C/C++ Program Design

LAB 15

CONTENTS

- Class Objects as members and private inheritance

- Learn Class Templates

2 Knowledge Points

2.1 Class Containment(Composition)

2.2 Private Inheritance

2.3 Template

2.1 Class Containment(Composition)

 Using class members that are themselves objects of another class is referred to as **_containment_** or **_composition_** or **_layering_**.

Containment is typically used to implement _**has-a**_ relationship(or **is-implemented-in-terms-of**" 根据某物实现"), that is, relationship for which the new class has an object of another class.

```cpp
class Student
{
private:
    string name;                  //use a string object for name
    valarray<double> scores;  // use valarray<double> object for scores
    ...
};
```

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

class Engine
{
 private:
    int cylinder;
 public:
    explicit Engine(int nc) :cylinder(nc) { cout << "Construtor:Engine(int)\n"; }

    void start() const
    {
        cout << getCylinder() <<" cylinder engine started" << endl;
    }

    int getCylinder() const { return cylinder; }

    ~Engine() { cout << "Destructor:~Engine()\n"; }
};
class Car
{
 private:
    Engine eng; // Car has-an Engine

 public:
    explicit Car(int n = 4): eng(n) { cout << "Construtor:Car(int)\n";}

    void start() const
    {
        cout << "Car with " << eng.getCylinder() << " cylinder engine started" << endl;
        eng.start();
    }

    ~Car() { cout << "Destructor:~Car()\n";}
};
```

use **explicit** keyword to turns off implicit conversions.

Define an object of Engine as Car's attribute

Initialize the object by its own constructor via initialization list in Car's constructor

```cpp
#include "car.h"

int main( )
{
    Car car1;
    Car car2(8);

    car1.start();
    car2.start();

    return 0;
}
```

Call the Car's default constructor
First, constructs the composition object(Engine) in Car class, then, constructs the Car's object.

```
Construtor:Engine(int)
Construtor:Car(int)
Construtor:Engine(int)
Construtor:Car(int)
Car with 4 cylinder engine started
4 cylinder engine started
Car with 8 cylinder engine started
8 cylinder engine started
Destructor:~Car()
Destructor:~Engine()
Destructor:~Car()
Destructor:~Engine()
```

When an object is destructed, the complier first destructs Car's object, and then destructs the composition object(Engine) in Car class.

Initializing Contained Objects

For inherited objects, constructors use the class name in the member initializer list to invoke a specific base-class constructor.

invoke the base-class constructor by its base-class name

```
SalariedEmployee(const char* name, const char* ssn,double s) :Employee(name, ssn), salary(s)
{
    cout << "The derived class constructor is invoked." << endl;
}
```

initialize the new data in subclass

For member objects, constructors use the member name.

```
explicit Car(int n = 4): eng(n) { cout << "Construtor:Car(int)\n";}
```

invoke the member object constructor by its object name

If you omit the initialization list, C++ uses the default constructors for the member objects' classes.

Using an Interface for a Contained Object

The interface for a contained object isn't public, but it can be used within the class methods.

```cpp
class Car
{
private:
    Engine eng; // Car has-an Engine

public:
    explicit Car(int n = 4): eng(n) { cout << "Construtor:Car(int)\n";}
```
define a member function of the Car class
```cpp
    void start() const
    {
        cout << "Car with " << eng.getCylinder() << " cylinder engine started" << endl;
        eng.start();
    }
```
use the object's methods in Car class by the object name
```cpp
    ~Car() { cout << "Destructor:~Car()\n";}
};
```

2.2 Private Inheritance

With **private inheritance**, **public** and **protected** members of the base class become **private** of the derived class. Thus, the member functions of base-class are not a part of the interface of derived objects, but they can be used in derived class member functions.

With **private inheritance** , the derived class does not inherit the base-class interface, it does inherit the implementation. It can be used to implement a **has-a** or **is-implement-in-terms-of** relationship.

Private inheritance is most likely to be a legitimate design strategy when you're dealing with two classes not related by **is-a** where one either needs access to the protected members of another or needs to redefine one or more its virtual functions.

