C++程序设计课程设计

实习指导书

夏军宝

中国地质大学（北京）信息工程学院

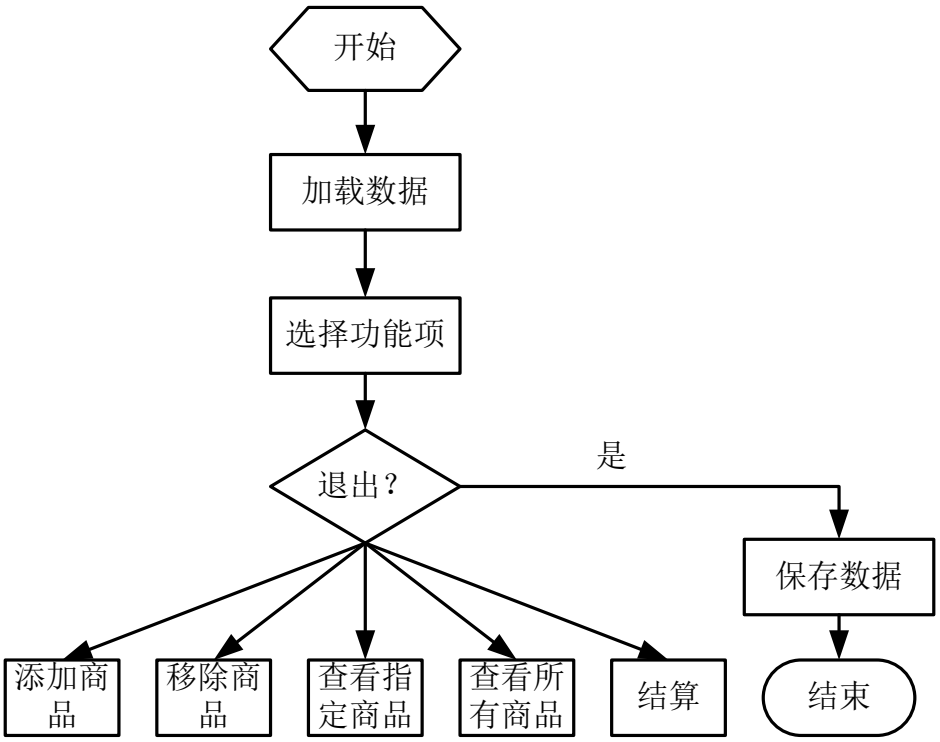
2021 年 11 月

目　　录

1　过程化编程综合实践

1.1　购物篮管理系统版本 1（数组版本）

开发购物篮管理系统，实现对购物篮中商品的基本管理，包括添加商品、移除商品、查看指定商品、查看所有商品、结算等操作，系统通过控制台界面显示菜单并提示用户操作，最后的数据要支持磁盘存储（序列化）。



1.1.1 交互式界面设计

1.1.1.1 基本实现（main.cpp）

1) 创建纯 C++语言项目，项目名称 Commodity。
2) 修改自动生成的 main.cpp。实现基本的交互式界面，提示若干选项，由用户进行选择，根据用户选择执行不同的功能。

```
#include <iostream>
using namespace std;
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    cout<<"    0) 退出系统\n";
    cout<<"    1) 向购物篮添加商品\n";
```

```
        cout<<"    2) 从购物篮移除商品\n";
        cout<<"    3) 查看指定商品\n";
        cout<<"    4) 查看所有商品\n";
        cout<<"    5) 结算\n";
        cout<<"请输入功能选项:";
        char choice;
        while(true){
            cin>>choice;
            if(choice=='0')
                break;
            switch(choice)          {
            case '1':     //具体功能省略
                break;
            case '2':   //具体功能省略
                break;
            case '3':   //具体功能省略
                break;
            case '4':   //具体功能省略
                break;
            case '5':   //具体功能省略
                break;
            default:
                cout<<"无效输入！请重试!\n";
                break;
            }
        }
        cout<<"再见!\n";
        return 0;
    }
```

程序中没有将 choice 定义为 int，主要考虑到程序的容错处理。如果将 choice 定义为 int 类型，while 循环中通过 cin 读取整数，如果用户输入非数字字符，则程序将陷入死循环，引入输入的非数字字符一直保存在键盘输入缓冲区中，每次 cin 都将解析数据失败。

1.1.1.2 问题剖析

1) 为保持主程序的简洁，需要将复杂功能提取出来，定义为函数，将 menu 功能保存到独立的函数中，并定义相应的头文件；

2) 程序基本结构设计：在项目中添加 function.cpp，定义系统主要功能函数（如 menu()）；添加 function.h，声明功能函数原型，以便在主程序或其它代码中调用；主程序文件 main.cpp 仅仅包含顶层功能逻辑。

3）添加功能函数的原型声明（function.h）

```
char menu();
```

4）添加功能函数的定义（function.cpp）

```
#include <iostream>
using namespace std;
char menu(){
    cout<<"   0) 退出系统\n";
    cout<<"   1) 向购物篮添加商品\n";
    cout<<"   2) 从购物篮移除商品\n";
    cout<<"   3) 查看指定商品\n";
    cout<<"   4) 查看所有商品\n";
    cout<<"   5) 结算\n";
    cout<<"请输入功能选项:";
    char choice;
    cin>>choice;
    return choice;
}
```

5) 修改主程序（main.cpp）

```
#include <iostream>
#include "function.h"
using namespace std;
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    char choice;
    while(true){
        choice=menu();
        if(choice=='0')
            break;
        switch(choice){
        case '1':    //具体功能省略
            break;
        case '2':    //具体功能省略
            break;
        case '3':   //具体功能省略
            break;
        case '4':   //具体功能省略
            break;
        case '5':   //具体功能省略
            break;
        default:
            cout<<"无效输入！请重试!\n";
            break;
```

```
            }
        }
        cout<<"再见!\n";
        return 0;
    }
```

1.1.2 定义数据

1.1.2.1 数据结构（datatype.h）

添加 datatype.h 文件，存放描述单个商品信息的结构体。将数据结构定义放在独立的头文件中。

```
#include <string>       //使用标准库中的 string 处理字符串
struct CommodityInfo{
    long id;     //商品编号
    std::string name;   //商品名称，头文件中避免使用 using namespace 命令
    double price;   //定价
    int num;     //数量
    double discount;     //折扣
};
const int MAX_COMMODITY_NUM=100;   //允许的最大商品数量常量
```

1.1.2.2 定义商品全局数据（data.cpp）

添加 data.cpp，在 data.cpp 文件中，定义存储所有商品信息的数组，为跟踪当前商品的数量，定义辅助变量，以记录实际商品数量。

```
#include "datatype.h"
CommodityInfo commodities[MAX_COMMODITY_NUM];
int commodityNum=0;
```

1.1.2.3 定义商品全局数据的声明（data.h）

添加 data.h 文件，定义全局变量的声明，以方便在程序中引用。全局变量可以在所有模块中访问，尽量只在主程序中访问，其它功能模块尽量以传参方式访问，不要直接访问。

```
#include "datatype.h"
extern CommodityInfo commodities[MAX_COMMODITY_NUM];
extern int commodityNum;
```

1.1.2.4 头文件的处理技巧

在 main.cpp 中包含了 datatype.h 和 function.h 头文件，将来会添加更多功能模块，可能也会访问 datatype.h 的结构体信息，这样在 function.h 中也会包含 datatype.h；

如此，会导致 main.cpp 文件重复包含 datatype.h 文件中的内容，引起编译错误，解决头文件重复包含的方法。

1）修改商品信息头文件（datatype.h）

```
#ifndef DATATYPE_HEADER
#define DATATYPE_HEADER
struct CommodityInfo{
    long id;
    std::string name;
    double price;
    int num;
    double discount;
};
const int MAX_COMMODITY_NUM=100;  //允许的最大商品数量常量
#endif
```

2）修改功能函数的原型声明头文件（function.h）

```
#ifndef FUNCTION_HEADER
#define FUNCTION_HEADER
#include "datatype.h"
char menu();
#endif
```

3）修改全局数据的声明头文件（data.h）

```
#ifndef DATA_HEADER
#define DATA_HEADER
#include "datatype.h"
extern CommodityInfo commodities[MAX_COMMODITY_NUM];
extern int commodityNum;
#endif
```

1.1.3 添加业务功能

1.1.3.1 添加功能函数原型声明（function.h）

针对主程序中的每个菜单项，需要编写 1 个配套的功能函数，功能函数的定

义放在 function.cpp 中，同时在 function.h 中添加功能函数声明，在主程序的相应菜单下调用功能函数。

因商品信息的数组和计数变量虽然是全局变量，尽量不要直接访问（最好只在 main 方法中引用 1 次），在调用功能函数时将数据作为参数传递给函数。数组可以通过指针传递，如果函数需要修改计数变量 commodityNum，需要传递引用；如果不修改传值即可。

```
#ifndef FUNCTION_HEADER
#define FUNCTION_HEADER
#include "datatype.h"
char menu();
void displayCommodities(CommodityInfo *pCommodities, int num);
void addCommodity(CommodityInfo *pCommodities, int &num);
void removeCommodity(CommodityInfo *pCommodities, int &num);
void viewCommodity(CommodityInfo *pCommodities, int num);
void checkOut(CommodityInfo *pCommodities, int num);
#endif
```

1.1.3.2 功能函数需求分析

1) 向购物篮中添加商品：为简化处理，新增商品添加到现有数据的尾部，商品数量加 1，当实际商品数量达到 MAX_COMMODITY_NUM 时，应禁止添加。添加商品时，若商品 Id 号已经存在时，只是累加商品数量（忽略其它信息）。添加商品信息时，提示用户输入商品的 id、名称、价格、数量和折扣。
2) 从购物篮中移除指定的商品：需要用户指定商品的 Id，系统查找指定 Id 的商品是否存在，若存在则删除相应的商品信息，若不存在则提示信息。删除商品成功后，commodityNum 减 1。因为采用数组存储信息，删除某个商品后，应将后续的商品数据往前移动。
3) 查看指定商品：需要用户指定商品 Id，系统查找指定 Id 的商品是否存在，若存在则显示该商品的信息，若不存在则提示信息。
4) 查看所有商品：遍历显示所有商品的信息。
5) 结算：计算所有商品的总价钱和结算信息。

分析以上的功能需求，发现存在许多重复的功能，应将其提炼出来，作为公共模块加以调用。

根据商品 Id 号查找商品在多个模块中重复（查看商品、添加商品、移除商品），将其提取出来作为内部公共模块（findCommodityById）。

查看商品模块找到特定商品后显示该商品的信息，而显示所有商品模块是逐一显示所有商品的信息，可以将显示单个商品信息的功能提取出来作为内部公共模块（showCommodityInfo）。

结算模块需要累计所有商品的价格，将计算单个商品价格的功能提取出来作为内部公共模块（getCommodityPrice）。

添加商品的模块中，需要用户输入商品的所有信息，可以将设定商品信息的模块提出作为独立的模块（setCommodityInfo）。

1.1.3.3 辅助功能实现（help.h、help.cpp）

四个辅助模块（findCommodityById、showCommodityInfo、getCommodityPrice、setCommodityInfo）是供功能函数使用的内部模块，不会被主程序调用，它们的声明不应放在 function.h 中（应该封装隐藏起来），可以考虑将它们放在独立的文件中，如 help.h 和 help.cpp 中，甚至做成动态链接库发布。

findCommodityById 查找特定 Id 的商品，若存在则返回该商品的地址，若不存在则返回 nullptr（空指针）。

1）添加 help.h 文件，辅助函数的原型声明

```
#ifndef HELP_HEADER
#define HELP_HEADER
#include "datatype.h"
CommodityInfo *findCommodityById(CommodityInfo *pCommodities,
                int num, long id);
void showCommodityInfo(CommodityInfo *pCommodity);
void setCommodityInfo(CommodityInfo *pCommodity);
double getCommodityPrice(CommodityInfo *pCommodity);
#endif
```

2）添加 help.cpp，定义辅助函数

```
#include <iostream>
#include "datatype.h"
void setCommodityInfo (CommodityInfo *pCommodity){ //读取商品信息
    fflush(stdin);    //确保之前残留的回车被清理，商品名称取整行
    cout<<"   输入商品名称: ";
    getline(cin,pCommodity->name);
    cout<<"   输入商品价格: ";
    cin>>pCommodity->price;
    cout<<"   输入商品数量: ";
    cin>>pCommodity->num;
    cout<<"   输入商品折扣: ";
```

```
    cin>>pCommodity->discount;
}
CommodityInfo *findCommodityById(CommodityInfo *pCommodities,
                int num, long id){
    CommodityInfo *pCommodity=pCommodities;
    for(; pCommodity < pCommodities+num; pCommodity++){
        if(pCommodity->id==id){
            return pCommodity;
        }
    }
    return nullptr;
}
double getCommodityPrice(CommodityInfo *pCommodity){
    return pCommodity->price*pCommodity->num*pCommodity->discount;
}
void showCommodityInfo(CommodityInfo *pCommodity){
    cout<<"商品编号(id):"<<pCommodity->id<<endl;
    cout<<"      商品名称:"<<pCommodity->name<<endl;
    cout<<"      商品总价:"<<getCommodityPrice(pCommodity)
        <<" (价格:"<<pCommodity->price<<", 数量:"
        <<pCommodity->num<<", 折扣:"<<pCommodity->discount<<" )\n";
}
```

1.1.3.4 功能函数实现（function.cpp）

在 function.cpp 文件中，给出所有功能函数的定义

```
#include <iostream>
#include "function.h"
#include "help.h"
#include "datatype.h"
using namespace std;
char menu(){
    cout<<"   0) 退出系统\n";
    cout<<"   1) 向购物篮添加商品\n";
    cout<<"   2) 从购物篮移除商品\n";
    cout<<"   3) 查看指定商品\n";
    cout<<"   4) 查看所有商品\n";
    cout<<"   5) 结算\n";
    cout<<"请输入功能选项:";
    char choice;
    cin>>choice;
    return choice;
}
```

```
void displayCommodities(CommodityInfo *pCommodities, int num){
    int i;
    cout<<"商品种类: "<<num<<endl;
    for(i=0;i<num;i++)  {
        showCommodityInfo(&pCommodities[i]);
    }
    cout<<endl;
}
void addCommodity(CommodityInfo *pCommodities, int &num){
    int id;
    cout<<"    输入商品编号(id): ";
    cin>>id;
    CommodityInfo *pCommodity=
            findCommodityById(pCommodities,num,id);
    if(pCommodity!=nullptr){//找到商品，累加数量
        cout<<"编号为"<<id<<"的商品已经存在!\n";
        cout<<"请输入增加的商品数量: ";
        int number;
        cin>>number;
        pCommodity->num+=number;
        return;
    }
    if(num==MAX_COMMODITY_NUM){
        cout<<"没有足够空间了!\n\n";
        return;
    }
    pCommodity = &pCommodities[num];
    pCommodity->id=id;
    setCommodityInfo(pCommodity);
    num++;
    cout<<"商品添加成功!\n\n";
}
void removeCommodity(CommodityInfo *pCommodities, int &num){
    int id;
    cout<<"    输入商品编号(id): ";
    cin>>id;
    CommodityInfo *pCommodity=
            findCommodityById(pCommodities,num,id);
    if(pCommodity==nullptr){
        cout<<"编号为"<<id<<"的商品不存在!\n\n";
        return;
    }
    num--;
    CommodityInfo *pNext=pCommodity+1;
```

```
        while(pCommodity < pCommodities + num){
            pCommodity->id=pNext->id;
            pCommodity->name=pNext->name;
            pCommodity->price=pNext->price;
            pCommodity->num=pNext->num;
            pCommodity->discount=pNext->discount;
            pCommodity++;
            pNext++;
        }
        cout<<"商品移除成功!\n\n";
    }
    void viewCommodity(CommodityInfo *pCommodities, int num){
        int id;
        cout<<"    输入商品的编号(id): ";
        cin>>id;
        CommodityInfo *pCommodity=
                findCommodityById(pCommodities,num,id);
        if(pCommodity==nullptr){
            cout<<"编号为"<<id<<"的商品不存在!\n\n";
            return;
        }
        showCommodityInfo(pCommodity);
        cout<<endl;
    }
    void checkOut(CommodityInfo *pCommodities, int num){
        double totalPrice=0;
        int totalNum=0;
        cout<<"商品种类: "<<num<<endl;
        cout<<"    商品名称\t\t"<<"价格\t"<<"件数\t"<<"折扣\t"<<"总价\n";
        for(int i=0;i<num;++i){
            double price=getCommodityPrice(pCommodities+i);
            cout<<"    "<<pCommodities[i].name<<"\t";
            cout<<pCommodities[i].price<<"\t"
                <<pCommodities[i].num<<"\t"
                <<pCommodities[i].discount<<"\t"
                <<price<<endl;
            totalPrice+=price;
            totalNum+=pCommodities[i].num;
        }
        cout<<"购物篮商品总件数: "<<totalNum<<"\n";
        cout<<"购物篮结算总价: "<<totalPrice<<endl;
    }
```

1.1.3.5 主程序中调用功能函数（main.cpp）

修改主程序 main.cpp，集成对功能函数的调用。

```
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    char choice;
    while(true){
        choice=menu();
        if(choice=='0')
            break;
        switch(choice){
        case '1':
            addCommodity (commodities,commodityNum);
            break;
        case '2':
            removeCommodity(commodities,commodityNum);
            break;
        case '3':
            viewCommodity(commodities,commodityNum);
            break;
        case '4':
            displayCommodities(commodities,commodityNum);
            break;
        case '5':
            checkOut(commodities,commodityNum);
            break;
        default:
            cout<<"无效输入！请重试!\n";
            break;
        }
    }
    cout<<"再见!\n";
    return 0;
}
```

1.1.3.6 进一步优化头文件结构（header.h）

随着系统中头文件的增加，在 main 方法中包含头文件的代码会不断增加，显得凌乱而不方便，需要进一步优化。将常用的头文件再集中在 1 个独立的头文件中，主要代码只要引用这 1 个头文件即可。

1）添加 header.h，包含其它主要的头文件：

```
#include "datatype.h"
#include "function.h"
#include "help.h"
#include "data.h"
```

2）简化 main 方法的头文件包含

```
#include <iostream>
#include "header.h"
using namespace std;
int main(){
    …
}
```

3）简化 function.cpp 文件中的头文件包含

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

1.1.4 实现文件存储

1.1.4.1 存储结构设计

要实现文件的写入和读取，需要设计合理的结构，考虑到商品数量的动态变化，首先存入的信息应该是商品数量，然后是每个商品的详细信息。考虑到商品名称可能会包含空格，所以需要独占 1 行，以便于解析。

商品数量
商品 1 的 id
name
price　num　discount
商品 2 的 id
name
price　num　discount
…

1.1.4.2 读写文件的功能函数原型声明（function.h）

```
#ifndef FUNCTION_HEADER
#define FUNCTION_HEADER
#include "datatype.h"
```

```
#include <string>
char menu();
void displayCommodities(CommodityInfo *pCommodities, int num);
void addCommodity(CommodityInfo *pCommodities, int &num);
void removeCommodity(CommodityInfo *pCommodities, int &num);
void viewCommodity(CommodityInfo *pCommodities, int num);
void checkOut(CommodityInfo *pCommodities, int num);
void readData(std::string filename);
void writeData(std::string filename);
#endif
```

1.1.4.3 读写文件的功能函数定义（function.cpp）

```
#include <iostream>
#include <fstream>
#include "header.h"
using namespace std;
void readData(std::string filename){
    ifstream in(filename);
    if(in){
        in>>commodityNum;
        if(commodityNum>MAX_COMMODITY_NUM)
            commodityNum=MAX_COMMODITY_NUM;
        string buf;
        for(int i=0;i<commodityNum;++i){
            in>>commodities[i].id;
            getline(in,buf);
            getline(in,commodities[i].name);
            in>>commodities[i].price>>commodities[i].num
                >>commodities[i].discount;
        }
    }
}
void writeData(std::string filename){
    ofstream out(filename);
    if(out){
        out<<commodityNum<<endl;
        for(int i=0;i<commodityNum;++i){
            out<<commodities[i].id<<endl;
            out<<commodities[i].name<<endl;
            out<<commodities[i].price<<" "<<commodities[i].num
               <<" "<<commodities[i].discount<<endl;
        }
    }
```

```
}
```

1.1.4.4 主程序中集成文件读写操作（main.cpp）

```
#include <iostream>
#include "header.h"
using namespace std;
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    readData("commodity.dat");
    char choice;
    while(true){
        …
    }
    writeData("commodity.dat");
    cout<<"再见!\n";
    return 0;
}
```

1.2 商品管理系统版本 2（动态扩容）

版本 1 中的代码，定义数组保存商品信息，需要在代码中硬编码数组大小，修改大小后需要重新编译发布程序。定义数组时，大小是固定的，便无法添加数量超过数组大小的商品。

可以考虑在程序运行过程中动态分配内存，系统启动后，分配初始大小的内存，随着用户不断添加数据，当初始内容用完后，重新分配内存，实现内存的动态管理。

定义全局变量（maxCommodityNum），以保存用户设置的最大商品数量。为便于对全局变量的访问和管理，将 maxCommodityNum 定义在 data.cpp 文件中，在 data.h 中定义 maxCommodityNum 的声明。将原先的数组修改为指针，在程序中根据需要动态分配内存。

通过 addCommodity 方法添加商品，当初始分配的内存用完之后，调用 reAllocMemory 动态扩容（倍增容量）。

创建纯 C++语言项目，项目名称 Commodity02，将版本 1 中的所有 cpp 文件和头文件复制到当前项目路径下，右击项目 Commodity02，在快捷菜单中选择“添加现有文件”，将其添加到当前项目中，再对其进行改造。

1.2.1.1 修改全局数据定义和声明（data.cpp、data.h）

1）修改 data.cpp 中的全局数据定义

```
#include "datatype.h"
int maxCommodityNum=1;
CommodityInfo *pCommodities;
~~CommodityInfo commodities[MAX_COMMODITY_NUM];~~
int commodityNum=0;
```

2）修改 data.h 中的全局数据声明

```
#ifndef DATA_HEADER
#define DATA_HEADER
extern int maxCommodityNum;
extern CommodityInfo *pCommodities;
~~extern CommodityInfo commodities[MAX_COMMODITY_NUM];~~
extern int commodityNum;
#endif
```

3）删除 datatype.h 中的常量定义

```
#ifndef DATATYPE_HEADER
#define DATATYPE_HEADER
#include <string>       //使用标准库中的 string 处理字符串
struct CommodityInfo{
    long id;     //商品编号
    std::string name;     //商品名称
    double price;   //定价
    int num;     //数量
    double discount;     //折扣
};
~~const int MAX_COMMODITY_NUM=100;   //允许的最大商品数量常量~~
#endif
```

1.2.1.2 修改辅助函数（help.cpp、help.h）

在 help.cpp 中添加“重新分配内存”辅助函数定义，并在 help.h 中添加辅助函数声明。

1）help.cpp 文件中添加 reAllocMemory 定义

```
#include <iostream>
#include "header.h"
using namespace std;
void reAllocMemory(CommodityInfo *&pCommodities,int num){
    maxCommodityNum*=2;
```

```
        CommodityInfo *temp=pCommodities;
        pCommodities=new CommodityInfo[maxCommodityNum];
        for(int i=0;i<num;++i)
            pCommodities[i]=temp[i];
        delete[] temp;
    }
```

... //其它代码省略

代码解析：在 reAllocMemory 函数中，maxCommodity 直接访问全局变量，pCommodities 是指针引用，因为要重新分配内存，原先的指针需要指向新分配的内存空间，通过引用才能在 reAllocMemory 中使得修改生效。

2）help.cpp 文件中添加 reAllocMemory 原型声明

```
#ifndef HELP_HEADER
#define HELP_HEADER
#include "dataType.h"
CommodityInfo *findCommodityById(CommodityInfo *pCommodities,
            int num, long id);
void showCommodityInfo(CommodityInfo *pCommodity);
void setCommodityInfo(CommodityInfo *pCommodity);
double getCommodityPrice(CommodityInfo *pCommodity);
void reAllocMemory(CommodityInfo *&pCommodities,int num);
#endif
```

1.2.1.3 修改添商品功能实现（function.cpp、function.h）

1）在 function.cpp 中，修改 addEmployee 定义。

```
#include <iostream>
#include "header.h"
using namespace std;
void addCommodity(CommodityInfo *&pCommodities, int &num){
    int id;
    cout<<"   输入商品编号(id): ";
    cin>>id;
    CommodityInfo *pCommodity=
            findCommodityById(pCommodities,num,id);
    if(pCommodity!=nullptr){//找到商品，累加数量
        cout<<"编号为"<<id<<"的商品已经存在!\n";
        cout<<"请输入增加的商品数量: ";
        int number;
        cin>>number;
        pCommodity->num+=number;
        return;
    }
```

```
    if(num==MAX_COMMODITY_NUM){
        cout<<"没有足够空间了!\n\n";
        return;
    }
    if(num==maxCommodityNum){
        reAllocMemory(pCommodities,num);
    }
    pCommodity = &pCommodities[num];
    pCommodity->id=id;
    setCommodityInfo(pCommodity);
    num++;
    cout<<"商品添加成功!\n\n";
}
```

