C++语言程序设计：MOOC版

清华大学出版社（ISBN 978-7-302-42104-7）

第10章 C++标准库

中国農業大學 阚道宏

第10章 C++标准库

- C++标准库
  - 基于结构化程序设计方法所提供的**系统函数**
  - 基于面向对象程序设计方法所提供的**系统类库**

- 使用系统类库

  系统类库是一组预先定义好的类。程序员可以直接使用这些类来定义对象，也可以通过组合或继承来定义新的类。系统类库极大地扩展了C++语言的功能，使得程序员可以在更高的起点上开发程序

- 模板技术：函数模板、类模板

第10章 C++标准库

- 本章内容
  - 10.1 函数模板
  - 10.2 类模板
  - 10.3 C++标准库
  - 10.4 C++语言的异常处理机制
  - 10.5 数据集合及其处理算法
  - 10.6 结语

10.1 函数模板

- **函数模板**的基本原理是通过数据类型的参数化，将一组算法相同但所处理数据类型不同的重载函数凝练成一个函数模板。编译时，再由编译器按照函数模板自动生成针对不同数据类型的重载函数定义代码

```
int     max( int x, int y )        { return (x>y ? x:y); } // 求2个整数的最大值
double  max( double x, double y ) { return (x>y ? x:y); } // 求2个实数的最大值
char    max( char x, char y )      { return (x>y ? x:y); } // 求2个字符的最大值

template <typename T> // 定义函数模板max
T  max( T x, T y )
{ return ( x>y ? x : y ); }
```

10.1 函数模板

C++语法：**定义函数模板**

```
template <类型参数列表>
函数类型 函数名(形式参数列表)
{
   函数体
}
```

语法说明：

- 定义函数模板以关键字“**template**”开头；
- **类型参数**是一种表示数据类型的参数。**类型参数列表**可定义一个或多个类型参数，每个类型参数以“**typename 类型参数名**”或“**class 类型参数名**”的形式定义，类型参数之间用“,”隔开。类型参数名需符合标识符的命名规则；
- 函数模板定义的其余部分，包括函数类型、函数名、形式参数列表以及函数体，它们和普通函数的定义形式没有什么区别。唯一不同的是，类型参数将会作为一种新的数据类型出现函数模板定义中；
- 使用typename或class所声明的类型参数就像是一种新的数据类型，可以用它来定义函数类型（即返回值的类型），也可以用来定义形参或在函数体中定义局部变量。类型参数是表示数据类型的参数，调用时可被替换成任意一种实际数据类型，例如某种基本数据类型、自定义数据类型或类类型等。类型参数也可理解成是一种通用数据类型；
- 函数模板定义中的前2行被称为是**函数模板头**。

10.1 函数模板

例10-1 一个完整的求最大值函数模板max的C++演示程序

```
1  #include <iostream>
2  using namespace std;
3
4  template  <typename T>  // 定义函数模板max
5  T  max( T x, T y )  // 使用类型参数T来指定函数的类型、形参x和y的类型
6  {
7      return ( x>y ? x : y );
8  }
9
10 int main()
11 {
12     cout << max( 5, 10 ) << endl;  // 调用函数模板max求2个整数的最大值，显示结果：10
13     cout << max( 5.2, 10.2 ) << endl;  // 调用函数模板max求2个实数的最大值，显示结果：10.2
14     return 0;
15 }
```

10.1 函数模板

- 函数模板的编译原理

```
template  <typename T>  // 定义函数模板max，T为类型参数
T  max( T x, T y )
{  return ( x>y ? x : y );  }
```

– cout << max(5, 10) << endl;

```
int  max( int x, int y )  // 用int取代类型参数T
{  return ( x>y ? x : y );  }
```

– cout << max(5.2, 10.2) << endl;

```
double  max( double x, double y )  // 用double取代类型参数T
{  return ( x>y ? x : y );  }
```

10.1 函数模板

- 函数模板的**编译原理**

  函数模板是具有类型参数的函数。类型参数是表示数据类型的参数，可指代任意一种实际数据类型。编译器在编译到函数模板调用语句时，根据位置对应关系从实参数据类型推导出类型参数所指代的数据类型，然后按照函数模板自动生成一个该类型的函数定义代码。不同类型实参的函数模板调用语句将生成不同类型的重载函数

- 函数模板将数据类型参数化，调用时会呈现出**参数多态性**

10.1 函数模板

例10-2 一个演示函数模板语法的C++程序

(a) 使用**函数模板**的程序

```
1  #include <iostream>
2  using namespace std;
3
4  template  <typename T>  // 函数模板
5  T  max( T x, T y )
6  {
7      return ( x>y ? x : y );
8  }
9
10 int main()
11 {
12     cout << max( 5, 10 ) << endl;
13     cout << max( 5.2, 10.2 ) << endl;
14     return 0;
15 }
```

(b) 等价的**重载函数**程序

```
#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()
{
   cout << max( 5, 10 ) << endl;
   cout << max( 5.2, 10.2 ) << endl;
   return 0;
}
```

- 使用函数模板可以减少程序员的编码工作量，但所编译出的可执行代码不会减少，执行效率也不变

10.1 函数模板

- 函数模板的声明

template <类型参数列表>
函数类型 函数名(形式参数列表

template <类型参数列表> 函数

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

template  <typename T>
T  fun( T x, int y ) ;  // 函数模板fun的声明

int main ( )
{
   cout << fun( 10, 5 ) << endl;  // 函数模板fun的调用
   cout << fun( 10.2, 5 ) << endl;  // 函数模板fun的调用
   return 0;
}

template  <typename T>  // 函数模板fun的定义
T  fun( T x, int y )
{
   T  z;  // 定义T类型的变量z
   z = (T)( x+y );  // 将x+y的结果转换成T类型
   return  z;
}
```

10.1 函数模板

- 程序员编程时可灵活运用模板技术。在定义多个重载函数时应考虑是否可以将它们合并成一个函数模板，这样可以凝练代码。在定义单个函数时应考虑是否可以将该函数升级成一个函数模板，这样可以提高函数代码的可重用性

- 在调用函数模板的程序员看来，函数模板与普通函数没有什么区别。唯一不同的是，函数模板就像是一个具有通用类型的函数，可以处理不同类型的数据

10.2 类模板

- 应用模板技术，可以将一组功能相同但所处理数据类型不同的类凝练成一个类模板

- 编译时，再由编译器按照类模板自动生成针对不同数据类型的类定义代码

10.2 类模板

C++语法：**定义类模板**

