第九章　拾遗补缺

本章导读

C++一个有机的整体，各部分知识点之间有紧密的联系。为了循序渐进地组织内容以方便学习，有些知识点在不影响大局的情况下，采取略讲策略；有些知识点充满细节，为避免纠缠细节而失去宏观，可能采取退避。更深入地讲解某些知识点，可能涉及到尚未学习的知识，也只能点到为止。为弥补上述遗憾，特增设“补充”一章。

学习目标：

1. 深入了解数据类型及其相关；
2. 指针；
3. 关键字补阙；
4. 预处理；

本章目录

第一节　数据类型及其相关

程序设计语言大都涉及到数据类型，如在int a中，int就是数据类型。在C++中，可以将数据类型分为基本数据类型、构造类型、指针类型、引用类型以及空类型(void)，如图9-1所示。

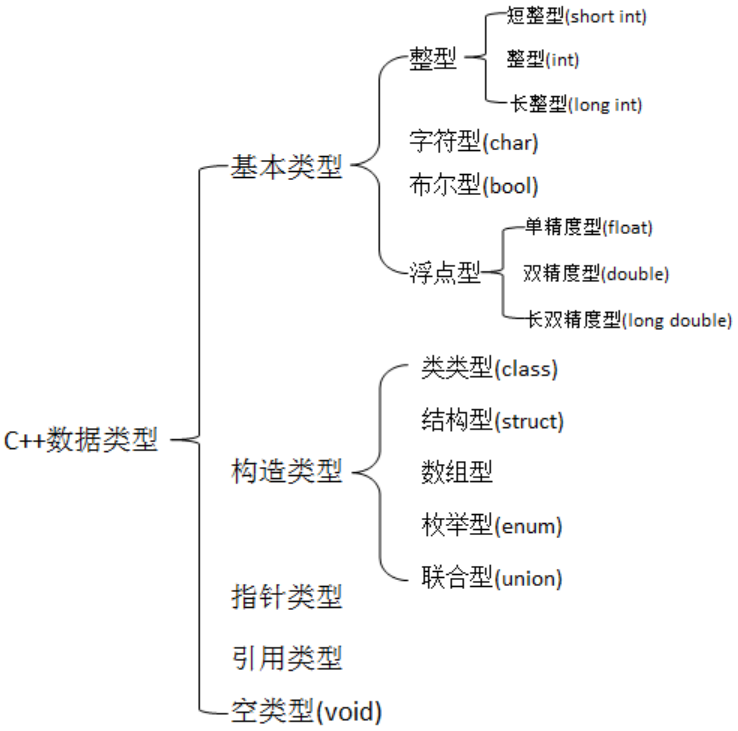


图9-1　C++中的数据类型

对于整型和字符型，还可以加上修饰符signed和unsigned关键字，分别表示有符号类型和无符号类型。当数据类型为各种类型的int时，默认为singed即有符号整数，能表示负数、零和正数；而unsigned用于修饰int时，则只能表示零和正数。对于系列char类型而言，在C++中有三种类型即：char、singed char和unsigned char。当使用char时，一般默认为singed char，在有些系统中默认为unsigned char，有编译器确定。当为signed char时，表示范围为-128~127，unsigned char则表示0-255。

1、typeid的应用

typeid是C++关键字之一，用于在编译或运行时刻确定表达式或变量的类型，其语法格式如下，例程9-1是其应用示例，用typeid(类型或表达式).name()可获得typeid()返回值的显式字符串表达。如果表达式的类型是类类型且至少包含有一个虚函数，则typeid操作符返回表达式的动态类型，在运行时确定；否则，typeid操作符返回表达式的静态类型，在编译时确定。另外，typeid()是运算符，不是函数。

typeid(类型或表达式)

例程9-1

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

struct baseClassA{
    virtual void print()=0;
};
struct derivedClassA1:baseClassA{
    void print(){cout<<"I am in derivedClassA1!!!"<<endl;}
};
struct derivedClassA2:baseClassA{
    void print(){cout<<"I am in derivedClassA2!!!"<<endl;}
};
void Print(baseClassA &objIN){
    cout<<typeid(objIN).name()<<endl;
```

