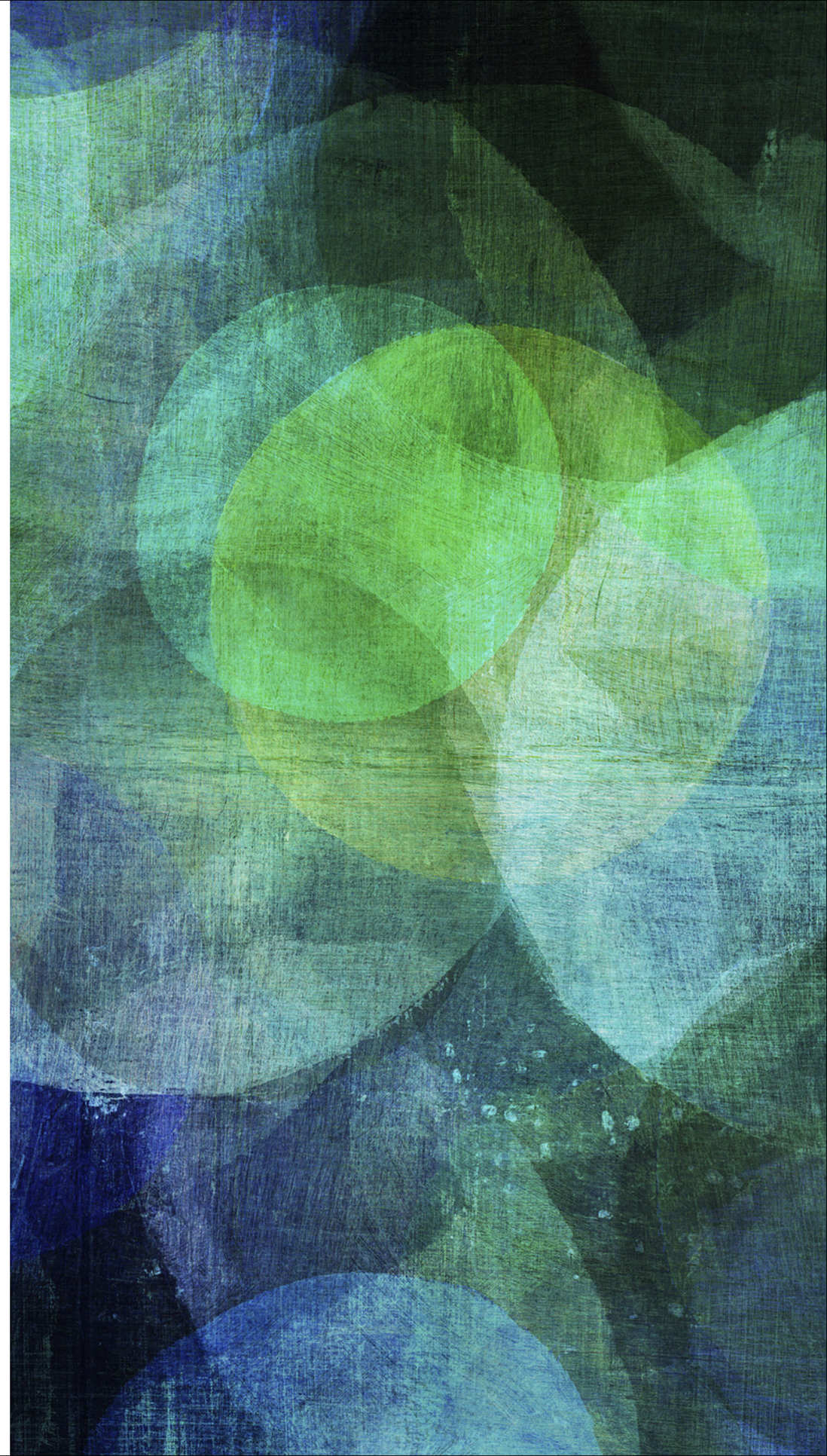
```
class people{
    public:
        int height, weight;
        string name;
        int fun() {
            return phone_number;
        }
    private:
        int phone_number;
        int get_phone_number() {
            return this->phone_number;
        }
    protected:
        int id; //id card number
};
```

```
class 类名称
{
public:
    公有成员（外部接口）
private:
    私有成员 （只允许本类中的函数访问，而类外部的任何函数都不能访问）
protected:
    保护成员（与private类似，差别表现在继承与派生时）
};
```

- public：哪里都能访问
- private：仅限于类的内部
- protected：限于类的内部，派生类的内部

继承

inheritation



- 继承的大前提：一般情况下private成员无法被继承
- public继承：除了private的都原样继承，派生类内部中只能访问基类中public/protected 成员，派生类对象次之，只能访问public成员
- private继承：继承后各成员全部变成了private，派生类内部中只能访问基类中public/protected 成员，派生类的对象什么都不能访问
- protected继承：继承后各成员全部变成protected，

