Глава 10

Виртуальные функции

В этой главе рассматривается следующий важный аспект C++: *виртуальные функции* (*virtual functions*). Виртуальные функции важны потому, что они используются для поддержки динамического полиморфизма (run-time polymorphism). Как вы знаете, в C++ полиморфизм поддерживается двумя способами. Во-первых, при компиляции он поддерживается посредством перегрузки операторов и функций. Во-вторых, во время выполнения программы он поддерживается посредством виртуальных функций. Здесь вы узнаете, как с помощью динамического полиморфизма можно повысить гибкость программ.

Основой виртуальных функций и динамического полиморфизма являются указатели на производные классы. Поэтому эта глава начинается с обсуждения указателей на производные классы.

10.1. Указатели на производные классы

Хотя в главе 4 довольно обстоятельно обсуждались указатели C++, одна их специфическая особенность до сих пор опускалась, поскольку она тесно связана с виртуальными функциями. Этой особенностью является следующее: указатель, объявленный в качестве указателя на базовый класс, также может использоваться, как указатель на любой класс, производный от этого базового. В такой ситуации представленные ниже инструкции являются правильными:

```
base *p; // указатель базового класса

base base_ob; // объект базового класса
derived derived ob; // объект производного класса

// Естественно, что указатель p может указывать
// на объект базового класса
p = &base_ob; // указатель p для объекта базового класса

// Кроме базового класса указатель p может указывать
//на объект производного класса

p = &derived_ob; // указатель p для объекта производного класса
```

Как отмечено в комментариях, указатель базового класса может указывать на объект любого класса, производного от этого базового и при этом ошибка несоответствия типов генерироваться не будет.

Для указания на объект производного класса можно воспользоваться указателем базового класса, при этом доступ может быть обеспечен только к тем объектам производного класса, которые были унаследованы от базового. Объясняется это тем, что базовый, указатель

"знает" только о базовом классе и ничего не знает о новых членах, добавленных в производном классе.

Указатель базового класса можно использовать для указания на объект производного класса, но обратный порядок недействителен. Указатель производного класса нельзя использовать для доступа к объектам базового класса. (Чтобы обойти это ограничение, можно использовать приведение типов, но на практике так действовать не рекомендуется.)

И последнее: запомните, что арифметика указателей связана с типом данных (т. е. классом), который задан при объявлении указателя. Таким образом, если указатель базового класса указывает, на объект производного класса, а затем инкрементируется, то он уже не будет указывать на следующий объект производного класса. Этот указатель будет указывать на следующий объект базового класса. Помните об этом.

Примеры

1. В этой короткой программе показано, как указатель базового класса может использоваться для доступа к объекту производного класса:

```
// Демонстрация указателя на объект производного класса
#include <iostream>
using namespace std;

class base {
   int x;
public:
   void setx(int i) { x = i; }
   int getx() { return x; }
};

class derived : public base {
   int y;
public:
   void sety(int i) { y = i; }
   int gety() {return y; }
};

int main()
{
   base *p;      // указатель базового класса
   base b_ob;    // объект базового класса
   derived d_ob; // объект производного класса

   // использование указателя p для доступа к объекту базового класса
   p = &b_ob;
   p->setx(10); // доступ к объекту базового класса
   cout << "Объект базового класса x: " << p->getx() << '\n';

   // использование указателя p для доступа к объекту производного класса
   p = &d_ob;   // указывает на объект производного класса
   p->setx(99); // доступ к объекту производного класса

   // т.к. p нельзя использовать для установки y, делаем это напрямую
   d_ob.sety(88);
   cout << "Объект производного класса x: " << p->getx() << ' ';
   cout << "Объект производного класса y: " << d_ob.gety() << '\n';
```

```
  return 0;
}
```

Нет смысла использовать указатели на объекты базового класса так, как показано в этом примере. Однако в следующем разделе вы увидите, почему для объектов производного класса столь важны указатели на объекты базового класса.

10.2. Знакомство с виртуальными функциями

Виртуальная функция (*virtual function*) является членом класса. Она объявляется внутри базового класса и переопределяется в производном классе. Для того, чтобы функция стала виртуальной, перед объявлением функции ставится ключевое слово **virtual**. Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция переопределяется. По существу, виртуальная функция реализует идею "один интерфейс, множество методов", которая лежит в основе полиморфизма. Виртуальная функция внутри базового класса определяет *вид интерфейса* этой функции. Каждое переопределение виртуальной функции в производном классе определяет ее реализацию, связанную со спецификой производного класса. Таким образом, переопределение создает *конкретный метод*. При переопределении виртуальной функции в производном классе, ключевое слово **virtual** не требуется.