```
第16行      objIN.print();
第17行  }
第18行
第19行  int main(){
第20行      int a=100;
第21行      cout<<typeid(int).name()<<endl;//输出：int
第22行      cout<<typeid(a).name()<<endl;//输出：int
第23行      cout<<typeid(100).name()<<endl;//输出：int
第24行      cout<<typeid(100l).name()<<endl;//输出：long
第25行      cout<<typeid(unsigned int).name()<<endl;//输出：unsigned int
第26行      cout<<typeid(unsigned long).name()<<endl;//输出：unsigned long
第27行      cout<<typeid(unsigned long int).name()<<endl;//输出：unsigned long
第28行      derivedClassA1 myClass1;
第29行      cout<<typeid(baseClassA).name()<<endl;//输出：struct baseClassA
第30行      cout<<typeid(myClass1).name()<<endl;//输出：struct derivedClassA1
第31行      Print(myClass1);//先输出：struct derivedClassA1，后输出：I am in derivedClassA1!!!
第32行      derivedClassA2 myClass2;
第33行      Print(myClass2);//先输出：struct derivedClassA2，后输出：I am in derivedClassA2!!!
第34行
第35行      cout<<typeid(cout).name()<<endl;//class std::basic_ostream<char,struct std::char_traits<char> >
第36行
第37行      return 0;
第38行  }
```

通过例程9-1可以看出，typeid()能得到表达式和类型的类型，对于带有虚函数的类，能根据执行场景，正确地得到类类型。另外，通过其name()成员函数能得到其类型的字符串表达。

2、数据类型转换

C++中常见数据类型转换，有隐式转换，也有系列cast函数的显式转换、更有通过定义类的成员函数完成转换。例程9-2是一个数据类型的隐式转换。

例程9-2

```
第1行   #include<iostream>
第2行   #include<typeinfo>
第3行   using namespace std;
第4行
第5行   int main(){
第6行       unsigned int i=100;
第7行       cout<<-1*i<<endl;//输出：4294967196
第8行       cout<<typeid(-1*i).name()<<endl;//输出：unsigned int
第9行
第10行      int a=100;
第11行      cout<<-1*a<<endl;//输出：-100
第12行      cout<<typeid(-1*a).name()<<endl;//输出：int
第13行
第14行      return 0;
第15行  }
```

注意第7行，i=100，-1*i理当为-100，但由于i为无符号整数，因此将-1*i也变为无符号整数，-100的原码为：1000 0000 0000 0000 0000 0000 0110 0100，其反码为：11111 1111 1111 1111 1111 1111 1001 1011，其补码为：11111 1111 1111 1111 1111 1111 1001 1100。负数在内存中以补码形式存在，当将该负数视为无符号整数时，则原来最高位用于表示正负号的符号位用于表达正数或零，因此，此时输出4294967196。

对于二元运算符，C++允许两个操作数是不同类型，当操作数类型不同时，C++将自动对数据类型进行转换，这种转换称为隐式转换，当被转换的数据类型为C++基本类型时，其规则如下：

- 当其中一个操作数是long double时，则将另外一个转换为long double；
- 当其中一个操作数是double时，则将另外一个转换为double；
- 当其中一个操作数是float时，则将另外一个转换为float；
- 当其中一个操作数是unsigned long时，则将另外一个转换为unsigned long；
- 当其中一个操作数是long时，则将另外一个转换为long；
- 当其中一个操作数是unsigned int时，则将另外一个转换为unsigned int；
- 如不满足上述规则，则都转换为int类型。

对于例程9-2第7行，-1为int，i为unsigned int，因此被转换为unsigned int。

除了隐式类型转换，C++还支持显示类型转换，如例程9-3所示，其中第7、8行是C模式类型转换，第8行为C++模式的类型转换。在C++中，当需要类似转换时，建议采用static_cast。

例程9-3

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int main(){
第5行      unsigned int i=100;
第6行      cout<<-1*(int)i<<endl;//输出：-100
第7行      cout<<-1*int(i)<<endl;//输出：-100
第8行      cout<<-1*static_cast<int>(i)<<endl;//输出：-100
第9行
第10行     return 0;
第11行 }
```

C++数据类型转换除了static_cast外，还有dynamic_cast和reinterpret_cast，其中的dynamic_cast的格式如下：

```
dynamic_cast<typeid>(Expression)
```

dynamic_cast把Expression转换成typeid类型的对象。typeid必须是类的指针、类的引用或者void*。如果typeid是类指针，那么Expression也必须是一个指针，如果typeid是一个引用，那么Expression也必须是一个引用。dynamic_cast主要用于类层次间的向上转换或向下转换。当进行向上转换时，dynamic_cast和static_cast的效果相同；当进行向下转换时，dynamic_cast具有类型检查，比static_cast更安全。例程9-4是dynamic_cast与static_cast的应用示例。

