第1章　C++复习

简要复习描述算法需要使用的C++基础知识，介绍C++标准模板库的使用，构建描述算法所需的支持程序。

- C++对C的过程性扩充
- 类和对象
- 运算符重载
- 模板
- C++ STL使用举例
- C++的开发环境*
- 几个常用的支持函数

1.1 C++对C的过程性扩充

1.1.1 C++版的“Hello”程序

下列程序是使用C++标准模版库的“Hello”程序。

```
#include<iostream> // cin, cout, endl;
#include<string> // string
using namespace std;
int main() // C++标准要求返回int值
{   // 显示Input your name.
    std::cout << "Input your name." << std::endl; // 提示“Input your name.”
    std::string name; // 保存姓名
    std::cin >> name; // 假设输入John
    cout << "Hello, " << name << "!" << endl; // 显示Hello, John!
    return 0; // 可省略
}
```

1.1.2 输入输出

C++通常标准输入流对象std::cin和标准输出流对象std::cout进行输入和输出。

1. 输入

std::cin必须与输入运算符“>>”配套使用，例如“std::cin >> x;”将输入数据转换为变量x的类型并存入x。运算符“>>”允许连续输入数据，如“std::cin >> x >> y >> z;”。

2. 输出

std::cout必须与输出运算符“<<”配套使用。例如“std::cout << y;”将变量y的值显示在屏幕上。使用运算符“<<”进行输出操作时，可以把多个不同类型的数据组合在一条语句中，也可以输出表达式的值。例如“std::cout << "a + b= " << 2.5 + 3.5 << std::endl;”将依次输出字符串“a + b=”和表达式2.5 + 3.5的值并换行，结果为“a + b= 6”。

3. 说明

注意，std::cin、std::cout和std::endl等在文件iostream中定义。在使用“using namespace std;”后，std::cin、std::cout和std::endl等可以简写为cin、cout和endl等。

1.1.3 变量声明和作用域

1. 变量声明和类型转换

- C++允许在源程序的任何合适位置声明变量，只要符合“先定义，后使用”的规定即可。
- C++ 17允许使用auto由编译器推断变量类型，这在变量类型比较复杂时非常有用。例如，假设有一个int变量i，则“auto p = &i;”相当于“int *p = &i;”。
- C++ 17也允许使用auto由编译器将return语句中的值类型作为函数的返回值类型。当然，如果函数体中有多个return语句，则值类型必须相同。
- 在C++中，结构名、联合名、枚举名等都是类型名，在定义变量时，不需要在这些名字前冠以struct、union或enum等关键字。
- C++ 17允许使用结构化的绑定声明，使用方法如下例。

```
struct Test { int id; string name; };
Test X = { 1, "Zhang" };
auto [id, name] = X; // 等价于auto id = X.id, name = X.name;
```

- C++增加了一种类似于函数调用的类型转换格式。例如，“x = double(i)”和“p = (double *)(&i)”。

2. 作用域

- 局部变量。在函数定义中声明的变量，只能在此函数内部使用；在类定义中声明的变量，只能在此类内部使用。
- 全局变量。在所有类及函数定义之外声明的变量，在源程序文件中全局有效。
- 同名变量。如果一个文件中全局变量与局部变量同名，默认情况使用局部变量。如果要使用全局变量，应该在全局变量的前面加上域运算符::（也称域解析符）。

3. 名字空间

- C++用名字空间来解决名字重复的问题。
- 用形如“namespace X { int n = 2; }”的方式创建名字空间，其中，“{ }”中可以定义任何变量、类型和函数等。
- 名字空间中的名字通过域解析符“::”引用。例如，cout对象属于std名字空间，通过std::cout引用。
- 名字空间中的名字也可以通过“using namespace *;”引用。例如，在使用“using namespace std;”命令后，也可以通过cout引用std::cout。

4. 举例

```
#include<iostream>
using namespace std;
int m = 0, n = 1; // 全局变量
namespace X { int n = 2; } // 名字空间
int main()
{   int n = 3; // 局部变量
    cout << "global m: " << m << endl; // global m: 0
    cout << "local n: " << n << endl; // local n: 3
    cout << "global n: " << ::n << endl; // global n: 1
    cout << "X::n: " << X::n << endl; // X::n: 2
}
```

1.1.4 引用

1. 引用的声明

引用是变量的一个新名字。引用的定义与变量的定义很相似，Type &表示对Type类型变量的引用。必须在定义引用时说明该引用表示哪个变量。

【注】C++ 17也允许对引用使用结构化的绑定声明。

```
#include<iostream>
using namespace std;
struct Test { int u, v; };
int main()
{   Test o = { 10, 15 }; // 变量定义
    Test &r = o; // 引用说明, r是对变量o的引用, 不是新变量
    o = (Test) { 20, 30 }; // 修改变量的值
    cout << (r.u == o.u && r.v == o.v) << endl; // 1 (r与o值相同)
    cout << (&r == &o) << endl; // 1 (r与o地址相同)
    auto &[u, v] = o; // u是o.u的引用, v是o.v的引用
    cout << (&u == &o.u && &v == &o.v) << endl; // 1 (u与o.u, v与o.v地址相同)
}
```

2. 引用参数

C++支持传引用的参数传递方式。这种方式在形参声明时，在参数前加上符号&。在调用函数时，传递实参的引用。函数通过引用存取实参，可以修改实参的值。

例如，在下述例子中，调用Swap(x, y)会交换变量x和y的值。

```
#include<iostream>
using namespace std;
void Swap(int &x, int &y)
{   int temp = x;
    x = y;
    y = temp;
}
int main()
{   int x = 1, y = 2;
    Swap(x, y);
    cout << x << ", " << y << endl; // 2, 1
}
```

3. 返回引用

可以使用引用类型作为函数的返回值类型，返回某个对象的引用。当然，必须保证该对象在函数返回后依然存在。采用这种方法可以将函数调用放在赋值运算符的左边。

```
#include<iostream>
using namespace std;
int &at(int X[], int i)
{   return X[i - 1]; // 便于使用从1开始的下标
}
int main()
{   int X[10];
    at(X, 1) = 5; // 等价于x[0]=5
    cout << X[0] << endl; // 5
}
```

1.1.5 控制结构上的扩充

1. if语句的扩充

C++ 17中的if语句允许在条件表达式前使用表达式语句或声明语句，具体使用方法如下例所示。

```
#include<iostream>
using namespace std;
int main()
{   int m = 50;
    if(int r = m % 3; r == 0) cout << m << "能被3整除" << endl;
    else cout << m << "不能被3整除，余数是" << r << endl;
    if(m += 60; m > 100) cout << "太大了" << endl;
    cout << "终值是" << m << endl;
}
```

```
50不能被3整除，余数是2
太大了
终值是110
```

2. while语句的扩充

从C++ 11开始，while语句允许使用带声明的条件表达式，具体使用方法如下例所示。

```
#include <iostream>
using namespace std;
int main()
{   unsigned char s[] = "计算机";
    int k = 0;
    while(auto c = s[k])
    {   cout << (int)c << ' ';
        ++k;
    }
    cout << endl;
}
```

```
188 198 203 227 187 250
```

3. 基于范围的for循环

从C++ 11开始，C++提供了基于范围的for循环。对于字符串、数组和容器等可以获得起始位置和结束位置，且可以按照某种顺序依次访问每个元素的对象，基于范围的for循环能够以统一、简洁的方式进行遍历，且可以避免越界访问，使用起来比较方便。下面举例说明这种循环的用法。

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

```
struct Test
{   int u, v;
};
```

```
void show(const Test &e)
{   cout << "(" << e.u << ", " << e.v << ") ";
}
```

```
int main()
{   Test X[5];
    for(Test &e : X) // e是各元素的引用
        e = (Test) { rand() % 100, rand() % 100 };
    for(const auto &e : X) // e是各元素的常值引用
        show(e);
    cout << endl;
    for(auto e : X) // e是各元素的值
        swap(e.u, e.v), show(e);
    cout << endl;
    for(const auto &[u, v] : X) // 结构化绑定声明
        cout << "(" << u << "," << v << ") ";
    cout << endl;
    for(const auto &e : {1, 2, 3, 4, 5}) // 遍历初始值列表
        cout << e << ' ';
    cout << endl;
}
```

```
(41, 67) (34, 0) (69, 24) (78, 58) (62, 64)
(67, 41) (0, 34) (24, 69) (58, 78) (64, 62)
(41, 67) (34, 0) (69, 24) (78, 58) (62, 64)
1 2 3 4 5
```

1.1.6　函数重载

1. 什么是函数重载

在C++中，两个或者两个以上的函数可以使用相同的函数名。但是，这些函数的形参个数、顺序或者类型不同，这就是函数重载。被重载的函数称为重载函数。当用户调用这些函数时，编译器根据实参的个数、顺序和类型来确定到底调用哪个重载函数。