虚函数与多态

A.cpp buffers

```
 9 class oier {
 8     public:
 7         void solve_problem() {
 6             cout << "an oier solved a problem" << endl;
 5         }
 4 };
 3 class rookie: public oier {
 2     public :
 1         void solve_problem() {
16             cout << "a rookie solved a problem" << endl;
 1         }
 2 };
 3 class master : public oier {
 4     public:
 5         void solve_problem() {
 6             cout << "a master solved a problem" << endl;
 7         }
 8 };
 9 class people{
```

NORMAL ~/A.cpp cpp 32% ≡ 16: 1

```
5       oier *test = new rookie();
4       test->solve_problem();
3       test = new master();
2       test->solve_problem();
```

- 我们本来想实现当基类的指针指向某一个派生类对象的地址的时候，调用相应的成员函数能自动识别
- 但是我们发现输出的都是基类里面的成员函数
- 我们再来看看下面的代码

A.cpp buffers

```
 9 class oier {
 8     public:
 7         virtual void solve_problem() {
 6             cout << "an oier solved a problem" << endl;
 5         }
 4 };
 3 class rookie: public oier {
 2     public :
 1         void solve_problem() {
16             cout << "a rookie solved a problem" << endl;
 1         }
 2 };
 3 class master : public oier {
 4     public:
 5         void solve_problem() {
 6             cout << "a master solved a problem" << endl;
 7         }
 8 };
```

- 虚函数有两类，一类是普通的虚函数，直接在函数名字前面加virtual，函数有一个缺省的实现方式，另一类是纯虚函数，函数体不做任何实现，带有纯虚函数的类也称为抽象类，这种类一般用来声明一些接口，不具体实现代码，抽象类也无法被实例化。

- 普通的虚函数，派生类可以选择重写也可以选择不重写，不重写就访问基类里面的，重写就访问派生类里面的。
- 纯虚函数要求派生类一定要实现此函数
- 利用虚函数的理念，我们可以实现动态绑定，在运行时决定该执行派生类里面的成员函数，还是基类里面的成员函数

```
class oier {
    public:
        virtual void solve_problem()  = 0;
};
class rookie: public oier {
    public :
/*          void solve_problem() {
              cout << "a rookie solved a problem" << endl;
          } */
};
```

```
A.cpp: In function 'int main()':
A.cpp:43:29: error: invalid new-expression of abstract class type 'rookie'
     oier *test = new rookie();
                             ^
A.cpp:11:7: note:   because the following virtual functions are pure within 'roo
kie':
 class rookie: public oier {
       ^~~~~~
A.cpp:9:22: note:       virtual void oier::solve_problem()
         virtual void solve_problem()  = 0;
                      ^~~~~~~~~~~~~~
```