例程9-4

```
第1行  #include<iostream>
第2行  #include<typeinfo>
第3行  using namespace std;
第4行  struct classRoot{
第5行      classRoot(){cout<<"I am in classRoot.\n";}
第6行      virtual void funA(){cout<<"I am in funA()@classRoot.\n";};
第7行  };
第8行  struct classBranchA:public classRoot{
第9行      int num;
```

```
第10行      classBranchA(){this->num=100;cout<<"I am in classBranchA.\n";}
第11行      void funA(){cout<<"this->num="<<this->num<<". I am in funA()@classBranchA.\n";}
第12行  };
第13行  struct classBranchB:public classRoot{
第14行      int num;
第15行      classBranchB(){this->num=99;cout<<"I am in classBranchB.\n";}
第16行      void funA(){cout<<"this->num="<<this->num<<". I am in funA()@classBranchB.\n";}
第17行  };
第18行  void funTest(classRoot &refObj){
第19行      cout<<"\n\n";
第20行
第21行      cout<<typeid(refObj).name()<<endl;
第22行      refObj.funA();
第23行
第24行      classBranchB &refB=static_cast<classBranchB&>(refObj);
第25行      classBranchA *refA=dynamic_cast<classBranchA*>(&refObj);
第26行
第27行      refB.funA();
第28行      if(refA!=NULL)refA->funA();
第29行  }
第30行  int main(){
第31行      classBranchA a;
第32行      classBranchB b;

第33行      funTest(a);funTest(b);
第34行
第35行      classBranchA *pA=static_cast<classBranchA *>(&a);//如将a修改b将导致编译错误或者语法错误
第36行      classBranchB *pB=dynamic_cast<classBranchB *>(&a);//此处a或b均可，当a时，pB为NULL
第37行
第38行      return 0;
第39行  }
```

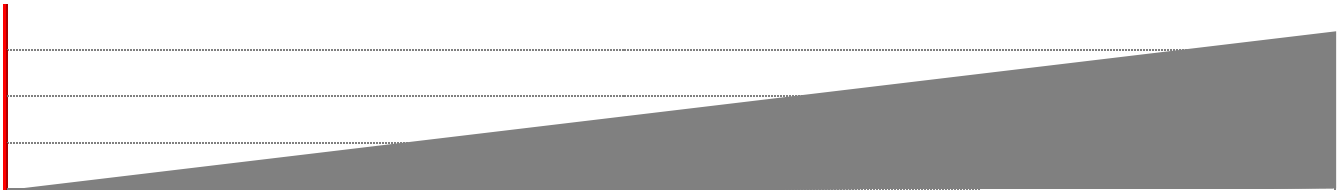
和static_cast相比，dynamic_cast要求基类必须有虚函数，而static_cast则没有该限制。dynamic_cast有类型检查。当将第25行修改为classBranchA &refA=dynamic_cast<classBranchA&>(refObj)，则当refObj为classBranchB时，将触发std::bad_cast异常；当不进行修改时，则当refObj为classBranchB时，refA的值将为NULL，删除第28行前的if(refA!=NULL)将报错。和static_cast相比，dynamic_cast更加安全。

另外，dynamic_cast支持交叉转换，如例程9-4第36行所示。当将classBranchA类型的对象a转换为classBranchB时，其结果为null，没有语法错误；当将&a改变为&b时，则正确转换。对第35行的static而言，不支持交叉转换，即将&a改变为&b，则将出现编译错误或者语法错误。

reinterpret_cast也用于类型转换，其格式如下。reinterpret_cast可以将任意指针类型转换成任意指针类型，即便两者没有任何关系，转换结果是二进制的简单拷贝。reinterpret_cast可以将指针转换为整数，或将整数转换为指针。整数值所代表的指针与平台相关，仅仅保证指针转换为整数时有足够容量存储整数，并保证将整数转回为合法的指针。reinterpret_cast转换是一种低层次操作，其转换为该类型的二进制表达，与平台相关，跨平台性不强。例程9-5是reinterpret_cast的使用示例。

reinterpret_cast<typeid>(Expression)

例程9-5



```
第19行      myFile.close();
第20行
第21行      cout<<readingData.a<<" "<<readingData.b<<endl;
第22行
第23行      return 0;
第24行  }
```

const_cast用于修改类型的const或volatile属性，其格式如下，其中除const修饰符外，typeid与Expression类型一致。const_cast有两种功能，一是将常量指针转化为非常量的指针，并且仍然指向原来的对象；二是将常量引用转换为非常量引用，并且仍然指向原来的对象。例程9-7是其应用示例。从示例可以看出，const_cast只能对类、结构和指针做从const到非const的转换，对基本数据类型，虽然能完成转换，但通过指针不能修改其值(第20-23行)。

```
const_cast<typeid>(Expression)
```