```
template <类型参数列表>
class 类名 // 类声明部分
{
   类成员声明
}
// 类实现部分：所有类外定义的函数成员，必须按如下的语法形式将它们定义成函数模板
template <类型参数列表>
函数类型 类名<类型参数名列表> :: 函数名( 形式参数列表 )
{ 函数体 }
```

语法说明：

- 定义类模板以关键字“**template**”开头；
- **类型参数**是一种表示数据类型的参数。**类型参数列表**可定义一个或多个类型参数，每个类型参数以“**typename 类型参数名**”或“**class 类型参数名**”的形式定义，类型参数之间用“,”隔开。类型参数名需符合标识符的命名规则；
- 类模板定义的其余部分，包括类名、类成员声明以及类实现部分，它们和普通类的定义形式基本相同。所不同的是，类型参数就像是一种新的数据类型，可以用它来定义数据成员，也可以用来定义函数成员；
- 定义模板类的函数成员，如果在类内（即声明部分）定义，其语法形式与普通类的函数成员没有区别；如果在类外（即类实现部分）定义，则必须按照函数模板的语法形式来定义，并且要在函数名前加“类名**<类型参数名列表>**::”限定。类型参数名列表应列出类型参数列表中的所有类型参数名，多个参数名之间用“,”隔开；
- 类模板中定义的类型参数是表示数据类型的参数。使用类模板时，类型参数可被替换成任意一种实际数据类型，例如某种基本数据类型、自定义数据类型或类类型等。类型参数也可理解成是一种通用数据类型。

例10-3 两个演示类模板语法的C++程序

(a) 在**类内**定义**函数模板成员**（内联）

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T> // 类模板A
5  class A // 类声明部分
6  {
7  private: // 声明以下2个数据成员
8      T a1;
9      int a2;
10 public: // 定义以下3个函数成员
11     A(T p1, int p2) // 构造函数
12     { a1 = p1; a2 = p2; }
13     T Sum( ) // 求数据成员的和
20 int main( )
21 {
22     // 用类模板A定义对象，并访问其成员
23     A <double> o1(10.5, 6); //double型对象o1
24     o1.Show( ); // 显示：10.5, 6
25     cout << o1.Sum( ) << endl; // 显示：16.5
26
27     A <int> o2(10, 6); // int型对象o2
28     o2.Show( ); // 显示：10, 6
29     cout << o2.Sum( ) << endl; // 显示：16
30     return 0;
31 }
```

(b) 在**类外**定义**函数模板成员**

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

template <typename T> // 类模板A
class A // 类声明部分
{
private: // 声明以下2个数据成员
    T a1;
    int a2;
public: // 声明以下3个函数成员
    A(T p1, int p2); // 构造函数
    T Sum( ); // 求数据成员的和
    void Show( ); // 显示数据成员

template <typename T>
T A <T> :: Sum( ) // 求数据成员的和
{ return (T)( a1 + a2 ); }

template <typename T>
void A <T> :: Show( ) // 显示数据成员
{ cout << a1 << " " << a2 << endl; }

int main( )
{ // ..... 主函数代码同(a)，省略 }
```

使用类模板定义对象时需要明确给出类型参数所指代的实际数据类型。

类模板名 <实际数据类型列表> 对象名1, 对象名2, ... ;

10.2 类模板

- 类模板的编译原理

A <**double**> o1(10.5, 6);

– 该语句使用类模板A定义对象o1，定义时给出了实际数据类型double。编译该语句，编译器首先用数据类型“double”取代类模板A中所有的类型参数T，从而自动生成一个double型类A。double型类A是一个普通的类。最终编译器是用这个double型类A来定义对象o1

– 编译时将类模板中类型参数绑定到某个具体数据类型的过程，称为类模板的实例化。实例化所生成的类称为类模板的实例类。实例类是一个普通的类，可以用来定义对象

10.2 类模板

- 类模板的编译原理

  A <int> o2(10, 6);

  – 该语句使用类模板A定义对象o2，定义时给出了实际数据类型int。编译该语句，编译器将再次使用类模板A进行实例化，自动生成一个int型类A。然后再用这个int型类A定义出对象o2

- 类模板的**编译原理**是：类模板是具有类型参数的类。类型参数是表示数据类型的参数，可指代任意一种实际数据类型。编译器在编译到使用类模板定义对象语句时，将首先按照所给定的实际数据类型对类模板进行实例化，生成一个实例类。最终，编译器是使用实例类来定义所需要的对象

10.2 类模板

- 使用typedef类型定义**显式地**实例化类模板

```
int main( )
{
   typedef  A<double>  DoubleA;  // 实例化类模板A，生成实例类DoubleA
   DoubleA  o1(10.5, 6);  // 定义类DoubleA的对象o1
   o1.Show( );  // 显示：10.5, 6
   cout << o1.Sum( ) << endl;  // 显示：16.5

   typedef  A<int>  IntA;  // 实例化类模板A，生成实例类IntA
   IntA  o2(10, 6);  // 定义类IntA的对象o2
   o2.Show( );  // 显示：10, 6
   cout << o2.Sum( ) << endl;  // 显示：16
   return 0;
}
```

- 使用类模板可以减少程序员的编码工作量，但所编译出的可执行代码不会减少，执行效率也不变

10.2 类模板

- 类模板的继承与派生
  - 类模板可以被继承，派生出新类
  - 以类模板为基类定义派生类时
    - 可以在派生时实例化
    - 也可以继续定义派生类模板。

10.2 类模板

- 定义类模板的实例化派生类

例10-4 一个实例化派生类的C++演示程序

(a) 基类：类模板Base　　(b) 派生类：实例化类Derived

```
1  #include <iostream>
2  using namespace std;
3
4  // 基类Base：是一个类模板
5  template <typename T>
6  class Base // 声明部分
7  {
8  private:
9      T a; // 数据成员
10 public:
11     Base(T x) // 构造函数
12     { a = x; }
13     void Show( ) // 显示数据成员
14     { cout << "a=" << a << ", "; }
15 };
16
17
18
19
20
21
22
```

```
// 派生类Derived：公有继承类模板Base，派生时实例化
class Derived : public Base <double>
{
private:
    int b; // 新增数据成员
public:
    // 注意派生类构造函数的写法
    Derived(double p1, int p2) : Base <double>( p1 )
    { b = p2; }
    void Show( ) // 新增函数成员，覆盖基类同名函数Show
    {
        Base <double>::Show( ); // 调用基类同名函数
        cout << "b=" << b << endl;
    }
};

int main( )
{
    Derived obj(10.5, 6); // 定义派生类Derived的对象obj
    obj.Show( ); // 显示结果：a=10.5, b=6
    return 0;
}
```

10.2 类模板

- 定

例10-5 一个派生类模板的C++演示程序

(a) **基类**：类模板Base

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

// 基类Base：是一个类模板
template  <typename T>
class  Base  // 声明部分
{
private:
    T  a;  // 数据成员
public:
    Base(T  x)  // 构造函数
    {  a = x;  }
    void Show( )  // 显示数据成员
    {  cout << "a=" << a << “, “;  }
};
```