注意，如果两个函数的函数名、形参个数、顺序和类型都相同，只是返回值不同，则不能重载。

2. 举例

假设需要两个Max函数，其中一个返回两个int类型对象中的较大者，另一个返回两个double类型对象中的较大者。

```
#include<iostream>
using namespace std;
int Max(int x, int y)
{   return x > y ? x : y;
}
double Max(double x, double y)
{   return x > y ? x : y;
}
int main()
{   int i = Max(1, 2);
    double d = Max(1.1, 2.2);
    cout << i << ", " << d << endl; // 2, 2.2
}
```

3. 默认参数

C++允许在说明函数原型时为参数列表右侧的一个或多个形参指定默认值。例如，有一函数原型说明为“void init(int x, int y = 5, int z = 10);”，则y和z的默认值分别为5和10。

当进行函数调用时，编译器按照从左向右的顺序将实参与形参结合。如果没有指定足够的实参，则编译器按顺序用函数原型中的默认值来补足缺少的实参。例如，“init(0, 25)”相当于“init(0, 25, 10)”，“init(0)”相当于“init(0, 5, 10)”。

注意，所有默认参数都必须出现在非默认参数的右边。函数声明和函数调用都必须遵守这项规定。

1.1.7 动态存储分配

1. 运算符new

运算符new可用于动态存储分配。该运算符返回一个指向所分配空间的指针。

运算符new的使用格式如下。

- new 对象类型{初始值}
- new 对象类型[对象个数]{初始值}

第一种格式适合于建立单个对象（“{}”可换成“()”），第二种格式实际上是建立一个对象数组。“{初始值}”可以省略，使用默认初始值。

2. 运算符delete

动态分配的存储空间不再需要时应该及时释放。在C++中，用运算符delete来释放new分配的空间。

delete命令的使用格式如下。

- delete 被释放对象的地址
- delete []被释放对象的地址

第一种格式用于释放单个对象，第二种格式用于释放对象数组。

3. 一维数组

为了在运行时创建一个大小可动态变化的一维Type类型数组，可以先将数组名声明为一个Type类型的指针，然后用new运算符为数组动态分配存储空间。

```
#include<iostream>
using namespace std;
void print_n(int *X, int n)
{   for(; n > 0; --n) cout << *X << ' ', ++X;
}
int main()
{   int *X = new int, n = 3;
    *X = 10;
    int *Y = new int{20}; //  或new int(20)
    cout << *X << ' ' << *Y << endl; // 10 20
    int *Z = new int[n];
    fill_n(Z, n, 5); // Z的每个元素都是5
    print_n(Z, n), cout << endl; // 5 5 5
    int *W = new int[n] {1, 2, 3};
    print_n(W, n), cout << endl; // 1 2 3
    delete X, delete Y, delete[]Z, delete[]W;
}
```

1.2 类和对象

类是一种组织数据和函数的自定义数据类型。数据称为这个类的成员变量，函数称为这个类的成员函数。一个类可以有多个对象。

1.2.1 类的组成部分

C++的类由4个部分组成。

- 类名。
- 数据成员，也称成员变量。
- 函数成员，也称成员函数。
- 访问级别。

1.2.2 类成员的访问级别

对类成员的访问有3种不同的级别：公有（public）、私有（private）和保护（protected）级别。

- 在public域中声明的成员在程序中可对其进行直接访问。
- 在private域中声明的成员只能由此类的成员函数和友元函数（friend）访问。
- 在protected域中声明的成员只能由此类的成员函数、友元函数以及派生类的成员函数访问。

在C++中，结构体与类只有一个区别，就是在结构体中，默认访问级别是public，而在类中，默认访问级别是private。

下列Position类描述二维整数坐标，成员函数GetX()用于返回x坐标，GetY()用于返回y坐标。

```
// Position_.h
#pragma once // 只编译一次
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    Position &operator=(const Position &); // 运算符重载
    friend bool Equal(const Position &, const Position &); // 友元函数
};
```

这里先列出相关函数的定义，然后逐一介绍。其中，友元函数见第1.2.6节，运算符重载见第1.3节。

```
Position::Position(int x, int y): x(x), y(y) {} // 构造函数
Position::Position(const Position &q): x(q.x), y(q.y) {} // 拷贝构造函数
Position::~Position() {} // 析构函数
```

```
int Position::GetX() const { return x; } // 返回x坐标
int Position::GetY() const { return y; } // 返回y坐标
```

```
Position &Position::operator=(const Position &p) // 运算符重载
{   x = p.x, y = p.y;
    return *this;
}
```

```
bool Equal(const Position &p, const Position &q) // 友元函数
{   return p.x == q.x && p.y == q.y;
}
```

```
bool operator==(const Position &p, const Position &q) // 运算符重载
{   return Equal(p, q);
}
```

1.2.3 成员函数的定义

一般将类的声明放在头文件中，而将类的实现放在源程序文件中，在源程序文件中成员函数实现通过作用域运算符::来表示该函数属于哪个类。

下列程序给出了类Position的成员函数GetX和GetY的定义。

```
int Position::GetX() const { return x; } // 返回x坐标
int Position::GetY() const { return y; } // 返回y坐标
```

这两个成员函数的参数列表后都带有const修饰，表示不允许在这2个成员函数的函数体中修改当前对象的任何数据成员。

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    Position &operator=(const Position &); // 运算符重载
    friend bool Equal(const Position &, const Position &); // 友元函数
};
```

1.2.4 构造函数与析构函数

1. 构造函数

在定义一个对象时，将自动调用构造函数，初始化该对象的数据成员。构造函数名与类名相同，可以有一个或多个参数，在定义对象时传递实际参数值。构造函数通常声明为类的公有成员函数（私有构造函数只能在类的内部使用）。构造函数不可以有返回值也不得指明返回值类型（包括void）。

2. 默认构造函数

默认构造函数是不带参数的构造函数，用于以默认方式创建对象，也就是对象的每个数据成员都使用默认值。如果没有显式定义任何构造函数，编译器会生成一个默认构造函数，否则，必须显式定义允许在调用时不带参数的构造函数。

3. 拷贝构造函数

拷贝构造函数是参数为同类型对象引用的构造函数，用于在函数中创建某个对象的副本。如果没有显式定义拷贝构造函数，编译器会生成一个默认的拷贝构造函数，使得当前对象每个数据成员的值都和参数对象对应数据成员相同。

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改
```

4. 析构函数

当对象被释放时，将自动调用析构函数。析构函数的函数名由“~”字符和类名构成，析构函数不可以有参数。析构函数主要用于在含有指针的数据成员中自动释放动态数据成员。如果没有动态数据成员，则析构函数通常为空。空析构函数的说明和定义都可以省略（必须同时省略）。

5. 示例

下列程序给出了类Position的构造函数和析构函数的定义。

```
Position::Position(int x, int y): x(x), y(y) {} // 构造函数
Position::Position(const Position &q): x(q.x), y(q.y) {} // 拷贝构造函数
Position::~Position() {} // 析构函数
```

其实，这里定义的拷贝构造函数与默认的拷贝函数完成同样的工作，无需显式定义。同样，析构函数也可以省略。当然，必须同时省略说明和定义都。

1.2.5 类的对象

类对象的声明与创建方式类似于变量的声明与创建方式。对一个对象成员进行访问或调用可采用直接选择（.）或间接选择（->）来实现。

下列代码说明了如何声明类Position的对象，以及如何调用其成员函数。

```cpp
// Position_.cpp
#include <iostream>
#include "Position_.h"
using namespace std;
int main()
{   Position X{1, 2}; // 构造函数, {}可换成()
    Position *Y = new Position{3, 4}; // 构造函数, {}可换成()
    Position Z{}; // 构造函数, 使用默认参数值, {}可省略
    cout << X.GetX() << ", " << X.GetY() << endl; // 1, 2(成员函数)
    cout << Y->GetX() << ", " << Y->GetY() << endl; // 3, 4
    cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
    cout << Equal(Z, X) << endl; // 0(友元函数)
    Z = X; // 运算符重载(成员函数)
    cout << (Z == X) << endl; // 1, 运算符重载(常规函数)
}
```

1.2.6 友元函数

在类声明中使用关键字friend声明的函数称为友元函数。友元函数不是类的函数成员，但友元函数能引用类的私有成员和保护成员。友元函数可以是一个常规函数，也可以是另一个类的成员函数，可以在类声明中的任意区域声明。下列程序定义了类Position的友元函数Equal()。

```
bool Equal(const Position &p, const Position &q) // 友元函数
{   return p.x == q.x && p.y == q.y;
}
```

注意，必须在类中声明该友元函数。声明形式为

```
friend bool Equal(const Position &, const Position &); // 友元函数
```

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    Position &operator=(const Position &); // 运算符重载
    friend bool Equal(const Position &, const Position &); // 友元函数
};
```

1.2.7 对象参数

- 值参数。传递给函数的实参是一个对象，在函数中创建该对象的副本。在创建这个副本时调用该对象的拷贝构造函数，在函数调用结束时调用该副本的析构函数。
- 引用参数。采用引用方式传递对象，在函数中不创建该讨象的副本，因而也不需要撤销副本。引用参数可能改变引用传递的对象。

通常，一个对象作为值参数传递时，为了节省传递时的资源开销，一般声明为const Type&，也就是常值引用。此时，在函数体内部不能修改该参数，否则将出现编译错误。