例程9-7

```
第1行   #include<iostream>
第2行   using namespace std;
第3行   struct UpDown{
第4行       UpDown(){this->Up=1;this->Down=1;}
第5行       int Up,Down;
第6行   };
第7行   int main(){
第8行       const UpDown UD;
第9行
第10行      UpDown *udA=const_cast<UpDown *>(&UD);
第11行      udA->Up=1;udA->Down=2;
第12行      cout<<"udA="<<"("<<udA->Up<<"/"<<udA->Down<<")"<<endl;//输出:udA=(1/2)
第13行      cout<<"UD="<<"("<<UD.Up<<"/"<<UD.Down<<")"<<endl;//输出:UD=(1/2)
第14行
第15行      UpDown &udB=const_cast<UpDown &>(UD);
第16行      udB.Up=2;udB.Down=3;
第17行      cout<<"udB="<<"("<<udB.Up<<"/"<<udB.Down<<")"<<endl;//输出:udB=(2/3)
第18行      cout<<"UD="<<"("<<UD.Up<<"/"<<UD.Down<<")"<<endl;//输出:UD=(2/3)
第19行
第20行      const int a=100;
第21行      int *pA=const_cast<int *>(&a);
第22行      *pA=200;
第23行      cout<<"a="<<a<<" *pA="<<*pA<<endl;//输出a=100 *pA=200
第24行
第25行      return 0;
第26行  }
```

3、函数指针类型

函数指针是指向函数的指针，如例程9-8第32行、第33行所示，分别定义pA和pB为函数指针，pA为二元函数，pB为一元函数。函数指针定义如int (*pB)(int)与定义一个变量为整数如int a、浮点数如float b差别较大，这是应用函数指针的可能障碍之一。在定义函数指针时，需要指明函数指针所指向函数的返回值类型及其参数。

```
函数返回值类型 (*函数指针变量名)(形参列表);
```

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

int plus(int a,int b){
    return a+b;
}
int minus(int a,int b){
    return a-b;
}
int negate(int a){
    return -a;

}
void print(int a){
    cout<<a<<endl;
}
template<typename T>
void tellType(T t){
    cout<<typeid(t).name()<<endl;
}
//函数重载
void funPointer(int (*f)(int,int)){
    cout<<typeid(f).name()<<endl;
}
void funPointer(int (*f)(int)){
    cout<<typeid(f).name()<<endl;
}
void funPointer(void (*f)(int)){
    cout<<typeid(f).name()<<endl;
}
int main(){
    int (*pA)(int,int)=NULL;
    int (*pB)(int)=NULL;
    cout<<typeid(pA).name()<<endl;//输出：int (__cdecl*)(int,int)
    cout<<typeid(pB).name()<<endl;//输出：int (__cdecl*)(int)

    int a=100;
    tellType(pA);//输出：int(__cdecl*)(int,int)
    tellType(pB);//输出：int(__cdecl*)(int)
    tellType(10);//输出：int
    tellType(a);//输出：int

```

```
第43行        funPointer(pA);//funPointer()函数重载
第44行        funPointer(pB);//funPointer()函数重载
第45行        funPointer(print);//funPointer()函数重载
第46行
第47行        pA=plus;//f指向plus函数
第48行        cout<<pA(20,10)<<endl;//输出：30
第49行        pA=minus;
第50行        cout<<pA(20,10)<<endl;//输出：10
第51行
第52行        //pA=negate(10);//错误!
第53行        //pB=minus(20,10);//错误!
第54行
第55行        //定义pFun为指向两个参数分别为int返回值也为int的二元函数类型pFun
第56行        typedef int (*pFun)(int,int);
第57行        pFun pPP=minus;
第58行        cout<<pPP(20,10)<<endl;
第59行        pPP=plus;
第60行        cout<<pPP(20,10)<<endl;
第61行
第62行
第63行        return 0;
第64行    }
```

第17-20行定义了tellType()模板函数，其功能是根据输出参数输出类型信息。第37-41行是调用tellType()函数，由此可可见pA、pB类型与int等都属于类型范畴。

第21-30行分别重载了funPointer()函数，分别适用于二元函数和一元函数。注意：void funPointer(int (*f)(int))和void funPointer(void (*f)(int))虽然都是一元函数，但是返回值不同，是不同的函数形态，是两种函数，或者说是两种不同的数据类型。第43-45行是调用funPointer()函数，参数分别是pA、pB以及print()。

指针可以指向符合约定的多个内存空间，函数指针也不例外，第47行pA指向plus()，第48行pA指向minus，不同的指向后分别执行pA(20,10)相当于分别执行plus(20,10)和minus(20,10)。注意：pA是指向二元函数，且返回值为int类型，如果指向其他函数，将报错如：pA=negate(20)指向了一元函数；pB指向一元函数，如用其指向二元函数minus()也将报错。

如果习惯int a类似的类型说明方式，可以用typedef定义类型别名，如第56行所示，然后如第57行定义函数指针pPP，其形似int a。

函数指针还可以指向类的成员函数，对于普通成员函数(非静态成员函数，非友元函数)，定义函数指针时必须指明函数指针的隶属关系，如例程9-9所示。第17行代码int (XYZ::*pFunA)(int,int)的功能是定义pFunA为XYZ类的函数指针，指向两个参数分别是int类型且返回类型int，与例程9-8中的函数指针相比，多了类的名称。第18行代码pFunA=&XYZ::plus将pFunA指向plus(注意：XYZ前的&不能省略)。当调用函数，需要指明对象，如第19行代码int result=(xyz.*pFunA)(a,b)中xyz是XYZ类的对象。

例程9-9