(b) **派生类**：类模板Derived

```
// 派生类Derived：公有继承类模板Base，派生类仍为类模板
template  <typename T, typename TT>  // 新增类型参数TT
class  Derived : public Base <T>
{
private:
    TT  b;  // 新增数据成员
public:
    // 注意派生类构造函数的写法
    Derived(T  p1, TT  p2) : Base <T>( p1 )
    {  b = p2;  }
    void Show( )  // 新增函数成员，覆盖基类同名函数Show
    {
        Base <T>::Show( );  // 调用基类同名函数
        cout << "b=" << b << endl;
    }
};

int main( )
{
  // 使用派生类模板Derived定义对象obj
    Derived <double, int>  obj(10.5, 6);
    obj.Show( );  // 显示结果：a=10.5, b=6
    return 0;
}
```

10.2 类模板

- 程序员编程时可灵活运用模板技术。在定义多个功能相同但所处理数据类型不同的类时应考虑是否可以将它们合并成一个类模板，这样可以凝练代码。在定义单个类时应考虑是否可以将它升级成一个类模板，这样可以提高类代码的可重用性

- 在使用类模板的程序员看来，类模板与普通类没有什么太大区别。只是在使用类模板时需要给出类型参数所指代的实际数据类型

10.3 C++标准库

- 10大类功能
  - 语言支持（Language Support）类
  - 通用工具（General Utilities）类
  - 输入/输出（Input/Output）类
  - 字符串（String）类
  - 诊断（Diagnostics）类
  - 容器（Container）类、迭代器（Iterator）类型和算法（Algorithm）
  - 数值（Numerics）类
  - 本地化（Locale）类

10.3 C++标准库

- 为了凝练代码，C++标准库广泛采用了模板技术，因此C++标准库有时也被称作标准模板库（Standard Template Library, STL）

- 因为使用了模板技术，C++标准库能以较少的代码量却提供了很强大的功能

10.3 C++标准库

```
// 输入/输出流类模板（此处仅列出类模板的声明）
template <class charT> class  basic_ios;
template <class charT> class  basic_istream;
template <class charT> class  basic_ostream;
template <class charT> class  basic_iostream;
template <class charT> class  basic_ifstream;
template <class charT> class  basic_ofstream;
template <class charT> class  basic_fstream;
template <class charT> class  basic_istringstream;
template <class charT> class  basic_ostringstream;
template <class charT> class  basic_stringstream;
```

10.3 C++标准库

```
// 使用类模板定义基于ANSI编码的输入/输出流类
typedef  basic_ios<char>  ios;
typedef  basic_istream<char>  istream;
typedef  basic_ostream<char>  ostream;
typedef  basic_iostream<char>  iostream;
typedef  basic_ifstream<char>  ifstream;
typedef  basic_ofstream<char>  ofstream;
typedef  basic_fstream<char>  fstream;
typedef  basic_istringstream<char>  istringstream;
typedef  basic_ostringstream<char>  ostringstream;
typedef  basic_stringstream<char>  stringstream;
```

10.3 C++标准库

```
// 使用类模板定义基于Unicode编码的输入/输出流类
typedef  basic_ios<wchar_t>  wios;
typedef  basic_istream<wchar_t>  wistream;
typedef  basic_ostream<wchar_t>  wostream;
typedef  basic_iostream<wchar_t>  wiostream;
typedef  basic_ifstream<wchar_t>  wifstream;
typedef  basic_ofstream<wchar_t>  wofstream;
typedef  basic_fstream<wchar_t>  wfstream;
typedef  basic_istringstream<wchar_t>  wistringstream;
typedef  basic_ostringstream<wchar_t>  wostringstream;
typedef  basic_stringstream<wchar_t>  wstringstream;
```

10.4 C++语言的异常处理机制

- 程序错误可分为3类
  - 语法错误
  - 语义错误（逻辑错误）
  - 运行时错误

- C++语言针对程序运行时错误设计了专门的异常处理机制，即try-catch机制

- C++标准库为异常处理机制提供多种不同功能的异常类

10.4 C++语言的异常处理机制

- 语法错误

例10-6 一个简单的C++除法运算程序（存在**语法**错误）

```
1  #include <iostream>
2  using namespace std;
3
4  int  Div( int n )  // 求100 ÷ n
5  { return  ( 100 / n ) ; }
6
7  int  main( )
8  {
9      int  N;  // 定义变量N，保存键盘输入的人数
10     cin << N;  // 语法错误：输入N的值，正确语句应为：cin >> N;
11     int  result = Div( N );  // 调用函数Div进行除法运算
12     cout << "100÷" << N << "=" << result << endl;  // 显示100 ÷ N的结果
13     return 0;
14 }
```

‘<<’: class istream does not define this operator

10.4 C++语言的异常处理机制

- 语义错误

例10-7 一个简单的C++除法运算程序（存在**语义**错误）

```
1  #include <iostream>
2  using namespace std;
3
4  int  Div( int n )  // 求100 ÷ n
5  { return  ( 100 * n ) ; } // 语义错误：将除法错误地写成了乘法，语法正确但语义错误
6
7  int  main( )
8  {
9      int  N; // 定义变量N，保存键盘输入的人数
10     cin >> N;
11     int  result = Div( N ); // 调用函数Div进行除法运算
12     cout << "100÷" << N << "=" << result << endl; // 显示100 ÷ N的结果
13     return 0;
14 }
```

```
2<回车键>
100 ÷2=200
```

10.4 C++语言的异常处理机制

- 运行时错误

例10-8 一个简单的C++除法运算程序（无任何**语法**或**语义**错误）

```
1  #include <iostream>
2  using namespace std;
3
4  int  Div( int n )  // 求100 ÷ n
5  {  return  ( 100 / n ) ;  }
6
7  int  main( )
8  {
9      int  N;  // 定义变量N，保存键盘输入的人数
10     cin >> N;
11     int  result = Div( N );  // 调用函数Div进行除法运
12     cout << "100÷" << N << "=" << result << endl;  //
13     return 0;
14 }
```

2<回车键>
100 ÷2=50

0 <回车键>

10.4 C++语言的异常处理机制

- 程序运行时，因运行环境差异或用户操作不当所造成的程序错误被统称为**运行时错误**
  - 运行环境存在差异、或用户出现操作不当，这些都被称为程序运行时的**异常情况**
  - 程序员无法阻止异常情况的出现，但可以在程序中添加**异常处理机制**，避免因异常情况而导致程序死机等严重的运行时错误