```
#include<iostream>
using namespace std;
struct Test_t // 默认访问级别public
{   Test_t() { cout << "构造函数" << endl; }
    Test_t(const Test_t &X) { cout << "拷贝构造函数" << endl; }
    ~Test_t() { cout << "析构函数" << endl; }
};
Test_t Test(Test_t X)
{   cout << "测试" << endl;
    Test_t Y = X; // 拷贝构造
    return Y;
} // 析构
int main()
{   Test_t X; // 构造
    cout << "调用" << endl;
    Test_t Y = Test(X); // 拷贝构造
    cout << "返回" << endl;
} // 析构
```

```
构造函数
调用
拷贝构造函数
测试
拷贝构造函数
析构函数
返回
析构函数
析构函数
```

1.3 运算符重载

1.3.1 运算符重载的含义

运算符重载是一项非常有用的技术，可以使得程序更加直观。运算符重载是通过创建运算符函数实现的，用关键字operator加上运算符来表示函数名的函数称为运算符函数。例如，两个坐标相等可定义如下函数。

```
bool Equal(const Position &, const Position &);
```

但人们习惯用“==”来表示相等，这可用运算符重载来表示。

```
bool operator==(const Position &, const Position &);
```

运算符与普通函数使用时的不同之处是普通函数参数出现在圆括号内，而运算符的参数出现在其左、右侧。

```
class Position
{   int x, y; // 坐标, 默认访问级别private
public: // 访问级别public
    Position(int = 0, int = 0); // 构造函数
    Position(const Position &); // 拷贝构造函数
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员)
    int GetY() const; // y坐标
    Position &operator=(const Position &); // 运算符重载
    friend bool Equal(const Position &, const Position &); // 友元函数
};
```

1.3.2 运算符重载为常规函数

如果运算符被重载为常规函数，那么只有一个参数的运算符叫做一元运算符，有两个参数的运算符叫做二元运算符。例如，两个坐标相等的运算符“==”可重载如下。

```
bool operator==(const Position &, const Position &);
```

下列程序实现了对类Position的运算符“==”的重载。

```
bool operator==(const Position &p, const Position &q) // 运算符重载
{   return Equal(p, q);
}
```

经重载运算符“==”后，即可用运算符“==”比较两个Position对象是否相等。

注意，如果运算符重载的实现需要引用对象的私有成员或保护成员，则必须将该运算符重载声明为类的友元函数。声明形式为

```
friend bool operator==(const Position &, const Position &);
```

```
    ~Position(); // 析构函数
    int GetX() const; // x坐标(不允许该函数修改数据成员
    int GetY() const; // y坐标
    Position &operator=(const Position &); // 运算符重载
    friend bool Equal(const Position &, const Position &); //
};
```

1.3.3 运算符重载为成员函数

如果运算符重载为类的成员函数，那么一元运算符没有参数，二元运算符只有一个右侧参数，因为对象自己就是左侧参数。例如，坐标复制的运算符“=”可重载如下。

```
Position &operator=(const Position &);
```

下列程序实现了对类Position的运算符“=”的重载。

```
Position &Position::operator=(const Position &p) // 运算符重载
{   x = p.x, y = p.y;
    return *this;
}
```

其中，在成员函数内部，this指向调用该成员函数的对象，因此该对象可用*this来表示。

经重载运算符“=”后，就可以使用运算符“=”将一个Position对象的值规定为另一个Position对象的值。

注意，必须在类的public域中声明该成员函数。声明形式为

```
Position &operator=(const Position &); // 运算符重载
```

1.3.4 几点注意

① 不能重载作用域运算符“::”、成员运算符“.”、指向成员运算符“.*”、sizeof运算符和三目条件运算符“?:”。

② 类型转换运算符、成员运算符“->”、指向成员运算符“->*”、函数调用运算符“()”、下标运算符“[]”和赋值运算符“=”只能重载为成员函数。

③ 不要过分热心使用运算符重载。如果不能使代码变得更加易读、易写，就不要使用运算符重载，否则，只会自找麻烦。

1.3.5 允许重载的常用运算符

如表1-1所示。表1-1中列出的运算符是按分类顺序排列的，其中加上“*”号的重载方式表示通常使用的重载方式。

表1-1 允许重载的常用运算符

运算符	原始含义	使用示例	重载方式
++, --	自增，自减	i++; ++j;	成员*
[]	下标	array[4] = 2	成员
->	通过指针访问成员	ptr->mem = 34	成员
*, &	间接访问，地址	u = *p; q = &v;	成员*
+, -	正，负	x = -i	成员*
*, /, %	积，商，模(余数)	m = 5 % 3	常规*
+, -	和，差	j = 4 - 2	常规*
<<, >>	左移位，右移位	x = 33 << 1	常规*
<, <=	小于，小于或等于	if(i < 42) ++i;	常规*
>, >=	大于，大于或等于	if(i > 42) --i;	常规*
==, !=	等于，不等于	if(i != 42) j = 42;	常规*
~	位反	y = ~x	成员*

运算符	原始含义	使用示例	重载方式
&, ^, |	位与, 位异或, 位或	y = x & 7	常规*
!	逻辑非	if(!done) ++i;	成员*
&&, ||	逻辑与, 逻辑或	if(x > 0 && x < n) y = 0;	常规*
=	赋值	y = x	成员
*=, /=, %=	积(商, 模)赋值	z %= 3	成员*
+=, -=	和赋值, 差赋值	x -= 3	成员*
<<=, >>=	左移赋值, 右移赋值	x <<= 2	成员*
&=, ^=, |=	位与(异或, 或)赋值	x |= 2	成员*
()	函数调用	w = max(x, y)	成员
type(), (type)	类型转换	i = int(3.5); j = (int)13.5;	成员
,	逗号	i = 0, j = 1, k=2	常规*

1.4 模板

模板是C++提供的一种新机制，用于增强类和函数的重用性。在有模板的算法中，通常将函数声明与函数实现放在同一个文件中。为方便与统一起见，本书其它章节都将函数声明与函数实现放在同一个文件中。

1.4.1 函数模板

1. 如何使用函数模板

对于前面讨论的函数Max，使用函数重载需要定义两个Max函数。但是，这两个函数的函数体几乎相同，定义多个函数增加了工作量和复杂性，且容易造成不一致。通过使用模板，可以定义一个通用的函数Max。

下述Max模板定义了一个Max函数的家族系列，它们分别对应于不同的类型T。编译器根据需要创建适当的Max函数。

```
#include<iostream>
using namespace std;
template<class T>
T Max(T x, T y)
{   return x > y ? x : y;
}
int main()
{   int i = Max(1, 2); //  创建Max<int>()函数
    double d = Max(1.1, 2.2); //  创建Max<double>()函数
    cout << i << ", " << d << endl; // 2, 2.2
}
```

本例根据main函数的调用情况，编译器创建两个Max函数，一个的参数类型为int，另一个的参数类型为double。

2. 多参数函数模板

上述Max模板有时使用起来不够方便，例如，对于调用“Max(10, 2.2)”，编译器不能正确推导出模板参数T代表的实际类型。因为在该调用中，一个参数的类型为int，另一个参数的类型为double。这个问题可以通过使用2个模板参数解决。

```
template<class T1, class T2>
auto Max(T1 x, T2 y)
{   return x > y ? x : y;
}
```

此时，对于调用“Max(11, 2.2)”，编译器创建Max<int, double>()函数，该函数返回一个double值。

1.4.2 类模板

除了定义通用函数外，模板还可用于定义通用类。例如，前面介绍的Position类用于描述整数坐标。如果要求使用另一元素类型的坐标，就必须另写一个Position类。可以使用模板定义一个通用的坐标类Position。

```
#include<iostream>
using namespace std;
template<class T>
class Position // 通用的坐标类Position
{   T x, y;
public:
    Position(T x = 0, T y = 0): x(x), y(y) {}
    ~Position() {} // 默认析构函数
    T GetX() const { return x; }
    T GetY() const { return y; }
    Position &operator=(const Position &p)
    {   x = p.x, y = p.y;
        return *this;
    }
    friend bool operator==(const Position &p, const Position &q)
    {   return p.x == q.x && p.y == q.y;
    }
};
```

```
int main()
{   Position<int> X(1, 2); // 或Position<int> X{1, 2}, 创建Position<int>类
    auto Y = new Position<double>{3.5, 4}; // 创建Position<double>类
    Position<int> Z;
    cout << X.GetX() << ", " << X.GetY() << endl; // 1, 2
    cout << Y->GetX() << ", " << Y->GetY() << endl; // 3.5, 4
    cout << Z.GetX() << ", " << Z.GetY() << endl; // 0, 0
    cout << (Z == X) << endl; // 0
    Z = X;
    cout << (Z == X) << endl; // 1
    delete Y;
}
```

根据main函数中的使用情况，编译器创建Position<int>类和Position<double>类。

【注】C++ 17允许根据构造函数的实际参数进行类模板参数推导。例如，上述例子中的Position<int> X{1, 2}可以写成Position X{1, 2}。这里，构造函数的两个实际参数都是整数，从而推导出类模板参数T代表的实际类型就是int。但是，Position<double>{3.5, 4}不能写成Position{3.5, 4}。因为构造函数的两个实际参数类型不同，不能推导出类模板参数T代表的实际类型。

1.5 C++ STL使用举例

1.5.1 常用算法