```
第1行   #include<iostream>
第2行   #include<typeinfo>
第3行   using namespace std;
第4行   struct XYZ{
第5行       int plus(int a,int b){return a+b;}
第6行       int minus(int a,int b){return a-b;}
第7行       static void print(int a){cout<<a<<endl;}
第8行       friend int sum(int a,int b);
```

```
第9行  };
第10行 int sum(int a,int b){
第11行     return a+b;
第12行 }
第13行 int main(){
第14行     int a=20,b=10;
第15行     XYZ xyz;
第16行
第17行     int (XYZ::*pFunA)(int,int);
第18行     pFunA=&XYZ::plus;
第19行     int result=(xyz.*pFunA)(a,b);
第20行     cout<<result<<endl;
第21行     pFunA=&XYZ::minus;
第22行     result=(xyz.*pFunA)(a,b);
第23行     cout<<result<<endl;
第24行
第25行     //静态函数成员
第26行     void (*pFunB)(int);
第27行     pFunB=&XYZ::print;

第28行     pFunB(10);
第29行
第30行     //友元函数
第31行     int (*pFunC)(int,int);
第32行     pFunC=sum;
第33行     cout<<pFunC(a,b)<<endl;
第34行
第35行     return 0;
第36行 }
```

对于指向静态函数成员，定义时无需指明类关系，如第26行所示，但指向具体函数时，如同普通成员函数一样，需要指明函数的隶属关系，如第27行所示。对于友元函数，与非类中成员函数相同，如第31-33行所示。

4、void

void在C/C++比较常见，如例程9-10所示。当用于函数时，如果函数没有返回值，则将返回值类型设置为void；如果函数没有参数，则将参数列表设置为void。可以将各种类型的指针转换为void*，如第14行代码所示，但不能将void *赋值给其他类型的指针，如第15行代码所示。当用void *作为函数参数时，可以接受各种类型的指针；如果用void *作为返回值，也可以返回各种类型的指针。第6-8行函数voidAnyPointer()就是void*的使用，第20-25行是voidAnyPointer()函数应用，分别支持int *和double*(注：某些编译器需要将int *p=voidAnyPointer(&b)修改为int *p=(int *)voidAnyPointer(&b)即将void *转换为制定类型的指针，第24行代码也需要类似转换)。

例程9-10

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  void sayHelloA(){
第4行      cout<<"asdf"<<endl;
第5行  }
第6行  void sayHello(void){
第7行      cout<<"Hello!C++!"<<endl;
```

```
第8行  }
第9行  void * voidAnyPointer(void *p){
第10行     return p;
第11行 }
第12行 int main(){
第13行     int a=10,b=100;

第14行     int *pA=&a,*pB=&b;
第15行     void *pVoid;
第16行
第17行     pVoid=pA;
第18行     //pB=pVoid;//错误!
第19行
第20行     int *p=voidAnyPointer(&b);
第21行     cout<<*p<<endl;
第22行
第23行     double dblNum=99.99;
第24行     double *dblP=voidAnyPointer(&dblNum);
第25行     cout<<*dblP<<endl;
第26行
第27行     return 0;
第28行 }
```