2）在 function.h 文件中修改 addEmployee 的原型声明

```
#ifndef FUNCTION_HEADER
#define FUNCTION_HEADER
#include "datatype.h"
char menu();
void displayCommodities(CommodityInfo *pCommodities, int num);
void addCommodity(CommodityInfo *&pCommodities, int &num);
void removeCommodity(CommodityInfo *pCommodities, int &num);
void viewCommodity(CommodityInfo *pCommodities, int num);
void checkOut(CommodityInfo *pCommodities, int num);
void readData(std::string filename);
void writeData(std::string filename);
#endif
```

1.2.1.4 修改读写文件操作（funcition.cpp）

在动态扩容的情况下，加载文件时需要根据实际存储的数据量动态分配内存，因此需要在文件存储时同时存储当前最大商品数（maxCommodityNum）和实际商品数（commodityNum），加载文件时根据这两个变量分配内存空间。

```
#include <iostream>
#include <fstream>
#include "header.h"
using namespace std;
void readData(std::string filename){
    ifstream in(filename);
    if(in){
        in>>maxCommodityNum;
        pCommodities=new CommodityInfo[maxCommodityNum];
        in>>commodityNum;
```

```
            if(commodityNum> maxCommodityNum)
                commodityNum= maxCommodityNum;
            string buf;
            for(int i=0;i<commodityNum;++i){
                in>> pCommodities [i].id;
                getline(in,buf);
                getline(in, pCommodities [i].name);
                in>> pCommodities [i].price>> pCommodities [i].num
                    >> pCommodities [i].discount;
            }
        }
        else
            pCommodities=new CommodityInfo[maxCommodityNum];
    }
    void writeData(std::string filename){
        ofstream out(filename);
        if(out){
            out<<maxCommodityNum<<endl;
            out<<commodityNum<<endl;
            for(int i=0;i<commodityNum;++i){
                out<< pCommodities [i].id<<endl;
                out<< pCommodities [i].name<<endl;
                out<< pCommodities [i].price<<" "<< pCommodities [i].num
                    <<" "<< pCommodities [i].discount<<endl;
            }
        }
    }
```

1.2.1.5 修改主程序，集成程序功能（main.cpp）

```
    #include <iostream>
    #include "header.h"
    using namespace std;
    int main(){
        cout<<"欢迎使用购物篮管理系!\n";
        readData("commodity2.dat");
        char choice;
        while(true)      {
            …//省略其余代码
        }
        writeData("commodity2.dat");
        delete [] pCommodities;
         return 0;
```

}

1.3 实习任务

1.3.1 实习任务一

将 findCommodityById 修改为返回商品在数组中的索引位置，重新实现系统。其原型如下

```
int findCommodityById (CommodityInfo *pCommodities,
                       int num, long id);
```

当找到商品时，返回找到商品的下标索引；未找到商品时，返回-1。

1.3.2 实习任务二

增加 1 个功能模块，修改商品信息。首先输入商品 id，查找商品是否存在。若商品存在，提示用户选择要修改的商品信息项，可以修改商品的数量、价格和折扣。

1.3.3 实习任务三

为“查看所有商品”增加子菜单，允许用户选择不同的排序方式（按商品 Id、按商品名称、按价格、按折扣、按总价格等），然后按照选定的排序方式输出商品清单信息。

1.3.4 实习任务四

参考商品管理系统，设计并实现简单的信息管理系统，参考但不限定于的选题如下：

员工管理系统、学生管理系统、课程管理系统、图书管理系统等，需求尽量简化，主要是练习系统构建的过程。

2 面向对象编程综合实践

2.1 购物篮管理系统版本 3（类与对象）

2.1.1 系统分析与设计

版本 1 和版本 2 实现的购物篮管理系统基于过程化方法实现，基于全局数据和大量操作数据的方法构成，抽象和封装不够。版本 3 将基于面向对象的思想，对程序中的主要实体进行封装。

购物篮管理系统包括 2 个类：Commodity 描述商品基本信息和对商品的操作；CommodityManage 实现对购物篮中商品的操作，包括添加、移除、查看指定商品、查看所有商品、结算等操作，主界面和交互保持不变。

创建纯 C++语言项目，项目名称为 Commodity03。

2.1.2 封装商品类（Commodity）

Commodity 类封装对单个商品信息，存储商品 id、商品名称、价格、购买数量、折扣等信息，提供相应的构造函数，提供设置和读取名称、价格、数量、折扣，读取总价格，输出基本信息等功能接口。

2.1.2.1 商品类定义（Commodity.h）

添加 commodity.h 文件，给出商品类的定义。

```
#ifndef COMMODITY_H
#define COMMODITY_H
#include <string>
class Commodity{
public:
    Commodity()=default;
    Commodity(long i,std::string n,double p=0,int nu=0,double d=0);
    void setPrice(double price){this->price=price;}
    void setNum(int num){this->num=num;}
    void setDiscount(double discount){this->discount=discount;}
    void setName(std::string name){this->name=name;}
    std::string getName()const{return name;}
```

```
        long getId()const {return id;}
        double getPrice()const{return price;}
        int getNum()const{return num;}
        double getDiscount()const{return discount;}
        double getNetPrice()const;
        void output()const;
    private:
        long id;
        std::string name;
        double price;
        int num;
        double discount;
    };
    #endif // COMMODITY_H
```

构造函数中使用缺省值，可以省略重载多个版本的构造函数。注意缺省值只能在类定义时给出（函数声明），在构造函数定义（CPP 文件）中不能再给出缺省值。

2.1.2.2 商品类成员函数定义（Commodity.cpp）

添加 commodity.cpp 文件，给出商品类的成员函数定义。

```
    #include <iostream>
    #include <string>
    #include "Commodity.h"
    using namespace std;
    Commodity::Commodity(long i,std::string n,double p,int nu,double d)
        :id(i),name(n),price(p),num(nu),discount(d){}
    double Commodity::getNetPrice()const{
        return price*num*discount;
    }
    void Commodity::output()const{
        cout<<"   商品编号(id):"<<id<<endl;
        cout<<"      商品名称:"<<name<<endl;
        cout<<"      商品总价:"<<getNetPrice()
            <<" (价格:"<<price<<", 数量:"<<num<<", 折扣:"
            <<discount<<" )\n";
    }
```

2.1.2.3 简单测试（main.cpp）

修改项目中自动生成的 main.cpp，对 Commodity 执行简单测试。

```
#include <iostream>
#include "Commodity.h"
using namespace std;
int main(){
    Commodity c(101,"C++ programming",108,2,0.9);
    c.output();
    cout<<"商品总价:"<<c.getNetPrice()<<endl;
    return 0;
}
```

2.1.2.4 解决商品类的 id 自增问题（Commodity）

Commodity 类的 id 值应由内部管理而不应该由用户指定，每次创建商品对象时，新商品的 id 应该能够自增，但是当我们从文件或数据库中加载数据时应按读取的 id 创建对象，不能采用自增值。

目前采用一种简单的处理办法，创建静态数据成员 nextId 并指定初始值为 100，定义静态方法 autoNextId，每次创建对象时读取并返回 id 值，同时递增该 id 值，从而有效管理 id 值。

为了便于管理，提供不带 id 参数的构造函数，创建对象时系统自增该 id，同时提供带 id 的构造函数，此时使用用户传入的 id，而不自增静态成员 id。

1）添加静态数据成员及配套构造函数（Commodity.h）

```
class Commodity{
public:
    Commodity()=default;
    Commodity(long i,std::string n,double p=0,int nu=0,double d=0);
    Commodity(std::string n,double p=0,int nu=0,double d=0);
    void setPrice(double price){this->price=price;}
    void setNum(int num){this->num=num;}
    void setDiscount(double discount){this->discount=discount;}
    void setName(std::string name){this->name=name;}
    std::string getName()const{return name;}
    long getId()const {return id;}
    double getPrice()const{return price;}
    int getNum()const{return num;}
    double getDiscount()const{return discount;}
    double getNetPrice()const;
    void output()const;
private:
    static long nextId;
    long autoNextId(){return nextId++;}
```

```
    long id;
    std::string name;
    double price;
    int num;
    double discount;
};
```

2）定义静态成员并实现新增构造函数（Commodity.cpp）

```
long Commodity::nextId=100;
Commodity::Commodity(string n,double p,int nu,double d)
    :Commodity(autoNextId(),n,p,nu,d){}
Commodity::Commodity(long i,std::string n,double p,int nu,double d)
    :id(i),name(n),price(p),num(nu),discount(d){}
```

3）修改测试程序（main.cpp）

```
int main(){
    Commodity c("C++ programming",108,2,0.9);
    c.output();
    cout<<"商品总价:"<<c.getNetPrice()<<endl;
    Commodity c2("STL programming",80,1,0.8);
    c2.output();
    return 0;
}
```

2.1.3 封装商品管理类（CommodityManage）

2.1.3.1 商品管理类分析

商品管理类 CommodityManage 实现对购物篮中商品数据集合的管理，包含 Commodity 指针作为数据成员（指向动态分配的 Commodity 数组），实现商品的添加、移除等操作。动态数组在创建时根据初始大小动态分配内存，当分配的内存占满后，再动态扩充容量。

向购物篮添加商品：为简化处理，新增商品添加到现有数据的尾部，商品数量加 1；如果添加商品的 Id 已经存在，只是累加购买数量。

从购物篮删除商品：根据用户指定的商品 Id，查找指定 Id 的商品是否存在，若存在则删除相应的商品信息，若不存在则提示信息。删除商品成功后，商品数量减 1。

查看指定商品：根据用户指定的商品 Id，查找指定 Id 的商品是否存在，若存在则显示该商品的信息，若不存在则提示信息。

查看所有商品：遍历显示所有商品的信息。

结算：根据购物篮中的商品信息，给出结算信息。

分析以上的功能需求，发现存在重复的功能，应将其提炼出来，作为CommodityManage 类的内部模块加以调用（私有方法）。根据商品 Id 号查找商品（findCommodityById）将被多个模块重复调用，将其定义为私有方法。

2.1.3.2 商品管理类定义（CommodityManage.h）

添加 commodityManage.h 文件，给出商品管理类的定义。

```
#ifndef COMMODITYMANAGE_H
#define COMMODITYMANAGE_H
class CommodityManage{
public:
    CommodityManage();
    CommodityManage(int s);
    ~CommodityManage(){delete[] pCommodities;}
    CommodityManage(const CommodityManage& c)=delete;
    CommodityManage& operator=(const CommodityManage& c)=delete;
    void addCommodity(const Commodity& c);
    void removeCommodity(int id);
    void viewCommodity(int id)const;
    void viewAllCommodities()const;
    void checkOut()const;
private:
    Commodity *pCommodities;   //指向动态创建的商品数组
    int maxSize;   //当前分配的空间上限
    int size;     //实际的商品数量
    Commodity* findCommodityById(int id)const;
};
#endif // COMMODITYMANAGE_H
```

通过赋值为 delete，将 CommdityManage 的拷贝构造函数和赋值运算符禁用，商品管理对象只能有 1 个对象，不能创建多个，更加有效的方法可以参照设计模式中的单件模式。

2.1.3.3 商品管理类成员函数定义（CommodityManage.cpp）

1）构造函数实现

```
#include "CommodityManage.h"
#include "Commodity.h"
```

```
CommodityManage::CommodityManage()
    :CommodityManage(100){}
CommodityManage::CommodityManage(int s)
    :maxSize(s),size(0){
    pCommodities=new Commodity[maxSize];
}
```

2）辅助函数（findCommodityById）

```
Commodity* CommodityManage::findCommodityById(int id)const{
    for(int i=0;i<size;++i)
        if(pCommodities[i].getId()==id)
            return pCommodities+i;
    return nullptr;
}
```

3）主要业务函数的实现

```
void CommodityManage::addCommodity(const Commodity& c){
    if(size==maxSize){
        cout<<"没有足够空间了!\n";
        return ;
    }
    Commodity* pCommodity=findCommodityById(c.getId());
    if(pCommodity!=nullptr){
        cout<<"编号为"<<c.getId()<<"的商品已经存在!累加其数量\n";
        pCommodity->setNum(pCommodity->getNum()+c.getNum());
        return;
    }
    pCommodities[size]=c;
    size++;
}
void CommodityManage::removeCommodity(int id){
    Commodity* pCommodity=findCommodityById(id);
    if(pCommodity==nullptr){
        cout<<"编号为"<<id<<"的商品不存在!\n";
        return;
    }
    Commodity *pNext;
    size--;
    while(pCommodity<pCommodities+size){//后面的商品往前移动
        pNext=pCommodity+1;
        *pCommodity=*pNext;
        pCommodity++;
    }
}
void CommodityManage::viewCommodity(int id)const{
```

```
    Commodity* pCommodity=findCommodityById(id);
    if(pCommodity==nullptr){
        cout<<"编号为"<<id<<"的商品不存在!\n";
        return;
    }
    pCommodity->output();
}
void CommodityManage::viewAllCommodities()const{
    cout<<"商品种类:"<<size<<endl;
    for(int i=0;i<size;++i)
        pCommodities[i].output();
}
void CommodityManage::checkOut()const{
    double totalPrice=0;
    int totalNum=0;
    cout<<"商品种类: "<<size<<endl;
    cout<<"    商品名称\t\t"<<"价格\t"<<"件数\t"<<"折扣\t"<<"总价\n";
    for(int i=0;i<size;++i){
        double price=(pCommodities+i)->getNetPrice();
        cout<<"    "<<pCommodities[i].getName()<<"\t";
        cout<<pCommodities[i].getPrice()<<"\t"
            <<pCommodities[i].getNum()<<"\t"
            <<pCommodities[i].getDiscount()<<"\t"
            <<price<<endl;
        totalPrice+=price;
        totalNum+=pCommodities[i].getNum();
    }
    cout<<"购物篮商品总件数: "<<totalNum<<"\n";
    cout<<"购物篮结算总价: "<<totalPrice<<endl;
}
```

2.1.3.4 实现动态扩容（CommodityManage.h、CommodityManage.cpp）

1）定义私有接口（CommodityManage.h）

```
class CommodityManage{
public:
    …
private:
    Commodity *pCommodities;   //指向动态创建的商品数组
    int maxSize;   //当前分配的空间上限
    int size;     //实际的商品数量
    Commodity* findCommodityById(int id)const;
    void reAllocMemory();
```

```
};
```

2）实现接口并修改添加商品模块（CommodityManage.cpp）

```
void CommodityManage::reAllocMemory(){
    maxSize*=2;
    int i;
    Commodity *temp=pCommodities;
    pCommodities=new Commodity[maxSize];
    for(i=0;i<size;++i)
        pCommodities[i]=temp[i];
    delete[] temp;
}
void CommodityManage::addCommodity(const Commodity& c){
    if(size==maxSize){
        reAllocMemory();
    }
    Commodity* pCommodity=findCommodityById(c.getId());
    if(pCommodity!=nullptr){
        cout<<"编号为"<<c.getId()<<"的商品已经存在!累加其数量\n";
        pCommodity->setNum(pCommodity->getNum()+c.getNum());
        return;
    }
    pCommodities[size]=c;
    size++;
}
```

2.1.3.5 简单测试（main.cpp）

```
#include <iostream>
#include "Commodity.h"
#include "CommodityManage.h"
using namespace std;
int main(){
    CommodityManage cm(1);
    cm.addCommodity(Commodity("Java programming",100,2,0.8));
    cm.addCommodity(Commodity("C++ programming",120,3,0.9));
    cm.checkOut();
    return 0;
}
```

2.1.4 搭建主界面

2.1.4.1 辅助函数原型声明（function.h）

```
#ifndef FUNCTION_H
#define FUNCTION_H
#include "CommodityManage.h"
char menu();
void doAddCommodity(CommodityManage& cm);
void doRemoveCommodity(CommodityManage& cm);
void doViewCommodity(const CommodityManage& cm);
void doViewAllCommodity(const CommodityManage& cm);
void doCheckout(const CommodityManage& cm);
#endif // FUNCTION_H
```

2.1.4.2 定义辅助函数（function.cpp）

```
#include <iostream>
#include "function.h"
using namespace std;
char menu(){
    cout<<"   0) 退出系统\n";
    cout<<"   1) 向购物篮添加商品\n";
    cout<<"   2) 从购物篮移除商品\n";
    cout<<"   3) 查看指定商品\n";
    cout<<"   4) 查看所有商品\n";
    cout<<"   5) 结算\n";
    cout<<"请输入功能选项:";
    char choice;
    cin>>choice;
    return choice;
}
void doAddCommodity(CommodityManage& cm){
    string name;
    double price,discount;
    int num;
    fflush(stdin);
    cout<<"Input name:";
    getline(cin,name);
    cout<<"Input price,num,discount:";
    cin>>price>>num>>discount;
    cm.addCommodity(Commodity(name,price,num,discount));
```

```
}
void doRemoveCommodity(CommodityManage& cm){
    long id;
    cout<<"Input id:";
    cin>>id;
    cm.removeCommodity(id);
}
void doViewCommodity(const CommodityManage& cm){
    long id;
    cout<<"Input id:";
    cin>>id;
    cm.viewCommodity(id);
}
void doViewAllCommodity(const CommodityManage& cm){
    cm.viewAllCommodities();
}
void doCheckout(const CommodityManage& cm){
    cm.checkOut();
}
```

代码说明：在 doAddCommodity 中，输入 name 时，先调用 fflush 清空缓冲区，防止其它输入的残留。传参时，doAddCommodity 和 doRemoveCommodity 需要修改 cm 状态，所以传入非常引用，其它方法传入常引用。

2.1.4.3 主程序实现（main.cpp）

```
#include <iostream>
#include "Commodity.h"
#include "CommodityManage.h"
#include "function.h"
using namespace std;
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    char choice;
    CommodityManage cm;
    while(true){
        choice=menu();
        if(choice=='0')
            break;
        switch(choice){
        case '1':
            doAddCommodity(cm);
            break;
```

```
            case '2':
                doRemoveCommodity(cm);
                break;
            case '3':
                doViewCommodity(cm);
                break;
            case '4':
                doViewAllCommodity(cm);
                break;
            case '5':
                doCheckout(cm);
                break;
            default:
                cout<<"无效输入！请重试!\n";
                break;
            }
        }
        cout<<"再见!\n";
        return 0;
    }
```

2.1.5 集成文件存储功能

2.1.5.1 增加文件读写接口（CommodityManage.h）

```
class CommodityManage{
public:
    CommodityManage();
    CommodityManage(int s);
    void readData(std::string filename);
    void saveData(std::string filename);
    …
};
```

2.1.5.2 读写文件的功能实现（CommodityManage.cpp）

```
#include <iostream>
#include <string>
#include <fstream>
#include "CommodityManage.h"
#include "Commodity.h"
using namespace std;
```

```
void CommodityManage::saveData(string filename){
    ofstream out(filename);
    if(out){
        out<<maxSize<<endl;
        out<<size<<endl;
        for(int i=0;i<size;++i){
            out<< pCommodities[i].getId()<<endl;
            out<< pCommodities[i].getName()<<endl;
            out<< pCommodities[i].getPrice()<<" "
                << pCommodities[i].getNum()<<" "
                << pCommodities[i].getDiscount()<<endl;
        }
    }
}
void CommodityManage::readData(string filename){
    ifstream in(filename);
    if(in){
        int fileMax,fileSize;
        in>>fileMax>>fileSize;
        long id;
        string name,buf;
        double price,discount;
        int num;
        for(int i=0;i<fileSize;++i){
            in>>id;
            getline(in,buf);   //读取 id 后残留的回车要读到 buf 中
            getline(in,name);
            in>>price>>num>>discount;
            addCommodity(Commodity(id,name,price,num,discount));
        }
    }
}
```

2.1.5.3 修改主程序（main.cpp）

```
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    char choice;
    CommodityManage cm;
    cm.readData("d:\\commodity03.data");
    while(true){
        …
    }
    cout<<"再见!\n";
```

```
        cm.saveData("d:\\commodity03.data");
        return 0;
    }
```

2.1.6 解决自增 id 的 bug

目前自增 id 的解决办法比较简单，在 Commodity 中定义静态数据成员 nextId，不指定 id 的构造函数执行时，取 nextId 作为自己的 id，之后递增 nextId。

如果每次都是手工创建商品对象，该解决方案没有问题。但是，我们从文件中读取商品信息时，每次的 id 都是从文件中读取，nextId 还是保留在之前的初始值，此后用户在手工创建商品时，会导致 id 的冲突。

一种简单解决方案：每次存储文件时将最后的 nextId 也保存到文件中，在读取文件时，读取并恢复 nextId 到 Commodity 中。

2.1.6.1 增加读写 nextId 的接口（Commodity.h）

```
class Commodity{
public:
    static void setNextId(long id){nextId=id;}
    static long getNextId(){return nextId;}
    …
};
```

2.1.6.2 增加读写文件的实现（CommodityManage.cpp）

```
void CommodityManage::saveData(string filename){
    ofstream out(filename);
    if(out){
        out<<maxSize<<endl;
        out<<size<<endl;
        out<<Commodity::getNextId()<<endl;
        …
    }
}
void CommodityManage::readData(string filename){
    ifstream in(filename);
    if(in){
        int fileMax,fileSize;
        long nextId;
```

```
            in>>fileMax>>fileSize>>nextId;
            Commodity::setNextId(nextId);
            …
        }
    }
```

2.2 购物篮管理系统版本 4（继承与多态）

2.2.1 继承与多态的分析与设计

Commodity 类中定义 id、name、price、num、discount 数据成员，表达的只是普通商品的信息，不具有普遍性；可能还存在二手商品、海外购商品、特卖商品，结算价格的算法也不相同。对于二手商品，没有折扣的概念，但有折旧率的算法；海外购商品，除了折扣以外，还需要额外的关税；特卖商品，设置特价，没有其它影响因素。

不同商品存在共性（基本属性），同时存在差异（结算结果的计算方法、输出信息不同），可以采用继承体系来描述它们。

将版本 3 中的 Commodity 类改造为基类，只定义所有商品都具备的共性属性和方法，在商品基类基础上，派生 SecondhandCommodity、NormalCommodity、OverseaCommodity、SpecialCommodity，实现各自特殊的处理逻辑。

考虑到 CommodityManage 类中定义 Commodity（基类）数组（通过指针动态分配内存），其中存储的可能是不同商品子类对象，为实现统一的处理接口，应该采用多态技术。

创建纯 C++语言项目，项目名称为 Commodity04。

2.2.2 商品基类（Commodity）

点击“文件”→“新建文件或项目”菜单项，在弹出对话框中，左侧选择“C++”，中间选择“C++ Class”，在向导中指定类名称为“Commodity”，文件名保持默认值，确定后自动创建头文件和 cpp 文件，并给出程序框架。

2.2.2.1 商品基类定义（Commodity.h）

```
#ifndef COMMODITY_H
#define COMMODITY_H
class Commodity{
public:
    virtual ~Commodity(){}
    Commodity()=default;
    Commodity(long i,std::string n,double p,int nn);
    Commodity(std::string n,double p,int nn);
    void setPrice(double price){this->price=price;}
    void setNum(int num){this->num=num;}
    void setName(std::string name){this->name=name;}
    std::string getName()const{return name;}
    double getPrice()const{return price;}
    int getNum()const{return num;}
    long getId()const{return id;}
    double getNetPrice()const;
    void output()const;
    static void setNextId(long id){nextId=id;}
    static long getNextId(){return nextId;}
private:
    long id;
    std::string name;
    double price;   //商品数量
    int num; //购买数量
    static long nextId;
    long autoNextId(){return nextId++;}
};
#endif   // COMMODITY_H
```

2.2.2.2 商品基类成员函数定义（Commodity.cpp）

```
#include <iostream>
#include "commodity.h"
using namespace std;
long Commodity::nextId=100;
Commodity::Commodity(string n,double p,int nu)
    :Commodity(autoNextId(),n,p,nu){}
Commodity::Commodity(long i,std::string n,double p,int nu)
    :id(i),name(n),price(p),num(nu){}
double Commodity::getNetPrice()const{
    return price*num;
```

```
}
void Commodity::output()const{
    cout<<"   商品编号(id):"<<id<<endl;
    cout<<"      商品名称:"<<name<<endl;
}
```

2.2.2.3 简单测试（main.cpp）

```
#include <iostream>
#include "commodity.h"
using namespace std;
int main(){
    Commodity c1("Java Programming",108,2);
    c1.output();
    return 0;
}
```