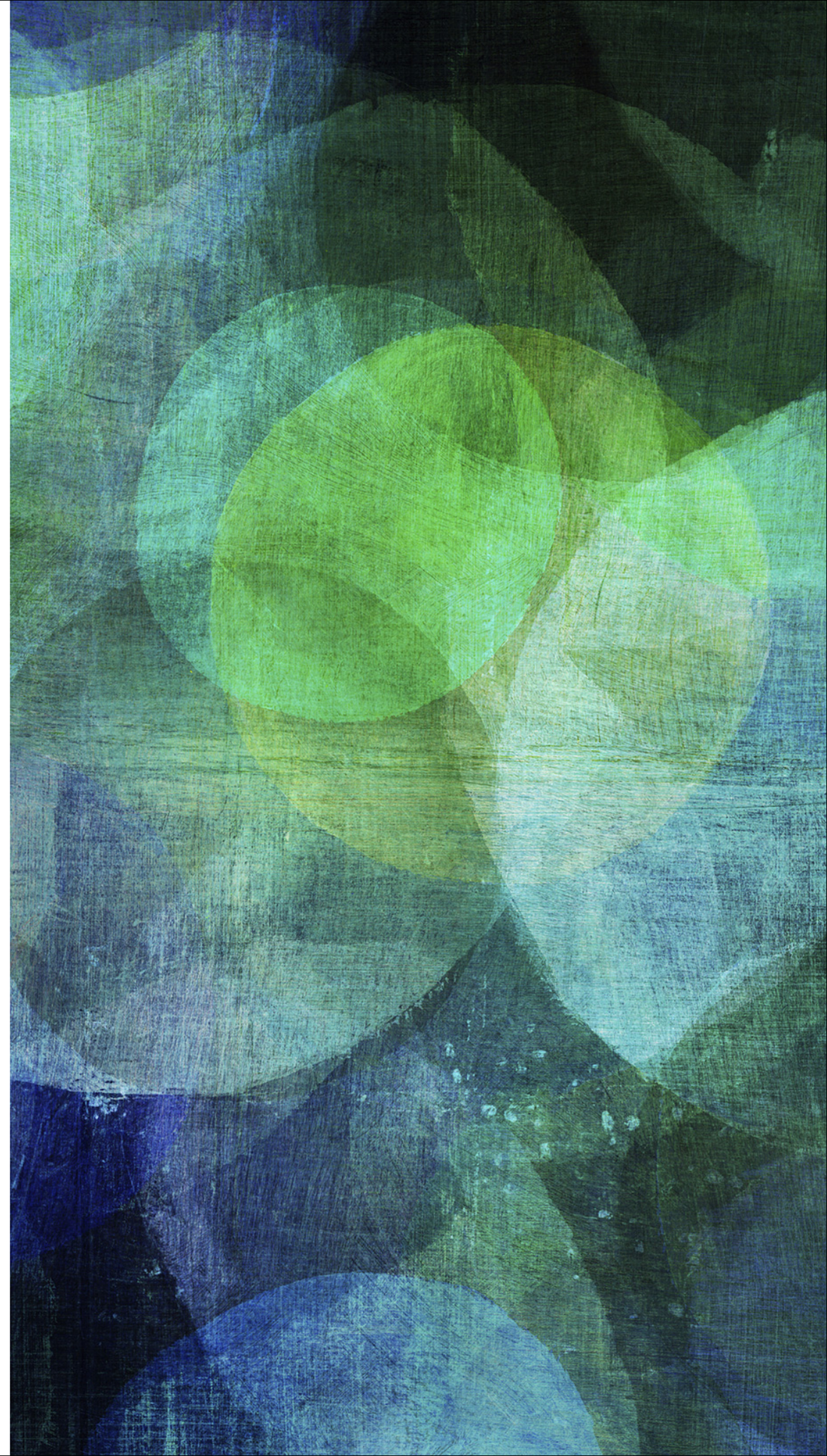
```
template <typename T>
class graph {
  public:
  struct edge {
    int from;
    int to;
    T cost;
  };

  vector<edge> edges;
  vector< vector<int> > g;
  int n;

  graph(int n) : n(n) {
    g.resize(n);
  }

  virtual int add(int from, int to, T cost) = 0;
};
```

```cpp
template <typename T>
class forest : public graph<T> {
  public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  forest(int n) : graph<T>(n) {
  }

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = edges.size();
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};
```

- 代码阅读
- http://codeforces.com/contest/860/submission/32156938

模版类封装

template



```cpp
template <typename T>
class fenwick {
  public:
  vector<T> fenw;
  int n;

  fenwick(int n) : n (n) {
    fenw.resize(n);
  }

  void modify(int x, T v) {
    while (x < n) {
      fenw[x] += v;
      x |= (x + 1);
    }
  }

  T get(int x) {
    T v{};
    while (x >= 0) {
      v += fenw[x];
      x = (x & (x + 1)) - 1;
    }
    return v;
  }
};
```

```cpp
fenwick <int> bit(n);
```

输入输出流

字符串流

stringstream
#include<sstream>

```
4
5       string s;
6       getline(cin, s);
7       stringstream sin;
8       sin << s;
9       string x;
10      while (sin >> x) {
11          cout << x << endl;
12      }
13
14      {
15          sin.clear();
16          sin << "123" << ' ' << "456";
17          int a, b;
18          sin >> a >> b;
19          cout << a << endl;
20          cout << b << endl;
21      }
```

文件流

#include <fstream>

```cpp
int main() {
    int a, b;
    ifstream fin ("A.txt");
    ofstream fout("B.txt");
    fin >> a >> b;
    fout << a + b << endl;
    fout.close();
    fout.open("C.txt");
    fout << "hello" << endl;
}
```

向量

vector

```cpp
vector <int> a, b, c;
void init(int n) {
    a.resize(n, -1);
    b.resize(n, 0);
    c.resize(n, numeric_limits<int>::max());
}

int main() {
    vector <int> tmp = {1, 2, 3, 4 ,5};
    for (int i = 0; i < 10; i++) {
        tmp.push_back(i);
    }
    vector <int> arr(10, -1); //initialize an array of 10 -1
    arr.resize(5, 0); // resize the vector
    cout << arr.size() << endl;
    for (int i = 0; i < (int)arr.size(); i++) { // visit like a normal array
        cout << arr[i] << " " ;
    }

//    arr.erase(arr.begin() + 3, arr.end()); //delete all the element after the third element
    sort (arr.begin(), arr.end());
    arr.erase(unique(arr.begin(), arr.end()), arr.end());
    return 0;
}
```