```
#include<iostream>
#include<algorithm>
#include<numeric> // accumulate, etc.
#include<random> // mt19937, etc.
using namespace std;
```

```
template<class T> // 输出数组中的前n个元素
void print_n(T X[], int n)
{   for(int i = 0; i < n; ++i) cout << X[i] << " ";
    cout << endl;
}
```

```
int main()
{   int n = 10, X[n], Y[n];
    fill(X, X + n, 1); // 或fill_n(X, n, 1);
    cout << "fill: ", print_n(X, n);
    copy(X, X + n, Y); // 或copy_n(X, n, Y);
    cout << "copy: ", print_n(Y, n);
    iota(X, X + n, 0);
    cout << "iota: ", print_n(X, n);
    cout << "accumulate: " << accumulate(X, X + n, 0) << endl;
    shuffle(X, X + n, mt19937{});
    cout << "shuffle: ", print_n(X, n);
    cout << "find 0(location): " << find(X, X + n, 0) - X << endl;
    int min_loc = min_element(X, X + n) - X; // max_element
    cout << "min_element(location): " << min_loc << endl;
```

```
fill: 1 1 1 1 1 1 1 1 1 1
copy: 1 1 1 1 1 1 1 1 1 1
iota: 0 1 2 3 4 5 6 7 8 9
accumulate: 45
shuffle: 2 9 0 5 4 6 7 1 3 8
find 0(location): 2
min_element(location): 2
sort: 0 1 2 3 4 5 6 7 8 9
lower, upper(value): 5, 6
reverse: 9 8 7 6 5 4 3 2 1 0
replace: 9 8 7 6 5 4 3 2 1 9
```

```
    sort(X, X + n);
    cout << "sort: ", print_n(X, n);
    auto low = lower_bound(X, X + n, 5); // ≥5的第一个位置
    auto up = upper_bound(X, X + n, 5); // >5的第一个位置
    cout << "lower, upper(value): " << *low << ", " << *up << endl;
    reverse(X, X + n);
    cout << "reverse: ", print_n(X, n);
    replace(X, X + n, 0, 9);
    cout << "replace: ", print_n(X, n);
}
```

```
fill: 1 1 1 1 1 1 1 1 1 1
copy: 1 1 1 1 1 1 1 1 1 1
iota: 0 1 2 3 4 5 6 7 8 9
accumulate: 45
shuffle: 2 9 0 5 4 6 7 1 3 8
find 0(location): 2
min_element(location): 2
```

```
sort: 0 1 2 3 4 5 6 7 8 9
lower, upper(value): 5, 6
reverse: 9 8 7 6 5 4 3 2 1 0
replace: 9 8 7 6 5 4 3 2 1 9
```

1.5.2 λ表达式

可以使用λ表达式达到在函数内部定义函数对象的目的。这里首先举例说明无递归λ表达式的使用，然后介绍有递归λ表达式的使用。

1. 无递归的λ表达式

同时说明常用方法generate()和for_each()的使用。

```
#include<iostream>
#include<algorithm>
using namespace std;
template<class T> // 输出数组中的前n个元素
void print_n(T X[], int n)
{   for(; n > 0; --n) cout << *X << " ", ++X;
    cout << endl;
}
```

```
int main()
{   int n = 10, X[n];
    // 命名的λ表达式
    auto digit = []() // []填写需要引用的外部变量, 这里没必要
    {   return rand() % 100;
    };
    generate(X, X + n, digit); // 可用generate_n
    cout << "generate: ";
    print_n(X, n);
    // 匿名的λ表达式
    for_each(X, X + n, [n](int &e) { e += n; });
    // 可用for_each_n, 这里[n]可改为[=], [&]或[&n]
    // [n]和[=]表示使用值, [&]和[&n]表示使用引用
    cout << "for_each: ";
    print_n(X, n);
}
```

```
generate: 41 67 34 0 69 24 78 58 62 64
for_each: 51 77 44 10 79 34 88 68 72 74
```

2. 有递归的λ表达式

在C++ STL中，使用类模板std::function表示函数对象。使用std::function模板类可以创建有递归的λ表达式，从而达到在函数内部定义递归函数的目的。

```
#include<functional> // function, etc.
#include<iostream>
using namespace std;
int main()
{   function<void(int, char, char, char)> // 不可用auto
    hanoi = [&](int n, char a, char b, char c) // 必须使用引用
    {   if(n <= 0) return;
        hanoi(n - 1, a, c, b);
        cout << a << "-->" << b << "\t";
        hanoi(n - 1, b, a, c);
    }; // 这里[&]等价于[&hanoi]
    hanoi(3, 'A', 'B', 'C');
}
```

A-->C	A-->B	C-->B	A-->C	B-->A	B-->C	A-->C

1.5.3 vector

线性表是一种最简单的数据组织形式，由某个集合中的一些元素组成。顺序表是用一个可变长的一维数组表示的线性表。

在C++ STL中，使用类模板std::vector表示可变长的数组（顺序表）。vector使用与元素数目成正比例的操作完成元素的插入和删除。

这里举例说明vector的使用方法。

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

```
template<class T> // 输出vector中的所有元素，形如{ 1 2 3 }
ostream &operator<<(ostream &out, const vector<T> &X)
{   auto F = begin(X), L = end(X);
    // begin()和end()返回迭代器，指示元素位置，类似于指针
    out << "{ ";
    for(; F != L; ++F) out << *F << " ";
    return out << "}";
}
```

```
int main()
{   int n = 3;
    vector<double> X(n), Y(n, 5), Z = {1, 2, 3, 4, 5};
    cout << "Construct(X, Y, Z): " << X << Y << Z << endl;
    X = Z, Y = {1, 2, 3};
    cout << "operator=(X, Y): " << X << Y << endl;
    cout << "size(X): " << X.size() << endl;
    cout << "front(X): " << X.front() << endl;
    cout << "back(X): " << X.back() << endl;
    cout << "operator[](X): " << X[2] << endl;
    X.assign({1, 2, 3, 4, 5}), Y.assign(n, 5);
    cout << "assign(X, Y): " << X << Y << endl;
    X.resize(4), Y.resize(4, 1);
    cout << "resize(X, Y): " << X << Y << endl;
    X.push_back(3), X.push_back(2);
    cout << "push_back(X): " << X << endl;
    X.clear();
    cout << boolalpha << "clear, empty: " << X.empty();
}
```

```
Construct(X, Y, Z):
    { 0 0 0 }{ 5 5 5 }
    { 1 2 3 4 5 }
operator=(X, Y):
    { 1 2 3 4 5 }{ 1 2 3 }
size(X): 5
front(X): 1
back(X): 5
operator[](X): 3
assign(X, Y):
    { 1 2 3 4 5 }{ 5 5 5 }
resize(X, Y):
    { 1 2 3 4 }{ 5 5 5 1 }
push_back(X):
    { 1 2 3 4 3 2 }
clear, empty: true
```

```
Construct(X, Y, Z): { 0 0 0 }{ 5 5 5 }{ 1 2 3 4 5 }
operator=(X, Y): { 1 2 3 4 5 }{ 1 2 3 }
size(X): 5
front(X): 1
back(X): 5
operator[](X): 3
assign(X, Y): { 1 2 3 4 5 }{ 5 5 5 }
resize(X, Y): { 1 2 3 4 }{ 5 5 5 1 }
push_back(X): { 1 2 3 4 3 2 }
clear, empty: true
```

1.5.4 stack

栈（stack）是一种特殊的线性表。栈的插入和删除都在称为栈顶（top）的一端进行。

栈的运算意味着如果依次将元素1, 2, 3, 4插入栈，那末从栈中移出的第一个元素必定是4，即最后进入栈的元素将首先移出，因此栈又称为后进先出（LIFO）表。

在C++ STL中，使用类模板std::stack表示栈。stack使用常数时间的操作完成元素的插入和删除。这里以将数组元素反向为例说明C++ STL中stack的使用方法。

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

```
template<class T>
void Reverse(T X[], int n)
{   stack<T> S;
    for(int i = 0; i < n; ++i) S.push(X[i]); // 进栈
    for(int i = 0; !S.empty(); ++i) // 是否空栈
        X[i] = S.top(), S.pop(); // 栈顶，出栈
}
```

```
template<class T> // 输出数组中的前n个元素
void show(T X[], int n)
{   for(int i = 0; i < n; ++i) cout << X[i] << ' ';
    cout << endl;
}
```

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    show(X, n); // 6 7 9 8 4 3 2 9 6
    Reverse(X, n);
    show(X, n); // 6 9 2 3 4 8 9 7 6
}
```