第二节　关键字补阙

由于多种原因，部分C++关键字尚未得到介绍，本节将予以补充介绍。

1、explict

在例程9-11中，XYZ类有单参数构造函数，默认情况下，可以用第10行的方式调用构造函数，虽然XYZ类和整数40没有关系，注意此时，不会调用"="运算符。C++的这种隐式规则并不是都合适，此时可以用explicit关键字予以抑制，如例程9-12所示，此时执行xyz=40将导致错误。

例程9-11

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  struct XYZ{
第4行      XYZ(void){};
第5行      XYZ(int a):Num(a){}
第6行      int Num;
第7行  };
第8行  int main(){

第9行      XYZ xyz;
第10行     xyz=40;
第11行
第12行     return 0;
第13行 }
```

例程9-12

```
#include<iostream>
using namespace std;
struct XYZ{
    XYZ(void){};
    explicit XYZ(int a):Num(a){cout<<"In XYZ(int a):Num(a)"<<endl;}
    int Num;
};
int main(){
    XYZ xyz(40);

    return 0;
}
```

如果类中有重载operator=()，则默认的单参数构造函数被调用的情况将被operator=()所代替或将优先调用operator=()。如果将operator=()置为私有成员函数，则将不允许调用该函数，屏蔽了单参数赋值现象，如例程9-13所示。

例程9-13

```
#include<iostream>
using namespace std;
struct XYZ{
    XYZ(void){};
    XYZ(int a):Num(a){}
    int Num;
private:
    XYZ& operator=(int a){
        this->Num=a;

        return *this;
    }
};
int main(){
    XYZ xyz;
    //xyz=40;//错误!

    return 0;
}
```

2、mutable

观察例程9-14，类udC中有三个私有成员，分别numA、numB和readCount，但readCount前面多了mutable关键字。成员函数print()原型后有关键字const，即在该函数内部不能修改数据成员的值。但第8行代码readCount++很明显将修改readCount的值，违背const规定。但是这项规定仅对非mutable修饰数据成员有效，对于mutable修饰的数据成员，在const语境中，仍然可以改变其数值。

例程9-14

```
#include<iostream>
using namespace std;
class udC{
public:
    udC(){readCount=0;};
```

```
第6行       udC(int a,int b):numA(a),numB(b){readCount=0;}
第7行       void print()const{
第8行           readCount++;
第9行           cout<<"numA="<<numA<<" numB="<<numB<<endl;
第10行          cout<<"readCount="<<readCount<<endl;
第11行      }
第12行  private:
第13行      int numA,numB;
第14行      mutable int readCount;
第15行  };
第16行  int main(){
第17行      udC myUDc(10,20);

第18行      myUDc.print();
第19行      myUDc.print();
第20行      myUDc.print();
第21行      system("pause");
第22行      return 0;
第23行  }
```

很明显，mutable用于在const语境中修改数据成员的值。在类设计过程中，应该仅有极少与数据成员使用mutable修饰符，如果太多应用，就失去了应用的意义。

3、auto

auto关键字在C++98与C++11中差别较大，本小节以C++98为准，有关auto在C++11中的应用请参看其他相关章节。

auto关键字很少被应用，因为在程序块中，变量默认被申明为auto存储类型，仅仅在程序块中有效。使用auto关键字，仅仅是将隐式申明编程显示申明。也正因为如此，auto关键字在C++11之前的版本中实用性不强，在C++11版本中，auto成为自动推断变量类型的关键字。

例程9-15

```
第1行   #include<iostream>
第2行   using namespace std;
第3行
第4行   int main(){
第5行       {
第6行           auto int i=10;
第7行           int j=10;
第8行       }
第9行
第10行      return 0;
第11行  }
```

4、extern

extern关键字置于变量或者函数前，以表明变量或者函数在别的文件中定义，当编译器遇到该关键字时将在其他模块中寻找该变量或者函数的定义。例程9-16和例程9-17是一个工程或项目中的两个CPP文件，在other.cpp中定义了i和print()函数，在main.cpp中，并没有定义i和print()函数，但可以告诉编译器，这俩定义在外部(extern)，到外部查找这俩的定义。extern常用于大型或超大型软件编写中。

例程9-16