2.2.2.4 定义抽象方法（Commodity.h）

```
class Commodity{
public:
    Commodity()=default;
    virtual double getNetPrice()const;
    virtual void output()const;
    static void setNextId(long id){nextId=id;}
    static long getNextId(){return nextId;}
    …
};
```

2.2.3 派生商品子类

2.2.3.1 普通商品类（NormalCommodity）

点击“文件”→“新建文件或项目”菜单项，创建 NormalCommodity，指定基类名称为 Commodity。

1）普通商品类定义（NormalCommodity.h）

```
#ifndef NORMALCOMMODITY_H
#define NORMALCOMMODITY_H
#include <string>
#include "commodity.h"
```

```
class NormalCommodity : public Commodity{
public:
    virtual ~NormalCommodity(){}
    NormalCommodity(long id,std::string name,
        double p=0,int n=0,double d=0);
    NormalCommodity(std::string name,double p=0,int n=0,double d=0);
    void setDiscount(double discount){this->discount=discount;}
    double getDiscount()const{return discount;}
    virtual double getNetPrice()const;
    virtual void output()const;
private:
    double discount;
};
#endif // NORMALCOMMODITY_H
```

2）普通商品类成员函数定义（NormalCommodity.cpp）

```
#include <iostream>
#include "normalcommodity.h"
using namespace std;
NormalCommodity::NormalCommodity(long id,std::string name,
        double p,int n,double d)
    :Commodity(id,name,p,n),discount(d){}
NormalCommodity::NormalCommodity(std::string name,double p,
        int n,double d)
    :Commodity(name,p,n),discount(d){}
double NormalCommodity::getNetPrice()const{
    return Commodity::getNetPrice()*discount;
}
void NormalCommodity::output()const{
    Commodity::output();
    cout<<"      商品总价:"<<getNetPrice()<<" (价格:"
        <<getPrice()<<", 数量:"<<getNum()<<", 折扣:"
        <<discount<<" )\n";
}
```

2.2.3.2 海外购商品类（OverseaCommodity）

1）海外购商品类定义（OverseaCommodity.h）

```
#ifndef OVERSEACOMMODITY_H
#define OVERSEACOMMODITY_H
#include <string>
#include "commodity.h"
class OverseaCommodity : public Commodity{
public:
```

```
    virtual ~OverseaCommodity(){}
    OverseaCommodity(long id,std::string name,
        double p=0,int n=0,double d=1.0,double t=0);
    OverseaCommodity(std::string name,double p=0,
        int n=0,double d=1.0,double t=0);
    void setDiscount(double discount){this->discount=discount;}
    void setTariff(double tariff){this->tariff=tariff;}
    double getDiscount()const{return discount;}
    double getTariff()const{return tariff;}
    virtual double getNetPrice()const;
    virtual void output()const;
private:
    double discount;
    double tariff;
};
#endif // OVERSEACOMMODITY_H
```

2）海外购商品类成员函数定义（OverseaCommodity.cpp）

```
#include <iostream>
#include "overseacommodity.h"
using namespace std;
OverseaCommodity::OverseaCommodity(long id,std::string name,
        double p,int n,double d,double t)
    :Commodity(id,name,p,n),discount(d),tariff(t){}
OverseaCommodity::OverseaCommodity(std::string name,double p,
        int n,double d,double t)
    :Commodity(name,p,n),discount(d),tariff(t){}
double OverseaCommodity::getNetPrice()const{
    return Commodity::getNetPrice()*discount+tariff;
}
void OverseaCommodity::output()const{
    Commodity::output();
    cout<<"      商品总价:"<<getNetPrice()<<" (价格:"
        <<getPrice()<<", 数量:"<<getNum()<<", 折扣:"
        <<discount<<", 关税:"<<tariff<<" )\n";
 }
```

2.2.3.3 简单测试（main.cpp）

```
#include <iostream>
#include "commodity.h"
#include "normalcommodity.h"
#include "overseacommodity.h"
using namespace std;
```

```
int main(){
    NormalCommodity c1("Java Programming",108,2,0.9);
    c1.output();
    OverseaCommodity c2("C++ Programming",100,3,0.8,23.4);
    c2.output();
    Commodity *p=&c1;
    p->output();
    p=&c2;
    p->output();
    return 0;
}
```

2.2.4 商品管理类（CommodityManage）

商品管理类的主要功能逻辑没有变化，从 Commodity03 中将商品管理类的头文件和 cpp 文件复制到当前项目路径下，右击项目 Commodity04，在快捷菜单中选择“添加现有文件”，将其添加到当前项目中，再对其进行改造。

多态必须要通过指向基类的指针或引用才能实现，在商品管理类中需要保存和管理的是商品类的指针，每个指针指向动态创建的商品，因为需要管理多个商品，所以数据结构应该使用二级指针。

2.2.4.1 修改商品管理类定义（CommodityManage.h）

```
class CommodityManage{
public:
    CommodityManage();
    CommodityManage(int s);
    ~CommodityManage();{delete[] pCommodities;}
    CommodityManage(const CommodityManage& c)=delete;
    CommodityManage& operator=(const CommodityManage& c)=delete;
    void addCommodity(Commodity* p);
    void removeCommodity(int id);
    void viewCommodity(int id)const;
    void viewAllCommodities()const;
    void checkOut()const;
    void readData(std::string filename);
    void saveData(std::string filename);
private:
    Commodity **pCommodities;   //指向动态创建的商品指针数组
    int maxSize;   //当前分配的空间上限
```

```
    int size;     //实际的商品数量
    Commodity* findCommodityById(int id)const;
    void reAllocMemory();
};
```

2.2.4.2 修改商品管理类基础成员函数定义（CommodityManage.cpp）

```
CommodityManage::CommodityManage(int s)
    :maxSize(s),size(0){
    pCommodities=new Commodity*[maxSize];
}
CommodityManage::~CommodityManage(){
    for(int i=0;i<size;++i)
        delete pCommodities[i];
    delete[] pCommodities;
    pCommodities=nullptr;
}
void CommodityManage::addCommodity(Commodity* p){
    if(size==maxSize){
        reAllocMemory();
    }
    Commodity* pCommodity=findCommodityById(p->getId());
    if(pCommodity!=nullptr){
        cout<<"编号为"<<p->getId()<<"的商品已经存在!累加其数量\n";
        pCommodity->setNum(pCommodity->getNum()+p->getNum());
        return;
    }
    pCommodities[size]=p;
    size++;
}
void CommodityManage::removeCommodity(int id){
    Commodity* pCommodity=findCommodityById(id);
    if(pCommodity==nullptr){
        cout<<"编号为"<<id<<"的商品不存在!\n";
        return;
    }
    delete pCommodity;
    size--;
    Commodity **pos=pCommodities+size;
    while(*pos!=pCommodity){
        pos--;
    }
    while(pos<pCommodities+size){     //移动指针数组中的元素
```

```
            *pos=*(pos+1);
            pos++;
        }
    }
    void CommodityManage::viewCommodity(int id)const{
        Commodity* pCommodity=findCommodityById(id);
        if(pCommodity==nullptr){
            cout<<"编号为"<<id<<"的商品不存在!\n";
            return;
        }
        pCommodity->output();
    }
    void CommodityManage::viewAllCommodities()const{
        cout<<"商品种类:"<<size<<endl;
        for(int i=0;i<size;++i)
            pCommodities[i]->output();
    }
    Commodity* CommodityManage::findCommodityById(int id)const{
        for(int i=0;i<size;++i)
            if(pCommodities[i]->getId()==id)
                return pCommodities[i];
        return nullptr;
    }
    void CommodityManage::reAllocMemory(){
        maxSize*=2;
        int i;
        Commodity **temp=pCommodities;
        pCommodities=new Commodity*[maxSize];
        for(i=0;i<size;++i)
            pCommodities[i]=temp[i];
        delete[] temp;
    }
```

2.2.4.3 调整结算成员函数功能（CommodityManage.cpp）

不同类型商品包含的属性不同，用简单表格列出不同类型商品的信息比较困难，对 checkOut 的输出功能进行调整，只输出必要的信息。在查看商品信息的功能菜单中通过多态的 output 输出每个商品的完整信息。

```
    void CommodityManage::checkOut()const{
        double totalPrice=0;
        int totalNum=0;
        cout<<"商品种类: "<<size<<endl;
```

```
    cout<<"    商品名称\t\t"<<"价格\t"<<"件数\t"<<"折扣\t"<<"总价\n";
    for(int i=0;i<size;++i){
        double price=pCommodities[i]->getNetPrice();
        cout<<"    "<<pCommodities[i]->getName()<<"\t";
        cout<<pCommodities[i]->getPrice()<<"\t"
            <<pCommodities[i]->getNum()<<"\t"
            <<pCommodities[i].getDiscount()<<"\t"
            <<price<<endl;
        totalPrice+=price;
        totalNum+=pCommodities[i]->getNum();
    }
    cout<<"购物篮商品总件数: "<<totalNum<<"\n";
    cout<<"购物篮结算总价: "<<totalPrice<<endl;
}
```

2.2.4.4 读写数据功能的问题分析与处理

每种商品子类的属性数量不同，保存这些商品的信息时，需要保存每个商品的不同属性数据，这样才能在读取数据时正确恢复该商品信息。

除了属性数据数量的不同之外，还需要存储商品所属子类的标识，这样才能正确恢复商品的子类类型。

为了简化处理，我们需要在基类中增加多态抽象方法，getType 返回子类类型编码，getInfo 返回保存对象属性值内容的字符串（每个子类知道本类有哪些属性），保存数据时，直接将字符串存入文件即可。

1）修改商品基类定义（Commodity.h）

```
class Commodity{
public:
    virtual double getNetPrice()const;
    virtual void output()const;
    virtual int getType()const=0;   //纯虚函数
    virtual std::string getInfo()const;
    …
};
```

2）商品类获取属性信息方法实现（Commodity.cpp）

```
#include <sstream>
string Commodity::getInfo()const{
    ostringstream ostr;
    ostr<<getId()<<endl;
    ostr<<getName()<<endl;
```

```
        ostr<<price<<" "<<num<<" ";
        return ostr.str();
    }
```

3）普通商品类定义的修改（NormalCommodity.h）

```
class NormalCommodity : public Commodity{
public:
    NormalCommodity(long id,std::string name,double p,int n,double d);
    NormalCommodity(std::string name,double p,int n,double d);
    void setDiscount(double discount){this->discount=discount;}
    double getDiscount()const{return discount;}
    virtual double getNetPrice()const;
    virtual void output()const;
    virtual int getType()const;
    virtual std::string getInfo()const;
private:
    double discount;
};
```

4）普通商品类虚函数的实现（NormalCommodity.cpp）

```
#include <sstream>
int NormalCommodity::getType()const{
    return 0;     //0 表示普通商品
}
string NormalCommodity::getInfo()const{
    ostringstream ostr;
    ostr<<getType()<<" ";   //先输出类型编码
    ostr<<Commodity::getInfo();   //输出基类的信息
    ostr<<discount<<endl;   //输出子类信息
    return ostr.str();
}
```

5）海外购商品类定义的修改（OverseaCommodity.h）

```
class OverseaCommodity : public Commodity{
public:
    …
    virtual double getNetPrice()const;
    virtual void output()const;
    virtual int getType()const;
    virtual std::string getInfo()const;
private:
    double discount;
    double tariff;
};
```

6）海外购商品类虚函数的实现（OverseaCommodity.cpp）

```
#include <sstream>
int OverseaCommodity::getType()const{
    return 1;   //1 表示海外购商品
}
string OverseaCommodity::getInfo()const{
    ostringstream ostr;
    ostr<<getType()<<" ";   //先输出类型编码
    ostr<<Commodity::getInfo();   //输出基类的信息
    ostr<<discount<<" "<<tariff<<endl;   //输出子类信息
    return ostr.str();
}
```

7）保存数据的修改（CommodityManage.cpp）

保存数据时，前面 3 个数据项保持不变，对于每个商品，首先存储商品类型（读取数据时能正确恢复对象），再存储该对象的属性值信息（不同商品的属性值不同，具体内容由多态的 getInfo 方法获得）。

```
void CommodityManage::saveData(string filename){
    ofstream out(filename);
    if(out){
        out<<maxSize<<endl;
        out<<size<<endl;
        out<<Commodity::getNextId()<<endl;
        for(int i=0;i<size;++i){
            out<< pCommodities[i]->getInfo();
            out<< pCommodities[i].getId()<<endl;
            out<< pCommodities[i].getName()<<endl;
            out<< pCommodities[i].getPrice()<<" "
                << pCommodities[i].getNum()<<" "
                << pCommodities[i].getDiscount()<<endl;
        }
    }
}
```

7）读取数据的修改（CommodityManage.cpp）

```
#include "normalcommodity.h"
#include "overseacommodity.h"
void CommodityManage::readData(string filename){
    ifstream in(filename);
    if(in){
        int fileMax,fileSize;
        long nextId;
        in>>fileMax>>fileSize>>nextId;
        Commodity::setNextId(nextId);
```

```
            int type;
            long id;
            string name,buf;
            double price,discount;
            double tariff;
            int num;
            for(int i=0;i<fileSize;++i){
                in>>type;
                in>>id;
                getline(in,buf);
                getline(in,name);
                in>>price>>num;
                if(type==0){
                    in>>discount;
                    addCommodity(new
                        NormalCommodity(id,name,price,num,discount));
                }
                else if(type==1){
                    in>>discount>>tariff;
                    addCommodity(new
                        OverseaCommodity(id,name,price,num,discount,tariff));
                }
            }
        }
    }
```

2.2.5 主程序修改（main.cpp）

主程序的主要功能逻辑没有变化，从 Commodity03 中将 main.cpp、function.h 和 function.cpp 文件复制到当前项目路径中，右击项目 Commodity04，在快捷菜单中选择“添加现有文件”，将其添加到当前项目中，再对其进行改造。

1）修改添加商品模块（function.cpp）

```
#include "normalcommodity.h"
#include "overseacommodity.h"
void doAddCommodity(CommodityManage& cm){
    string name;
    double price,discount;
    double tariff;
    int num,type;
    cout<<"选择商品类型(0-普通商品，1-海外购商品)";
    cin>>type;
```

```
        fflush(stdin);
        cout<<"输入商品名称:";
        getline(cin,name);
        cout<<"输入价格和商品数量:";
        cin>>price>>num;
        if(type==0){
            cout<<"输入商品折扣:";
            cin>>discount;
            cm.addCommodity(new
              NormalCommodity(name,price,num,discount));
        }
        else if(type==1){
            cout<<"输入商品折扣和关税:";
            cin>>discount>>tariff;
            cm.addCommodity(new
               OverseaCommodity(name,price,num,discount,tariff));
        }
    }
```

2）修改存储的文件路径（main.cpp）

```
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    char choice;
    CommodityManage cm;
    cm.readData("d:\\commodity04.data");
    while(true){
        …
    }
    cout<<"再见!\n";
    cm.saveData("d:\\commodity04.data");
    return 0;
}
```

2.3 实习任务

2.3.1 实习任务一

在购物篮管理系统版本 4 的基础之上，添加“二手商品”和“特价商品”2 种商品子类。二手商品包括折旧程度、折扣属性，计算净价时在原价基础之上乘以折扣和折旧；特价商品包含特价属性，计算净价时和原价没有关系，只是特价和数量的乘积。

在 Commodity 基础之上派生相应的子类，在主程序中修改代码，实现商品管理功能。

2.3.2 实习任务二

增加 1 个功能模块，修改商品信息。首先输入商品 id，查找商品是否存在。若商品存在，提示用户选择要修改的商品信息项，不同商品类型能修改的信息不同。尽量使用多态的处理手段，不同子类实现本类对象的数据修改。实现数据修改时，由基类负责价格和数量的修改，不同子类处理本类特有数据的修改。

2.3.3 实习任务三

以上一章所完成的信息管理系统为基础，寻找基类和子类，实现基于继承和多态的信息管理系统，进一步完善系统功能和架构。

3 STL 编程综合实践

3.1 购物车管理系统版本 5（STL）

3.1.1 版本 4 的问题分析

版本 4 采用面向对象技术，利用多态手段，实现了比较灵活的商品管理，但代码结构上还是存在一些问题。

首先，CommodityManage 承担了大量的任务，不但包括添加和移除商品、结算等功能，包含了大量的用于内存管理的职责，包括动态内存的分配与释放、内存不足情况下的内存扩容等，此外还包含了查找、排序等功能。应该尽量将底层数据管理功能和业务逻辑功能进行分离。

STL 中提供了大量的组件（容器）和算法，实现了众多的底层内存管理、查找、排序等算法，我们应该充分利用，让 CommodityManage 只负责和业务逻辑相关的内容，而底层管理功能由 STL 的容器和算法实现。

3.1.2 CommodityManage 类的改造

版本 4 的 CommodityManage 类中定义 Commodity 二级指针，同时定义 maxSize 和 size 成员变量，跟踪分配的空间和实际存储的商品数量。在改版的商品管理类中，定义 vector 容器保存所有的商品信息，vector 容器负责底层内存管理（动态扩容）及元素访问和查找等算法。为了支持多态效果，需要在 vector 中保存 Commodity 指针。

3.1.2.1 创建项目

创建纯 C++纯语言项目，项目名称为 Commodity05。将 Commodity04 版本中的 Commodity.h、Commodity.cpp、NormalCommodity.h、NormalCommodity.cpp、OverseaCommodity.h 、 OverseaCommodity.cpp 、 CommodityManage.h 和 CommodityManage.cpp 复制到项目路径中，并添加到项目中。

3.1.2.2 修改项目管理类定义（CommodityManage.h）

```
#include <vector>
```

```
class CommodityManage{
public:
    CommodityManage()=default;     缺省构造
    ~~CommodityManage(int s);   //不再需要该版本构造函数~~
    ~CommodityManage();
    CommodityManage(const CommodityManage& c)=delete;
    CommodityManage& operator=(const CommodityManage& c)=delete;
    void addCommodity(Commodity* p);
    void removeCommodity(int id);
    void viewCommodity(int id)const;
    void viewAllCommodities()const;
    void checkOut()const;
    void readData(std::string filename);
    void saveData(std::string filename);
private:
    ~~Commodity **pCommodities;   //指向动态创建的商品指针数组~~
    std::vector<Commodity*> pCommodities;
    ~~int maxSize;   //当前分配的空间上限~~
    ~~int size;     //实际的商品数量，不再需要跟踪内存使用情况~~
    ~~Commodity* findCommodityById(int id)const;~~
    ~~void reAllocMemory();     //不再需要管理内存~~
    Commodity* findCommodityById(int id);
    const Commodity* findCommodityById(int id)const;
    std::vector<Commodity*>::iterator getIterator(Commodity* p);
};
```

我们修改 findCommodityById 的返回类型保持不变，以减少代码的修改。此外，重载该函数为 2 个版本，通过普通对象调用普通版本，返回可修改元素内容的指针；通过常对象调用 const 版本，返回只能读取元素内容的指向常量的指针。

考虑到 vector 容器移除元素（删除商品）时，需要传入迭代器作为参数，而 findCommodityById 返回的是 Commodity 指针，定义辅助函数 getIterator，根据传入的指针返回指向该元素的迭代器。

3.1.2.3 修改项目管理类成员函数定义（CommodityManage.cpp）

```
~~CommodityManage::CommodityManage()~~
~~    :CommodityManage(100){}~~
~~CommodityManage::CommodityManage(int s)~~
~~    :maxSize(s),size(0){~~
~~    pCommodities=new Commodity*[maxSize];~~
~~}~~
CommodityManage::~CommodityManage(){
```

```
    for(int i=0;i<size;++i)
        delete pCommodities[i];
    delete[] pCommodities;
    pCommodities=nullptr;
    for(auto e: pCommodities)
        delete e;
}
void CommodityManage::addCommodity(Commodity* p){
    if(size==maxSize){
        reAllocMemory();
    }
    Commodity* pCommodity=findCommodityById(p->getId());
    if(pCommodity!=nullptr){
        cout<<"编号为"<<p->getId()<<"的商品已经存在!累加其数量\n";
        pCommodity->setNum(pCommodity->getNum()+p->getNum());
        return;
    }
    pCommodities[size]=p;
    size++;
    pCommodities.push_back(p);
}
void CommodityManage::removeCommodity(int id){
    Commodity* pCommodity=findCommodityById(id);
    if(pCommodity==nullptr){
        cout<<"编号为"<<id<<"的商品不存在!\n";
        return;
    }
    delete pCommodity;
    pCommodities.erase(getIterator(pCommodity));
    size--;
    Commodity **pos=pCommodities+size;
    while(*pos!=pCommodity){
        pos--;
    }
    while(pos<pCommodities+size){     //移动指针数组中的元素
        *pos=*(pos+1);
        pos++;
    }
}
void CommodityManage::viewCommodity(int id)const{
    const Commodity* pCommodity=findCommodityById(id);
    if(pCommodity==nullptr){
        cout<<"编号为"<<id<<"的商品不存在!\n";
        return;
```

```
        }
        pCommodity->output();
    }
```

代码分析：viewCommodity 为常成员函数，传入的隐含 this 指针为常量指针，调用的是常量版 findCommodityById，所以返回类型是 const Commodity*，通过该指针只能调用常方法（output）。

```
    void CommodityManage::viewAllCommodities()const{
        cout<<"商品种类:"<< pCommodities.size()<<endl;
        for(auto e:pCommodities)
            e->output();
     }
    void CommodityManage::checkOut()const{
        double totalPrice=0;
        int totalNum=0;
        cout<<"商品种类: "<< pCommodities.size()<<endl;
        cout<<"    商品名称\t\t"<<"价格\t"<<"件数\t"<<"总价\n";
        for(auto e : pCommodities){
            double price=e->getNetPrice();
            cout<<"    "<<e->getName()<<"\t";
            cout<<e->getPrice()<<"\t"
                <<e->getNum()<<"\t"
                <<price<<endl;
            totalPrice+=price;
            totalNum+=e->getNum();
        }
        cout<<"购物篮商品总件数: "<<totalNum<<"\n";
        cout<<"购物篮结算总价: "<<totalPrice<<endl;
    }
    Commodity* CommodityManage::findCommodityById(int id)const{
        for(auto e : pCommodities)
            if(e->getId()==id)
                return e;
        return nullptr;
    }
    const Commodity* CommodityManage::findCommodityById(int id)const{
        for(auto e : pCommodities)
            if(e->getId()==id)
                return e;
        return nullptr;
    }
    vector<Commodity*>::iterator CommodityManage::getIterator(Commodity* p){
        for(auto it=pCommodities.begin();it!=pCommodities.end();++it)
            if(*it==p)
```

```
                return it;
        return pCommodities.end();
}
void CommodityManage::reAllocMemory(){
    maxSize*=2;
    int i;
    Commodity **temp=pCommodities;
    pCommodities=new Commodity*[maxSize];
    for(i=0;i<size;++i)
        pCommodities[i]=temp[i];
    delete[] temp;
}
void CommodityManage::saveData(string filename){
    ofstream out(filename);
    if(out){
        out<<maxSize<<endl;
        out<< pCommodities.size()<<endl;
        out<<Commodity::getNextId()<<endl;
        for(auto e : pCommodities){
            out<<e->getInfo();
        }
    }
}
void CommodityManage::readData(string filename){
    ifstream in(filename);
    if(in){
        int fileMax,fileSize;
        long nextId;
        in>>fileMax>>fileSize>>nextId;
        Commodity::setNextId(nextId);
        int type;
        long id;
        string name,buf;
        double price,discount;
        double tariff;
        int num;
        for(int i=0;i<fileSize;++i){
            in>>type;
            in>>id;
            getline(in,buf);
            getline(in,name);
            in>>price>>num;
            if(type==0){
                in>>discount;
```

```
                addCommodity(new
                    NormalCommodity(id,name,price,num,discount));
            }
            else if(type==1){
                in>>discount>>tariff;
                addCommodity(new
                    OverseaCommodity(id,name,price,num,discount,tariff));
            }
        }
    }
}
```

3.1.2.4 修改主程序（CommodityManage.cpp）

将版本 4 中的 main.cpp、function.h、function.cpp 文件复制到 Commodity05 项目路径下，并添加到项目中。修改数据文件的保存路径。

```
int main(){
    cout<<"欢迎使用购物篮管理系统!\n";
    char choice;
    CommodityManage cm;
    cm.readData("d:\\commodity05.data");
    while(true){
        …
    }
    cout<<"再见!\n";
    cm.saveData("d:\\commodity05.data");
    return 0;
}
```

3.1.3 应用 STL 算法

算法是 STL 的重要构成部分，往往可以施加在不同容器之上，提供通用的泛型算法，应该尽量使用，而不是重新编写这些算法。在版本 4 的基础上，使用 STL 算法替代程序的查找、排序算法。