10.4 C++语言的异常处理机制

- 程序运行时，常见的异常情况有：
  - **用户操作不当**。例如，运行除法运算程序时输入了人数0
  - **输入文件不存在**。从文件中输入数据，但文件不存在，打开（open）文件将会失败。如果文件未能成功打开，继续对文件进行读/写操作将导致运行时错误
  - **非法访问内存单元**。例如，数组越界、或使用未经初始化的指针变量等，都将导致运行时错误
  - **网络连接中断**。程序可以通过网络发送/接收数据。在网络连接中断的情况下，继续发送/接收数据将导致运行时错误

10.4 C++语言的异常处理机制

- 一个**异常处理机制**由2部分组成
  - **发现异常**。程序员应在可能出现异常的程序位置增加检查异常的代码，其目的就是及时发现异常
  - **处理异常**。发现异常后，程序需要改变算法流程，否则将产生运行时错误。处理异常就是在算法中增加异常处理流程。没有异常时，程序执行正常算法流程。发现异常时，程序执行异常处理流程

10.4 C++语言的异常处理机制

例10-9 一个简单的C++除法运算程序（具有**异常处理机制**）

```
1  #include <iostream>
2  using namespace std;
3
4  int  Div( int n ) // 求100 ÷ n
5  {
6      if ( n <= 0 ) // 检查异常：如果n<=0，则属于异常情况
7          return ( -1 ); // 异常处理流程
8      // 如果没有异常，则执行如下的正常处理流程
9      return ( 100 / n ) ;
10 }
11
12 int  main( )
13 {
14     int  N; // 定义变量N，保存键盘输入的人数
15     cin >> N;
16     int  result = Div( N ); // 调用函数Div进行除法运算
17     if ( result == -1 ) // 检查函数Div的返回值：-1表示函数Div出现了异常
18        cout << “输入人数时必须为正整数” << endl; // 向用户显示错误信息
19     else
20         cout << "100÷" << N << "=" << result << endl; // 显示100 ÷ N的结果
21     return 0;
22 }
```

2<回车键>
100 ÷2=50

0 <回车键>
输入人数时必须为正整数

错误代码：用于表示不同的异常情况

10.4 C++语言的异常处理机制

- 运用异常处理机制，将同时涉及到主调函数与被调函数
  - 被调函数负责发现异常。发现异常后应处理异常，并向上级主调函数报告异常
  - 常规的报告方法是通过特定的函数返回值（即错误代码）来表示不同的异常情况
  - 主调函数通过检查被调函数的返回值来检查调用时是否出现了异常，发现异常后也需要做相应的异常处理
  - 如果是函数多级调用，主调函数还需要逐级向各自的上级函数报告异常

- 涉及到主调函数与被调函数之间的异常处理机制被称为**多级异常处理机制**
  - 发现异常
  - 处理异常
  - 报告异常

- C++语言提供了try-catch异常处理机制。

10.4 C++语言的异常处理机制

- try-catch异常处理机制

C++语法：**throw语句**

throw 异常表达式 ;

语法说明：

- **异常表达式**的结果可以是基本数据类型、自定义数据类型或类类型。异常表达式结果的数据类型被用于区分不同类型的异常，结果的值被用于描述异常的详细信息。异常表达式可以是单个常量、变量或对象；
- 计算机执行该语句，将抛出一个异常，并退出当前函数的执行。throw的功能是报告异常，该异常将被try-catch语句捕获并做相应的异常处理。

举例：

```
throw  15;  // 抛出1个int型异常，该异常的详细信息为15
throw  “Error name”;  // 抛出1个字符串型异常，该异常的详细信息为” Error name”
```

10.4 C++语言的异常处理机制

C++语法：**try-catch语句**

```
try
{
   受保护代码段
}
catch ( 异常类型1 )
{ 异常类型1的处理代码 }
catch ( 异常类型2 )
{ 异常类型2的处理代码 }
.....
```

语法说明：

- 如果预计某个程序代码段有可能发生异常，程序员可使用**try子句**将该代码段保护起来。受保护代码段在执行时将启用C++语言的异常保护机制，捕获该代码段执行过程中的任何异常报告，包括下级被调函数所报告的异常；
- **catch子句**负责捕获并处理异常。每个catch子句只负责一种类型的异常。异常类型就是抛出该异常的throw语句中表达式结果的类型；
- 若受保护代码段在执行过程中未报告任何异常，则所有的catch子句都不会执行；
- 若受保护代码段在执行过程中有异常，则根据异常类型依次匹配catch子句。如果匹配上某个catch子句（称异常被捕获），则执行所匹配catch子句的处理代码。每个异常最多只会有一个catch子句的处理代码被执行，其它catch子句都不会执行。如果异常未被任何catch子句捕获，则自动逐级向上级函数继续报告该异常，直到被上级函数中的某个catch子句所捕获。假设某个异常，连最上级的主函数main也未能捕获它，也就是说没有任何函数能够捕获并处理该异常，则自动使用**默认方法**来进行处理。默认的异常处理方法通常是中止当前程序的执行，并提示简单的错误信息；
- **catch(...)**形式的子句可匹配并捕获任意类型的异常，其后面的catch子句都将是无效子句。因此catch(...)形式子句应放在所有catch子句的最后。

10.4 C++语言的异常处理机制

例10-10 一个简单的C++除法运算程序（具有**try-catch**异常处理机制）

```
1  #include <iostream>
2  using namespace std;
3
4  int  Div( int n )  // 求100 ÷ n
5  {
6      if ( n <= 0 )  // 检查异常：如果n<=0，则属于异常情况
7          throw  ( -1 );  // 异常处理流程：抛出异常，然后退出当前函数的执行
8      // 如果没有异常，则执行如下的正常处理流程
9      return  ( 100 / n ) ;
10 }
11
12 int  main( )
13 {
14     int  N;  // 定义变量N，保存键盘输入的人数
15     cin >> N;
16     try
17     {
18         int  result = Div( N );  // 调用函数Div进行除法运算
19         cout << "100÷" << N << "=" << result << endl;  // 显示100 ÷ N的结果
20     }
21     catch ( int )  // 捕获int型异常
22     {
23       cout << “输入人数时必须为正整数” << endl;  // 向用户显示错误信息
24     }
25     catch ( ... )  // 捕获任意类型的异常
26     {
27       cout << “发生了其它异常” << endl;  // 向用户显示错误信息
28     }
29     return 0;
30 }
```

10.4 C++语言的异常处理机制

- 在catch子句中接收异常的详细信息

```
catch ( int x )
{
   cout << “输入人数时必须为正整数” << endl;
   cout << “异常的详细信息：” << x << endl;
}
```