1.5.5 queue

队列（queue）也是一种特殊的线性表。队列的插入只能在称为尾部的一端进行，删除则只能在称为首部的另一端进行。

队列的运算要求第一个插入队中的元素也第一个移出，因此队列是一个先进先出（FIFO)表。

在C++ STL中，使用类模板std::queue表示队列。queue使用常数时间的操作完成元素的插入和删除。

这里举例说明queue的使用方法。

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

```
int main()
{   int n = 9;
    double X[n] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    queue<double> Q;
    for(auto e : X)
        cout << e << " ", Q.push(e); // 进队
    cout << endl;
    while(!Q.empty()) // 是否空队
        cout << Q.front() << " ", Q.pop(); // 队首，出队
    cout << endl;
}
```

```
6 7 9 8 4 3 2 9 6
6 7 9 8 4 3 2 9 6
```

1.5.6 heap

堆分为最小堆和最大堆2种，是一种带有优先级的队列。在最小堆中，队首（堆顶）元素一定是最小的元素，在最大堆中，队首（堆顶）元素一定是最大的元素。

在C++ STL中，可用priority_queue表示堆。priority_queue使用与元素数目的对数成正比例的操作完成元素的插入和删除。但是C++ STL中的priority_queue使用起来不是很方便。这里对priority_queue进行特殊化，专门定义最小堆和最大堆，保存在文件queue.hpp中。

1. 最小堆和最大堆

```
// queue.hpp
#pragma once
#include <queue> // queue, priority_queue
```

```
namespace std
{   using namespace rel_ops; // 只需定义<和==即可在std中使用!=, <=, >, >=
}
```

```
#ifndef _STD_BEGIN
#define _STD_BEGIN namespace std {
#define _STD_END }
#endif
```

```
_STD_BEGIN
// 最小堆, 将STL的priority_queue特殊化, 需要大于运算
#ifndef minheap
template<class T, class C = vector<T>> //
using minheap = priority_queue<T, C, greater<T>>;
#endif
// 最大堆, 将STL的priority_queue特殊化, 需要小于运算
#ifndef maxheap
template<class T, class C = vector<T>> //
using maxheap = priority_queue<T, C, less<T>>;
#endif
_STD_END
```

2. 举例

以堆排序（升序排列）为例说明C++ STL中堆的使用方法。

```
#include "queue.hpp"
using namespace std;
```

```
template<class T> // 对数组X[0 to n-1]排序
void HeapSort(T X[], int n)
{   minheap<T> H; // 最小堆
    for(int i = 0; i < n; ++i) H.push(X[i]); // 进堆
    for(int i = 0; !H.empty(); ++i) // 是否空堆
        X[i] = H.top(), H.pop(); // 堆顶，出堆
} // 耗时O(n*log(n))
```

```
struct Test { int v; };
bool operator<(const Test &x, const Test &y)
{   return x.v < y.v;
}
void show(const Test X[], int n)
{   for(int i = 0; i < n; ++i)
        cout << X[i].v << ' ';
    cout << endl;
}
int main()
{   Test X[] = {6, 7, 9, 8, 4, 3, 2, 9, 6};
    int n = 9;
    show(X, n); // 6 7 9 8 4 3 2 9 6
    HeapSort(X, n);
    show(X, n); // 2 3 4 6 6 7 8 9 9
}
```

1.5.7 set

在C++ STL中，使用类模板std::set表示集合。set对象中的元素是有序的，不允许重复的。其中，第0号元素是最小元素，第1号元素是次小元素，依此类推。

set使用与元素数目的对数成正比例的操作完成元素的查找、插入和删除。

这里举例说明set的使用方法。

```
#include<iostream>
#include<set>
#include<cstdlib> // rand
using namespace std;
```

```
template<class T> // 输出set中的所有元素，形如{ 1 2 3 }
ostream &operator<<(ostream &out, const set<T> &X)
{   out << "{ ";
    for(const T &e : X) out << e << " ";
    return out << "}";
}
```

```
template<class K, class Pr, class A, class K2> // 在X中查找v
bool Find(const set<K, Pr, A> &X, const K2 &v)
{   return X.find(v) != end(X);
}
```

```
int main()
{   int n = 10, w;
    set<int> X = {1, 2, 3}, Y;
    cout << "Construct: ";
    cout << X << " " << Y;
    cout << endl;
    for(int i = 0; i < n; ++i)
        X.insert(rand() % 25);
    X.insert({7, 17});
    cout << "insert: " << X << endl;
    cout << "size(X, Y): " << X.size() << ", " << Y.size() << endl;
    cout << boolalpha;
    cout << "find(3, 10): " << Find(X, 3) << ", " << Find(X, 10) << endl;
    cout << "begin, end(value): " << *begin(X) << ", " << *(--end(X)) << endl;
    X.erase(3);
    cout << "erase(3): " << X << endl;
```

```
Construct: { 1 2 3 } { }
insert: { 0 1 2 3 7 8 9 12 14 16 17 19 24 }
size(X, Y): 13, 0
find(3, 10): true, false
begin, end(value): 0, 24
erase(3): { 0 1 2 7 8 9 12 14 16 17 19 24 }
```

```
    auto low = X.lower_bound(10); // >=10的第一个位置
    auto up = X.upper_bound(15); // >15的第一个位置
    cout << "lower, upper(value): " << *low << ", " << *up << endl;
    X.erase(low, up);
    cout << "erase(10 -- 15): " << X << endl;
    X.clear();
    cout << "clear, empty(X): " << X.empty() << endl;
}
```

```
Construct: { 1 2 3 } { }
insert: { 0 1 2 3 7 8 9 12 14 16 17 19 24 }
size(X, Y): 13, 0
find(3, 10): true, false
begin, end(value): 0, 24
erase(3): { 0 1 2 7 8 9 12 14 16 17 19 24 }
lower, upper(value): 12, 16
erase(10 -- 15): { 0 1 2 7 8 9 16 17 19 24 }
clear, empty(X): true
```

1.5.8 map

可以认为map是一种特殊的集合。模板类std::map<Key, T>描述可变长的std::pair<const Key, T>类型元素序列。每个对的first分量是排序键且second分量是键的关联值。可以使用first分量作为下标访问序列中的元素，返回值为相应second分量的引用，当first分量无效时插入一个元素。

map使用与元素数目的对数成正比例的操作完成元素的查找、插入和删除。这里举例说明map的使用方法。

```
#include<iostream>
#include<map>
using namespace std;
// 输出map中的所有元素，形如{ (1, 3) (2, 5) (3, 7) }
template<class K, class T>
ostream &operator<<(ostream &out, const map<K, T> &X)
{   out << "{ ";
    for(const auto &[key, val] : X)
        out << "(" << key << ", " << val << ") ";
    return out << "}";
}
```

```
int main()
{   map<double, int> X = {{1.5, 3}, {2.5, 5}, {3.5, 7}}, Y = X;
    cout << "Construct: " << X << endl;
    cout << "X[8]: " << X[8] << endl; // 新建X[8]
    X.insert({{7, 17}, {3.5, 8}}); // 插入, 去除重复
    cout << "insert: " << X << endl;
    cout << "find(3.5): " << (X.find(3.5) != end(X)) << endl;
    X.erase(3.5); // 删除
    cout << "erase(3.5): " << X << endl;
    int n = 3; // map数组大小
    map<double, int> W[n] = {X, Y}; // map数组
    for(int i = 0; i < n; ++i) cout << i << ": " << W[i] << endl;
}
```

```
Construct: { (1.5, 3) (2.5, 5) (3.5, 7) }
X[8]: 0
insert: { (1.5, 3) (2.5, 5) (3.5, 7) (7, 17) (8, 0) }
find(3.5): 1
erase(3.5): { (1.5, 3) (2.5, 5) (7, 17) (8, 0) }
0: { (1.5, 3) (2.5, 5) (7, 17) (8, 0) }
1: { (1.5, 3) (2.5, 5) (3.5, 7) }
2: { }
```

1.6 C++的开发环境*

1.6.1 开发环境的选择和安装

在本书中，IDE使用Dev-Cpp 5.11，C++编译器使用MinGW-W64中的GCC 7.1.0。

Dev-C++的原始版本为Bloodshed Dev-C++，最高版本号为4.9.9.2，4.9.9.2以后的版本为Orwell Dev-C++。Orwell Dev-C++的下载地址为“http://orwelldevcpp.blogspot.com/”。下载完成后直接运行安装程序，按照提示完成安装。

MinGW-W64的下载地址可以通过互联网搜索“MinGW-W64”找到。下载完成后直接运行安装程序，按照提示选定安装目录（例如C:\GCC 7）和GCC版本号，安装完成后目录结构如图1-1所示。