```
privateInheritance > C person.h > ...
     1    #include <iostream>
     2    class Person
     3    {
     4    public:
     5        void eat() const
     6        {
     7            std::cout << "person can eat." << std::endl;
     8        }
     9    };
    10
    11    class Student : public Person
    12    {
    13    public:
    14        void study() const
    15        {
    16            std::cout << "student can study." << std::endl;
    17        }
    18    };
```

```
privateInheritance > G+ testPerson.cpp > ⬡ main()
     1    #include <iostream>
     2    #include "person.h"
     3
     4    void eat(const Person& p);  // anyone can eat
     5    void study(const Student& s);   // only students study
     6    int main()
     7    {
     8        Person p;   // p is a Person
     9        Student s;  // s is a Student
    10        eat(p);     // fine, p is a Person
    11        eat(s);        // OK, s can convert to Person implicitly in public inheritance
    12        return 0;
    13    }
    14
    15    void eat(const Person& p)
    16    {
    17        p.eat();
    18    }
    19
    20    void study(const Student& s)
    21    {
    22        s.study();
    23    }
```

```
person can eat.
person can eat.
```

Call the member function of the base-class by derived-class object.

Can we call the **study** function by **p**?

```
study(p);
study(s);
```

p can not convert to the type of Student automatically.
Compilation fails.

```
privateInheritance > C person.h > ...
  1    #include <iostream>
  2    class Person
  3    {
  4    public:
  5        void eat() const
  6        {
  7            std::cout << "person can eat." << std::endl;
  8        }
  9    };
 10
 11    class Student : private Person
 12    {
 13    public:
 14        void study() const
 15        {
 16            std::cout << "student can study." << std::endl;
 17        }
 18    };
```

```
privateInheritance > G testPerson.cpp > ...
  1    #include <iostream>
  2    #include "person.h"
  3
  4    void eat(const Person& p);   // anyone can eat
  5    void study(const Student& s);   // only students study
  6    int main()
  7    {
  8        Person p;   // p is a Person
  9        Student s;   // s is a Student
 10        eat(p);        // fine, p is a Person
 11        eat(s);        // error! s isn't a Person
 12        return 0;
 13    }
 14
 15    void eat(const Person& p)
 16    {
 17        p.eat();
 18    }
 19
 20    void study(const Student& s)
 21    {
 22        s.study();
 23    }
```

```
testPerson.cpp: In function 'int main()':
testPerson.cpp:11:7: error: 'Person' is an inaccessible base of 'Student'
   11 |    eat(s);  // error! s isn't a Person
      |        ^
```

In contrast to public inheritance, compilers will generally not convert a derived class object **(Student)** into a base class object **(Person)** if the inheritance relationship between the classes is private.

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

class Engine
{
 private:
    int cylinder;
 public:
    explicit Engine(int nc) : cylinder(nc) { cout << "Construtor:Engine(int)\n"; }

    virtual void start() const
    {
        cout << getCylinder() <<" cylinder engine started" << endl;
    }

    int getCylinder() const { return cylinder;}

    virtual ~Engine() { cout << "Destructor:~Engine()\n"; }
};
class Car : private Engine
{    // Car has-an Engine
 public:
    Car(int nc = 4) : Engine(nc) { cout << "Construtor:Car(int)\n"; }

    void start() const override
    {
     cout << "Car with " << Engine::getCylinder() << " cylinder engine started" << endl;
     Engine::start();
    }

    ~Car() { cout << "Destructor:~Car()\n";}
};
```

Call the base-class constructor by base-class name

Call the base-class member functions by base-class name

```cpp
#include "car.h"

int main( )
{
    Car car1;
    Car car2(8);

    car1.start();
    car2.start();

    return 0;
}
```

```
Construtor:Engine(int)
Construtor:Car(int)
Construtor:Engine(int)
Construtor:Car(int)
Car with 4 cylinder engine started
4 cylinder engine started
Car with 8 cylinder engine started
8 cylinder engine started
Destructor:~Car()
Destructor:~Engine()
Destructor:~Car()
Destructor:~Engine()
```

As we see from the example, the **Car** class winds up with an inherited **Engine** component such as **cylinder** and the **Car** method can use the Engine method, **getCylinder()**, internally to access the Engine component, **cylinder**.