0<回车键>

输入人数时必须为正整数
异常的详细信息：-1

10.4 C++语言的异常处理机制

- 使用类描述异常
  - throw语句所抛出的异常可以是基本数据类型、自定义数据类型或类类型
  - 类类型的异常可以提供更多的异常信息和异常处理功能

例10-11 一个简单的C++除法运算程序（使用**异常类**）

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

class  Error  // 异常类Error
{
public:
     int errCode;  // 异常代码
     char errMsg[40];  // 异常信息
     Error( int code, char *msg )  // 构造函数
     {  errCode = code;  strcpy( errMsg, msg );  }
     void ShowError( )  // 显示异常的详细信息
     {  cout << errCode << ": " << errMsg << endl;  }
};

int  Div( int n )  // 求100 ÷ n
{
     if ( n <= 0 )  // 检查异常：如果n<=0，则属于异常情况
     {
          Error  err( -1, "输入人数时必须为正整数" );  // 定义异常类Error的对象err
          throw  ( err );  // 抛出异常对象err，然后退出当前函数的执行
     }
     // 如果没有异常，则执行如下的正常处理流程
     return  ( 100 / n ) ;
}

int  main( )
{
     int  N;  // 定义变量N，保存键盘输入的人数
     cin >> N;
     try
     {
          int  result = Div( N );  // 调用函数Div进行除法运算
          cout << "100÷" << N << "=" << result << endl;  // 显示100 ÷ N的结果
     }
     catch ( Error e )  // 捕捉Error类的异常，并定义参数e来接收异常对象
     {
       e.ShowError( );  // 向用户显示所捕捉到异常对象e的详细信息
     }
     return 0;
}
```

10.4 C++语言的异常处理机制

- 关键字throw的另外2个用法

```
catch( ... ) // 可捕捉到所有异常
{
    ..... // 处理异常
    throw ; // 将当前捕捉到的异常再次抛出
}
```

– 声明异常：

```
void fun( ) throw ( A, B, C ) ; // 函数fun能且只能抛出A、B或C共3种类型的异常
void fun( ) throw ( ) ; // 函数fun不会抛出任何类型的异常
void fun( ) ; // 函数fun可能会抛出任何类型的异常
```

10.4 C++语言的异常处理机制

- C++标准库中的异常类exception

例10-12 标准异常类exception的示意代码

```
1  class exception
2  {
3  private:    char *msg ;  // 保存异常信息的数据成员
4  public:
5     exception();  // 构造函数
6     exception( const char * );
7     exception( const exception & );
8     exception & operator=(const exception & );  // 重载赋值运算符
9     virtual ~exception();  // 析构函数
10    virtual const char *what();  // 函数成员：返回异常信息
11 };
```

10.4 C++语言的异常处理机制

例10-13 一个简单的C++除法运算程序（使用**标准异常类exception**）

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4
5  int  Div( int n )  // 求100 ÷ n
6  {
7      if ( n <= 0 )  // 检查异常：如果n<=0，则属于异常情况
8      {
9          exception  err( “输入人数时必须为正整数” );  // 定义exception类的对象err
10         throw  ( err );  // 抛出异常对象err，然后退出当前函数的执行
11     }
12     // 如果没有异常，则执行如下的正常处理流程
13     return  ( 100 / n ) ;
14 }
15
16 int  main( )
17 {
18     int  N;  // 定义变量N，保存键盘输入的人数
19     cin >> N;
20     try
21     {
22         int  result = Div( N );  // 调用函数Div进行除法运算
23         cout << "100÷" << N << "=" << result << endl;  // 显示100 ÷ N的结果
24     }
25     catch ( exception  &e )  // 捕捉exception类的异常，并定义其引用参数e来接收异常对象
26     {
27         cout << e.what( ) << endl;  // 向用户显示所捕捉到异常对象e的详细信息
28     }
29     return 0;
30 }
```

10.5 数据集合及其处理算法

- **数据集合**
  - **数据项**（Data Item）。数据项是数据集合的最小单位。例如，表10-1中第2行的姓名“张三”、成绩“92”都分别是一个独立的数据项
  - **数据元素**（Data Element）。数据元素是由多个具有内在关联关系的数据项所组成。例如，表10-1中第2行“张三，92”、第3行“李四，86”都分别构成一个数据元素
  - **数据集合**（Data Set）。数据集合由多个并列的数据元素所组成。例如，表10-1就是一个由多个数据元素（每个数据元素一行）组成的数据集合

表10-1

姓名	成绩
张三	92
李四	86
王五	95
……	……

10.5 数据集合及其处理算法

- 对数据集合的处理
  - 对数据集合的处理通常包括**增加**、**删除**、**修改**或**查找**数据元素，统称为**增删改查**（Create、Read、Update、Delete，简称CRUD）
  - 为了提高查找速度，可将数据集合中的数据元素按照某种规则事先进行**排序**