```
第1行   //Main.cpp文件
第2行   #include<iostream>
```

```
using namespace std;
extern int i;
extern void print();
int main(){
    cout<<i<<endl;//输出：100
    i++;
    print();

    return 0;
}
```

例程9-17

```
//other.cpp文件
#include<iostream>
using namespace std;
int i=100;
void print(){
    cout<<i<<endl;
}
```

5、wchar_t

wchar_t类似于char类型，用于处理宽字符。程序中的字符主要ASCII编码，但是也还有很多非ASCII码，如中文等。C++提供了一些列处理宽字符的类、函数等，这些函数大都与关键字wchar_t有关，或者说以此为基础，如例程9-18所示。

例程9-18

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

locale loc("chs");//设置为chinese simplified，即中文简体

int main(){
    wchar_t wStr[] =L"美丽的汉语言！";
    wstring myString=L"伟大的祖国！";
    cout<<wcslen(wStr)<<endl;//长度
    wcout.imbue(loc);//输出时按loc对象设置进行
    wcout<<myString[1]<<endl;
    wcout<<wStr[1]<<endl;

    return 0;
}
```

宽字符处理大都与locale库文件有关，且可通过locale类设置宽字符代码，在例程第6行即为申明loc对象为locale类，宽字符代码chs，即chinese simplified(简体中文)。第9行申明wStr变量的类型为wchar_t，字符串前面L字符表明字符串为宽字符；第10行的wstring是string的宽字符版本；第11行的wcslen()函数用于返回宽字符串的长度；第12行wcout.imbue()用于设置宽字符输出时关联的locale类，此处按loc对象的设置。与wcout对应还有wcin等用于宽字符输入。

对于myString[1]和wStr[1]而言，如果未采用宽字符版本，将导致莫名其妙的输出，原因是这些宽字符由两个字节组成，

myString[1]和wStr[1]将仅仅截取一个字符编码，将导致输出错误。

在全部C++关键字集合中，还有asm、register等关键字。asm用于在C++代码中嵌入汇编代码。register用于在局部作用域中把变量放入CPU寄存器中，以提高效率。在C++中，register一般不能用于全局变量，并且将register用于修饰全局变量，让该变量一直占用某个存储器，一般说来是很坏的选择。

第三节　预处理

预处理是C++的重要组成部分，如常见的#include<iostream>中的#include就是典型预处理。C++预处理主要有三种类型，分别是文件包含、条件编译和宏定义。在C++中，预处理命令均以“#”符号开头。

1、文件包含

文件包含的含义是一个文件包含另一个文件的内容，其格式如下：

```
#include<文件名>
或
#include"文件名"
```

在C++中，文件名所指向的文件一般称为头文件、库文件。头文件的内容除了函数原型、宏定义外，还可以含有结构体定义，全局变量定义等。在C++中，标准库文件采用#include<文件名>模式，如常见的#include<iostream>、#include<string>等，编译时编译器到系统指定位置查找库文件；自定义头文件则采用#include"文件名"模式，当头文件不含有位置信息时，默认为与源代码同一位置，如不在同一位置，则需要指明文件位置。

include可以嵌套，即被包含的头文件中还可以继续包含其他头文件。

2、宏定义

宏定义又称为宏代换、宏替换，简称“宏”，在C++中，#define实现宏定义。为更好地理解宏定义，先观察例程9-19。

例程9-19

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  #define PI 3.1415926
第5行  #define CA(r) PI*r*r
第6行
第7行  int main(){
第8行      long double radius=2.0;
第9行      cout<<CA(radius)<<endl;
第10行
第11行     return 0;
第12行 }
```

在例程9-19第4行、第5行就是宏定义。当编译时，编译器首先将CA(radius)替换为PI*r*r，然后再将PI替换为3.1415926，最后将r替换为radius。宏定义相关说明如下：

1. 宏名一般用大写；
2. 宏定义末尾不加分号；
3. 使用宏可提高程序的通用性和易读性，减少不一致性，减少输入错误且便于维护。如：数组大小常用宏定义；
4. 预处理不做语法检查，宏定义也不例外；
5. 宏定义的函数不在花括号外边，作用域为宏定义后的程序。宏定义函数通常在文件的最开头；
6. 宏定义不分配内存，变量定义分配内存；
7. 宏定义可以嵌套(如CA的定义)；
8. #undef命令可以终止宏定义的作用域，如#undef PI；
9. 字符串引号中永远不包含宏；
10. 宏名和参数的括号间不能有空格；

11. 宏只作替换，不做计算，不做表达式求解；
12. 宏函数的形参与实参结合不存在类型，也没有类型转换，仅作替换；
13. 宏函数的替换会使源程序变长，函数调用则不会；

观察例程9-20第9行，发现输出没有达到预期，其原因是宏定义是替换，首先将CA(1.0+1.0)替换为CA(r)，然后替换为PI*1.0+1.0*1.0+1.0，然后将PI替换后得到5.14159。为了避免类似错误，将CA(r)定义为PI*(r)*(r)即可，如例程9-21所示。

例程9-20

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  #define PI 3.1415926
第5行  #define CA(r) PI*r*r
第6行  int main(){
第7行      long double radius=2.0;
第8行      cout<<CA(radius)<<endl;//输出：12.5664
第9行      cout<<CA(1.0+1.0)<<endl;//输出：5.14159
第10行
第11行     return 0;
第12行 }
```