在 findCommodityById 方法中，可以考虑使用 find_if 算法。

因为 vector 中存储的是 Commodity 指针，不能直接用来作为查找、排序的比较运算，指针之间比较的是内存中的地址关系，应该通过函数对象、Lambda 表达式等形式自定义比较运算规则。

版本 4 中，输出商品列表时，根据添加商品的顺序进行输出，没有执行排序

操作。应该根据用户的选项，执行指定的排序操作。

3.1.3.1 应用 find_if 算法（CommodityManage.cpp）

```
Commodity* CommodityManage::findCommodityById(int id){
    for(auto e : pCommodities)
        if(e->getId()==id)
            return e;
    vector<Commodity*>::iterator it=find_if(pCommodities.begin(),
       pCommodities.end(), [=](Commodity* p){return p->getId()==id;});
    if(it!=pCommodities.end())
        return *it;
    return nullptr;
}
const Commodity* CommodityManage::findCommodityById(int id)const{
    for(auto e : pCommodities)
        if(e->getId()==id)
            return e;
    vector<Commodity*>::const_iterator it=find_if(pCommodities.begin(),
       pCommodities.end(), [=](const Commodity* p){return p->getId()==id;});
    if(it!=pCommodities.end())
        return *it;
    return nullptr;
}
```

3.1.3.2 应用 sort 算法（CommodityManage.cpp）

1）修改商品管理类定义（CommodityManage.h）

```
class CommodityManage{
public:
    …
private:
    std::vector<Commodity*> pCommodities;
    Commodity* findCommodityById(int id);
    const Commodity* findCommodityById(int id)const;
    std::vector<Commodity*>::iterator getIterator(Commodity* p);
    int sortType=0;   //记录当前排序类型
    void sortCommodities();
    void sortCommoditiesByType(int type);
};
```

sortType 跟踪当前排序的类型（0-商品 id,1-商品名称,2-商品净价），

sortCommodities 根据 sortType 执行实际的排序，sortCommoditiesByType 根据用户设置的新排序类型执行排序，如果传入的参数 type 和 sortType 相等则无需再次排序，如果不等则更新 sortType 值，调用 sortCommodities 实现排序。

2）修改商品管理类成员方法定义（CommodityManage.cpp）

```
void CommodityManage::addCommodity(Commodity* p){
    Commodity* pCommodity=findCommodityById(p->getId());
    if(pCommodity!=nullptr){
        cout<<"编号为"<<p->getId()<<"的商品已经存在!累加其数量\n";
        pCommodity->setNum(pCommodity->getNum()+p->getNum());
        return;
    }
    pCommodities.push_back(p);
    sortCommodities();  //添加商品后根据当前规则重新排序
}
void CommodityManage::sortCommodities(){
    switch(sortType){
    case 0:  //根据 id 排序
        sort(pCommodities.begin(),pCommodities.end(),
            [=](Commodity* p1,Commodity* p2){
                return p1->getId()<p2->getId();});
        break;
    case 1:  //根据名称排序
        sort(pCommodities.begin(),pCommodities.end(),
            [=](Commodity* p1,Commodity* p2){
                return p1->getName()<p2->getName();});
        break;
    case 2:  //根据净价排序
        sort(pCommodities.begin(),pCommodities.end(),
            [=](Commodity* p1,Commodity* p2){
                return p1->getNetPrice()<p2->getNetPrice();});
        break;
    }
}
void CommodityManage::sortCommoditiesByType(int type){
    if(type==sortType)      //已经按指定规则排序，直接返回
        return;
    sortType=type;
    sortCommodities();
}
void CommodityManage::viewAllCommodities()const{
    cout<<"商品种类:"<<pCommodities.size()<<endl;
    if(pCommodities.size()==0)
        return;
```

```
        cout<<"指定排序方式(0-商品 id,1-商品名称,2-商品净价):";
        int type;
        cin>>type;
        const_cast<CommodityManage*>(this)->sortCommoditiesByType(type);
        for(auto e:pCommodities)
            e->output();
    }
```

在 viewAllCommodities 方法中使用 const_cast，因为该方法是 const 方法，而 sortCommoditiesByType 是非常方法，不能直接调用，所有通过 const_cast 将 this 的 const 属性去掉。

```
    void CommodityManage::readData(string filename){
        ifstream in(filename);
        if(in){
            …
        }
        sortCommodities();
    }
```

3.2 实习任务

3.2.1 实习任务一

在购物篮管理系统版本 5 基础之上，增加按照购买数量、商品价格 2 种排序方式，修改相应程序代码并测试。

【扩展练习】如果系统中允许商品名称相同但 Id 不同的情况（不同的商品），修改按照名称排序算法，实现商品名称相同的情况下再按照商品 Id 排序。

3.2.2 实习任务二

根据新的需求，修改程序功能：

假定商品名称没有重复，添加商品时发现同名商品时只是累加商品数量，添加、移除等操作都以检测商品名称为依据，实现 findCommodityByName 方法，并据此修改相应的程序。

3.2.3 实习任务三

以上一章所完成的信息管理系统为基础，通过集合类实现数据的管理（添加、移除、内存管理等功能），实现业务逻辑管理与底层数据管理的分离。必要时通过应用适当的算法，优化程序结构与性能。

4 QT5 可视化编程基础

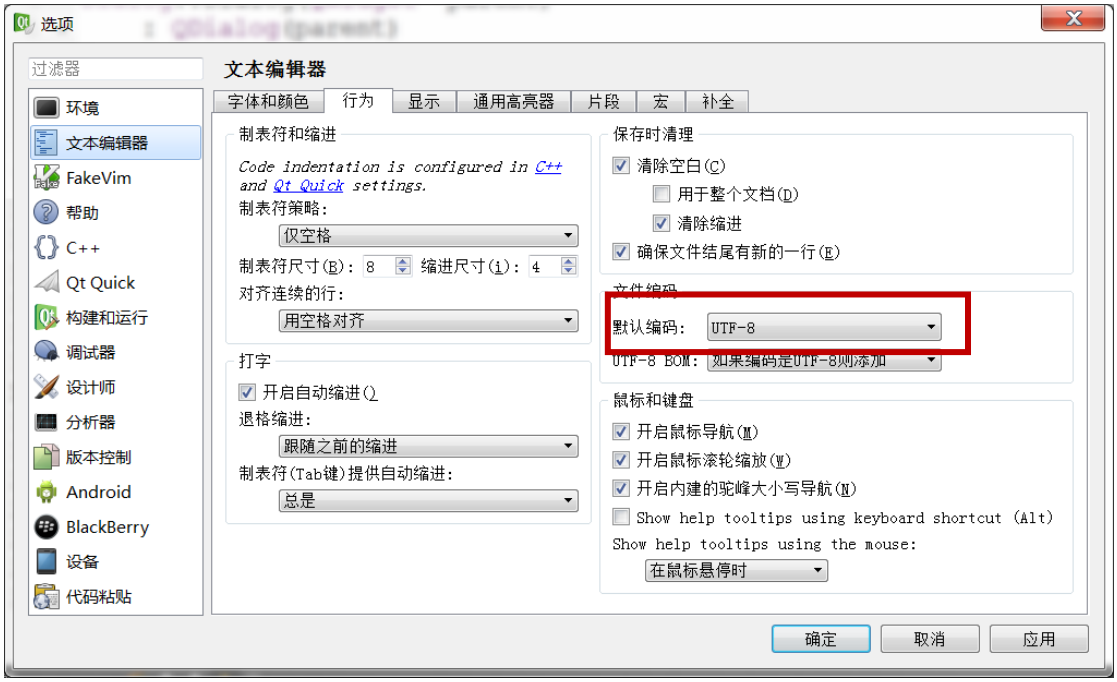
4.1 一个简单窗口界面示例

4.1.1 基于界面设计器的实现

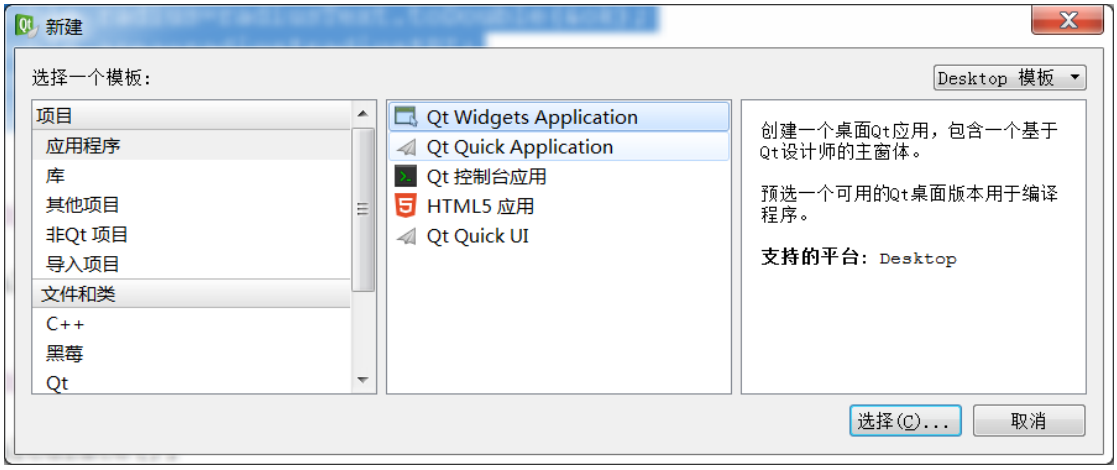
4.1.1.1 创建项目

在可视化界面项目中，不需要在控制台中输出信息，需要在系统中进行配置，将编码重新设置为 utf-8 编码，否则会出现中文乱码。

执行“工具”→“选项”菜单项，在弹出的对话框中，选择“文本编辑器”选项，在“行为”tab 页，设置默认编码为“utf-8”。

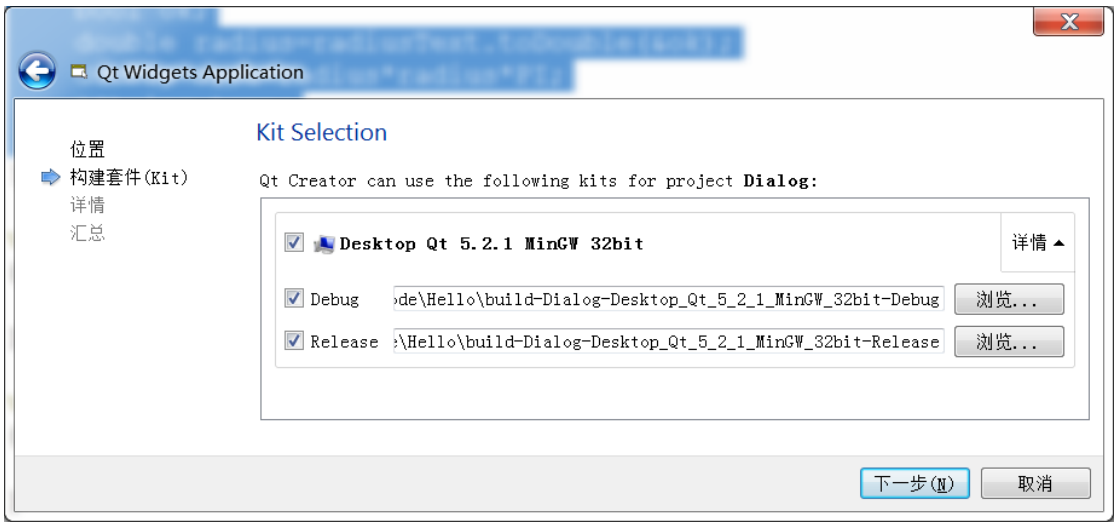


执行“文件”→“新建文件或项目”，左侧选择“应用程序”，中间选择“Qt Widgets Application”，点击“选择”按钮。



指定项目名称和项目存储路径。

点击“下一步”按钮，在弹出的对话框中，指定用于编译的套件，保持默认值即可。

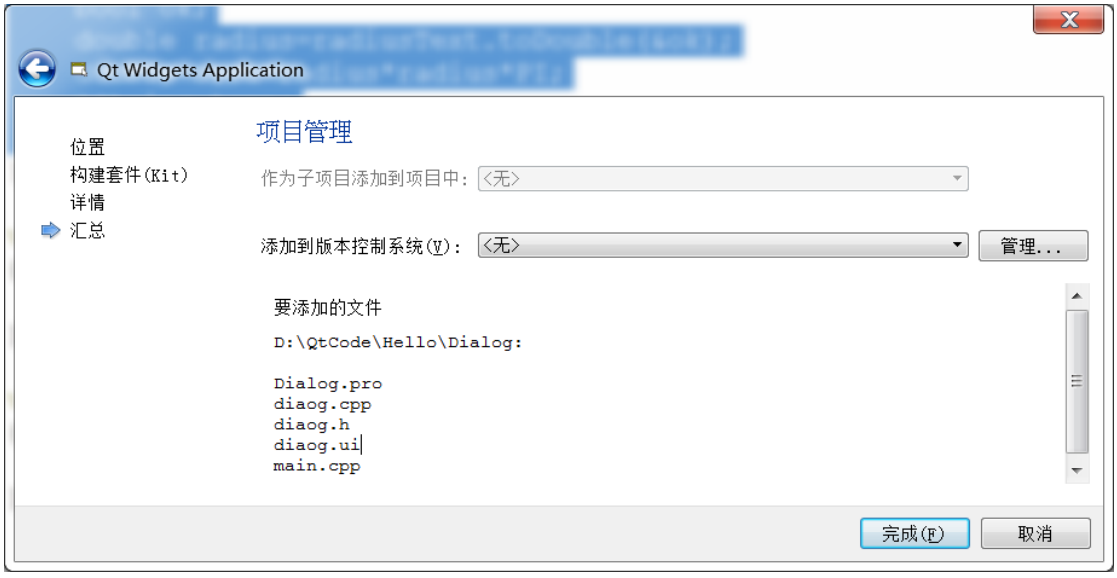


点击“下一步”，在弹出的对话框中，设置主界面类名称、基类以及对应头

文件和 cpp 文件的文件名称，此外选择是否创建界面文件。

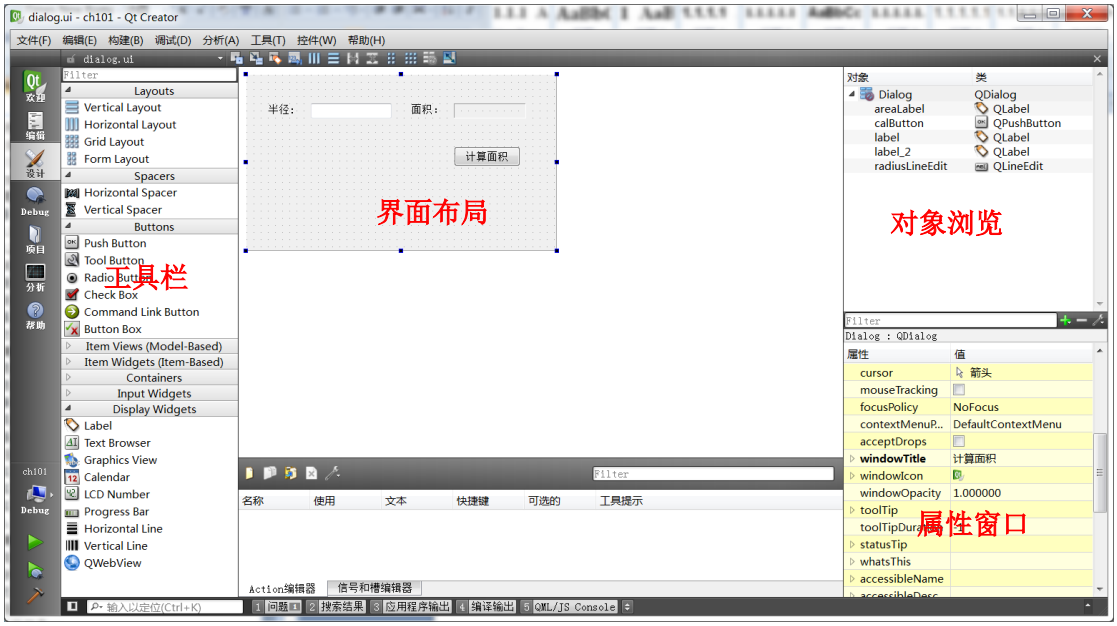
我们要创建的项目是一个基于对话框的简单应用，类名称设置为 Dialog，基类为 QDialog，类文件名保持默认。我们选择通过 QT 的界面设计器设计界面，由向导生成 ui 文件，并提供可视化的界面编辑工具。当然也可以通过纯写代码的方式创建界面。

点击“下一步”，查看项目的总体信息，点击“完成”按钮完成整个项目的创建。整个项目包括 main.cpp，配套的类文件，以及界面文件。



4.1.1.2 界面设计器简介

双击 dialog.ui，进入界面设计器界面，可以从工具栏中拖放控件到界面设计器中，由设计器生成后台的代码。



4.1.1.3 完成界面布局

根据上图的布局，拖放 3 个 Label、1 个 LineEdit 和 1 个 PushButton 控件到界面布局区， Label 用于显示信息，LineEdit 用于输入单行文本，PushButton 定义按钮。在属性窗口设置各控件的 name 及其它属性，便于后续代码能够有效访问。具体如下：

objectName 属性	text 属性（显示内容）	对应的类
radiusLabel	半径：	QLabel
radiusLineEdit		QLineEdit
areaLabel	面积：	QLabel
areaResultLabel		QLabel
calButton	计算面积	QPushButton

objectName 为控件的名称，代码将通过该名称访问所需要的控件；text 属性设置控件的显示内容。选择 areaResultLabel 控件，修改“frameShape”属性为“Panel”，修改“frameShadow”属性为“Sunken”。

另外点击界面布局的空白区，选中整个窗体，修改 wondowTitle 属性可改变窗口的标题栏内容，此处修改为“计算面积”。

运行程序，查看界面效果。

4.1.1.4 实现计算面积代码

1）处理按钮点击事件

选择“计算面积”按钮，右击选择“转到槽”命令，在弹出的对话框中选择“clicked()”，单击“确定”按钮，向导将在 dialog.cpp 文件中创建响应按钮点击事件的方法 on_calButton_clicked()，并在内部创建它们之间的映射，编写如下的代码。

```
const static double    PI=3.14156;
void Dialog::on_calButton_clicked(){
    QString radiusText=ui->radiusLineEdit->text();
    bool ok;
    double radius=radiusText.toDouble(&ok);
    double area=radius*radius*PI;
    QString temp;
    ui-> areaResultLabel->setText(temp.setNum(area));
}
```

代码分析：在界面编辑器中定义的控件，QT 将其封装在 Ui 命名空间下的 Dialog 类中，并在生成的 Dialog 类中定义其指针成员 ui，要访问界面设计器中定义的控件，需要通过“ui->控件名称”的形式访问。

通过 LineEdit 控件的 text 可以读取文本框框中文本，返回 QString 类型（封装了字符串的 QT 类），QString 类的 toDouble 可以将字符串转换为浮点数并返回，由传入的 ok（传指针）判断转换是否成功。QString 类的 setNum 方法可以将数值数据转换为字符串。Label 和 LineEdit 控件的 setText 方法可以重新设置控件的显示内容。

QString 是 QT 封装的字符串类，C++标准库通过 string 封装字符串操作，两者可以相互转化。

```
QString qstr;
string str;
str = qstr.toStdString();
qstr = QString::fromStdString(str);
```

2）处理文本框内容变化事件

选择 radiuLineEdit 文本框控件，右击选择“转到槽”命令，在弹出的对话框中选择“textChanged(Qstring)”，点击“确定”按钮后，向导将在 dialog.cpp 文件

中创建响应文本框内容变化事件的方法 on_radiusLineEdit_textChanged，并在内部创建它们之间的映射，编写如下的代码。

```
void Dialog::on_radiusLineEdit_textChanged(const QString &arg1){
    QString radiusText=ui->radiusLineEdit->text();
    bool ok;
    double radius=radiusText.toDouble(&ok);
    double area=radius*radius*PI;
    QString temp;
    ui-> areaResultLabel->setText(temp.setNum(area));
}
```

3）重构代码，减少冗余

在 dialog.h 声明 calculate 方法，在 dialog.cpp 文件中将重复的代码定义在 calculate 方法中，在 2 个事件处理代码中调用 calculate 方法，实现代码共用。

```
class Dialog : public QDialog{
    Q_OBJECT
public:
    explicit Dialog(QWidget *parent = 0);
    void calculate();
    ~Dialog();
private slots:
    void on_calButton_clicked();
    void on_radiusLineEdit_textChanged(const QString &arg1);
private:
    Ui::Dialog *ui;
};
void Dialog::calculate(){
    QString radiusText=ui->radiusLineEdit->text();
    bool ok;
    double radius=radiusText.toDouble(&ok);
    double area=radius*radius*PI;
    QString temp;
    ui-> areaResultLabel->setText(temp.setNum(area));
}
void Dialog::on_calButton_clicked(){
    calculate();
}
void Dialog::on_radiusLineEdit_textChanged(const QString &arg1){
    calculate();
}
```

4.1.2 基于代码的界面实现

4.1.2.1 创建项目

执行“文件”→“新建文件或项目”，通过向导创建项目，步骤和 4.1.1.1 相同，在“类详情”设置页面中，取消“创建界面”复选框。

在创建的项目中将不会出现 ui 文件，需要通过编写代码来创建所有界面元素，并通过信号与槽机制来绑定事件处理。

4.1.2.2 编写代码创建界面

1）定义界面元素（Dialog.h）

在 Dialog.h 中定义所有界面元素指针，后续代码中将通过这些指针访问界面元素，通常将它们放在 private 区域中。不同界面元素由不同的类封装，需要包含相应的头文件。

```
#ifndef DIALOG_H
#define DIALOG_H
#include <QDialog>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
class Dialog : public QDialog{
    Q_OBJECT
public:
```

```
    Dialog(QWidget *parent = 0);
    ~Dialog();
private:
    QLabel *radiusLabel, *areaLabel;
    QLineEdit *radiusLineEdit;
    QPushButton *calButton;
};
#endif // DIALOG_H
```

2）创建并布局控件（Dialog.cpp）

通常在父窗口的构造函数中创建所有界面元素控件，并通过布局管理器进行布局管理。创建界面元素使用 new 运算符动态创建，为了简化内存释放和管理，QT 封装了对这些界面元素的内存释放，不需要在析构函数中显示释放它们了。

```
#include "dialog.h"
#include <QGridLayout>
Dialog::Dialog(QWidget *parent)
    : QDialog(parent){
    radiusLabel=new QLabel(this);
    radiusLabel->setText(tr("半径："));
    radiusLineEdit=new QLineEdit(this);
    areaLabel=new QLabel(this);
    calButton=new QPushButton(this);
    calButton->setText(tr("计算面积"));
    QGridLayout *mainLayout=new QGridLayout(this);
    mainLayout->addWidget(radiusLabel,0,0);
    mainLayout->addWidget(radiusLineEdit,0,1);
    mainLayout->addWidget(areaLabel,1,0);
    mainLayout->addWidget(calButton,1,1);
    setLayout(mainLayout);
}
```

4.1.2.3 实现事件处理

1）添加槽声明（Dialog.h）

QT 通过信号和槽机制来处理事件，信号对应各种事件，如针对控件的操作（按钮被点击、文本框内容被修改等）、鼠标键盘操作、窗口事件的发生都会触发事件，槽对应某个类的处理方法，QT 通过 connect 将信号和槽绑定起来，一旦绑定完成，信号发生时，会自动调用对应的槽方法，从而建立事件响应的灵活方法。槽通常放在 private 区域。

```
class Dialog : public QDialog{
    Q_OBJECT
public:
    Dialog(QWidget *parent = 0);
    ~Dialog();
private:
    QLabel *radiusLabel, *areaLabel;
    QLineEdit *radiusLineEdit;
    QPushButton *calButton;
private slots:
    void showArea();
};
```

2）槽方法定义（Dialog.cpp）

```
const static double PI=3.14159;
void Dialog::showArea(){
    QString radiusText=radiusLineEdit->text();
    bool ok;
    double radius=radiusText.toDouble(&ok);
    double area=radius*radius*PI;
    QString temp;
    areaLabel->setText(temp.setNum(area));
}
```

3）建立信号和槽的绑定关系（Dialog.cpp）

```
Dialog::Dialog(QWidget *parent)
    : QDialog(parent){
    …
    connect(calButton,SIGNAL(clicked()),this,SLOT(showArea()));
}
```

connect 通常有 4 个参数，第 1 个参数为发出信号的对象，此处为界面中的按钮；第二个参数为发出信号的名称，不同的控件或窗体能够发出的信号不同，按钮控件常用的信号是 clicked()，表示被点击了；第 3 个参数为接收信号并执行处理的对象，此处的 this 表示 Dialog 窗体对象；第 4 个参数指定执行处理的方法名称。此语句表明，当 calButton 按钮被点击时，将会调用 Dialog 类对象的 showArea()方法执行处理。SIGNAL 和 SLOT 宏将返回方法名称字符串。