- 处理数据集合的计算机程序
  - 如何组织和存储数据集合，即**数据结构**
  - 如何基于数据结构进行数据处理，即**算法**

10.5 数据集合及其处理算法

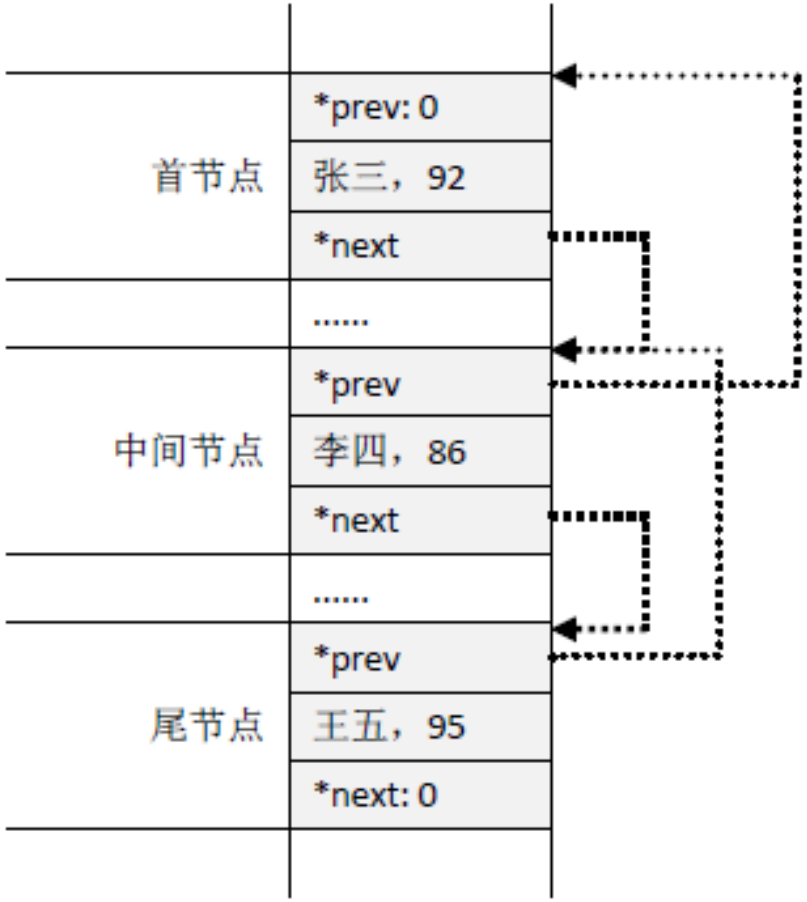
- 数据集合的存储和处理

例10-14 一个存储和处理学生成绩单的C++程序（使用**结构体数组**）

```
#include <iostream>
#include <string.h>
using namespace std;

typedef  struct  // 定义保存学生成绩的结构体Student
{
    char  name[9] ;
    float  score ;
} Student ;

int main()
{
    Student  s[ 3 ] ;  // 定义保存学生成绩单的结构体数组s
    strcpy( s[0].name, “张三” );  s[0].score = 92;  // 通过下标访问结构体数组元素
    strcpy( s[1].name, “李四” );  s[1].score = 86;
    strcpy( s[2].name, “王五” );  s[2].score = 95;

    Student  *p = s ;  // 通过指针变量p访问结构体数组元素
    for  (int n=0; n < 3; n++)  // 显示学生成绩单
    {
       cout << p->name << “, “ << p->score << endl;
       p++ ;  // 将指针变量p移到下一个数组元素
    }
    return 0;
}
```

10.5 数据集合及其处理算法

- 链表
  - 节点（Node）
  - 第一个节点被称为**首节点**
  - 最后一个节点被称为**尾节点**
  - 同时具有前向和后向指针的链表称为**双向链表**
  - 只有一个方向指针的链表称为**单向链表**



例10-15 一个存储和处理学生成绩单的C++程序（使用**双向链表**）

```
#include <iostream>
#include <string.h>
using namespace std;

typedef  struct tagStudent  // 定义保存学生成绩的结构体Student，具有双向指针
{
     char  name[9] ;
     float  score ;
     struct tagStudent  *prev, *next;  // 定义前向指针prev和后向指针next
} Student ;

int main()
{
     Student  *begin ;  // 定义指向双向链表首节点的指针变量begin
     begin = new  Student;  // 创建首节点
     strcpy( begin->name, "张三" );   begin->score = 92;
     begin->prev = 0;   begin->next = 0;  // 首节点目前还没有前后节点，将前后指针置空
     Student  *curNode = begin;  // 定义指向当前节点的指针变量curNode

     Student  *p = new  Student;  // 创建第2个节点
     strcpy( p->name, "李四" );   p->score = 86;
     curNode->next = p;  // 将前一个节点的后向指针next指向新节点
     p->prev = curNode;  // 将新节点的前向指针prev指向前一个节点
     p->next = 0;  // 将新节点的后向指针next置空
     curNode = p;  // 新节点成为当前节点

     p = new  Student;  // 创建第3个节点
     strcpy( p->name, "王五" );   p->score = 95;
     curNode->next = p;  // 将前一个节点的后向指针next指向新节点
     p->prev = curNode;  // 将新节点的前向指针prev指向前一个节点
     p->next = 0;  // 将新节点的后向指针next置空
     curNode = p;  // 新节点成为当前节点

     curNode = begin ;  // 通过指针变量curNode访问链表中的各个节点，显示出学生成绩单
     while (curNode != 0 )  // 循环条件：当前节点不为空
     {
        cout << curNode ->name << ", " << curNode ->score << endl;
        curNode = curNode->next ;  // 将指针变量curNode移到下一个节点
     }
     return 0;
}
```

10.5 数据集合及其处理算法

- 数组和链表是存储数据集合的两种基本存储结构
  - **链表便于插入和删除操作**。数组中的数据元素是连续存储的，插入或删除数据元素需要移动该元素后面的所有元素。而链表中的数据元素是各自独立存储的，插入或删除数据元素只需要修改相关的前向/后向指针，不需要移动数据。换句话说，如果数据集合在生成之后比较稳定，不需要经常插入或删除数据元素，则选用数组存储比较适合，否则选用链表更加适合
  - **链表只能顺序访问**。数组可以通过下标随机访问任意指定位置的数据元素。而链表只能从头到尾，或从尾到头按顺序依次访问数据元素
  - **链表占用内存空间更多**。数组只存储数据元素，而链表除数据外还需要存储前向或后向指针

10.5 数据集合及其处理算法

- C++标准库中数据集合的存储和处理
  - 为了存储数据集合，C++标准库提供了7个数据结构类，它们被统称为**容器**（Container）类
  - **迭代器**（Iterator）为访问不同容器中的数据元素提供了一种统一的访问方法。每个容器类都定义了自己的迭代器类型iterator，它是一种类似于指针类型的类类型
  - 为了处理存储在容器中的数据集合，C++标准库提供近70个左右的函数，实现了丰富的数据处理功能，它们被统称为**算法**（Algorithm）。算法函数通过迭代器（请记住：迭代器类似于指针）访问容器中的数据元素

10.5 数据集合及其处理算法

- 指针类型

  int x[10]; // 定义数组x
  int *p; // 定义同类型的指针变量p
  p = &x[0]; // 取数组元素的地址

  – 指针访问： *p、p++、p--
  (*p).、p->
  – 下标访问：p[0]、p[1]、p[2]、……

- 迭代器类型

  vector <int> iv; // 定义数据集合iv
  vector <int> :: iterator p; // 定义迭代器变量p
  p = iv.beign( ); // 取向量元素的地址

  – 指针访问： *p、p++、p—
  (*p).、p->
  – 下标访问：p[0]、p[1]、p[2]、……

10.5 数据集合及其处理算法

- 迭代器类型
  - 输入迭代器（Input Iterator）
    ++（前置/后置）、*（读内容）、=（赋值）、==（等于）、!=（不等于）
  - 输出迭代器（Output Iterator）
    ++（前置/后置）、*（写内容）、=（赋值）
  - 正向迭代器（Forward Iterator）
    正向迭代器既有输入迭代器的功能，又有输出迭代器的功能。通过正向迭代器访问数据元素时既可以读取数据，也可以修改。同样，正向迭代器也只能按从头到尾的顺序单向访问容器中的数据元素
  - 双向迭代器（Bidirectional Iterator）
    双向迭代器在正向迭代器的基础上又重载了自减运算符“--”，因此双向迭代器既能按从头到尾的顺序，也能按从尾到头的顺序，双向访问容器中的数据元素
  - 随机访问迭代器（Random Access Iterator）
    在双向迭代器的基础上，又重载了如下的运算符：[ ]、+、-、+=、-=、>、>=、<、<=

10.5 数据集合及其处理算法

- 算法
  - 对数据集合的处理通常包括增加、删除、修改或查找数据元素。为了提高查找速度，可将数据集合中的数据元素按照某种规则事先进行排序
  - C++标准库将算法分成3大类，分别为非可变序列算法（即查找算法）、可变序列算法（即增删改算法）和排序相关算法