例程9-21

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  #define PI 3.1415926
第5行  #define CA(r) PI*(r)*(r)
第6行  int main(){
第7行      long double radius=2.0;
第8行      cout<<CA(radius)<<endl;//输出：12.5664
第9行      cout<<CA(1.0+1.0)<<endl;//输出：12.5664
第10行
第11行     return 0;
第12行 }
```

在宏定义中，#运算符有特定用途，如例程9-22所示。当#出现字符串之后时，相当于用函数中的变量替换变量，如#define REPLACE(name) "I am "#name，相当于用name的值替换#name，当输入REPLACE(John)，则name被替换为John。当出现##时，则相当于连接变量，如第5行和第8行所示。

例程9-22

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  #define REPLACE(name) "I am "#name
第5行  #define JOIN(a,b) a##b
第6行  int main(){
第7行      cout<<REPLACE(John)<<endl;//输出：I am John
第8行      cout<<JOIN(10,20)<<endl;//输出：1020
第9行
```

```
第10行      return 0;
第11行  }
```

与#define相对的是#undef，用于取消已定义的宏。

3、条件编译

一般情况下，源程序中所有的行都参加编译。但有时希望对其中一部分内容只在满足一定条件下才进行编译，即对一部分内容根据条件编译，这就是“条件编译”。条件编译指令将决定哪些代码被编译，而哪些是不被编译的。可根据表达式的值或某个特定宏是否被定义来确定编译条件。条件编译指令有：#if、#else、#elif和#endif指令组，#ifdef和#ifndef指令组，#error指令，#line指令，#pragma指令。

#ifdef与#ifndef的含义分别是如果已经定义或如果没有定义，其配对的结束指令是#endif，且经常与#if、#else、#elif和#endif指令组组合使用。

⑴ #if、#else、#elif和#endif指令组

这组指令有如下两种格式：

```
#if 表达式
  //语句段1
[#else
  //语句段2]
#endif
```

```
#if 表达式1
  //语句段1
#elif 表达式2
  //语句段2
#else
  //语句段3
#endif
```

例程9-23是#if、#else、#elif和#endif指令组的使用示例，其中#ifndef ARRAY_SIZE的含义是如果没有宏定义ARRAY_SIZE标志符。

例程9-23

```
第1行  #ifndef ARRAY_SIZE
第2行  #define ARRAY_SIZE 100
第3行  #endif
第4行
第5行  #if ARRAY_SIZE>200
第6行  #undef ARRAY_SIZE
第7行  #define ARRAY_SIZE 200
第8行
第9行  #elif ARRAY_SIZE<50
第10行  #undef ARRAY_SIZE
第11行  #define ARRAY_SIZE 50
第12行
第13行  #else
第14行  #undef ARRAY_SIZE
第15行  #define ARRAY_SIZE 100
第16行  #endif
```

```
第17行
第18行    int data[ARRAY_SIZE];
```

(2) #ifdef和#ifndef指令组

#ifdef的含义是如果已经定义(if define)，#ifndef的含义是如果没有定义(if no define)，这两种预处理与#endif配对使用，如例程9-24所示。

例程9-24

```
第1行    #ifdef ARRAY_SIZE
第2行    int data[ARRAY_SIZE];
第3行    #endif
第4行
第5行    #ifndef ARRAY_SIZE
第6行    #define ARRAY_SIZE 100
第7行    int data[ARRAY_SIZE]
第8行    #endif
```

(3) #error

当遇到#error指令时将终止编译过程，产生编译错误，并输出#error后定义的信息，如例程9-25所示。在这个例子中，如果宏_cplusplus没有定义，将终止编译过程(所有编译器都定义有__cplusplus宏)。

例程9-25

```
第1行    #ifndef __cplusplus
第2行    #error A C++ compiler is required!
第3行    #endif
```

(4) #pragma

在所有的预处理指令中，#pragma指令可能是最复杂，它的作用是设定编译器的状态或者是指示编译器完成一些特定的动作。#pragma指令与具体编译器密切相关，在保持与C和C++语言完全兼容的情况下，给出主机或操作系统专有的特征。当编译器不支持#pragma参数时，将忽略该参数，且不会产生语法错误。

#pragma Para