槽与信号机制是 QT 处理事件的基础机制，非常灵活，在实际应用中，可以将多个事件绑定到 1 个槽中，也可以将 1 个事件绑定到多个槽中。

4）增加槽绑定，响应文本框内容修改事件（Dialog.cpp）

```
Dialog::Dialog(QWidget *parent)
```

```
    : QDialog(parent){
    …
    connect(calButton,SIGNAL(clicked()),this,SLOT(showArea()));
    connect(radiusLineEdit,SIGNAL(textChanged(QString)),
        this,SLOT(showArea()));
}
```

4.2 基础界面元素的实现

4.2.1 主窗体程序

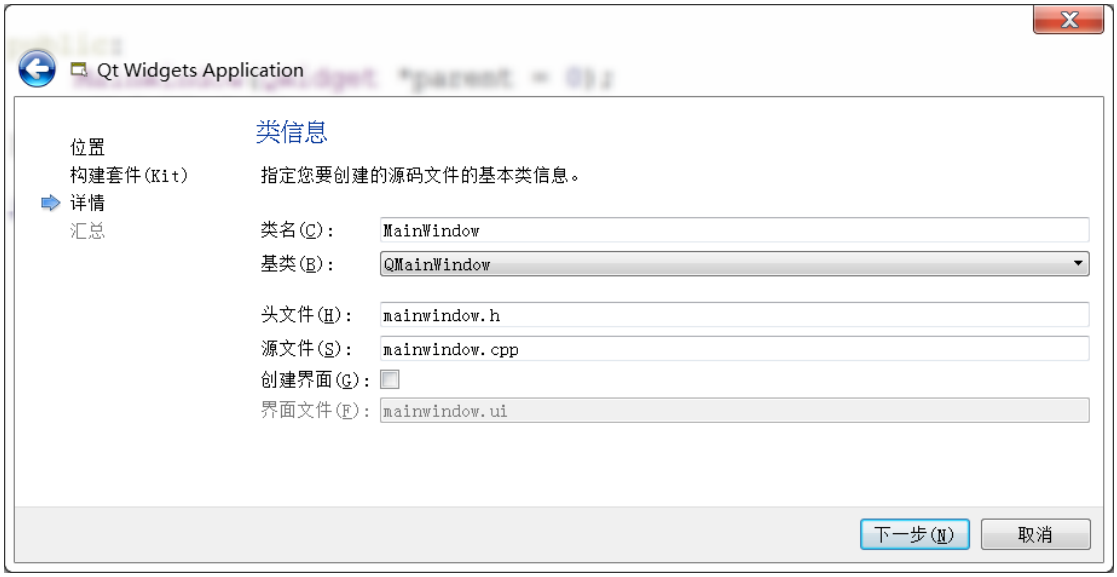
QT 通过 QMainWindow 封装了主窗口程序的基础功能，主窗口程序包括 1 个菜单栏、工具栏（可以多个）、停靠部件（可以多个）、1 个状态栏和 1 个中心部件，是窗口界面程序的基础。

QMainWindow 具有自己的布局管理器，无需为其再设置窗口布局管理器，但可以为其中心部件设置部件管理器。

主窗口程序的中心部件通常派生自 QWidget 类，主要用户实现用户可见的窗口和交互区域。

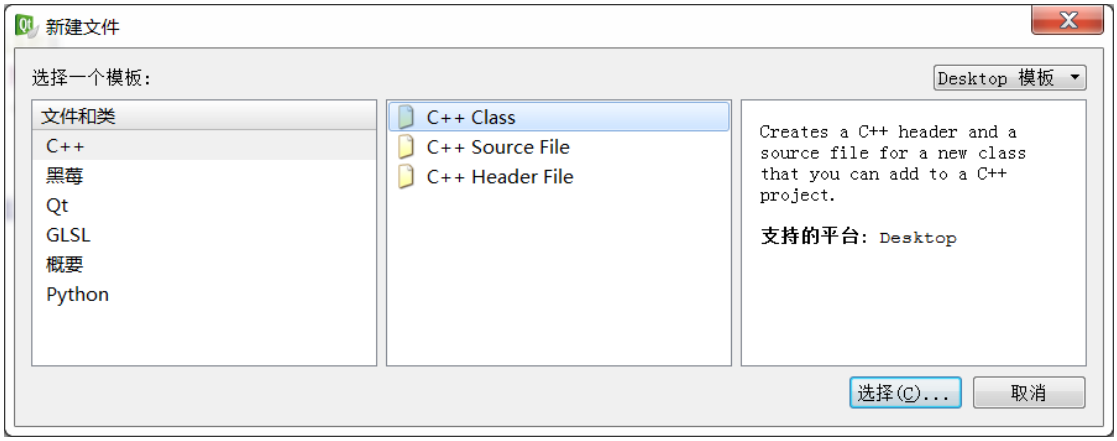
4.2.1.1 创建项目

执行“文件”→“新建文件或项目”，通过向导创建项目，步骤和 4.1.1.1 相同，在“类详情”设置页面中，取消“创建界面”复选框，选择基类为 QMainWindow。

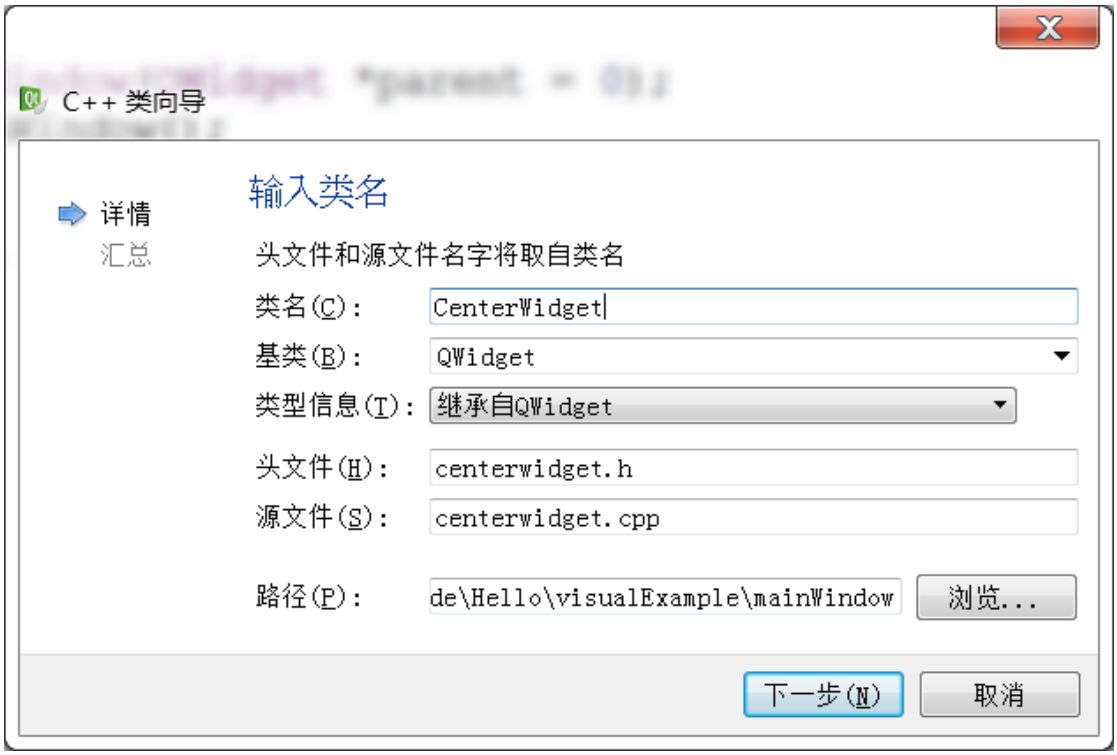


4.2.1.2 创建中心部件

右键点击当前项目，执行“添加新文件”，在弹出的对话框中，左侧选择“C++”，中间选择“C++ class”。



点击“选择”，在弹出对话框中，输入类名称为“CenterWidget”，选择基类为 QWidget，其它选项保持不变。



4.2.1.3 将中心部件添加到主程序中

1）在主程序类中定义中心部件（MainWindow.h）

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
```

```
#include <QMainWindow>
#include "centerwidget.h"
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
};
#endif // MAINWINDOW_H
```

2）创建中心部件（MainWindow.cpp）

```
#include "mainwindow.h"
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单窗口程序"));
    centerWidget=new CenterWidget(this);
    setCentralWidget(centerWidget);
}
MainWindow::~MainWindow(){
}
```

4.2.1.4 在中心部件输出简单内容

1）声明 paintEvent 虚函数（CenterWidget.h）

```
#ifndef CENTERWIDGET_H
#define CENTERWIDGET_H
#include <QWidget>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
signals:
public slots:
};
#endif // CENTERWIDGET_H
```

2）实现窗体输出（CenterWidget.cpp）

```
#include "centerwidget.h"
#include <QPainter>
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
```

```
        setMinimumSize(400,400);
    }
    void CenterWidget::paintEvent(QPaintEvent *){
        QPainter p(this);
        p.drawText(QRect(50,50,300,100),Qt::AlignCenter,
                        tr("欢迎进入 QT 可视化编程的世界！"));
    }
```

setMinimumSize 方法设置中心部件的最小尺寸，从而确定了 MainWindow 的初始大小及最小尺寸。QPainter 封装了窗口绘制与输出环境，调用 drawText 可以实现文本的输出，此外还包含了大量的输出图形的方法，还封装了设置画笔、字体、颜色等绘图环境的方法。

4.2.1.5 调整主窗体的大小

```
    #include <QDesktopWidget>
    #include <QApplication>
    MainWindow::MainWindow(QWidget *parent)
        : QMainWindow(parent){
        setWindowTitle(tr("简单窗口程序"));
        centerWidget=new CenterWidget(this);
        setCentralWidget(centerWidget);
        QDesktopWidget* desktop = QApplication::desktop();
        resize(desktop->width()/2,desktop->height()/2);
        move((desktop->width() - this->width())/2,
           (desktop->height() - this->height())/2);
    }
```

QApplication 封装了应用程序行为，其静态方法 desktop()可以获得桌面窗口组件，通过其 width()和 height()方法可以获得桌面大小。此处通过 resize 方法和 move 方法将主程序窗口调整为居于屏幕中央且大小为桌面的一半。

4.2.2 简单绘图

QWidget 及其派生类通过 paintEvent 虚函数实现窗口绘制。QT 中 QPainter 封装了绘图环境，可以实现文本、图形、图像等内容的绘制输出。在 QWidgit 类的 paintEvent 方法中可以实现“持久”绘制，屏幕刷新后会自动调用该方法重绘，当我们需要主动刷新窗口时，可以调用 update 实现窗口的刷新。

4.2.2.1 绘制基本图形

在 4.2.1 项目基础之上，在 CenterWidget 的 paintEvent 方法中根据设置的线条、颜色绘制椭圆。

```
void CenterWidget::paintEvent(QPaintEvent *){
    QPainter p(this);
    p.drawText(QRect(50,50,300,100),Qt::AlignCenter,
                tr("欢迎进入 QT 可视化编程的世界！"));
    QRect rect(200,200,300,200);
    QPen pen(Qt::red,2,Qt::SolidLine);
    p.setPen(pen);
    QBrush brush(Qt::blue);
    p.setBrush(brush);
    p.drawEllipse(rect);
}
```

代码分析：QRect 封装了矩形几何对象，前 2 个参数定义矩形左上角 x、y 坐标，后 2 个参数定义矩形长宽。矩形的边必须和坐标轴平行，特殊的矩形需要通过多边形定义（QPolygon）。绘图图形时，通过画笔绘制形状的轮廓，通过画刷填充封闭区域。默认画笔为黑色、1 个像素的实线，QPen 封装了画笔的属性，可以创建指定颜色、线宽（像素数）和线型的画笔，通过 setPen 方法将画笔选入画图设备环境。默认画刷为无填充（透明），QBrush 封装了画刷的属性，可以创建指定颜色和图案的画刷，通过 setBrush 方法将画刷选入画图设备环境。

线型通过枚举类型定义，常见的有：Qt::SolidLine（实线）、Qt::DashLine（短划线）、Qt::DotLine（点虚线）、Qt::DashDotLine（短划点虚线）等。

画刷填充方案也是通过枚举类型定义，常见的有：Qt::SolidPattern（实心均匀填充）、Qt::NoBrush（不填充）、Qt::HorPattern（水平填充）等，参照文档在程序中多尝试，以便理解其含义。

QPainter 的 drawEllipse 绘制椭圆，以 rect 为其外接矩形。

4.2.2.2 绘制文字

```
void CenterWidget::paintEvent(QPaintEvent *){
    QPainter p(this);
    QFont font(tr("宋体"),18,QFont::Bold,true);
    p.setFont(font);
    p.setPen(QPen(QColor(125,56,0)));
```

```
        p.drawText(QRect(50,50,300,100),Qt::AlignCenter,
                   tr("欢迎进入 QT 可视化编程的世界！"));
        QRect rect(200,200,300,200);
        QPen pen(Qt::red,2,Qt::SolidLine);
        p.setPen(pen);
        QBrush brush(Qt::blue);
        p.setBrush(brush);
        p.drawEllipse(rect);
    }
```

代码解析：绘制文字需要分别指定字体和颜色，绘制图形时画笔包含了颜色和线型等属性。QFont 封装了字体属性，创建字体对象时需要指定字体名称、字号、粗细程度和是否斜体，通过 setFont 方法将字体选入绘图设备环境。指定颜色时除了使用预设颜色枚举常量外（如 Qt::blue），还可以通过指定 rgb 分量直接创建一个 QColor 颜色对象。

4.2.2.3 绘制图像

1）在 CenterWidget 中定义 QImage 成员，保存要绘制的图像对象。

```
#ifndef CENTERWIDGET_H
#define CENTERWIDGET_H
#include <QWidget>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
signals:
public slots:
private:
    QPixmap pixmap;
};
#endif // CENTERWIDGET_H
```

2）在 CenterWidget 构造函数中，加载图像文件。

```
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    pixmap.load("flower.jpg");
    setMinimumSize(400,400);
}
```

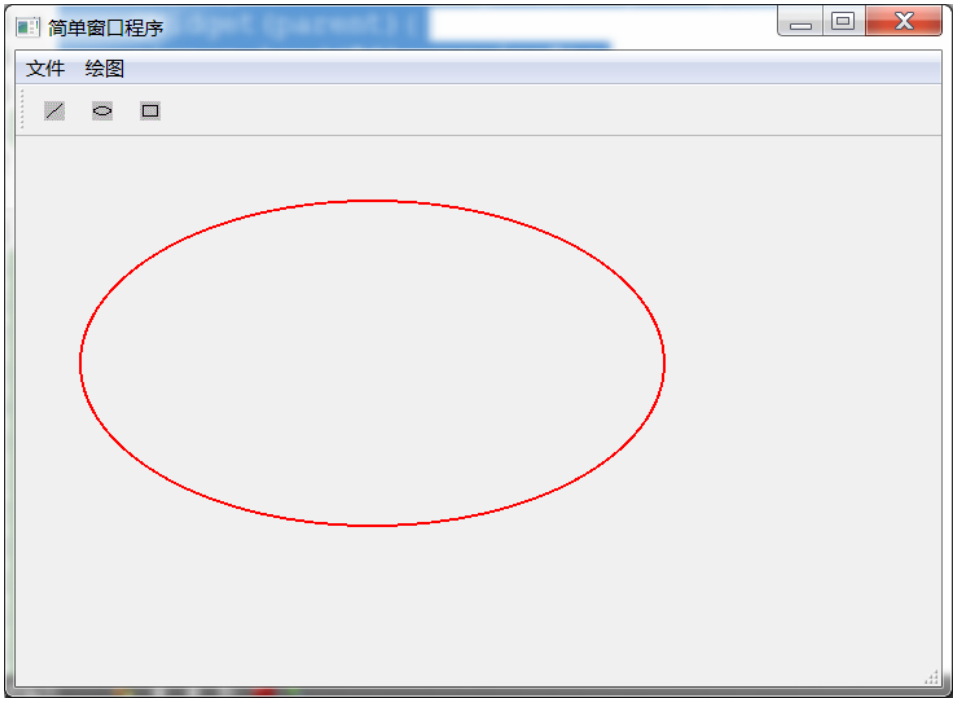
3）在 paintEvent 中输出图像

```
void CenterWidget::paintEvent(QPaintEvent *){
```

```
        QPainter p(this);
        p.drawPixmap(600,50,pixmap);
        QFont font(tr("宋体"),18,QFont::Bold,true);
        p.setFont(font);
        p.setPen(QPen(QColor(125,56,0)));
        p.drawText(QRect(50,50,300,100),Qt::AlignCenter,
                        tr("欢迎进入 QT 可视化编程的世界！"));
        QRect rect(200,200,300,200);
        QPen pen(Qt::red,2,Qt::SolidLine);
        p.setPen(pen);
        QBrush brush(Qt::blue);
        p.setBrush(brush);
        p.drawEllipse(rect);
}
```

4.2.3 菜单、工具栏等

在 4.2.2 的基础之上，为系统添加菜单组件，共有 2 个菜单，File 菜单下有 Exit 菜单项（退出系统），Draw 菜单下有 Line、Rectangle、Ellipse 菜单项，点击 Line、Rectangle、Ellipse 后在窗口组件中绘制线条、矩形、椭圆。为菜单项配备快捷键、加速键、图标等内容。



1）在 MainWindow.h 中定义菜单和工具栏对象

```
#include <QMainWindow>
#include <QMenu>
```

```
#include <QAction>
#include <QToolBar>
#include "centerwidget.h"
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
    QMenu *fileMenu;
    QMenu *drawMenu;
    QAction *exitAction;
    QAction *lineAction;
    QAction *ellipseAction;
    QAction *rectangleAction;
    QToolBar *drawToolBar;
};
```

代码解析：QT 中的菜单封装在 QMenu 类中，工具栏封装在 QToolBar 类中，状态栏封装在 QStatusBar 类中。QMenu 是放置菜单项的容器，QToolBar 中放置按钮等控件，菜单项和工具栏按钮往往绑定一个 QAction 对象，由该对象定义相应的快捷键、显示图标、显示文本、状态栏提示等，绑定触发事件等。

2）在 MainWindow 构造函数中创建菜单

```
#include <QStatusBar>
#include <QMenuBar>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单窗口程序"));
    centerWidget=new CenterWidget(this);
    setCentralWidget(centerWidget);
    QDesktopWidget* desktop = QApplication::desktop();
    resize(desktop->width()/2,desktop->height()/2);
    move((desktop->width() - this->width())/2,
        (desktop->height() - this->height())/2);

    fileMenu=menuBar()->addMenu(tr("文件"));
    exitAction=new QAction(tr("退出"),this);
    exitAction->setShortcut(tr("Ctrl+X"));
    exitAction->setStatusTip(tr("退出程序"));
    fileMenu->addAction(exitAction);
```

```
    lineAction=new QAction(QIcon("line.gif"),tr("线条"),this);
    lineAction->setShortcut(tr("Ctrl+L"));
    lineAction->setStatusTip(tr("绘制线条"));
    ellipseAction=new QAction(QIcon("ellipse.gif"),tr("椭圆"),this);
    ellipseAction->setShortcut(tr("Ctrl+E"));
    ellipseAction->setStatusTip(tr("绘制椭圆"));
    rectangleAction=new QAction(QIcon("rectangle.gif"),tr("矩形"),this);
    rectangleAction->setShortcut(tr("Ctrl+R"));
    rectangleAction->setStatusTip(tr("绘制矩形"));
    drawMenu=this->menuBar()->addMenu(tr("绘图"));
    drawMenu->addAction(lineAction);
    drawMenu->addAction(ellipseAction);
    drawMenu->addAction(rectangleAction);
```

```
    drawToolBar=addToolBar(tr("Draw"));
    drawToolBar->addAction(lineAction);
    drawToolBar->addAction(ellipseAction);
    drawToolBar->addAction(rectangleAction);
```

```
    statusBar()->show();
}
```

代码解析：QMainWindow 类（基类）中的 menuBar()可获得主窗口的菜单栏指针（顶层菜单栏），通过 QMenuBar 的 addMenu()可以添加下一级菜单栏。创建 QAction 后，通过 QMenu 的 addAction()方法，可以将其作为菜单项添加到菜单栏中，通过 QToolBar 的 addAction()方法可以将其添加到工具栏中。通过 QToolBar 的 setAllowedAreas 可以设置工具栏的停靠方式，大家可以通过查看帮助文档获得相应的设置方法。

QAction 创建时可以通过构造函数指定菜单项或工具栏按钮的显示图标、显示文本等内容，通过 setShortcut()方法可以设置快捷键，通过 setStatusTip 方法可以设置状态栏提示文本。

默认情况下，系统不显示状态栏。通过 QmainWindow 的 statusBar 方法可以获取指向状态栏的指针，从而可以操作状态栏。此处调用其 show 方法，显示状态栏。

到此为止，我们只是创建了菜单和工具栏，并没有定义菜单的事件处理代码，点击菜单和工具栏按钮后并没有响应。

3）在 MainWindow 类中定义处理菜单事件的槽

```
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
    QMenu *fileMenu;
    QMenu *drawMenu;
    QAction *exitAction;
    QAction *lineAction;
    QAction *ellipseAction;
    QAction *rectangleAction;
    QToolBar *drawToolBar;
protected slots:
    void line();
    void ellipse();
    void rectangle();
};
```

菜单项和工具栏按钮的点击响应也是通过槽与信号机制来实现的，菜单项和工具栏按钮点击后会触发 triggered()的事件，需要在 QMainWindow 中定义槽（事件对应的处理方法），在构造函数中通过 connect 将其绑定起来。

3）定义槽函数，在 MainWindow 的构造函数中绑定信号和槽

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    …
    statusBar()->show();
    connect(exitAction,SIGNAL(triggered()),this,SLOT(close()));
    connect(lineAction,SIGNAL(triggered()),this,SLOT(line()));
    connect(ellipseAction,SIGNAL(triggered()),this,SLOT(ellipse()));
    connect(rectangleAction,SIGNAL(triggered()),this,SLOT(rectangle()));
}
void MainWindow::line(){
}
void MainWindow::ellipse(){
}
void MainWindow::rectangle(){
}
```

line、ellipse 和 rectangle 是我们定义的响应菜单项点击事件的槽函数，close 是 QMainWindow 类中预定义的槽函数，用于结束应用程序。

4）在 centerWidget.h 中定义标识绘图类型的变量及访问方法

```
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void setDrawType(int type);
signals:
public slots:
private:
    QPixmap pixmap;
    int drawType;
};
```

drawType 用于标识用户选择的绘图类型，0 表式线条，1 表式椭圆，2 表式矩形，在构造函数中初始化为 0。由 setDrawType 接口设置其值。

4）在 centerWidget.cpp 中实现 setDrawType 接口

```
void CenterWidget::setDrawType(int type){
    drawType=type;
}
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    pixmap.load("flower.jpg");
    setMinimumSize(400,400);
    drawType=0;
}
```

5）根据 drawType 绘制不同的图形

```
void CenterWidget::paintEvent(QPaintEvent *){
    QPainter p(this);
    QPen pen(Qt::red,2,Qt::SolidLine);
    p.setPen(pen);
    QPoint p1(50,50),p2(500,300);
    switch(drawType){
    case 0:
        p.drawLine(p1,p2); break;
    case 1:
        p.drawEllipse(QRect(p1,p2)); break;
    case 2:
        p.drawRect(QRect(p1,p2)); break;
    }
    p.drawPixmap(600,50,pixmap);
    QFont font(tr("宋体"),18,QFont::Bold,true);
    p.setFont(font);
```

```
    p.setPen(QPen(QColor(125,56,0)));
    p.drawText(QRect(50,50,500,100),Qt::AlignCenter,
                    tr("欢迎进入 QT 可视化编程的世界！"));
    QRect rect(200,200,300,200);
    QPen pen(Qt::red,2,Qt::SolidLine);
    p.setPen(pen);
    QBrush brush(Qt::blue);
    p.setBrush(brush);
    p.drawEllipse(rect);
}
```

6）实现 MainWindow 中菜单响应槽函数

```
void MainWindow::line(){
    centerWidget->setDrawType(0);
    centerWidget->update();
}
void MainWindow::ellipse(){
    centerWidget->setDrawType(1);
    centerWidget->update();
}
void MainWindow::rectangle(){
    centerWidget->setDrawType(2);
    centerWidget->update();
}
```