10.5 数据集合及其处理算法

- 算法函数模板

  算法函数通过形参接收迭代器，再通过迭代器访问容器中的数据元素。为了接收不同类型的迭代器，C++标准库将算法函数都定义成了函数模板

  – 排序函数sort

```
template  <typename RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last );

template  <typename RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
```

10.5 数据集合及其处理算法

- C++标准库中的容器类
  - 为了存储数据集合，C++标准库提供了7个数据结构类，它们被统称为**容器**（Container）类
  - 比较常用的容器类有：
    - 向量类vector
    - 列表类list
    - 集合类set
    - 映射类map
  - 为了存储不同类型的数据，C++标准库将容器类都定义成了类模板

10.5 数据集合及其处理算法

- 向量类vector

  向量类vector是一种容器类，可以存储一维有序数据序列。向量类具有如下特点：
  - 向量的内部存储结构是数组。
  - 向量中的所有元素都属于同一个数据类型。
  - 向量中可以存储的元素数量不受限制。当元素个数超出向量的存储容量时，向量会自动扩展其内存空间。扩展时会额外多分配若干个元素的存储单元，这样可以避免频繁地扩展内存。由此带来的后果是，向量的后面几个元素可能是空元素，即未使用的元素。
  - 向量的迭代器类型是随机访问迭代器，可使用下标访问元素。
  - 向量比数组功能更强，可取代数组。

  - 向量类vector是一个类模板，可使用该类模板定义出不同数据类型的向量对象
  - 使用向量类需包含其相应的类声明头文件<vector>

10.5 数据集合及其处理算法

- 向量类vector

使用语法	说明
vector <class T> v;	定义1个空向量容器对象v，元素类型为T
vector <class T> v(n);	定义1个初始元素个数为n的向量容器对象v，元素类型为T
vector <class T>::iterator it;	定义1个向量迭代器对象it，该迭代器是随机访问迭代器
v.begin()	返回指向第一个元素位置的迭代器
v.end()	返回指向最后一个元素后面那个空元素位置的迭代器
v.size()	返回向量中元素的个数
v.push_back(e)	在向量末尾添加1个元素e，e的类型应为T
v.insert(it, e)	在it所指向元素之前插入1个新元素e，e的类型应为T
v.erase(it)	删除it所指向的元素，返回指向其下一个元素位置的迭代器
v.clear()	删除向量中的所有元素

10.5 数据集合及其处理算法

例10-16 一个向量类vector的C++演示程序（**int型**向量）

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

int main()
{
    vector <int>  iv;  // 定义1个int型向量iv
    for (int n=0; n < 5; n++)  // 添加5个向量元素
        iv.push_back( n*n );  // 第n个元素的值等于n²

    vector <int> :: iterator  vit;  // 定义1个向量类的容器迭代器vit
    for ( vit = iv.begin( ); vit < iv.end( ); vit++ )  // 通过容器迭代器vit遍历向量
      cout << *vit << ", ";  // 显示向量内容
    cout << endl << endl;  // 向量显示结果：0, 1, 4, 9, 16,
    cout << "size= " << iv.size( ) << endl;  // 显示向量中已存放数据的元素个数：size= 5

    iv.pop_back( ) ;  // 删除向量中的最后一个元素，再显示其中的元素个数
    cout << "size= " << iv.size( ) << endl;  // 显示向量中已存放数据的元素个数：size= 4
    return 0;
}
```

10.5 数据集合及其处理算法

例10-17 一个对int型向量进行排序的C++演示程序

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

int main()
{
    vector <int>  iv;  // 定义1个int型向量iv
    // 添加4个向量元素，数值依次为：3, 7, 9, 5
    iv.push_back( 3 );   iv.push_back( 7 );   iv.push_back( 9 );   iv.push_back( 5 );

    sort( iv.begin( ), iv.end( ) );  // 使用算法函数sort对向量iv进行排序，排序区间为整个向量

    // 显示排序后的结果
    vector <int> :: iterator  vit;  // 定义1个向量类的容器迭代器vit
    for ( vit = iv.begin( ); vit < iv.end( ); vit++ )  // 通过容器迭代器vit遍历向量
      cout << *vit << ", ";  // 显示向量内容
    cout << endl;  // 向量排序后的显示结果：3, 5, 7, 9,
    return 0;
}
```

例10-18 一个学生成绩单向量的C++演示程序（**结构体类型**向量）

```
1  #include <iostream>
2  #include  <string.h>
3  #include <vector>
4  #include <algorithm>
5  using namespace std;
6
7  typedef  struct  // 定义保存学生成绩的结构体Student
8  {
9      char  name[9] ;
10     float  score ;
11 } Student ;
12
13 bool compStudent( Student x, Student y)  // 比较函数：按姓名比较大小
14 {
15    if (strcmp(x.name, y.name) < 0)  return true;  // 按字符串规则来比较姓名
16    else  return false;
17 }
18
19 int main()
20 {
21     vector <Student>  sv;  // 定义1个结构体Student类型的向量sv
22     // 添加3个向量元素
23     Student  s;
24     strcpy( s.name,  "张三");    s.score = 92;    sv.push_back( s );
25     strcpy( s.name,  "李四");    s.score = 86;    sv.push_back( s );
26     strcpy( s.name,  "王五");    s.score = 95;    sv.push_back( s );
27
28     sort( sv.begin( ), sv.end( ), compStudent );  // 对向量sv进行排序，比较函数为compStudent
29
30     // 显示排序后的结果
31     vector <Student> :: iterator  svit;  // 定义1个向量类的容器迭代器svit
32     for ( svit = sv.begin( ); svit < sv.end( ); svit++ )  // 通过容器迭代器svit遍历向量
33         cout << svit->name << ", " << svit->score << endl;  // 显示向量内容
34     return 0;
35 }
```

例10-19 一个学生成绩单向量的C++演示程序（**类类型**向量）

```
#include <iostream>
#include  <string.h>
#include <vector>
#include <algorithm>
using namespace std;

class  Student  // 定义保存学生成绩的类Student
{
public:
    char  name[9] ;
    float  score ;
    bool operator<(Student x)  // 重载运算符<：按成绩比较大小。其用途是指定排序规则
    {
        if (score < x.score)  return true;  // 小于返回true则为升序，反之则为降序
        else  return false;
    }
} ;

int main()
{
    vector <Student>  sv;  // 定义1个类Student类型的向量sv
    // 添加3个向量元素
    Student  s;
    strcpy( s.name,  "张三");   s.score = 92;   sv.push_back( s );
    strcpy( s.name,  "李四");   s.score = 86;   sv.push_back( s );
    strcpy( s.name,  "王五");   s.score = 95;   sv.push_back( s );

    sort( sv.begin( ), sv.end( ) );  // 对向量sv进行排序。排序时将使用重载的运算符“<”

    // 显示排序后的结果
    vector <Student> :: iterator  svit;  // 定义1个向量类的容器迭代器svit
    for ( svit = sv.begin( ); svit < sv.end( ); svit++ )  // 通过容器迭代器svit遍历向量
        cout << svit->name << ", " << svit->score << endl;  // 显示向量内容
    return 0;
}
```