Composition vs. Private Inheritance

1. Similarities
- In both cases, there is exactly one **Engine** member object contained in every **Car** object.
- In both cases the **Car** class has a **start()** method that calls the **start()** method on the contained **Engine** object.
- In neither case can users (outsiders) convert a **Car*** to an **Engine***.

2. Differences
- The **composition** is needed if you want to contain several **Engine**s per **Car**.
- The **private inheritance** can introduce unnecessary multiple inheritance.
- The **private inheritance** allows members of **Car** to convert a **Car*** to an **Engine*** by explicitly cast.
- The **private inheritance** allows access to the **protected** members of the base class.
- The **private inheritance** allows **Car** to override **Engine**'s **virtual** functions.

   Use **composition** when you can, but use **private** inheritance when you have to.

2.3 Template

A template is a mechanism in C++ that lets you write a function or a class that uses a generic data type. A placeholder is used instead of a real type and a substitution is done by the compiler whenever a new version of the function or class is needed by your program.

The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.

Template provides us two advantages:
First, it gives us type flexibility by using generic programming.
Second, its performance is near optimal.

Class Templates

Much like function templates, class templates offer the ability to define a single class that can be utilized with a variety of data types.

or <class T>

multiple parameters

```cpp
template<typename T>
class class_name
{
    // class definition

};
```

```cpp
template<typename T1, typename T2, typename T3>
class class_name
{
    // class definition
};
```

nontype template argument

multiple and default parameters

```cpp
template<typename T, size_t size>
class array
{
    T arr[size];

};
```

```cpp
template<typename T1, typename T2, typename T3 = char>
class class_name
{
    // class definition
};
```

1. Template Definition

```cpp
#ifndef CLASSTEMPLATE_MATRIX_H
#define CLASSTEMPLATE_MATRIX_H

#define MAXSIZE 5

template<class T>
class Matrix
{
private:
    T matrix[MAXSIZE];
    size_t size;

public:
    // constructor Initialize all the values of matrix to zero
    Matrix();  // Set size to MAXSIZE

    //print Function
    void printMatrix();

    // Setter Functions
    void setMatrix(T[]);    //set the array to what is sent
    void addMatrix(T[]);    //add an array to matrix

    // No destructor needed
};

#endif //CLASSTEMPLATE_MATRIX_H
```
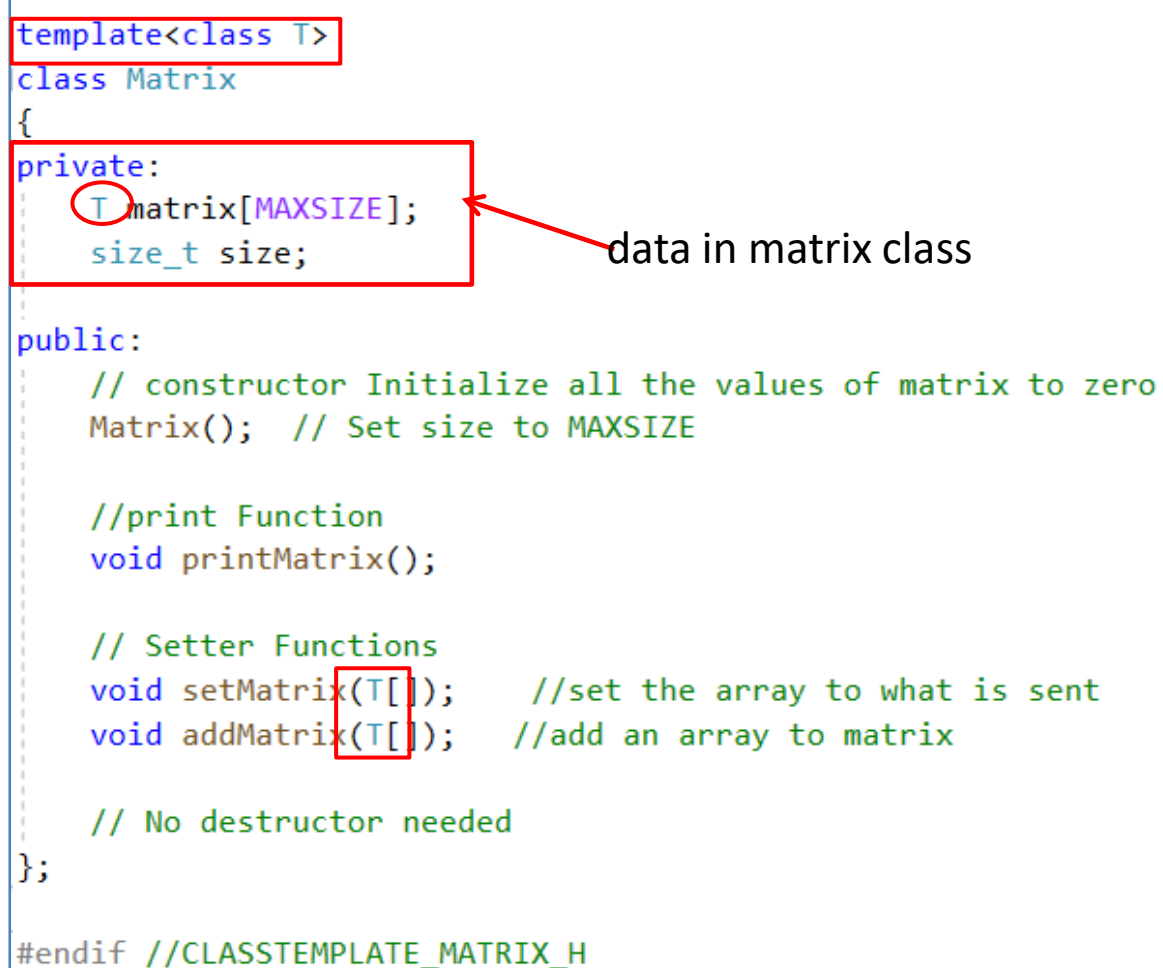
data in matrix class