根据用户点击的菜单项，将绘图类型通过 setDrawType 接口传给 centerWidget 对象，调用其 update 方法刷新窗口显示。

4.2.4 鼠标与键盘事件

QMainWindow 类定义虚函数 mousePressEvent()、mouseMoveEvent()、mouseReleaseEvent()、mouseDoubleClickEvent()方法用于处理鼠标的点击、移动、释放、双击事件。在我们自定义的窗口类中要重新定义这些事件的处理，需要覆盖对应的方法。

类似与鼠标事件处理，键盘事件对应的虚函数为 keyPressEvent()，覆盖该方法可以重新对键盘事件的响应。

在 4.2.3 示例程序的基础之上，处理鼠标点击、移动事件，鼠标点击时在窗口显示点击的位置，鼠标移动时在窗口显示当前鼠标位置。

1）定义鼠标事件处理函数（CenterWidget.h）

```
#include <QMouseEvent>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void setDrawType(int type);
protected:
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
signals:
public slots:
private:
    QPixmap pixmap;
    int drawType;
    QString mouseClickInfo;   //保存点击信息的字符串
    QString mousePosInfo;     //保存鼠标位置的字符串
};
```

2）鼠标点击事件处理（CenterWidget.cpp）

```
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    pixmap.load("flower.jpg");
    setMinimumSize(400,400);
    drawType=0;
    mouseClickInfo=tr("");
    mousePosInfo=tr("");
    setMouseTracking(true);
}
```

在构造函数中，初始化要显示的字符串为空串。定义鼠标事件的处理函数后，需要在窗体类的构造函数中调用 setMouseTracking(true)才能启动对鼠标事件的捕获。

```
void CenterWidget::mousePressEvent(QMouseEvent *e){
    mouseClickInfo=tr("Mouse Click at: ")+QString::number(e->x())
            +","+QString::number(e->y());
    update();
}

void CenterWidget::mouseMoveEvent(QMouseEvent *e){
    mousePosInfo=tr("Mouse pos: ")+QString::number(e->x())
            +","+QString::number(e->y());
    update();
}
```

QMouseEvent 封装了鼠标事件发生时的很多状态信息，包括鼠标位置、按键情况等，通过调用其 x()和 y()方法，可以获得事件发生时相对于当前窗口的坐标位置，通过 QString 类的 number 静态方法可以将整数值转换为字符串，将要显示的内容保存在 mouseClickInfo 和 mousePosInfo 字符串中，最后通过 update 刷新窗口显示，在 paintEvent 方法中输出 mouseClickInfo 和 mousePosInfo 的内容。

```
void CenterWidget::paintEvent(QPaintEvent *){
    QPainter p(this);
    QPen pen(Qt::red,2,Qt::SolidLine);
    p.setPen(pen);
    QPoint p1(50,50),p2(500,300);
    switch(drawType){
    case 0:
        p.drawLine(p1,p2); break;
    case 1:
        p.drawEllipse(QRect(p1,p2)); break;
    case 2:
        p.drawRect(QRect(p1,p2)); break;
    }
    p.drawText(550,200,mouseClickInfo);
    p.drawText(550,300,mousePosInfo);
}
```

QPainter 的 drawText 方法在窗口指定位置输出字符串。大家要理解这种异步处理的方法，在鼠标事件中生成字符串，在 paintEvent 方法中异步显示字符串内容。

4.2.5 在状态栏中显示鼠标当前位置

在 4.2.4 的基础之上，在状态栏中添加标签控件，用以显示鼠标的当前坐标位置。

1）添加显示信息的标签控件（CenterWidget.h）

```
#include <QMouseEvent>
#include <QLabel>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void setDrawType(int type);
```

```
protected:
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
signals:
public slots:
private:
    QPixmap pixmap;
    int drawType;
    Qlabel *mousePosLabel;     //显示鼠标位置的控件
    QString mouseClickInfo;   //保存点击信息的字符串
    QString mousePosInfo;      //保存鼠标位置的字符串
};
```

2）在构造函数中创建标签控件（CenterWidget.cpp）

```
#include <QStatusBar>
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    pixmap.load("flower.jpg");
    setMinimumSize(400,400);
    drawType=0;
    mouseClickInfo=tr("");
    mousePosInfo=tr("");
    mousePosLabel=new QLabel;
    mousePosLabel->setText(“”);       //初始显示内容为空
    mousePosLabel->setFixedWidth(100);       //控件设置为固定大小
    MainWindow *p=(MainWindow *)parent;
    p->statusBar()->addPermanentWidget (mousePosLabel);
    setMouseTracking(true);
}
```

3）鼠标移动事件处理（CenterWidget.cpp）

```
void CenterWidget::mouseMoveEvent(QMouseEvent *e){
    mousePosInfo=tr("Mouse pos: ")+QString::number(e->x())
            +","+QString::number(e->y());
    mousePosLabel->setText(mousePosInfo);      //直接更新状态栏中的控件
    update();      //通过刷新窗口，在 paintEvent 中更新窗口中的显示内容
}
```

4.3 实习任务

4.3.1 实习任务一

研究 keyPressEvent 方法及其 QKeyEvent 参数，响应用户按键操作，在窗体指定位置处显示用户按键的键值。同时将按键的键值显示在状态栏中。

4.3.2 实习任务二

基于 4.2.3 的示例，添加 Color 菜单，添加 Black、Green、Yellow 菜单项，默认选中 Black 菜单项，切换后以选中的颜色重新绘制图形。添加分隔符，再添加菜单项 Fill，点击后切换是否填充图形，默认为不填充。

【提示】需要在组件窗口中定义颜色和是否填充的标识变量，paientEvent 方法根据标识变量进行绘制，用户点击菜单后改变标识变量的值，再刷新窗口显示，从而更新显示内容。

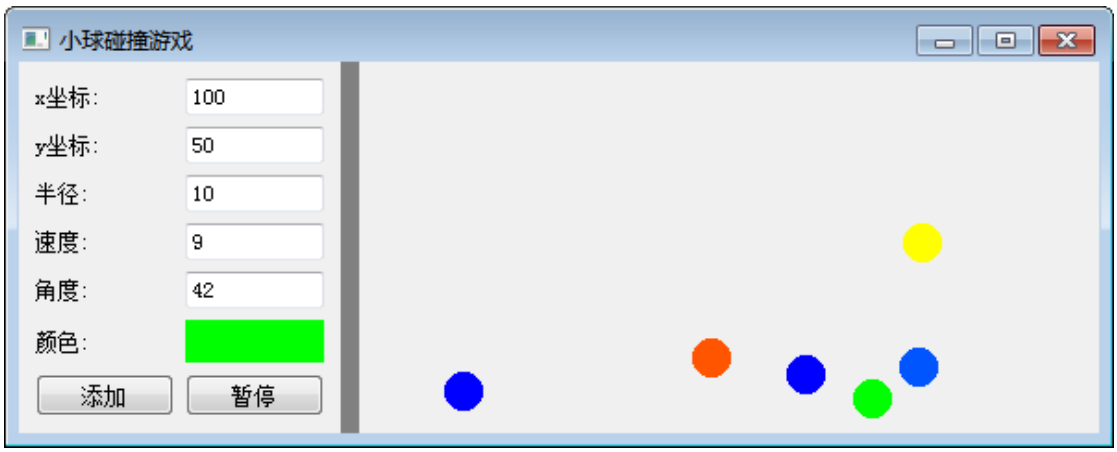
【提示】QAction 的 setCheckable(bool)方法设置是否启用对应菜单项或工具栏按钮的选中状态，在启用的情况下，setChecked(bool)方法可以设置对应菜单项或工具栏按钮的选中状态。

【提示】QMenu 和 QToolBar 的 addSeparator()方法可以在菜单项或工具栏按钮之间增加分割符效果。

5　综合编程：小球碰撞模拟

5.1　演示案例

【任务目标】实现如下图所示的小球碰撞程序。在一个矩形区域内，模拟多个小球的移动和碰撞（小球和边框、小球之间），整个视图切分为左右 2 个部分。左侧视图显示当前待发射小球的参数（位置、角度、速度和颜色），颜色由用户指定，点击“添加”按钮后添加待发射的小球，点击“启动”按钮后右侧视图内小球开始移动，点击“暂停”按钮暂停小球移动。小球与边框、小球之间会产生弹性碰撞效果。



【物理模型】小球移动过程中与边框、小球之间发生碰撞时，应遵循一定的物理模型，才能产生真实感效果。本例为简化程序，假定小球移动过程中没有摩擦，小球与边框、小球之间的碰撞是完全弹性碰撞（没有能量损失），小球与边框碰撞为简单的反射过程（只改变角度），小球碰撞后仅仅交换速度和方向。

5.1.1　构建程序框架

1) 创建项目，项目名称为 BallGame，类型选择“Qt Widgets Application”。在“类详情”设置页面中，取消“创建界面”复选框，选择基类为 QMainWindow。添加新类“LeftWidget”，选择基类为 QWidget，添加新类“RightWidget”，选择基类为 QWidget。

2) 在主窗口类中添加 LeftWidget 和 RightWidget 对象，通过 QSplitter 类构建切分视图框架（MainWindow.h）

```
#include <QMainWindow>
#include <QSplitter>
```

```
#include "leftwidget.h"
#include "rightwidget.h"
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    LeftWidget *left;
    RightWidget *right;
    QSplitter *splitter;
};
```

3) 构造切分窗体（MainWindow.cpp）

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle("小球碰撞游戏");
    splitter=new QSplitter(Qt::Horizontal,this);   //水平切分
    left=new LeftWidget(this);
    right=new RightWidget(this);
    splitter->addWidget(left);
    splitter->addWidget(right);
    splitter->setStretchFactor(0,1);   //设置切分的默认比例，1:4
    splitter->setStretchFactor(1,4);
    splitter->setStyleSheet("QSplitter::handle { background-color: gray }");
    splitter->setHandleWidth(10);   //设置分割条的大小和样式
    this->setCentralWidget(splitter);
}
```

【代码分析】在 MainWindow 中构建 splitter，再构建 left 和 right，将 left 和 right 添加到 splitter 中，最后将 splitter 添加到主窗口中。

5.1.2 1 个小球的模拟运动

1) 添加 Ball 类。对应头文件 ball.h 和 ball.cpp。

Ball 类的初步设计，每个小球具备的属性应包括中心坐标、半径、颜色、移动速度、移动方向，此外为了方便检测小球与窗体边界的碰撞检测，在 Ball 类中增加 1 个表示边界的 QRect 对象。

2) Ball 类初步定义（ball.h）

```
#ifndef BALL_H
#define BALL_H
```

```
#include <QColor>
#include <QRect>
#include <QPainter>
class Ball
{
public:
    Ball();
    Ball(double xpos,double ypos,double r,double s,double a,
         QColor c);
    void setRectangle(const QRect& rect){
        this->rect=rect;
    }
    void draw(QPainter *p);   //绘制小球
private:
    double x,y;   //小球的中心为位置
    double radius;   //小球半径
    double speed;   //小球移动速度
    double angle;   //小球移动角度
    QColor color;    //小球填充颜色
    QRect rect;      //用于检测的窗口矩形区域
};
#endif // BALL_H
```

3) Ball 类成员函数的初步定义（ball.cpp）

```
#include <cmath>
#include <QPen>
#include <QBrush>
Ball::Ball():Ball(40,100,10,10,45,Qt::red)
{    //通过委托构造实现缺省构造函数
}
Ball::Ball(double xpos,double ypos,double r,double s,double a,QColor c)
    :x(xpos),y(ypos),radius(r),speed(s),angle(a),color(c)
{
}
void Ball::draw(QPainter *p)
{
    QPen pen(color,1,Qt::SolidLine);
    QBrush brush(color);
    p->setPen(pen);
    p->setBrush(brush);
    QRect r(x-radius,y-radius,radius*2,radius*2);
    p->drawEllipse(r);
}
```

4) 在 right 窗体中定义定义 1 个小球（RightWidget.h）

```
#ifndef RIGHTWIDGET_H
#define RIGHTWIDGET_H
#include "ball.h"
#include <QWidget>
class RightWidget : public QWidget
{
    Q_OBJECT
public:
    explicit RightWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
private:
    Ball ball;
signals:
public slots:
private:
};
#endif // RIGHTWIDGET_H
```

paintEvent 方法是基类的虚函数，重载后用于更新窗口显示。

5) 在 right 窗体中显示小球（RightWidget.cpp）

```
#include "rightwidget.h"
#include <QPainter>
RightWidget::RightWidget(QWidget *parent) :
    QWidget(parent)
{
    setMinimumSize(400,200);
}
void RightWidget::paintEvent(QPaintEvent *)
{
    QPainter p(this);
    ball.draw(&p);
}
```

到此为止，可以在右侧窗口中显示 1 个位置固定的小球，下面将通过定时器实现小球的移动效果。

6) 定义小球移动接口（ball.h）

```
class Ball
{
public:
    Ball(double xpos,double ypos,double r,double s,double a,
         QColor c);
    Ball();
    void setRectangle(const QRect& rect){
        this->rect=rect;
```

```
    }
    void draw(QPainter *p);   //绘制小球
    void move();     //在指定的方向上移动 1 步
private:
    …
};
```

7) 小球移动方法的实现（ball.cpp）

```
void Ball::move(){
    double dx,dy;
    const double PI=3.14159;
    dx=speed*std::sin(angle*PI/180);
    dy=speed*std::cos(angle*PI/180);
    x+=dx;
    y+=dy;
}
```

8) 封装小球移动接口（RightWidget.h）

```
class RightWidget : public QWidget
{
    Q_OBJECT
public:
    explicit RightWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void updateBalls();   //移动小球的接口
private:
    Ball ball;
signals:
public slots:
private:
};
```

9) 封装小球移动接口的实现（RightWidget.cpp）

```
void RightWidget::updateBalls()   //封装的控制小球移动的接口
{
    ball.move();
    update();       //更新窗口显示，重绘小球
}
```

10) 主窗口类中定义 Timer 对象（MainWindow.h）

```
#include <QMainWindow>
#include <QSplitter>
#include <QTimer>
#include "leftwidget.h"
#include "rightwidget.h"
```

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
    void StopTimer();   //停止定时器的接口
    void resumeTimer();   //重启定时器的接口
protected slots:
    void timeToShow();      //定时器事件对应的槽方法
private:
    LeftWidget *left;
    RightWidget *right;
    QSplitter *splitter;
    QTimer *timer;      //定时器对象
};
```

11) 启动定时器并控制小球移动（MainWindow.cpp）

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    …
    this->setCentralWidget(splitter);
    timer=new QTimer(this);
    connect(timer,SIGNAL(timeout()),this,SLOT(timeToShow()));
    timer->start(100);
}
void MainWindow::timeToShow(){
    right->updateBalls();
}
void MainWindow::StopTimer(){
    timer->stop();
}
void MainWindow::resumeTimer(){
    timer->start(100);
}
```

到此为止，通过定时器事件，可以控制小球沿指定的角度和速度移动，遇到边界后会移出边界，下面将实现小球碰到边界后的反弹效果。如果碰撞时完全弹性碰撞，则小球碰到边界后只是改变移动角度。

12) 定义小球与边界碰撞检测方法（ball.h）

```
class Ball
{
    QRect rect;
```

```
private:
    …
    void checkBoundary();
};
```

13) 小球与边界碰撞方法的实现（ball.cpp）

```
void Ball::move(){
    double dx,dy;
    const double PI=3.14159;
    dx=speed*std::sin(angle*PI/180);
    dy=speed*std::cos(angle*PI/180);
    x+=dx;
    y+=dy;
    checkBoundary();
}
void Ball::checkBoundary(){
    if(y+radius>rect.height()){
        angle=180-angle;
        y=rect.height()-radius;
    }
    if(y-radius<0){
        angle=180-angle;
        y=radius;
    }
    if(x+radius>rect.width()){
        angle=-angle;
        x=rect.width()-radius;
    }
    if(x-radius<0){
        angle=-angle;
        x=radius;
    }
}
```

14) 设置小球的检测边界（RightWidget.cpp）

```
void RightWidget::updateBalls()   //封装的控制小球移动的接口
{
    ball.setRectangle(this->geometry());   //为小球设置检测的窗口边界
    ball.move();
    update();       //更新窗口显示，重绘小球
}
```

至此，完成 1 个小球的移动，能够处理小球与边框的碰撞处理。下面将实现多个小球的模拟及其之间的碰撞处理。

5.1.3 多个小球的模拟运动

15) 定义存储多个小球的数据结构（RightWidget.h）

```
#include <QList>
class RightWidget : public QWidget
{
    Q_OBJECT
public:
    explicit RightWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void updateBalls();
    void addBall(const Ball& b);   //添加小球的接口
private:
    Ball ball;
    QList<Ball> balls;
signals:

public slots:
private:
};
```

16) 多个小球的构造及管理（RightWidget.cpp）

```
RightWidget::RightWidget(QWidget *parent) :
    QWidget(parent)
{
    setMinimumSize(400,200);
    balls.clear();   //清空后手工添加 3 个小球，用于测试
    addBall(Ball(30,20,10,10,49,Qt::red));
    addBall (Ball(56,200,10,4,39,Qt::blue));
    addBall (Ball(80,100,10,7,69,Qt::yellow));
}
void RightWidget::addBall(const Ball &b){
    balls.append(b);
}
void RightWidget::updateBalls(){
    ball.setRectangle(this->geometry());
    ball.move();
    for(auto &b:balls){
        b.setRectangle(this->geometry());
        b.move();
    }
    update();
}
```

```
void RightWidget::paintEvent(QPaintEvent *){
    QPainter p(this);
    ball.draw(&p);
    for(auto& b:balls)
        b.draw(&p);
}
```

17) 定义小球之间碰撞检测方法（ball.h）

```
class Ball
{
public:
    Ball(double xpos,double ypos,double r,double s,double a,
         QColor c);
    Ball();
    void setRectangle(const QRect& rect){
        this->rect=rect;
    }
    void move();
    void draw(QPainter *p);
    void checkCollision(Ball &b);
private:
    …
};
```

18) 实现小球间的碰撞检测与处理（ball.cpp）

```
void Ball::checkCollision(Ball &b){
    double dx,dy;
    dx=x-b.x;
    dy=y-b.y;
    double dis=std::sqrt(dx*dx+dy*dy);
    if(dis<=radius+b.radius){   //碰撞后只是交换速度和角度
        double temp;
        temp=speed;
        speed=b.speed;
        b.speed=temp;
        temp=angle;
        angle=b.angle;
        b.angle=temp;
    }
}
```

19) 集成碰撞检测方法（RightWidget.cpp）

```
void RightWidget::updateBalls(){
    for(auto &b:balls){
        b.setRectangle(this->geometry());
```

```
            b.move();
        }   //先移动小球，处理与边框的碰撞，再处理小球间碰撞
        int i,j;
        for(i=0;i<balls.size()-1;++i){
            for(j=i+1;j<balls.size();++j)
                balls[i].checkCollision(balls[j]);
        }
        update();
    }
```

至此，程序的核心功能已经实现，可以模拟多个小球的移动，小球与边框的碰撞检测，小球之间的碰撞检测，也预留了动态添加小球的接口。下面将实现左侧的辅助控制功能，实现添加小球和定时器的启动与恢复控制。

5.1.4 小球的添加与控制

1) 左侧控制界面定义（LeftWidget.h）

```
#ifndef LEFTWIDGET_H
#define LEFTWIDGET_H
#include <QWidget>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
class MainWindow;
class LeftWidget : public QWidget
{
    Q_OBJECT
public:
    explicit LeftWidget(QWidget *parent = 0);
signals:
private:
    QLabel *xLabel,*yLabel,*radiusLabel,
        *speedLabel,*angleLabel,*colorLabel;
    QLineEdit *xEdit,*yEdit,*radiusEdit,*speedEdit,*angleEdit;
    QPushButton *addButton,*stopButton;
    MainWindow *pmain;   //指向主窗口的指针
};
#endif // LEFTWIDGET_H
```

以上界面元素实现控制界面中输入小球的位置、半径、速度、角度等属性，颜色设置将在后面再实现。在 LeftWidget 类中定义 MainWindow 指针（pmain），用于间接访问右侧的窗口类对象（RightWidget），以获得对小球数据的访问。

2) 界面元素构造（LeftWidget.cpp）

```
#include "leftwidget.h"
#include <QGridLayout>
#include <QPalette>
#include <QEvent>
#include <QPainter>
#include <QString>
#include "mainwindow.h"
LeftWidget::LeftWidget(QWidget *parent) :
    QWidget(parent)
{
    pmain=(MainWindow *)parent;
    xLabel=new QLabel(this);
    xLabel->setText(tr("x 坐标:"));
    yLabel=new QLabel(this);
    yLabel->setText(tr("y 坐标:"));
    radiusLabel=new QLabel(this);
    radiusLabel->setText(tr("半径:"));
    speedLabel=new QLabel(this);
    speedLabel->setText(tr("速度:"));
    angleLabel=new QLabel(this);
    angleLabel->setText(tr("角度:"));
    colorLabel=new QLabel(this);
    colorLabel->setText(tr("颜色:"));

    xEdit=new QLineEdit("100",this);
    yEdit=new QLineEdit("50",this);
    radiusEdit=new QLineEdit("10",this);
    speedEdit=new QLineEdit("5",this);
    angleEdit=new QLineEdit("40",this);

    addButton=new QPushButton(tr("添加"),this);
    stopButton=new QPushButton(tr("暂停"),this);

    QGridLayout *mainLayout=new QGridLayout(this);
    mainLayout->addWidget(xLabel,0,0);
    mainLayout->addWidget(xEdit,0,1);
    mainLayout->addWidget(yLabel,1,0);
    mainLayout->addWidget(yEdit,1,1);
    mainLayout->addWidget(radiusLabel,2,0);
    mainLayout->addWidget(radiusEdit,2,1);
    mainLayout->addWidget(speedLabel,3,0);
    mainLayout->addWidget(speedEdit,3,1);
    mainLayout->addWidget(angleLabel,4,0);
```

```
        mainLayout->addWidget(angleEdit,4,1);
        mainLayout->addWidget(colorLabel,5,0);
        mainLayout->addWidget(addButton,6,0);
        mainLayout->addWidget(stopButton,6,1);
        setLayout(mainLayout);
    }
```

在界面布局中使用了网格布局（QGridLayout），将控件布置为 6 行 2 列的布局，颜色预览的控件比较复杂，需要在普通的 QLabel 基础之上派生子类，通过重载该控件的 paintEvent 方法实现控件的自定义绘制。

3) 定义颜色预览条自定义类（PaintLabel.h）

```
    #ifndef PAINTLABEL_H
    #define PAINTLABEL_H
    #include <QLabel>
    #include <QColor>
    #include <QMouseEvent>
    class PaintLabel : public QLabel    //QLabel 是 QWidget 的子类
    {
        Q_OBJECT
    public:
        explicit PaintLabel(QWidget *parent = 0);
        void paintEvent(QPaintEvent *event);    //
          // QWidget 子类都可以覆盖该虚函数实现窗口绘制
        void setFillColor(QColor color);
        void mousePressEvent(QMouseEvent *e);
        QColor getFillColor();
    private:
        QColor fillColor;
    };
    #endif // PAINTLABEL_H
```

PaintLabel 继承自 QLabel，实现颜色条的预览效果，在该类中定义 QColor 对象表示当前颜色，定义 getFillColor 获取该颜色值，重载 paintEvent 方法实现对 Label 控件的填充颜色绘制操作，同时定义该控件对应的鼠标事件处理，响应鼠标点击事件。

4) 定义颜色预览条自定义类的成员函数（PaintLabel.cpp）

```
    #include "paintlabel.h"
    #include <QPainter>
    #include <QWidget>
    #include <QColorDialog>
    PaintLabel::PaintLabel(QWidget *parent):QLabel(parent)
    {
```

```
        fillColor=Qt::blue;
    }
    void PaintLabel::setFillColor(QColor color)
    {
        fillColor=color;
    }
    QColor PaintLabel::getFillColor()
    {
        return fillColor;
    }
    void PaintLabel::paintEvent(QPaintEvent *event)
    {
        QLabel::paintEvent(event);
        QPainter p(this);
        QPen pen(fillColor,2,Qt::SolidLine);
        QBrush brush(fillColor);
        p.setPen(pen);
        p.setBrush(brush);
        p.drawRect(0,0,this->width(),this->height());
        //根据 fillColor 填充 Label 控件，实现颜色预览
    }
    void PaintLabel::mousePressEvent(QMouseEvent *e){
        QColor chooseColor=QColorDialog::getColor(fillColor);    //返回颜色值
        if(chooseColor.isValid()==true)  //如果用户取消，则返回无效颜色值
        {
            fillColor=chooseColor;
            update();
        }
    }
```

5) 添加颜色预览控件（LeftWidget.h）

```
    #include "paintlabel.h"
    class LeftWidget : public QWidget
    {
        Q_OBJECT
    public:
        explicit LeftWidget(QWidget *parent = 0);
    signals:

    public slots:
        void addBall();
        void stopBall();
    private:
        QLabel *xLabel,*yLabel,*radiusLabel,
            *speedLabel,*angleLabel,*colorLabel;
```