Виртуальная функция может вызываться так же, как и любая другая функция-член. Однако наиболее интересен вызов виртуальной функции через указатель, благодаря чему поддерживается динамический полиморфизм. Из предыдущего раздела вы знаете, чти указатель базового класса можно использовать в качестве указателя на объект производного класса. Если указатель базового класса ссылается на объект производного класса, который содержит виртуальную функцию и для которого виртуальная функция вызывается через этот указатель, то компилятор определяет, какую версию виртуальной функции вызвать, основываясь при этом на *типе объекта, на который ссылается указатель*. При этом определение конкретной версии виртуальной функции имеет место не в процессе компиляции, а в процессе выполнения программы. Другими словами, тип объекта, на который ссылается указатель, и определяет ту версию виртуальной функции, которая будет выполняться. Поэтому, если два или более различных класса являются производными от базового, содержащего виртуальную функцию, то, если указатель базового класса ссылается на разные объекты этих производных классов, выполняются различные версии виртуальной функции. Этот процесс является реализацией принципа динамического полиморфизма. Фактически, о классе, содержащем виртуальную функцию, говорят как о *полиморфном классе (polymorphic class).*

Примеры

1. Рассмотрим короткий пример использования виртуальной функции:

```
// Простой пример использования виртуальной функции
#include <iostream>
using namespace std;

class base {
public:
  int i;
  base(int x) { i = x; }
  virtual void func()
```

```
  {
    cout << "Выполнение функции func() базового класса: ";
    cout << i << '\n';
  }
};

class derived1 : public base {
public:
  derived1(int x) : base(x) { }
  void func()
  {
     cout << "Выполнение функции func() класса derived1: ";
     cout << i * i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) { }
  void func()
  {
    cout << "Выполнение функции func() класса derived2: ";
    cout << i + i << '\n';
  }
};

int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);

  p = &ob;
  p->func(); // функция func() класса base

  p = &d_ob1;
  p->func(); // функция func() производного класса derived1

  p = &d_ob2;
  p->func(); // функция func() производного класса derived2

  return 0;
}
```

После выполнения программы на экране появится следующее:

```
Выполнение функции func() базового класса: 10
Выполнение функции func()  класса derived1: 100
Выполнение функции func()  класса derived2: 20
```

Переопределение виртуальной функции внутри производного класса может показаться похожим на перегрузку функций. Однако эти два процесса совершенно различны. Во-первых, перегружаемая функция должна отличаться типом и/или числом параметров, а переопределяемая виртуальная функция должна иметь точно такой же тип параметров, то же их число, и такой же тип возвращаемого значения. (На самом деле, если при переопределении виртуальной функции вы изменяете число или тип параметров, она просто становится перегружаемой функцией и её виртуальная природа теряется.) Далее, виртуальная функция должна быть членом класса. Это не относится к перегружаемым функциям. Кроме этого, если деструкторы могут быть виртуальными, то конструкторы нет.

Чтобы подчеркнуть разницу между перегружаемыми функциями и переопределяемыми виртуальными функциями, для описания переопределения виртуальной функции используется термин *подмена (overriding).*

В рассмотренном примере создаётся три класса. В классе **base** определяется виртуальная функция **func()**. Затем этот класс наследуется двумя производными классами: **derived1** и **derived2**. Каждый из этих классов переопределяет функцию **func()** по-своему. Внутри функции **main()** указатель базового класса **p** поочерёдно ссылается на объекты типа **base**, **derived1** и **derived2**. Первым указателю **p** присваивается адрес объекта **ob** (объекта типа **base**). При вызове функции **func()** через указатель **p** используется её версия из класса **base**. Следующим указателю **p** присваивается адрес объекта **d_ob1** и функция **func()** вызывается снова. Поскольку версия вызываемой виртуальной функции определяется типом объекта, на который ссылается указатель, то вызывается та версия функции, которая переопределяется в классе **derived1**. Наконец, указателю **p** присваивается адрес объекта **d_ob2**, и снова вызывается функция **func()**. При этом выполняется та версия функции **func()**, которая определена внутри класса **derived2**.

Ключевым для понимания предыдущего примера является тот факт, что, во-первых, тип адресуемого через указатель объекта определяет вызов той или иной версии подменяемой виртуальной функции, во-вторых, выбор конкретной версии происходит уже в процессе выполнения программы.