2. Member Function Definition

To refer to the class in a generic way you must include the placeholder in the class name like this:

template <class T>

return_type class_name <T>::
function_name(parameter_list,…)

```cpp
template<class T>
Matrix<T>::Matrix():size(MAXSIZE) { }

template<class T>
void Matrix<T>::setMatrix(T array[])
{
    for (size_t i = 0; i < size; i++)
        matrix[i] = array[i];
}

template<class T>
void Matrix<T>::printMatrix()
{
    for (size_t i = 0; i < size; i++)
        std::cout << matrix[i] << " ";
    std::cout << std::endl;
}

template<class T>
void Matrix<T>::addMatrix(T otherArray[])
{
    for (size_t i = 0; i < size; i++)
        matrix[i] += otherArray[i];
}
```

3. Class Instantiation

To make an instance of a class you use this form:

class_name <type> variablename;

For example, to create a Matrix with **int** you would type:

Matrix<int> m;

Matrix<int> becomes **the name of a new class**.

```cpp
#include <iostream>
#include "Matrix.h"

int main()
{
    int a[MAXSIZE]{ 1,2,3,4,5 };

    Matrix<int> m;

    m.setMatrix(a);
    m.printMatrix();

    return 0;
}
```

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

template<class T, size_t size>
class A
{
private:
    T arr[size];  // automatic array initialization.

public:
    void insert()
    {
        int i = 1;
        for (int j = 0; j < size; j++)
        {
            arr[j] = i;
            i++;
        }
    }

    void display()
    {
        for (int i = 0; i < size; i++)
        {
            std::cout << arr[i] << " ";
        }
    }
};
```

nontype template argument

```cpp
int main()
{
    A<int, 10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Nontype template arguments can be strings, constant expression and built-in types.

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

template<class T1, class T2>        multiple parameters
class A
{
private:
    T1 a;
    T2 b;
public:
    A(T1 x, T2 y):a(x),b(y) { }

    void display()
    {
        std::cout << "Values of a and b are : " << a << " ," << b << std::endl;
    }
};


int main()
{
    A<int, float> d(5, 6.5);
    d.display();
    return 0;
}
```

multiple and default parameters

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

template <class T, class U, class V = char>
class MultipleParameters
{
private:
    T var1;
    U var2;
    V var3;

public:
    MultipleParameters(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {}  // constructor

    void printVar() {
        cout << "var1 = " << var1 << endl;
        cout << "var2 = " << var2 << endl;
        cout << "var3 = " << var3 << endl;
    }
};
int main()
{
    // create object with int, double and char types
    MultipleParameters<int, double> obj1(7, 7.7, 'c');
    cout << "obj1 values: " << endl;
    obj1.printVar();

    // create object with int, double and bool types
    MultipleParameters<double, char, bool> obj2(8.8, 'a', false);
    cout << "\nobj2 values: " << endl;
    obj2.printVar();

    return 0;
}
```

```
obj1 values:
var1 = 7
var2 = 7.7
var3 = c

obj2 values:
var1 = 8.8
var2 = a
var3 = 0
```