```
    QLineEdit *xEdit,*yEdit,*radiusEdit,*speedEdit,*angleEdit;
    PaintLabel *colorPreview;
    QPushButton *addButton,*stopButton;
    MainWindow *pmain;
};
```

6) 构建颜色预览控件（LeftWidget.cpp）

```
LeftWidget::LeftWidget(QWidget *parent) :
    QWidget(parent)
{
    …
    colorPreview=new PaintLabel(this);
    colorPreview->setText(tr("          "));   //初始字符串确定控件大小
    angleEdit=new QLineEdit("40",this);
    addButton=new QPushButton(tr("添加"),this);
    stopButton=new QPushButton(tr("暂停"),this);
    …
    mainLayout->addWidget(colorLabel,5,0);
    mainLayout->addWidget(colorPreview,5,1);
    mainLayout->addWidget(addButton,6,0);
    mainLayout->addWidget(stopButton,6,1);
    setLayout(mainLayout);
}
```

7) 定义按钮点击事件的槽方法（LeftWidget.h）

```
class LeftWidget : public QWidget
{
    Q_OBJECT
public:
    explicit LeftWidget(QWidget *parent = 0);
signals:
public slots:
    void addBall();     //添加小球
    void stopBall();      //控制定时器的启动与暂停
    …
};
```

8) 主窗口类中封装获取右侧窗口的接口（MainWindow.h）

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
```

```
        RightWidget* getRightWidget(){return right;}
        …
    };
```

9) 实现按钮点击事件的槽方法（LeftWidget.cpp）

```
    LeftWidget::LeftWidget(QWidget *parent) :
        QWidget(parent)
    {
        …
        setLayout(mainLayout);
        connect(addButton,SIGNAL(clicked()),this,SLOT(addBall()));
        connect(stopButton,SIGNAL(clicked()),this,SLOT(stopBall()));
    }
    void LeftWidget::addBall(){
        RightWidget *right=pmain->getRightWidget();
        //pmain 时指向主窗口的指针，通过其接口 getRightWidget 获得
        //右侧窗口指针，再调用 right 的 addBall 接口添加小球
        double x,y;
        double radius,angle,speed;
        QColor fillColor;
        bool ok;
        x=xEdit->text().toDouble(&ok);
        y=yEdit->text().toDouble(&ok);
        radius=radiusEdit->text().toDouble(&ok);
        speed=speedEdit->text().toDouble(&ok);
        angle=angleEdit->text().toDouble(&ok);
        fillColor=colorPreview->getFillColor();
        right->addBall(Ball(x,y,radius,speed,angle,fillColor));
    }
    void LeftWidget::stopBall(){
        if(stopButton->text()==tr("暂停")){
            pmain->StopTimer();
            stopButton->setText(tr("开始"));
        }
        else{
            pmain->resumeTimer();
            stopButton->setText(tr("暂停"));
        }
    }
```

5.2 实习任务

5.2.1 实习任务一

在演示案例的基础之上，扩展程序应用，在小球的运动过程中增加阻尼效应。一种简单的模型如下：在小球移动过程中，每移动 1 步后速度下降到原来的 99%，小球碰撞后每个小球的速度下降到原来的 95%，当速度小于某个阈值（如 0.1）时，将速度置为 0。

5.2.2 实习任务二

在右侧窗口中，通过点击鼠标发射小球，点击鼠标处为起始点，拖动鼠标后可以绘制小球的预览效果（橡皮线），通过拖动的距离计算对应的速度（采用线性比例或其它模型），通过释放鼠标处位置和起始点位置确定发射角度。

5.2.3 实习任务三

解决碰撞模型的 Bug，速度相同、方向接近的小球碰撞后，简单交换速度和角度，会出现“粘球”现象。对于这种情况，应在小球碰撞后强行将其分开一段距离，使其保持未接触状态。

5.2.4 实习任务四

【扩展应用】实现更多的控制元素，比如在右侧窗口底部添加挡板，通过键盘控制挡板的移动，通过挡板控制小球的反弹，小球触碰到底边会移出界面并判定游戏结束。

6 综合编程：简单绘图程序

6.1 演示案例

【任务目标】实现简单的绘图功能，并支持绘图数据的保存和打开等功能。通过 Draw 菜单，由用户点击选择不同的图形，按下鼠标左键并拖动鼠标开始绘图，松开鼠标后保存图形（线条、椭圆、矩形）数据，并更新屏幕显示。

File 菜单下定义文件的基本操作，New 菜单项用于新建绘图文件，如果上一个文件修改了并没有保存，需要进行提示，Save 用于将绘图结果保存为磁盘文件，实现持久化存储，Open 用于打开存盘的图形文件。

6.1.1 简单绘图版本

1) 新建项目，名称为 SimpleDraw，类型选择“Qt Widgets Application”。在“类详情”设置页面中，取消“创建界面”复选框，选择基类为 QMainWindow。添加新类“CenterWidget”，选择基类为 QWidget。

2) 在主窗口类中添加 CenterWidget 对象（MainWindow.h）

```
#include <QMainWindow>
#include "centerwidget.h"
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
};
```

3) 构造 centerWidget（MainWindow.cpp）

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单绘图程序  未命名"));
    centerWidget=new CenterWidget(this);
    setCentralWidget(centerWidget);
}
```

4) 启动时让窗口最大化显示（main.cpp）

```
#include "mainwindow.h"
#include <QApplication>
```

```
int main(int argc, char *argv[]){
    QApplication a(argc, argv);
    MainWindow w;
    //w.show();
    w.showMaximized();   //主程序窗口最大化输出
    return a.exec();
}
```

5) 设计数据结构（CenterWidget.h）

目前系统中支持线条、椭圆和矩形 3 中图形，线条可以通过 QLine 保存，椭圆和矩形可以统购 QRect 保，椭圆的矩形定义其外接矩形。使用 Qt 中的集合类 QList 存储所绘制的图形数据，也可以考虑使用 STL 中的 list 等容器。

```
#ifndef CENTERWIDGET_H
#define CENTERWIDGET_H
#include <QList>
#include <QPoint>
#include <QRect>
#include <QWidget>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
signals:
public slots:
private:
    QList<QLine> lines;     //存储线条的集合
    QList<QRect> ellipses;     //存储椭圆的集合
    QList<QRect> rects;     //存储矩形的集合
    QPoint p1,p2;             //绘图过程的临时起点坐标和终点坐标
    int drawType;           //当前绘图类型，0-直线，1-椭圆，2-矩形
    bool beginDraw;
};
#endif // CENTERWIDGET_H
```

绘图时，按下鼠标左键时记录起点 p1，拖动鼠标显示绘制过程，松开鼠标时记录终点 p2，由 2 个点确定要绘制的线条、椭圆或矩形。drawType 记录当前用户选择的图形类型。bool 型的 beginDraw 记录是否正在绘图，按下鼠标左键时将 beginDraw 置为 true，标志进入绘图状态，松开鼠标时将 beginDraw 置为 false，标志结束绘图，将来在处理鼠标事件时，只有当 beginDraw 为 true 时才执行绘图处理操作。

6) centerWidget 构造及数据初始化（CenterWidget.cpp）

```
#include "centerwidget.h"
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    lines.clear();
    ellipses.clear();
    rects.clear();
    beginDraw=false;
    setMinimumSize(500,400);
}
```

QList 的 clear 方法用于清空容器中的元素。

7) 绘制所有图形数据（CenterWidget.h，CenterWidget.cpp）

```
//CenterWidget.h
#include <QList>
#include <QPoint>
#include <QRect>
#include <QWidget>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
signals:
public slots:
private:
    QList<QLine> lines;     //存储线条的集合
    QList<QRect> ellipses;     //存储椭圆的集合
    QList<QRect> rects;     //存储矩形的集合
    QPoint p1,p2;             //绘图过程的临时起点坐标和终点坐标
    int drawType;            //当前绘图类型，0-直线，1-椭圆，2-矩形
    bool beginDraw;
};

//CenterWidget.cpp
#include <QPainter>
void CenterWidget::paintEvent(QPaintEvent *){
    QPainter p(this);
    for(auto const& line:lines)
        p.drawLine(line);
    for(auto const& ellipse:ellipses)
        p.drawEllipse(ellipse);
    for(auto const& rect:rects)
        p.drawRect(rect);
}
```

8) 菜单和工具栏设计（MainWindow.h）

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QMenu>
#include <QMenuBar>
#include <QAction>
#include <QToolBar>
#include "centerwidget.h"
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
    QMenu *fileMenu;
    QMenu *drawMenu;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
    QAction *exitAction;
    QAction *lineAction;
    QAction *ellipseAction;
    QAction *rectangleAction;
    QToolBar *drawToolBar;
};
#endif // MAINWINDOW_H
```

QMenu 封装菜单，QAction 封装了对菜单和工具栏按钮的统一处理，QToolBar 封装了工具栏，QStatusBar 封装状态栏。状态栏栏只有 1 个，不需要单独定义，通过 QMainWindow 的 statusBar()方法可以直接得到指向状态栏对象的指针，所以不需要专门定义。

9) 菜单和工具栏构造（MainWindow.cpp）

```
#include "mainwindow.h"
#include <QStatusBar>
#include <QActionGroup>
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单绘图程序 未命名"));
    centerWidget=new CenterWidget(this);
```

```
setCentralWidget(centerWidget);
```

```
fileMenu=menuBar()->addMenu(tr("文件"));
newAction=new QAction(QIcon("new.gif"),tr("新建"),this);
newAction->setShortcut(tr("Ctrl+N"));
newAction->setStatusTip(tr("新建绘图文件"));
openAction=new QAction(QIcon("open.gif"),tr("打开"),this);
openAction->setShortcut(tr("Ctrl+O"));
openAction->setStatusTip(tr("打开保存的绘图文件"));
saveAction=new QAction(QIcon("save.gif"),tr("保存"),this);
saveAction->setShortcut(tr("Ctrl+S"));
saveAction->setStatusTip(tr("保存当前绘图文件"));
exitAction=new QAction(tr("退出"),this);
exitAction->setShortcut(tr("Ctrl+X"));
exitAction->setStatusTip(tr("退出程序"));
fileMenu->addAction(newAction);
fileMenu->addAction(openAction);
fileMenu->addAction(saveAction);
fileMenu->addSeparator();
fileMenu->addAction(exitAction);
```

```
lineAction=new QAction(QIcon("line.gif"),tr("线条"),this);
lineAction->setShortcut(tr("Ctrl+L"));
lineAction->setStatusTip(tr("绘制线条"));
ellipseAction=new QAction(QIcon("ellipse.gif"),tr("椭圆"),this);
ellipseAction->setShortcut(tr("Ctrl+E"));
ellipseAction->setStatusTip(tr("绘制椭圆"));
rectangleAction=new QAction(QIcon("rectangle.gif"),tr("矩形"),this);
rectangleAction->setShortcut(tr("Ctrl+R"));
rectangleAction->setStatusTip(tr("绘制矩形"));
drawMenu= menuBar()->addMenu(tr("绘图"));
lineAction->setCheckable(true);
ellipseAction->setCheckable(true);
rectangleAction->setCheckable(true);
QActionGroup *group=new QActionGroup(this);
group->addAction(lineAction);
group->addAction(ellipseAction);
group->addAction(rectangleAction);
group->setExclusive (true);
lineAction->setChecked(true);
drawMenu->addAction(lineAction);
drawMenu->addAction(ellipseAction);
drawMenu->addAction(rectangleAction);
```

```
        drawToolBar=addToolBar(tr("Draw"));
        drawToolBar->addAction(newAction);
        drawToolBar->addAction(openAction);
        drawToolBar->addAction(saveAction);
        drawToolBar->addSeparator();
        drawToolBar->addAction(lineAction);
        drawToolBar->addAction(ellipseAction);
        drawToolBar->addAction(rectangleAction);
```

```
        statusBar()->show();
    }
```

代码解析：QMainWindow 的 menuBar()方法返回整个程序的顶层菜单，调用它的 addMenu()方法可以添加菜单栏，并返回指向所添加菜单栏的指针。QAction 定义了菜单或工具栏按钮的图标、文本、助记符、快捷键、状态栏提示文本等内容，通过 QMenu 的 addAction()方法可以将其添加到菜单栏下，通过 QToolBar 的 addAction()方法也可以将其添加到工具栏中。

默认情况下，状态栏是不显示的，此处通过 statuBar()方法获得指向状态栏对象的指针，调用其 show()方法让其显示出来。

如果要实现一组互斥的菜单项，可以定义 1 个 QActionGroup 对象，然后将互斥的菜单项通过 addAction()方法加入其中，同时调用 setCheckable()方法设置这些菜单项的可勾选状态，然后调用 QActionGroup 对象的 setExclusive()方法设置它们的互斥状态。

10) 绘图菜单项事件处理框架（MainWindow.h，MainWindow.cpp）

```
//MainWindow.h
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
    QMenu *fileMenu;
    QMenu *drawMenu;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
    QAction *exitAction;
    QAction *lineAction;
    QAction *ellipseAction;
    QAction *rectangleAction;
```

```
    QToolBar *drawToolBar;
protected slots:     //定义槽，相应菜单和工具栏按钮的点击事件
    void line();
    void ellipse();
    void rectangle();
};

//MainWindow.cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单绘图程序  未命名"));
    centerWidget=new CenterWidget(this);
    setCentralWidget(centerWidget);
    …
    statusBar()->show();
    connect(exitAction,SIGNAL(triggered()),this,SLOT(close()));
    connect(lineAction,SIGNAL(triggered()),this,SLOT(line()));
    connect(ellipseAction,SIGNAL(triggered()),this,SLOT(ellipse()));
    connect(rectangleAction,SIGNAL(triggered()),this,SLOT(rectangle()));
}
void MainWindow::line(){
}
void MainWindow::ellipse(){
}
void MainWindow::rectangle(){
}
```

将绘制线条、椭圆、矩形菜单项的点击事件映射到我们定义的槽函数中，当用户点击相应的菜单项时，会自动调用绑定的槽函数。退出菜单项，用于结束程序，绑定到 QmainWindow 预定义的 close 槽方法中。

11) 绘图菜单项事件处理（MainWindow.h，MainWindow.cpp）

绘图分析：用户点击线条、椭圆、矩形菜单项时，只是切换要绘制的图形类型。实际绘图是通过鼠标操作完成的，按下鼠标开始绘图，记录绘图起点，松开鼠标完成绘图操作。我们在 CenterWidget 类中定义 drawType 标识绘图类型，为了在 MainWindow 类中访问，为 CenterWidget 封装了 setDrawType 方法。

```
//CenterWidget.h
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void setDrawType(int type);
```

```
signals:
public slots:
private:
    QList<QLine> lines;     //存储线条的集合
    QList<QRect> ellipses;     //存储椭圆的集合
    QList<QRect> rects;     //存储矩形的集合
    QPoint p1,p2;             //绘图过程的临时起点坐标和终点坐标
    int drawType;           //当前绘图类型，0-直线，1-椭圆，2-矩形
    bool beginDraw;
};

//CenterWidget.cpp
void CenterWidget::setDrawType(int type){
    drawType=type;
}
//MainWindow.cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单绘图程序  未命名"));
    ….
    centerWidget->setDrawType(0);
}
void MainWindow::line(){
    centerWidget->setDrawType(0);
}
void MainWindow::ellipse(){
    centerWidget->setDrawType(1);
}
void MainWindow::rectangle(){
    centerWidget->setDrawType(2);
}
```

12) 完成绘图处理（CenterWidget.h，CenterWidget.cpp）

```
//CenterWidget.h
#include <QMouseEvent>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void setDrawType(int type);
signals:
public slots:
```

```
private:
    QList<QLine> lines;     //存储线条的集合
    QList<QRect> ellipses;     //存储椭圆的集合
    QList<QRect> rects;     //存储矩形的集合
    QPoint p1,p2;             //绘图过程的临时起点坐标和终点坐标
    int drawType;           //当前绘图类型，0-直线，1-椭圆，2-矩形
    bool beginDraw;
protected:
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
    void mouseReleaseEvent(QMouseEvent *e);
};
```

为窗口监听相应的鼠标事件，需要覆盖对应的虚函数方法，以实现具体的事件处理。此外，还需要调用 setMouseTracking(true)获得对鼠标事件的响应。

```
//CenterWidget.cpp
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    lines.clear();
    ellipses.clear();
    rects.clear();
    beginDraw=false;
    setMinimumSize(500,400);
    setMouseTracking(true);
}
void CenterWidget::mousePressEvent(QMouseEvent *e){
    p1=e->pos();
    p2=p1;
    beginDraw=true;
}
void CenterWidget::mouseReleaseEvent(QMouseEvent *e){
    p2=e->pos();
    beginDraw=false;
    if(p1==p2)
        return;
    switch(drawType){
    case 0:
        lines.append(QLine(p1,p2));
        break;
    case 1:
        ellipses.append(QRect(p1,p2));
        break;
    case 2:
        rects.append(QRect(p1,p2));
```

```
            break;
        }
        update();
    }
```

按下鼠标左键，记录起点 p1，设置 beginDraw 为 true，标志开始绘图。松开鼠标后，完成绘图，记录终点 p2。根据所选择的图形类型（drawType），创建相应的图形对象，添加到对应图形集合中，调用 update 更新窗口显示。

13) 鼠标移动过程中橡皮线效果（CenterWidget.cpp）

```
    void CenterWidget::mouseMoveEvent(QMouseEvent *e){
        if(beginDraw==false)
            return;
        p2=e->pos();     //更新 p2
        update();     //绘制 p1 和 p2 之间构成的图形
    }

    void CenterWidget::paintEvent(QPaintEvent *){
        QPainter p(this);
        for(auto const& line:lines)
            p.drawLine(line);
        for(auto const& ellipse:ellipses)
            p.drawEllipse(ellipse);
        for(auto const& rect:rects)
            p.drawRect(rect);
        if(beginDraw==true){
            switch(drawType){
            case 0:
                p.drawLine(p1,p2);
                break;
            case 1:
                p.drawEllipse(QRect(p1,p2));
                break;
            case 2:
                p.drawRect(QRect(p1,p2));
                break;
            }
        }
    }
```

14) 鼠标移动过程中坐标显示（CenterWidget.h，CenterWidget.cpp）

绘图过程中，应该在鼠标移动过程中，动态显示鼠标当前位置信息，通常在状态栏中显示这些信息。默认情况下，状态栏中只包含显示菜单和工具栏提示信息的部分，如果要增加显示不同的内容，需要向其中动态添加组件，然后向相应

的组件中输出信息。

```
//CenterWidget.h
#include <QLabel>
class CenterWidget : public QWidget{
    Q_OBJECT
public:
    explicit CenterWidget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *);
    void setDrawType(int type);
signals:
public slots:
private:
    QList<QLine> lines;     //存储线条的集合
    QList<QRect> ellipses;     //存储椭圆的集合
    QList<QRect> rects;     //存储矩形的集合
    QPoint p1,p2;             //绘图过程的临时起点坐标和终点坐标
    int drawType;           //当前绘图类型，0-直线，1-椭圆，2-矩形
    bool beginDraw;
    QLabel *mousePosLabel;
protected:
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
    void mouseReleaseEvent(QMouseEvent *e);
};


//CenterWidget.cpp
#include <QStatusBar>
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    lines.clear();
    ellipses.clear();
    rects.clear();
    beginDraw=false;
    mousePosLabel=new QLabel;
    mousePosLabel->setText("");
    mousePosLabel->setFixedWidth(150);
    MainWindow *p=(MainWindow *)parent;
    p->statusBar()->addPermanentWidget(mousePosLabel);
    setMinimumSize(500,400);
    setMouseTracking(true);
}
```

通过调用状态栏对象的 addPermanentWidget()方法，可以动态在状态栏中添

加组件，添加的组件位于右侧。在后续的程序中，我们只要向相应的控件输出信息，即可完成状态栏输出。

```
void CenterWidget::mouseMoveEvent(QMouseEvent *e){
    mousePosLabel->setText("X:"+QString::number(e->x())+",Y:"
        +QString::number(e->y()));
    if(beginDraw==false)
        return;
    p2=e->pos();
    update();
}
```

15) 定义文件菜单的响应处理框架（MainWindow.h，MainWindow.cpp）

```
//MainWindow.h
class MainWindow : public QMainWindow{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    CenterWidget *centerWidget;
    QMenu *fileMenu;
    QMenu *drawMenu;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
    QAction *exitAction;
    QAction *lineAction;
    QAction *ellipseAction;
    QAction *rectangleAction;
    QToolBar *drawToolBar;
protected slots:     //定义槽，相应菜单和工具栏按钮的点击事件
    void line();
    void ellipse();
    void rectangle();
    void newDrawing();
    void openDrawing();
    void saveDrawing();
};

//MainWindow.cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent){
    setWindowTitle(tr("简单绘图程序  未命名"));
    centerWidget=new CenterWidget(this);
```

```
        setCentralWidget(centerWidget);
        …
        statusBar()->show();
        connect(exitAction,SIGNAL(triggered()),this,SLOT(close()));
        connect(lineAction,SIGNAL(triggered()),this,SLOT(line()));
        connect(ellipseAction,SIGNAL(triggered()),this,SLOT(ellipse()));
        connect(rectangleAction,SIGNAL(triggered()),this,SLOT(rectangle()));
        connect(newAction,SIGNAL(triggered()),this,SLOT(newDrawing()));
        connect(openAction,SIGNAL(triggered()),this,SLOT(openDrawing()));
        connect(saveAction,SIGNAL(triggered()),this,SLOT(saveDrawing()));
    }

    void MainWindow:: newDrawing (){
    }
    void MainWindow:: openDrawing (){
    }
    void MainWindow:: saveDrawing (){
    }
```

考虑到图形数据保存在 CenterWidget 类中，在 MainWindow 类中只是定义菜单处理的框架，具体新建、保存及打开文件操作应该在 CenterWidget 类中定义。

16) 实现文件操作（CenterWidget.h，CenterWidget.cpp）

文件操作的逻辑还是比较复杂，如执行“新建”菜单时，需要清空当前的图形数据，如果用户之前已经绘制图形数据而未保存，应提醒用户是否保存，执行“打开”菜单项时也需要考虑相同的问题。而用户针对存盘提示问题，也可能会选择不同的操作，“是”、“否”和“取消”。在执行“保存”菜单项时，如果之前没有保存过，应弹出文件保存对话框，由用户选择存储的路径和文件名，如果之前已经保存过，则直接存储。在未保存的状态下，如果用户直接退出程序，程序应该能够识别并提醒用户。

```
    //CenterWidget.h
    #include <QLabel>
    class CenterWidget : public QWidget{
        Q_OBJECT
    public:
        explicit CenterWidget(QWidget *parent = 0);
        void paintEvent(QPaintEvent *);
        void setDrawType(int type);
        void newDrawing();
        void openDrawing();
        void saveDrawing();
        bool getModifiedFlag();      //获取文档修改状态
```

```
signals:
public slots:
private:
    QList<QLine> lines;     //存储线条的集合
    QList<QRect> ellipses;     //存储椭圆的集合
    QList<QRect> rects;     //存储矩形的集合
    QPoint p1,p2;             //绘图过程的临时起点坐标和终点坐标
    int drawType;           //当前绘图类型，0-直线，1-椭圆，2-矩形
    bool beginDraw;
    QLabel *mousePosLabel;
    bool isModified;                  //标识文档是否发生了修改
    QString fileName;               //文件名，新建后为空
    void saveFile(QString fileName);   //写文件的操作
    void openFile(QString fileName);   //读文件的操作
protected:
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
    void mouseReleaseEvent(QMouseEvent *e);
};

//CenterWidget.cpp
#include <QFileDialog>
#include <QTextStream>
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    lines.clear();
    ellipses.clear();
    rects.clear();
    beginDraw=false;
    isModified=false;        //还没有绘制图形，未修改状态
    fileName=tr("");
    setMouseTracking(true);
    mousePosLabel=new QLabel;
    mousePosLabel->setText("");
    mousePosLabel->setFixedWidth(150);
    MainWindow *p=(MainWindow *)parent;
    p->statusBar()->addPermanentWidget(mousePosLabel);
    setMinimumSize(500,400);
}

bool CenterWidget::getModifiedFlag(){
    return isModified;
}
```

```
void CenterWidget::mouseReleaseEvent(QMouseEvent *e){
    p2=e->pos();
    beginDraw=false;
    if(p1==p2)
        return;
    switch(drawType){
    case 0:
        lines.append(QLine(p1,p2));
        break;
    case 1:
        ellipses.append(QRect(p1,p2));
        break;
    case 2:
        rects.append(QRect(p1,p2));
        break;
    }
    isModified=true;
    update();
}
```

```
void CenterWidget::newDrawing(){     //新建绘图
    lines.clear();
    ellipses.clear();
    rects.clear();              //清空图形内容
    beginDraw=false;
    isModified=false;
    fileName=tr("");
    parentWidget()->setWindowTitle(tr("简单绘图程序  未命名"));
    update();       //更新窗口显示
}
```