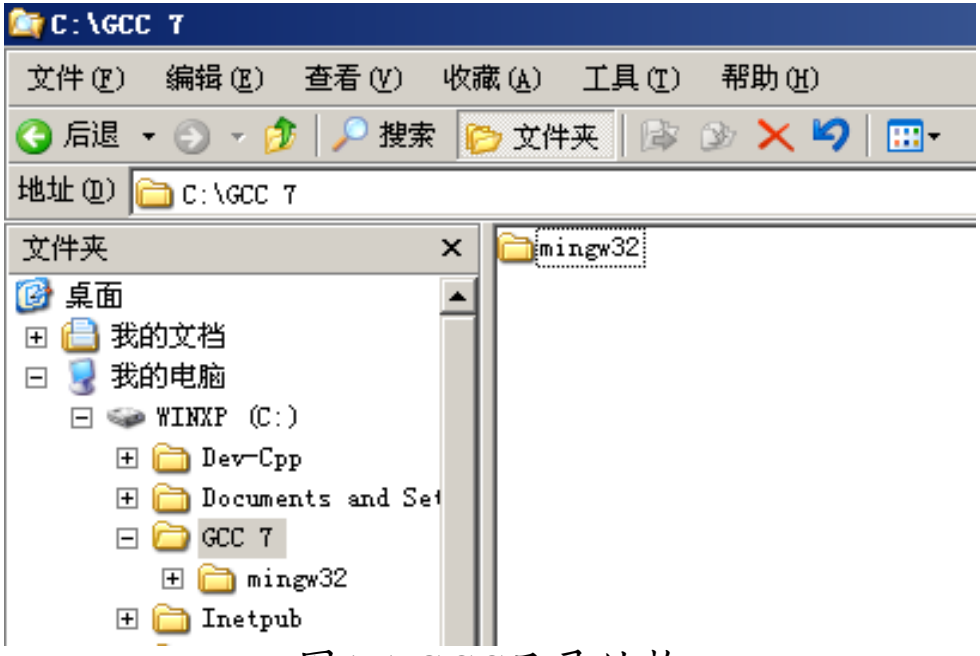
C:\GCC 7
文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)
后退 搜索 文件夹
地址(D) C:\GCC 7
文件夹
桌面
我的文档
我的电脑
WINXP (C:)
Dev-Cpp
Documents and Set
GCC 7
mingw32
Inetpub
mingw32

图1-1 GCC目录结构

1.6.2 编译器设置

1. 设置方法

从Tools菜单中选择Compiler Options打开Compiler Options对话框，在Compiler Options对话框中完成编译器设置。如图1-2所示。