Template specialization

In some cases, it isn't possible or desirable for a template to define exactly the same code for any type. In such cases you can define a *specialization* of the template for that particular type. When a user instantiates the template with that type, the compiler uses the specialization to generate the class, and for all other types, the compiler chooses the more general template. Specializations in which all parameters are specialized are **complete specializations**. If only some of the parameters are specialized, it is called a **partial specialization**.

A template specialization of a class requires a **primary** *class* and a type or parameters to specialize. A specialized template class behaves like a new class. There is no inheritance from the primary class. It doesn't share anything with the primary template class, except the name. Any and all methods and members will have to be implemented.

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

template <class Z>
class Test
{
public:
    Test()
    {
        cout << "It is a General template object \n";
    }
};
```

primary class

```cpp
template <>
class Test <int>
{
public:
    Test()
    {
        cout << "It is a Specialized template object\n";
    }
};
```

class specialization

```cpp
int main()
{
    Test<int> p;
    Test<char> q;
    Test<float> r;
    return 0;
}
```

```
It is a Specialized template object
It is a General template object
It is a General template object
```

Class templates can be **partially specialized**, and the resulting class is still a template. Partial specialization allows template code to be partially customized for specific types in situations, such as:

(1) A template has multiple types and only some of them need to be specialized. The result is a template parameterized on the remaining types.

(2) A template has only one type, but a specialization is needed for pointer, reference, pointer to member, or function pointer types. The specialization itself is still a template on the type pointed to or referenced.

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

template<class T1, class T2>
class Data
{
private:
    T1 a;
    T2 b;

public:
    Data(T1 m, T2 n) :a(m), b(n)
    {
        cout << "Original class template Data<T1,T2>\n";
    }

    void display()
    {
        cout << "Original class template Data:" << a << "," << b << endl;
    }
};
```

primary class

class partial specialization

```cpp
template<class T1>
class Data<T1, char>
{
private:
    T1 a;
    char b;

public:
    Data(T1 m, char c) :a(m), b(c)
    {
        cout << "Partial specialization Data<T1,char>\n";
    }

    void display()
    {
        cout << "Partial specialization Data:" << a << "," << b << endl;
    }
};
```

```cpp
#include <iostream>
#include "partial.h"

int main()
{
    Data<int, int> d_original(5, 8);
    d_original.display();

    Data<double,char> d_special(3.4, 'A');
    d_special.display();

    return 0;
}
```

```
Original class template Data<T1,T2>
Original class template Data:5,8
Partial specialization Data<T1,char>
Partial specialization Data:3.4,A
```

Two code listings side by side illustrating C++ template specialization.

Left listing — primary class (Original template class):

```cpp
template <class T>
class Bag
{
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T t)
    {
        T* tmp;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T[max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = t;
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = t;
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};
```

Annotations on left: "primary class / Original template class" pointing to `template <class T> class Bag`. The `void add(T t)` line is circled. The lines `tmp[size++] = t;` and `elem[size++] = t;` are boxed.

Right listing — class partial specialization (template partial specialization for pointer types):

```cpp
template <class T>
class Bag<T*>
{
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T* t)
    {
        T* tmp;
        if (t == NULL) {    // Check for NULL
            cout << "Null pointer!" << endl;
            return;
        }

        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T[max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = *t;  // Dereference
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = *t; // Dereference
    }

    void print()
    {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};
```

Annotations on right: "class partial specialization / template partial specialization for pointer types" pointing to `template <class T> class Bag<T*>`. The `void add(T* t)` line is circled. The lines `tmp[size++] = *t;  // Dereference` and `elem[size++] = *t; // Dereference` are boxed. "The values that are pointed to are added. If there is no partial specialization, only the pointers themselves are added."