CenterWidget 类的 newDrawing 只负责新建绘图的处理，至于未保存数据的提示应该放在 MainWindow 类中实现。

```
void CenterWidget::openDrawing(){     //打开绘图
    fileName=QFileDialog::getOpenFileName(this,
        tr("打开文件对话框"),"/","绘图文件(*.draw);;所有文件(*.*)");
    if(fileName==tr(""))
        return;
    lines.clear();
    ellipses.clear();
    rects.clear();
    beginDraw=false;
    isModified=false;
```

```
        openFile(fileName);        //存储文件
        parentWidget()->setWindowTitle(tr("简单绘图程序 ")+fileName);
        update();
    }
```

openDrawing 方法弹出文件选择对话框，由用户选择打开的文件，该方法由 QfileDialog 实现，静态函数 getOpenFileName 弹出系统文件选择对话框，第 1 个参数为父窗口指针，第 2 个参数为标题栏信息，第 3 个参数为初始路径，第 4 个参数为文件类型和扩展名过滤器。文件选择对话框返回用户所选文件的完整路径，若用户取消对话框，则返回空串。

用户选择文件后，首先清空之前绘制的图形，设置修改状态 isModified 为 false，beginDraw 置为 false，同时修改程序的标题栏，此处使用 parentWidget() 方法返回父窗口指针，因为我们修改的是 MainWindow 窗口标题。具体读取文件内容由私有函数 openFile 实现，最后调用 update 更新窗口显示。

```
    void CenterWidget::saveDrawing(){    //存储绘图
        if(fileName==tr("")){ //文件名为空，弹出文件选择对话框
            fileName=QFileDialog::getSaveFileName(this,
                tr("保存文件对话框"),"/","绘图文件(*.draw);;所有文件(*.*)");
            if(fileName==tr(""))
                return;
        }
        saveFile(fileName);       //存储文件
        parentWidget()->setWindowTitle(tr("简单绘图程序 ")+fileName);
        isModified=false;
    }
```

saveDrawing 方法首先判断文件名是否为空串，若为空串，是首次保存，弹出文件保存对话框，由用户选择存储的路径和文件名；若文件名不为空，则直接按照当前文件直接保存。弹出存储文件对话框由 QfileDialog 类的静态方法 getSaveFileName 实现，参数与 getOpenFileName 类似，该方法返回用户所选文件的完整路径。注意，用户在文件选择对话框中可能会选择取消，此时返回的文件路径为空串，需要进行判断并返回，不执行存储文件的后续操作。具体存储文件的操作由 saveFile 私有方法完成。

```
    void CenterWidget::saveFile(QString fileName){     //写文件操作
        QFile file(fileName);
        if(file.open(QFile::WriteOnly|QFile::Truncate)){
            QTextStream out(&file);
            out<<lines.length()<<endl;
```

```
                for(QLine line:lines)
                    out<<line.p1().x()<<" "<<line.p1().y()<<" "
                            <<line.p2().x()<<" "<<line.p2().y()<<endl;
                out<<ellipses.length()<<endl;
                for(QRect rect:ellipses)
                    out<<rect.topLeft().x()<<" "<<rect.topLeft().y()<<" "
                            <<rect.bottomRight().x()
                            <<" "<<rect.bottomRight().y()<<endl;
                out<<rects.length()<<endl;
                for(QRect rect:rects)
                    out<<rect.topLeft().x()<<" "<<rect.topLeft().y()<<" "
                            <<rect.bottomRight().x()
                            <<" "<<rect.bottomRight().y()<<endl;
                file.close();
            }
        }
```

QFile 封装了文件操作，构造 QFile 对象时关联 1 个文件，通过其 open 方法可以指定打开方式，QFile::WriteOnly|QFile::Truncate 表示写文件并清空文件原来的内容。为了方便输入输出操作，在 QFile 对象之上，通常使用封装的 QTextStream 流对象（文本方式）或 QDataStream（二进制方式），通过<<和>>运算符可以方便数据读写。

存储结构设计：我们以文本方式存储所有的线条、椭圆和矩形数据，为了能够打开时恢复状态，在存储数据时，首先需要存储每种图形的数量，再逐个图形存储其数据。比如当前绘制了 2 条直线，3 个椭圆，则可能的存储结构如下：

2

101 200 340 346

23 345 128 129

3

50 190 340 560

230 110 330 180

12 234 45 333

0

2 对应线条数量，下面是 2 条线条的具体数据，对应线条起点和终点坐标的 x 和 y 坐标对。3 对应椭圆的数量，下面依次存储 3 个椭圆的数据，对应椭圆左上角和右下角点的 x 和 y 坐标值。矩形数据的数量为 0，也要保存，但没有具体的矩形数据。

QList 的 length()方法可以返回容器中的元素数量。QLine 的 p1()方法返回线

条的起始点，类型为 QPoint，QPoint 的 x()方法和 y()方法分别返回点的 x 和 y 坐标值。QRect 的 topLeft()方法返回左上角点，bottomRight()方法返回右下角点。

```
void CenterWidget::openFile(QString fileName){       //读文件操作
    QFile file(fileName);
    if(file.open(QFile::ReadOnly)){       //只读方式打开
        QTextStream in(&file);
        int lineNums;
        in>>lineNums;
        int x1,y1,x2,y2;
        for(int i=0;i<lineNums;++i){
            in>>x1>>y1>>x2>>y2;
            lines.append(QLine(x1,y1,x2,y2));
        }
        int ellipseNums;
        in>>ellipseNums;
        for(int i=0;i<ellipseNums;++i){
            in>>x1>>y1>>x2>>y2;
            ellipses.append(QRect(QPoint(x1,y1),QPoint(x2,y2)));
        }
        int rectNums;
        in>>rectNums;
        for(int i=0;i<rectNums;++i){
            in>>x1>>y1>>x2>>y2;
            rects.append(QRect(QPoint(x1,y1),QPoint(x2,y2)));
        }
        file.close();
    }
}
```

读文件时，首先读取每种图形的数量，然后通过循环依次读取每个图形的数据，根据图形类型，重新构造相应的对象，通过 append 方法添加到对应的容器之中。

17) 主窗口中的文件操作逻辑控制（MainWindow.cpp）

```
void MainWindow::newDrawing(){
    if(centerWidget->getModifiedFlag()==true){   //文档已修改
        switch(QMessageBox::question(this,tr("文档保存提示"),
                    tr("文档已经修改，是否保存文档"),
                    QMessageBox::Ok|
                    QMessageBox::Cancel|QMessageBox::No,
                    QMessageBox::Ok)){
        case QMessageBox::Ok:
            centerWidget->saveDrawing();
```

```
                break;
            case QMessageBox::Cancel:
                return;
            case QMessageBox::No:
                break;
            default:
                break;
            }
        }
        centerWidget->newDrawing();
    }
```

在 MainWindow 的 newDrawing 方法中，首先判断文档是否已经修改（新建并绘图或打开文件后绘图），如果文档已经修改，通过 QMessageBox 的 question 弹出提示框，根据返回值分别进行处理。question 方法第 1 个参数为父窗口指针，第 2 个参数为标题栏文本，第 3 个参数为提示内容正文，第 4 个参数为窗口中显示的按钮，此处包括 Ok、No 和 Cancel 共 3 个按钮，第 5 个参数为缺省选中按钮。

如果用户选择 Ok，则执行 centerWidget 的 saveDrawing 方法，该方法根据文件名是否为空执行保存文件弹出框选择文件保存或直接保存。如果用户选择 Cancel，应直接返回方法，不做后续的处理。如果用户选择 No，在不保存文件的情况下直接新建文件，新建文件调用 centerWidget 的 newDrawing 方法实现。

```
    void MainWindow::openDrawing(){
        if(centerWidget->getModifiedFlag()==true){   //文档已修改
            switch(QMessageBox::question(this,tr("文档保存提示"),
                    tr("文档已经修改，是否保存文档"),
                    QMessageBox::Ok|QMessageBox::Cancel
                    |QMessageBox::No,
                    QMessageBox::Ok)){
            case QMessageBox::Ok:
                centerWidget->saveDrawing();
                break;
            case QMessageBox::Cancel:
                return;
            case QMessageBox::No:
                break;
            default:
                break;
            }
        }
```

```
        centerWidget->openDrawing();
    }
```

MainWindow 类中的 openDrawing 的处理逻辑与 newDrawing 的处理逻辑类似，首先判断文档是否已经修改，若修改了，需要弹出提示框，由用户选择相应的操作。

```
    void MainWindow::saveDrawing(){
        centerWidget->saveDrawing();
    }
```

18) 拦截关闭窗口事件（MainWindow.h，MainWindow.cpp）

```
    //MainWindow.h
    class MainWindow : public QMainWindow{
        Q_OBJECT
    public:
        MainWindow(QWidget *parent = 0);
        ~MainWindow();
    private:
        CenterWidget *centerWidget;
        QMenu *fileMenu;
        QMenu *drawMenu;
        QAction *newAction;
        QAction *openAction;
        QAction *saveAction;
        QAction *exitAction;
        QAction *lineAction;
        QAction *ellipseAction;
        QAction *rectangleAction;
        QToolBar *drawToolBar;
    protected slots:     //定义槽，相应菜单和工具栏按钮的点击事件
        void line();
        void ellipse();
        void rectangle();
        void newDrawing();
        void openDrawing();
        void saveDrawing();
        void closeEvent(QCloseEvent *event);
    };
```

```
    void MainWindow::closeEvent(QCloseEvent *event){
        if(centerWidget->getModifiedFlag()==true){   //文档已修改
            switch(QMessageBox::question(this,tr("文档保存提示"),
                    tr("文档已经修改，是否保存文档"),
```

```
                    QMessageBox::Ok|QMessageBox::Cancel
                    |QMessageBox::No,
                    QMessageBox::Ok)){
            case QMessageBox::Ok:
                centerWidget->saveDrawing();
                event->accept();
                break;
            case QMessageBox::Cancel:
                event->ignore();
                return;
            case QMessageBox::No:
                event->accept();
                break;
            default:
                event->accept();
                break;
            }
        }
    }
```

在 QmainWindow 中定义的虚函数 closeEvent 槽实现缺省的关闭窗口事件，包括点击关闭按钮或者执行 close 方法（关闭菜单绑定的槽方法），都会引起对 closeEvent 的调用，需要覆盖该方法，判断文档是否已经修改，并提示用户执行相应的处理。

closeEvent 方法传递的参数类型为 QCloseEvent，该对象的方法 accept()表示确认窗口关闭的行为，ignore()方法忽略窗口关闭行为。

6.1.2 多态版本绘图

目前的版本中，以 3 个容器分别存储线条、椭圆、矩形，程序的扩展性和灵活性不强，如果要增加一种图形，程序中需要修改的地方非常多。此外难以实现对每个图形颜色、线条样式的控制。

可以采用多态版本，定义 Shape 基类，在 Shape 类的基础之上派生不同的形状子类，在 Shape 类中可以封装颜色、线型等属性，从而实现更加灵活的绘图程序，在每个类中定义 draw 方法，实现自我绘制。

1) 创建 Shape 类。

2) 定义 Shape 基类（shape.h，shape.cpp）

```
//shape.h
#ifndef SHAPE_H
#define SHAPE_H
```

```
#include <QColor>
#include <QPainter>
#include <QTextStream>
class Shape{
public:
    Shape(QColor c);
    QColor getColor()const;
    void setColor(QColor c);
    virtual void draw(QPainter *p)=0;
    virtual void save(QTextStream& s)=0;
protected:
    QColor color;
};
#endif // SHAPE_H
```

在 Shape 类中定义纯虚函数 draw 和 save 接口，不同子类需要覆盖该抽象接口，实现各自不同的绘制和存储功能。

//shape.cpp

```
#include "shape.h"
Shape::Shape(QColor c):color(c){
}
void Shape::setColor(QColor c){
    color=c;
}
QColor Shape::getColor()const{
    return color;
}
```

3) 依次创建 Line、Ellipse 和 Rectangle 类，指定 Shape 类为基类。

4) 定义 Shape 类的子类线条类（line.h，line.cpp）

//line.h

```
#ifndef LINE_H
#define LINE_H
#include "shape.h"
#include <QPoint>
class Line : public Shape{
public:
    Line(QPoint p1,QPoint p2,QColor c=Qt::black);
    QPoint p1()const;
    QPoint p2()const;
    virtual void draw(QPainter *p);
    virtual void save(QTextStream& out);
    static Shape* read(QTextStream& in);
private:
```

```
    QPoint point1;
    QPoint point2;
};
#endif // LINE_H
```

Line 类中定义 point1 和 point2 数据成员，对应线条的起点和终点。在 cpp 文件需要给出 draw 和 save 方法的定义，draw 方法实现屏幕绘制，save 实现将 Line 的对象数据写入磁盘文件。同时定义静态方法 read，实现从磁盘文件中读取数据并构建 Line 对象，返回的是 Shape 指针，实际指向的是 Line 子类对象，将来可以利用多态实现灵活架构。

```
//line.cpp
#include "line.h"
Line::Line(QPoint p1,QPoint p2,QColor c):
    point1(p1),point2(p2),Shape(c){
}
QPoint Line::p1()const{
    return point1;
}
QPoint Line::p2()const{
    return point2;
}
void Line::draw(QPainter *p){
    QPen pen(color,1,Qt::SolidLine);
    p->setPen(pen);
    p->drawLine(point1,point2);      //绘制线条
}
void Line::save(QTextStream &out){
    out<<"0 "<<p1().x()<<" "<<p1().y()<<" "
        <<p2().x()<<" "<<p2().y()<<endl;
}
```

将来所有的图形对象都存储在同一个集合容器中，存储数据时，在 1 个循环中依次将所有数据都写入磁盘文件中。为了打开文件时能够识别并创建相应的图形类型，需要在写入数据时写入类型标识，此处在写入具体坐标数据之前写入标识图形类型的整数，0-线条，1-椭圆，2-矩形。因为 Line 类要保存数据，知道自己的类型码为 0，所以直接输出 0 即可。

```
Shape* Line::read(QTextStream &in){
    int x1,y1,x2,y2;
    in>>x1>>y1>>x2>>y2;
    return new Line(QPoint(x1,y1),QPoint(x2,y2));
}
```

在读取并恢复图形对象时，根据 save 写入的类型代码，可以判断将要读取

的数据类型，调用对应类的静态方法 read，该方法知道 Line 类应该读取哪些数据，并通过读取的数据创建 Line 对象，返回指向该对象的 Shape 指针，以便后续程序实现多态效果。

5) 定义 Shape 类的子类椭圆类（ellipse.h，ellipse.cpp）

```
//ellipse.h
#ifndef ELLIPSE_H
#define ELLIPSE_H
#include "shape.h"
class Ellipse : public Shape{
public:
    Ellipse(QPoint p1,QPoint p2,QColor c=Qt::black);
    QPoint topLeft()const;
    QPoint bottomRight()const;
    virtual void draw(QPainter *p);
    virtual void save(QTextStream& out);
    static Shape* read(QTextStream& in);
private:
    QPoint point1;
    QPoint point2;
};
#endif // ELLIPSE_H

//ellipse.cpp
#include "ellipse.h"
Ellipse::Ellipse(QPoint p1,QPoint p2,QColor c):
    point1(p1),point2(p2),Shape(c){
}
QPoint Ellipse::topLeft()const{
    return point1;
}
QPoint Ellipse::bottomRight()const{
    return point2;
}
void Ellipse::draw(QPainter *p){
    QPen pen(color,1,Qt::SolidLine);
    p->setPen(pen);
    p->drawEllipse(QRect(point1,point2));
}
void Ellipse::save(QTextStream &out){
    out<<"1 "<<topLeft().x()<<" "<<topLeft().y()<<" "
        <<bottomRight().x()<<" "<<bottomRight().y()<<endl;
}
Shape* Ellipse::read(QTextStream &in){
```

```
        int x1,y1,x2,y2;
        in>>x1>>y1>>x2>>y2;
        return new Ellipse(QPoint(x1,y1),QPoint(x2,y2));
    }
```

6) 定义 Shape 类的子类矩形类（rectangle.h，rectangle.cpp）

```
    //rectangl.h
    #ifndef RECTANGLE_H
    #define RECTANGLE_H
    #include "shape.h"
    class Rectangle : public Shape{
    public:
        Rectangle(QPoint p1,QPoint p2,QColor c=Qt::black);
        QPoint topLeft()const;
        QPoint bottomRight()const;
        virtual void draw(QPainter *p);
        virtual void save(QTextStream& out);
        static Shape* read(QTextStream& in);
    private:
        QPoint point1;
        QPoint point2;
    };
    #endif // RECTANGLE_H

    //rectangle.cpp
    #include "rectangle.h"
    Rectangle::Rectangle(QPoint p1,QPoint p2,QColor c):
        point1(p1),point2(p2),Shape(c){
    }
    QPoint Rectangle::topLeft()const{
        return point1;
    }
    QPoint Rectangle::bottomRight()const{
        return point2;
    }
    void Rectangle::draw(QPainter *p){
        QPen pen(color,1,Qt::SolidLine);
        p->setPen(pen);
        p->drawRect(QRect(point1,point2));
    }
    void Rectangle::save(QTextStream &out){
        out<<"2 "<<topLeft().x()<<" "<<topLeft().y()<<" "
            <<bottomRight().x()<<" "<<bottomRight().y()<<endl;
    }
```

```
Shape* Rectangle::read(QTextStream &in){
    int x1,y1,x2,y2;
    in>>x1>>y1>>x2>>y2;
    return new Rectangle(QPoint(x1,y1),QPoint(x2,y2));
}
```

7) 修改数据结构（CenterWidget.h）

```
#include "line.h"
#include "ellipse.h"
#include "rectangle.h"
class CenterWidget : public QWidget{
    Q_OBJECT
    ~CenterWidget();
    ….
signals:
public slots:
private:
    QList<Shape*> shapes;
    ~~QList<QLine> lines;~~
    ~~QList<QRect> ellipses;~~
    ~~QList<QRect> rects;~~
    QPoint p1,p2;
    int drawType;
    bool beginDraw;
    QLabel *mousePosLabel;
    bool isModified;
    QString fileName;
    void saveFile(QString fileName);
    void openFile(QString fileName);
};
```

代码分析：基于多态技术，我们将以统一方式处理所有的图形数据，原先 QList 存储的是具体的图形对象，将无法实现多态。为了实现多态，我们在 QList 容器中存储的数据类型是 Shape *。此外考虑到程序需要动态分配内存，为 CenterWidget 增加 1 个析构函数，以释放内存。

8) 修改程序处理逻辑（CenterWidget.cpp）

```
CenterWidget::~CenterWidget(){
    for(auto shape:shapes)
        delete shape;
}
```

shapes 容器中存储的是指针，每个指针指向堆中动态创建的图形对象，所以在析构函数中，需要释放所有在堆中创建的对象。

```
CenterWidget::CenterWidget(QWidget *parent) :
    QWidget(parent){
    lines.clear();
    ellipses.clear();
    rects.clear();
    shapes.clear();
    beginDraw=false;
    …
}

void CenterWidget::newDrawing(){
    lines.clear();
    ellipses.clear();
    rects.clear();
    for(auto shape:shapes)
        delete shape;
    shapes.clear();
    beginDraw=false;
    isModified=false;
    fileName=tr("");
    parentWidget()->setWindowTitle(tr("简单绘图程序  未命名"));
    update();
}

void CenterWidget::openDrawing(){     //打开绘图
    fileName=QFileDialog::getOpenFileName(this,
        tr("打开文件对话框"),"/","绘图文件(*.draw);;所有文件(*.*)");
    if(fileName==tr(""))
        return;
    lines.clear();
    ellipses.clear();
    rects.clear();
    for(auto shape:shapes)
        delete shape;
    shapes.clear();
    beginDraw=false;
    isModified=false;
    openFile(fileName);        //存储文件
    parentWidget()->setWindowTitle(tr("简单绘图程序  ")+fileName);
    update();
}

void CenterWidget::saveFile(QString fileName){     //写文件操作
    QFile file(fileName);
```

```
    if(file.open(QFile::WriteOnly|QFile::Truncate)){
        QTextStream out(&file);
        out<<lines.length()<<endl;
        for(QLine line:lines)
            out<<line.p1().x()<<" "<<line.p1().y()<<" "
               <<line.p2().x()<<" "<<line.p2().y()<<endl;
        out<<ellipses.length()<<endl;
        for(QRect rect:ellipses)
            out<<rect.topLeft().x()<<" "<<rect.topLeft().y()<<" "
               <<rect.bottomRight().x()
               <<" "<<rect.bottomRight().y()<<endl;
        out<<rects.length()<<endl;
        for(QRect rect:rects)
            out<<rect.topLeft().x()<<" "<<rect.topLeft().y()<<" "
               <<rect.bottomRight().x()
               <<" "<<rect.bottomRight().y()<<endl;
        out<<shapes.length()<<endl;
        for(auto shape:shapes)
            shape->save(out);
        file.close();
    }
}

void CenterWidget::openFile(QString fileName){      //读文件操作
    QFile file(fileName);
    if(file.open(QFile::ReadOnly)){      //只读方式打开
        QTextStream in(&file);
        int lineNums;
        in>>lineNums;
        int x1,y1,x2,y2;
        for(int i=0;i<lineNums;++i){
            in>>x1>>y1>>x2>>y2;
            lines.append(QLine(x1,y1,x2,y2));
        }
        int ellipseNums;
        in>>ellipseNums;
        for(int i=0;i<ellipseNums;++i){
            in>>x1>>y1>>x2>>y2;
            ellipses.append(QRect(QPoint(x1,y1),QPoint(x2,y2)));
        }
        int rectNums;
        in>>rectNums;
        for(int i=0;i<rectNums;++i){
            in>>x1>>y1>>x2>>y2;
```

```
                rects.append(QRect(QPoint(x1,y1),QPoint(x2,y2)));
            }
            int nums;
            in>>nums;
            int type;
            Shape *curShape;
            for(int i=0;i<nums;++i){
                in>>type;
                switch(type){
                case 0:
                    curShape=Line::read(in);
                    break;
                case 1:
                    curShape=Ellipse::read(in);
                    break;
                case 2:
                    curShape=Rectangle::read(in);
                    break;
                }
                shapes.append(curShape);
            }
            file.close();
        }
    }

    void CenterWidget::mouseReleaseEvent(QMouseEvent *e){
        p2=e->pos();
        beginDraw=false;
        if(p1==p2)
            return;
        switch(drawType){
        case 0:
            lines.append(QLine(p1,p2));
            break;
        case 1:
            ellipses.append(QRect(p1,p2));
            break;
        case 2:
            rects.append(QRect(p1,p2));
            break;
        }
        Shape *shape=nullptr;
        switch(drawType){
        case 0:
```

```
            shape=new Line(p1,p2);
            break;
        case 1:
            shape=new Ellipse(p1,p2);
            break;
        case 2:
            shape=new Rectangle(p1,p2);
            break;
        }
        shapes.append(shape);
        isModified=true;
        update();
    }

    void CenterWidget::paintEvent(QPaintEvent *){
        QPainter p(this);
    ~~    for(auto const& line:lines)~~
    ~~        p.drawLine(line);~~
    ~~    for(auto const& ellipse:ellipses)~~
    ~~        p.drawEllipse(ellipse);~~
    ~~    for(auto const& rect:rects)~~
    ~~        p.drawRect(rect);~~
        for(auto shape:shapes)
            shape->draw(&p);
    ~~    if(beginDraw==true){~~
    ~~        switch(drawType){~~
    ~~        case 0:~~
    ~~            p.drawLine(p1,p2);~~
    ~~            break;~~
    ~~        case 1:~~
    ~~            p.drawEllipse(QRect(p1,p2));~~
    ~~            break;~~
    ~~        case 2:~~
    ~~            p.drawRect(QRect(p1,p2));~~
    ~~            break;~~
    ~~        }~~
    ~~    }~~
        Shape *temp=nullptr;
        switch(drawType){
        case 0:
            temp=new Line(p1,p2);
            break;
        case 1:
            temp=new Ellipse(p1,p2);
```

```
            break;
        case 2:
            temp=new Rectangle(p1,p2);
            break;
        }
        temp->draw(&p);
        delete temp;
    }
```

6.2 实习任务

6.2.1 实习任务一

在演示案例的基础之上，添加菜单，点击后弹出颜色选择框，由用户设置当前画笔的颜色，此后将使用该颜色绘制所有形状，直到用户重新选择新的颜色。

6.2.2 实习任务二

在工具栏中添加颜色预览控件，显示当前用户所选择的颜色。