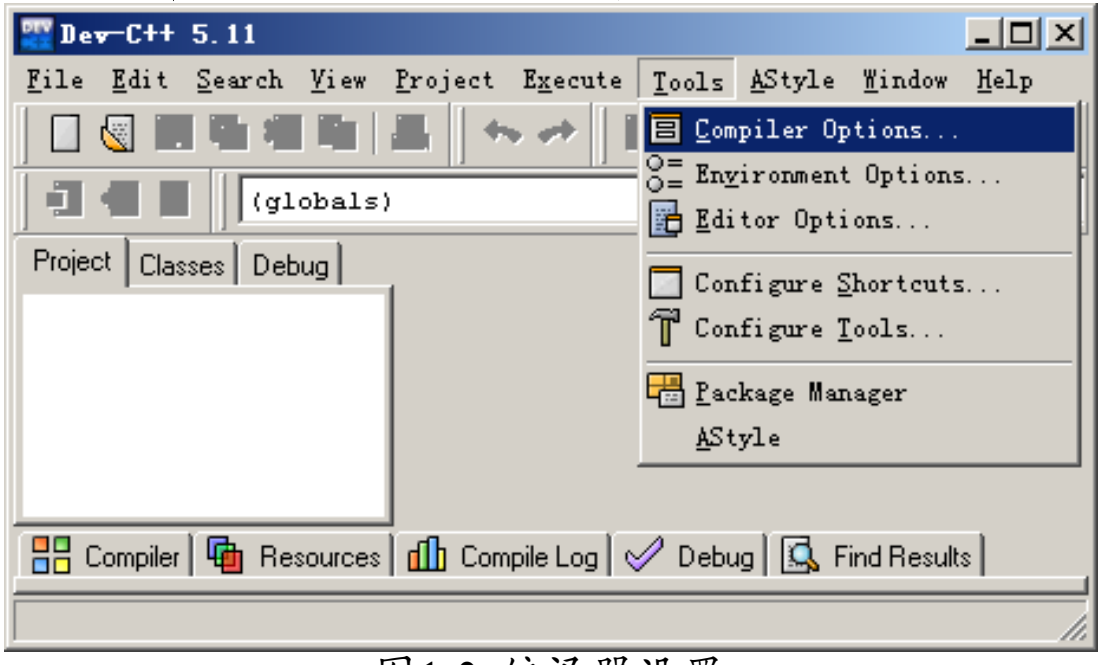

图1-2 编译器设置

2. 添加编译器程序组

使用“添加空程序组”按钮（第1个“+”号按钮）打开“新程序组”对话框，指定新程序组名称（如GCC 7）。如图1-3所示。

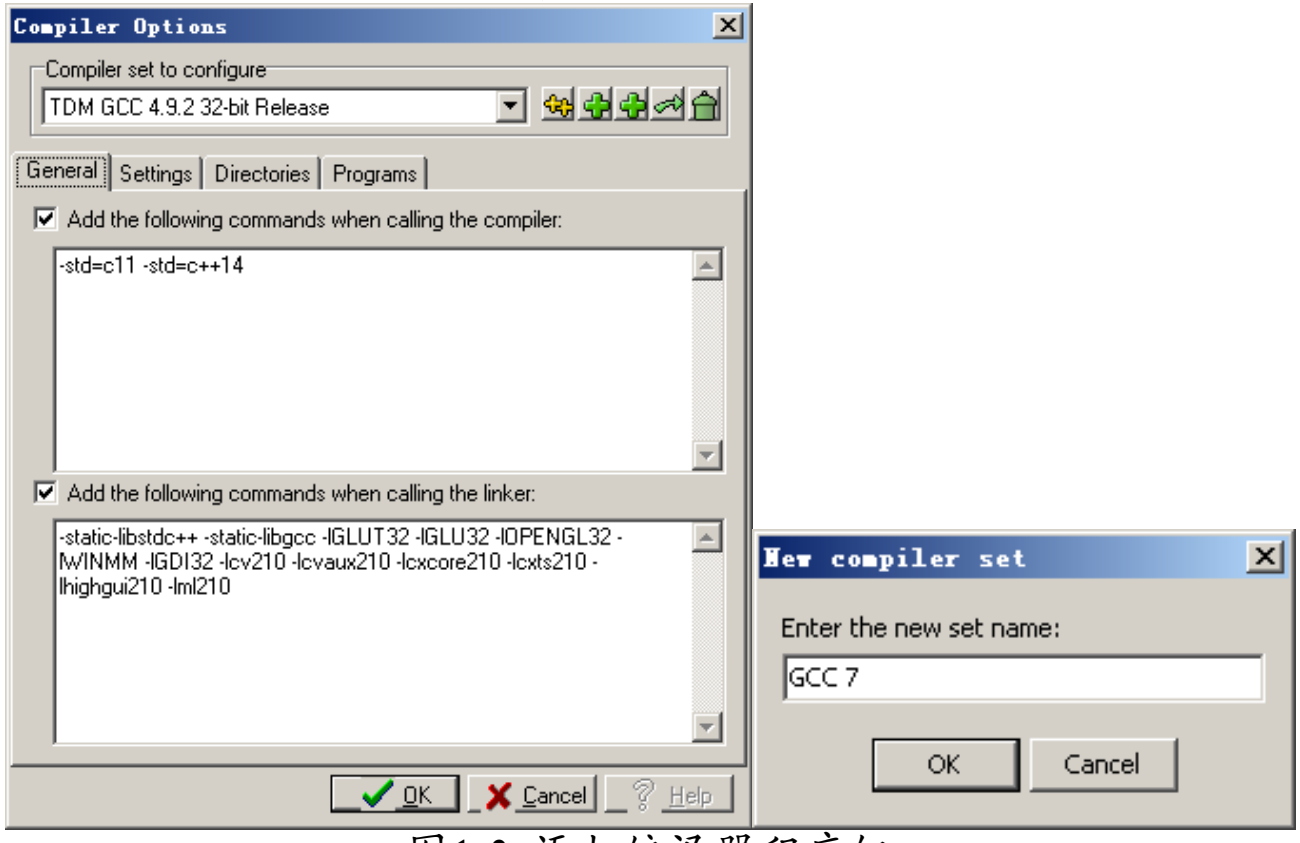

图1-3 添加编译器程序组

2. 语言支持

本教程有很多程序使用到了一些C++ 17新增的特性，可以使用选项-std=c++17使得编译器支持C++ 17。如图1-4所示。

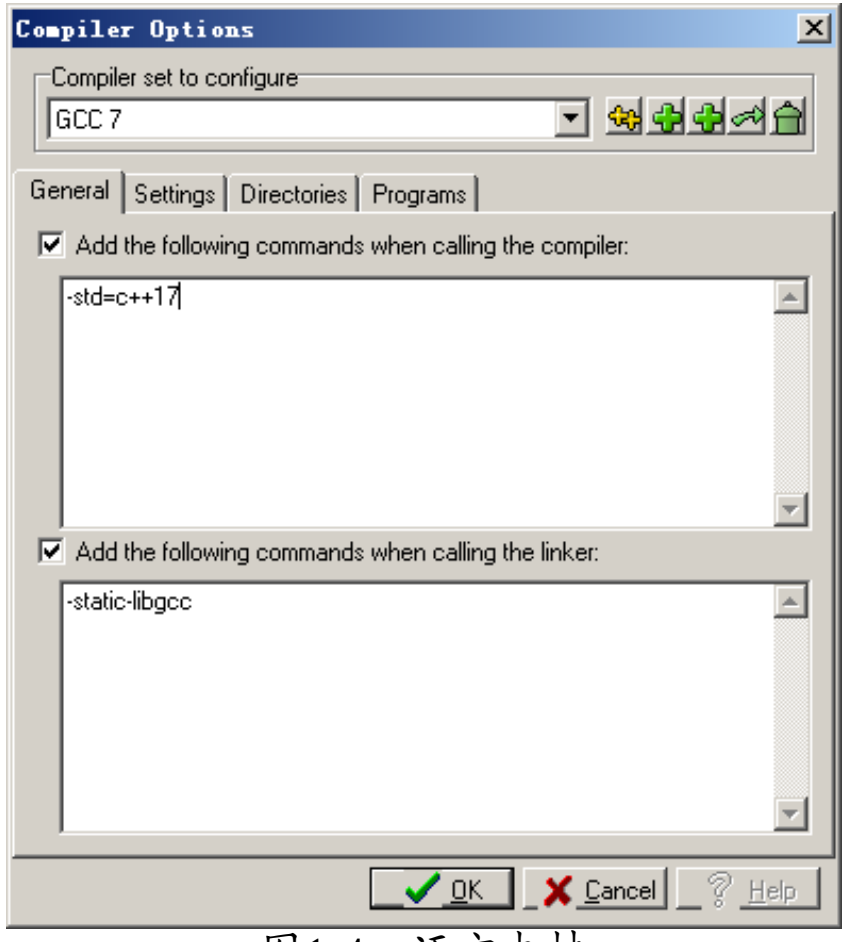

图1-4　语言支持

3. 可执行文件路径

首先从Directories页中选择Binaries，然后将编译和运行应用程序所需要的可执行文件的路径添加到列表中（可以使用“浏览”按钮选择文件路径），最后调整这些路径的顺序。如图1-5所示。

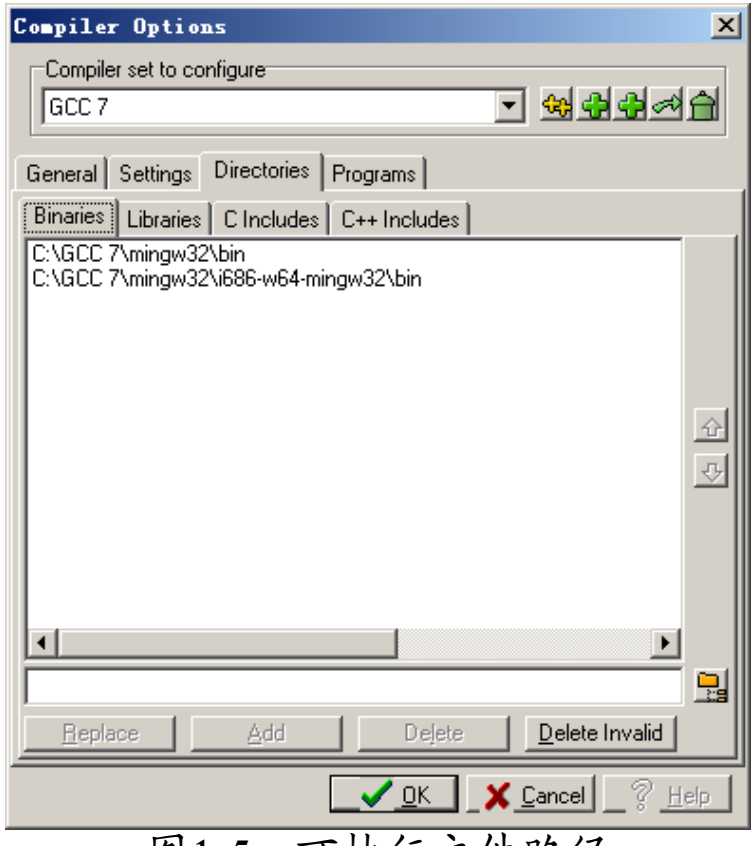

图1-5　可执行文件路径

4. 链接库文件路径

首先从Directories页中选择Libraries，然后将编译应用程序所需要的链接库文件的路径添加到列表中（可以使用“浏览”按钮选择文件路径），最后调整这些路径的顺序。如图1-6所示。

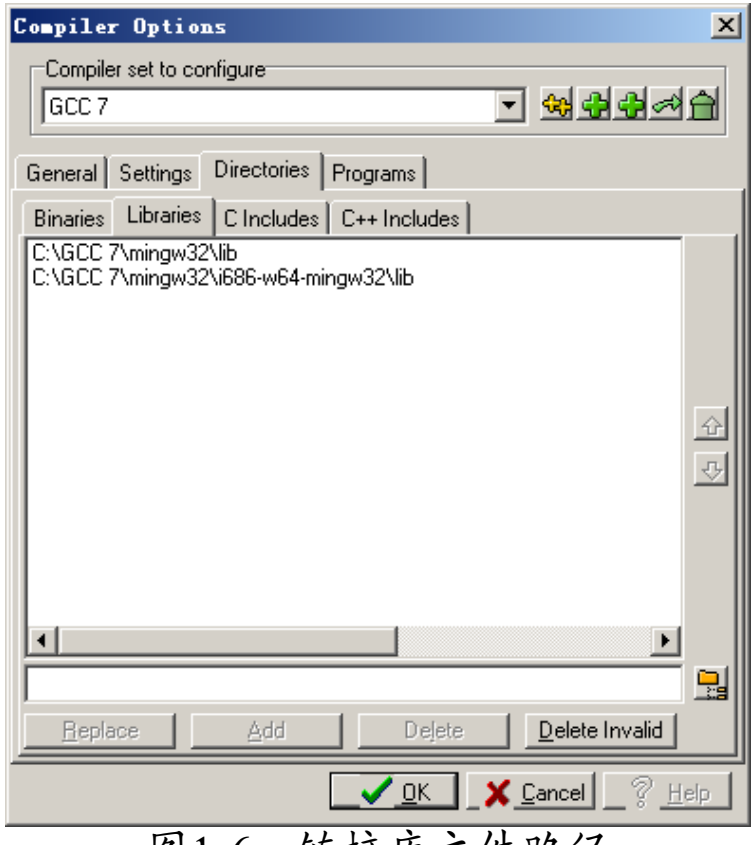

图1-6　链接库文件路径

5. C Include文件路径

首先从Directories页中选择C Includes，然后将编译应用程序所需要的C Include文件的路径添加到列表中（可以使用“浏览”按钮选择文件路径），最后调整这些路径的顺序。如图1-7所示。

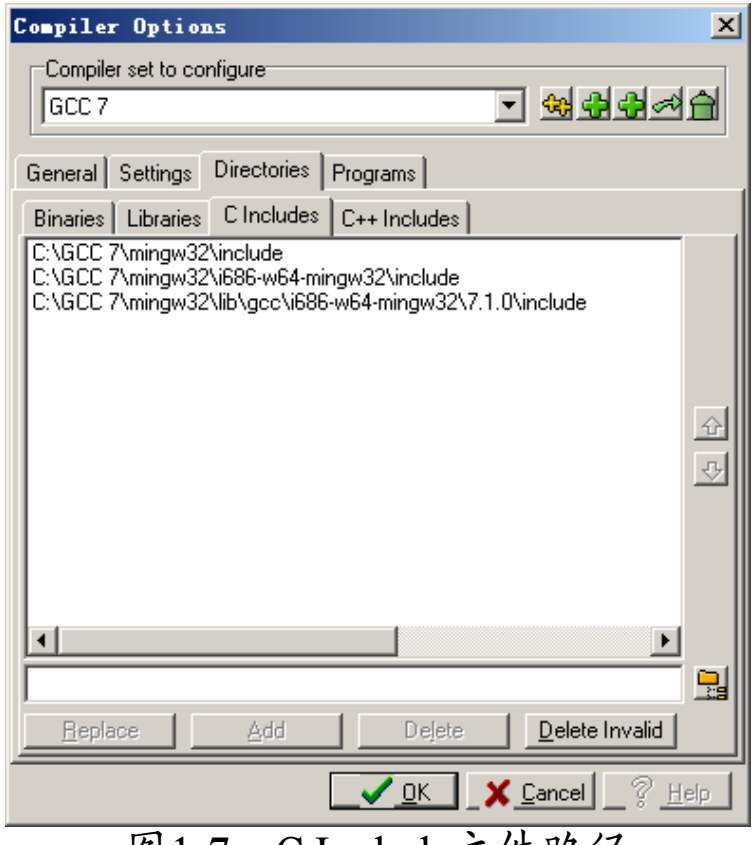

图1-7　C Include文件路径

6. C++ Include文件路径

首先从Directories中选择C++ Includes，然后将编译应用程序所需要的C++ Include文件的路径添加到列表中（可以使用“浏览”按钮选择文件路径），最后调整这些路径的顺序。如图1-8所示。