Bringing it All Together

Normally when you write a C++ class you break it into two parts: **a header file** with the interface, and a **.cpp file** with the implementation. With templates this doesn't work so well because the compiler needs to see the definition of the member functions to create new instance of the template class. Some compilers are smart enough to figure out what to do, but some don't. These are usually the most efficient way to use templates. **We recommend that template classes be declared and implemented in .h files to ensure proper linking.**

Template or Inheritance

Templates are powerful, but they are not magical. When you design or use a template you should be aware of what operations the data types you will use need to support.

Template should be used to **generate a set of classes** where the object type **does not affect** the function behavior of the class.

Inheritance should be **used on a set of classes** where the object type **does affect** the function behavior of the class.

3 Exercises

1. The declarations of Point class and Line class are as follows:

```
class Point {
private:
    double x, y;

 public:
    Point(double newX, double newY) ;

    double getX() const;
    double getY() const;

};
```

```
class Line
{
private:
    Point p1, p2;
    double distance;

public:
    Line(Point xp1, Point xp2);
    Line(Line& q);
    double getDistance() const;
};
```

Complete the member functions of the two classes. Write a program to test the classes.

```
test point a: x = 8, y = 9
test point b: x = 1, y = -1
----------------------------------
line1:10.6301
calling the copy constructor of Line
line2:10.6301
```

2. A template class named **Pair** is defined as follows. Please implement the overloading **operator<** which compares the value of the key, if this->key is smaller than that of p.key, return true. Then define a friend function to overload **<< operator** which displays the Pair's data members. At last, run the program. The output sample is as follows:

```cpp
#include <iostream>
#include <string>
using namespace std;
template <class T1,class T2>
class Pair
{
public:
    T1 key;
    T2 value;
    Pair(T1 k,T2 v):key(k),value(v) { };
    bool operator < (const Pair<T1,T2> & p) const;
};
```

```cpp
int main()
{
    Pair<string,int> one("Tom",19);
    Pair<string,int> two("Alice",20);

    if(one < two)
        cout << one;
    else
        cout << two;

    return 0;
}
```

Output: `Alice    20`

3. There is a definition of a template class **Dictionary**. Please write a template partial specialization for Dictionary class whose **Key** is specified to be **int,** and add a member function named sort() which sorts the elements in dictionary in ascending order. At last, run the program. The output sample is as follows:

```cpp
template <class Key, class Value>
class Dictionary {
  Key* keys;
  Value* values;
  int size;
  int max_size;
public:
  Dictionary(int initial_size) :  size(0) {
    max_size = 1;
    while (initial_size >= max_size)
      max_size *= 2;
    keys = new Key[max_size];
    values = new Value[max_size];
  }
  void add(Key key, Value value) {
    Key* tmpKey;
    Value* tmpVal;
    if (size + 1 >= max_size) {
      max_size *= 2;
      tmpKey = new Key [max_size];
      tmpVal = new Value [max_size];
      for (int i = 0; i < size; i++) {
        tmpKey[i] = keys[i];
        tmpVal[i] = values[i];
      }
      tmpKey[size] = key;
      tmpVal[size] = value;
      delete[] keys;
      delete[] values;
      keys = tmpKey;
      values = tmpVal;
    }
    else {
      keys[size] = key;
      values[size] = value;
    }
    size++;
  }

  void print() {
    for (int i = 0; i < size; i++)
      cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
  }

~Dictionary()
  {
    delete[] keys;
    delete[] values;
  }

};
```

```cpp
int main()
{
  Dictionary<const char*, const char*> dict(10);
  dict.print();
  dict.add("apple", "fruit");
  dict.add("banana", "fruit");
  dict.add("dog", "animal");
  dict.print();

  Dictionary<int, const char*> dict_specialized(10);
  dict_specialized.print();
  dict_specialized.add(100, "apple");
  dict_specialized.add(101, "banana");
  dict_specialized.add(103, "dog");
  dict_specialized.add(89, "cat");
  dict_specialized.print();
  dict_specialized.sort();
  cout << endl << "Sorted list:" << endl;
  dict_specialized.print();

  return 0;
}
```

Output:

```
{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}
```