10.5 数据集合及其处理算法

- 列表类list

  列表类list是一种容器类，可以存储一维有序数据序列。列表中的所有元素都属于同一个数据类型。与向量类vector相比，列表类list具有如下特点：
  - 列表的内部存储结构是链表，而向量的内部存储结构是数组。
  - 列表中每个元素的内存空间是独立分配的，而向量是连续存储的。
  - 列表适合于存储需频繁添加删除的数据集合，而向量适合于存储元素总数相对固定的数据集合，即向量不适合频繁地添加删除元素。
  - 列表的迭代器类型是双向迭代器，而向量的迭代器类型是随机访问迭代器（可使用下标访问元素）。

  - 列表类list是一个类模板，可使用该类模板定义出不同数据类型的列表对象
  - 使用列表类需包含其相应的类声明头文件<list>

10.5 数据集合及其处理算法

- 列表类list

例10-20 一个列表类list的C++演示程序（**int型**列表）

```
1  #include <iostream>
2  #include <list>
3  #include <algorithm>
4  using namespace std;
5
6  int main()
7  {
8      list <int>  il;  // 定义1个int型列表il
9      // 添加4个列表元素，数值依次为：3, 7, 9, 5
10     il.push_back( 3 );   il.push_back( 7 );   il.push_back( 9 );   il.push_back( 5 );
11
12     il.sort( );  // 对列表il进行排序。列表类list自带排序函数成员sort
13
14     // 显示排序后的结果
15     list <int> :: iterator  lit;  // 定义1个列表类的容器迭代器lit（属于双向迭代器）
16     for ( lit = il.begin( ); lit != il.end( ); lit++ )  // 通过容器迭代器lit遍历列表
17      cout << *lit << ", ";  // 显示列表内容
18    cout << endl;  // 列表排序后的显示结果：3, 5, 7, 9,
19    return 0;
20 }
```

10.5 数据集合及其处理算法

- 集合类set

  集合类set是一种容器类，可以存储一维无序数据序列。
  集合类set具有如下特点：
  - 集合中的所有元素都属于同一个数据类型。
  - 向集合中插入的元素总是会被按某种键值自动排序。
  - 可通过比较函数来指定排序键值。
  - 集合中每个元素的键值都是唯一的，不能相同。键值重复的元素会被自动丢弃。
  - 集合的迭代器类型是双向迭代器。

  - 集合类set是一个类模板，可使用该类模板定义出不同数据类型的集合对象。
  - 使用集合类需包含其相应的类声明头文件<set>

10.5 数据集合及其处理算法

- 集合类set

例10-21 一个集合类set的C++演示程序（**int型**集合）

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

int main()
{
    set <int>  is;  // 定义1个int型集合is
    // 插入4个集合元素，数值依次为：3, 7, 9, 5
    is.insert( 3 );   is.insert( 7 );   is.insert( 9 );   is.insert( 5 );

    is.insert( 3 );  // 重复元素：重复的集合元素将自动被丢弃

    // 显示集合中的元素
    set <int> :: iterator  sit;  // 定义1个集合类的容器迭代器sit（属于双向迭代器）
    for ( sit = is.begin( ); sit != is.end( ); sit++ )  // 通过容器迭代器sit遍历集合
     cout << *sit << ", ";  // 显示集合内容
    cout << endl;  // 集合会自动排序，显示结果：3, 5, 7, 9,
    return 0;
}
```

10.5 数据集合及其处理算法

- 映射类map

  映射类map是一种适合存储被称作“**键-值**”类型数据的容器类。映射类map具有如下特点：

  – “**键-值**”类型数据包括两个数据，一个称为“键”，一个称为“值”。例如：“张三, 92”，“李四, 86”，……，就是一种“姓名-分数”类型的键值对，其中姓名是“键”，分数是“值”

  – 向映射中插入的元素总是被按“键”自动排序，这样可以便于今后的快速查找

  – 映射中每个元素的“键”都是唯一的，不能相同。插入“键”重复的元素时，新元素将覆盖老元素

  – 映射的迭代器类型是双向迭代器

  – 映射类map是一个类模板，可使用该类模板定义出不同数据类型的映射对象

  – 使用映射类需包含其相应的类声明头文件<map>

10.5 数据集合及其处理算法

- 映射类map

例10-22 一个映射类map的C++演示程序

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8      map <string, int>  sim;  // 定义1个string-int型映射sim
9      // 插入3个映射元素，数值依次为：张三-92, 李四-86, 王五-95
10     sim["张三"] = 92;   sim["李四"] = 86;   sim["王五"] = 95;
11
12     sim["张三"] = 100;  // 重复元素：“键”重复的映射元素将自动覆盖之前的老元素
13
14    map <string, int> :: iterator  mit;  // 定义1个映射类的容器迭代器mit（属于双向迭代器）
15     for (mit = sim.begin( ); mit != sim.end( ); mit++)  // 通过容器迭代器mit遍历映射
16       cout << mit->first << ", " << mit->second << endl;  // 显示键（first）和值（second）
17     return 0;
18 }
```

10.6 结语

- 掌握了程序设计的基本原理和方法，并能够比较熟练地运用C++语言的语法知识来阅读或编写简单的计算机程序
- 通过重用别人的代码，例如标准C库中的系统函数或C++标准库中的系统类库，程序员可以站在更高的起点上开发程序，而不需要从零开始
- 使用第三方开发的函数或类库

10.6 结语

- 微软基础类库MFC（Microsoft Foundation Classes）
  - 基于Windows的图形界面程序。MFC类库提供了多种窗口及组件类，常用的有窗口类CWnd、对话框类CDialog、按钮类CButton、编辑框类CEdit、菜单类CMenu等
  - 网络应用程序。MFC类库提供了多种网络通讯类，常用的有套接字类CSocket、HTTP连接类CHttpConnection、FTP连接类CFtpConnection等
  - 数据库应用程序。MFC类库提供了统一的数据库操作类，常用的有数据库类CDaoDatabase、记录集类CRecordSet等

本章要点

- 了解如何使用模板技术来提高函数和类代码的可重用性
- 重点学习C++语言的异常处理机制
- 初步掌握如何使用C++标准库中的向量类、列表类、集合类和映射类来存储和处理数据集合