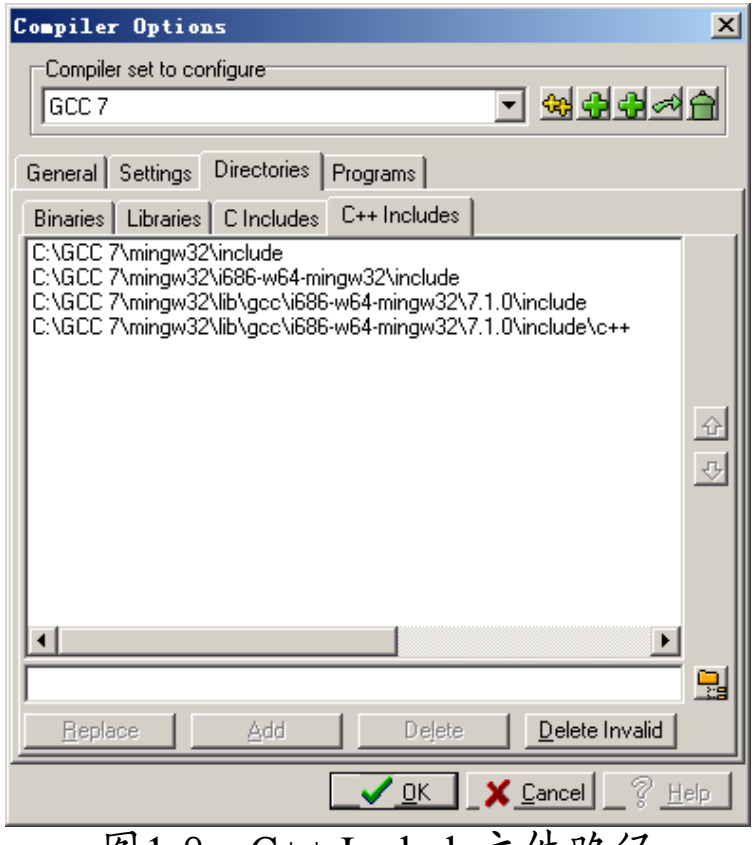

图1-8　C++ Include文件路径

7. 编译器文件

在Programs页中指定编译器各主要组成文件的文件名（可以使用“浏览”按钮选择文件路径）。如图1-9所示。

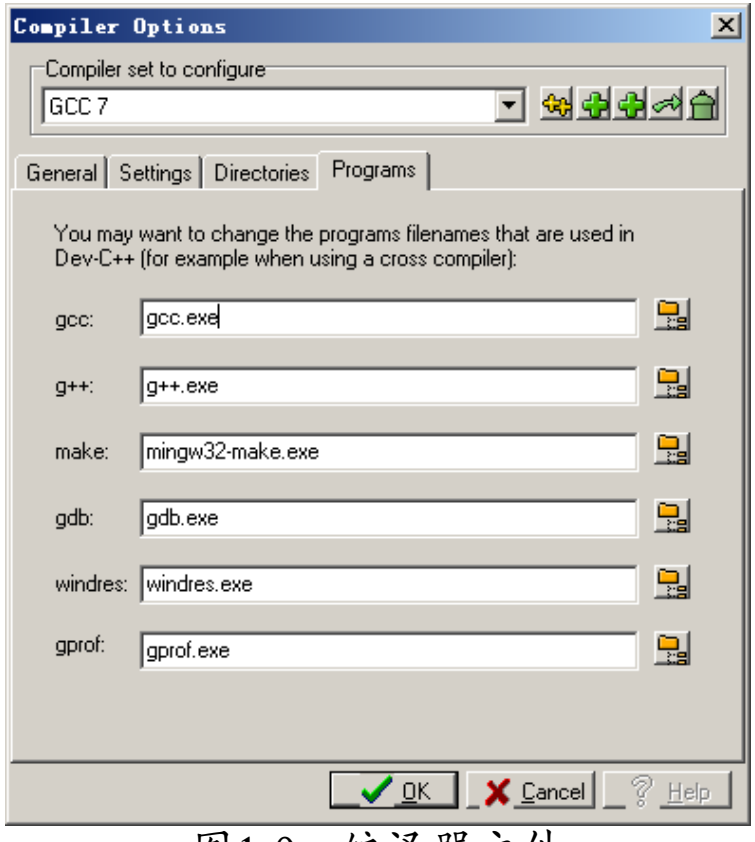

图1-9　编译器文件

1.7 几个I/O函数

本教程后续章节介绍的算法多次用到输出数组或容器中的元素等函数，这里给出这些函数的C++源程序。保存在文件ios.hpp中，需C++ 17支持。

1.7.1 预处理

```
// ios.hpp
#pragma once // 避免重复编译
#include <iostream> // io
#include <sstream> // ostringstream
#include <vector>
#include <set>
```

```
#ifndef _STD_BEGIN
#define _STD_BEGIN namespace std {
#define _STD_END }
#endif
```

```
_STD_BEGIN
```

1.7.2 数组或容器中的元素输出到字符串

1. 输出指定范围内的元素

```
// 数组或容器中的元素输出到字符串
template<class It> // 指定范围内的元素
string to_string(It F, It L, const char sp[] = " ")
{   if(F == L) return ""; // 没有元素
    ostringstream out;
    out << *F; // 首元素前不用分隔符
    // 后续元素前加上分隔符
    for(++F; F != L; ++F) out << sp << *F;
    return out.str();
}
```

2. 输出前n个元素

```
template<class T> // 数组的前n个元素
string to_string(T X[], int n, const char sp[] = " ")
{   return to_string(X, X + n, sp);
}
```

```
template<class Cont> // 容器的前n个元素
string to_string(const Cont &X, int n, const char sp[] = " ")
{   auto F = begin(X), L = next(F, n);
    return to_string(F, L, sp);
}
```

3. 输出指定序号范围的元素

输出数组或容器中序号属于区间[i, j]的元素。

```
template<class T> // 输出数组中指定序号范围的元素
string to_string(T X[], int i, int j, const char sp[] = " ")
{   return to_string(X + i, X + j + 1, sp);
}
```

```
template<class Cont> // 输出容器中指定序号范围的元素
string to_string(const Cont &X, int i, int j, const char sp[] = " ")
{   auto F = next(begin(X), i), L = next(begin(X), j + 1);
    return to_string(F, L, sp);
}
```

1.7.3 使用operator<<输出容器的元素

本书后续章节介绍的实例包含的容器对象只有vector对象和set对象，所以这里只给出vector对象和set对象的输出函数，其他容器对象的输出函数可类似实现。

```
// 使用operator<<输出vector和set的全部元素
template<class T, class A>
ostream &operator<<(ostream &out, const vector<T, A> &X)
{   return out << to_string(begin(X), end(X));
}
```

```
template<class K, class Pr, class A>
ostream &operator<<(ostream &out, const set<K, Pr, A> &X)
{   return out << to_string(begin(X), end(X));
}
```

另外，为了方便向vector对象和set对象中添加元素，这里还顺便给出用于vector对象和set对象的插入运算，向其他容器对象添加元素的函数可类似实现。

```
// 使用operator<<向vector和set添加元素
template<class T, class A, class T2>
auto &operator<<(vector<T, A> &X, const T2 &v)
{   X.push_back(v);
    return X;
}
template<class K, class Pr, class A, class K2>
auto &operator<<(set<K, Pr, A> &X, const K2 &v)
{   X.insert(v);
    return X;
}
_STD_END
```

1.7.4 使用方法举例

```
#include "ios.hpp"
using namespace std;
int main()
{   int n = 4;
    char X[] = {'A', 'B', 'C', 'D', 'E'};
    cout << to_string(X, n) << endl; //  数组: A B C D
    vector<bool> Y = {1, 0, 1, 0, 1};
    cout << to_string(Y, n) << endl; // vector: 1 0 1 0
    set<int> Z = {1, 2, 3, 4, 5};
    cout << to_string(Z, n) << endl; // set: 1 2 3 4
    cout << "\"" << X << "\"" << endl; // C串: "ABCDE8?"
    cout << "{" << Y << "}" << endl; // vector: {1 0 1 0 1}
    cout << "{" << Z << "}" << endl; // set: {1 2 3 4 5}
    cout << "{" << (set<bool>) {1, 0, 1, 0, 1} << "}" << endl; // {0 1}
    cout << to_string(X, 1, 3) << endl; //  数组: B C D
    cout << to_string(Z, 1, 3) << endl; // set: 2 3 4
}
```