2. Виртуальные функции имеют иерархический порядок наследования. Кроме того, если виртуальная функция *не* подменяется в производном классе. То используется версия функции, определённая в базовом классе. Например, ниже приведена слегка изменённая версия предыдущей программы:

```
// Иерархический порядок виртуальных функций
#include <iostream>
using namespace std;

class base {
public:
   int i;
   base(int x) { i = x; }
   virtual void func()
   {
      cout << "Выполнение функции func() базового класса: ";
      cout << i << '\n';
   }
};

class derived1 : public base {
public:
   derived1(int x) : base(x) { }
   void func()
   {
      cout << "Выполнение функции func() класса derived1: ";
      cout << i * i << '\n';
   }
};

class derived2 : public base {
public:
   derived2(int x) : base(x) { }
   // в классе derived2 функция func() не подменяется
};
```

```
int main()
{
   base *p;
   base ob(10);
   derived1 d_ob1(10);
   derived2 d_ob2(10);
   p = &ob;
   p->func(); // функция func() базового класса
   p = &d_ob1;
   p->func(); // функция func() производного класса derived1
   p = &d_ob2;
   p->func(); // функция func() базового класса
   return 0;
}
```

После выполнения программы на экране появится следующее:

```
Выполнение функции func() базового класса: 10
Выполнение функции func()  класса derived1: 100
Выполнение функции func() базового класса : 10
```

В этой программе в классе **derived2** функция **func()** не подменяется. Когда указателю **p** присваивается адрес объекта **d_ob2** и вызывается функция **func()**, используется версия функции из класса **base**. Поскольку она следующая в иерархии классов. Обычно, если виртуальная функция не определена в производном классе, используется её версия из базового класса.

3. В следующем примере показано, как случайные события во время работы программы влияют на вызываемую версию виртуальной функции. Программа выбирает между объектами **d_ob1** и **d_ob2** на основе значений, возвращаемых стандартным генератором случайных чисел **rand()**. Запомните, выбор конкретной версии функции **func()** происходит во время работы программы. (Действительно, при компиляции этот выбор сделать невозможно, поскольку он основан на значениях, которые можно получить только во время работы программы.)

```
/* В этом примере показана работа виртуальной функции при наличии случайных событий во время выполнения программы.
*/
#include <iostream>
#include <cstdlib>
using namespace std;

class base {
public:
   int i;
   base(int x) { i = x; }
   virtual void func()
   {
      cout << "Выполнение функции func() базового класса: ";
      cout << i << '\n';
   }
};

class derived1 : public base {
public:
   derived1(int x) : base(x) { }
   void func()
   {
```

```
    cout << "Выполнение функции func() класса derived1: ";
    cout << i * i << '\n';
  }
};

class derived2 : public base {
public:
  derived2(int x) : base(x) { }
  void func()
  {
    cout << "Выполнение функции func() класса derived2: ";
    cout << i + i << '\n';
  }
};

int main()
{
  base *p;
  derived1 d_ob1(10);
  derived2 d_ob2(10);
  int i, j;

  for(i=0; i<10; i++) {
    j = rand();
    if((j%2)) p = &d_ob1; // если число нечетное
                // использовать объект d_ob1
    else p = &d_ob2;    // если число четное
                // использовать объект d_ob2
    p->func();          // вызов подходящей версии функции
  }

  return 0;
}
```

4. Теперь более реальный пример использования виртуальной функции. В этой программе создаётся исходный базовый класс **area**, в котором сохраняются две размерности фигуры. В нём также объявляется виртуальная функция **getarea()**, которая, при её подмене в производном классе, возвращает площадь фигуры, вид которой задаётся в производном классе. В этом случае определение функции **getarea()** внутри базового класса задаёт интерфейс. Конкретная реализация остаётся тем классам, которые наследуют класс **area**. В этом примере рассчитывается площадь треугольника и прямоугольника.

```
// Использование виртуальной функции для определения интерфейса
#include <iostream>
using namespace std;

class area {
  double dim1, dim2; // размеры фигуры
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea()
  {
```

```
        cout << "Вам необходимо подменить эту функцию\n";
        return 0.0;
    }
};

class rectangle : public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle : public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    rectangle r;
    triangle t;
    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);
    p = &r;
    cout << "Площадь прямоугольника: " << p->getarea() << '\n';
    p = &t;
    cout << "Площадь треугольника: " << p->getarea() << '\n';
    return 0;
}
```

Обратите внимание, что определение **getarea()** внутри класса **area** является только "заглушкой" и, в действительности, не выполняет никаких действий. Поскольку класс **area** не связан с фигурой конкретного типа, то нет значимого определения, которое можно дать функции **getarea()** внутри класса area. При этом, для того чтобы нести полезную нагрузку, функция **getarea()** должна быть переопределена в производном классе. В следующем разделе вы узнаете об этом подробнее.

10.3. Дополнительные сведения о виртуальных функциях

Как показано в примере 4 предыдущего раздела, иногда, когда виртуальная функция объявляется в базовом классе, она не выполняет никаких значимых действий. Это вполне обычная ситуация, поскольку часто в базовом классе законченный тип данных не определяется. Вместо этого в нем просто содержится базовый набор функций-членов и переменных, для которых в производном классе определяется все недостающее. Когда в виртуальной функции базового класса отсутствует значимое действие, в любом классе производном от этого базового, такая функция *обязательно* должна быть переопределена.

Для реализации этого положения в C++ поддерживаются та называемые *чистые виртуальные функции (pure virtual function).*

Чистые виртуальные функции не определяются в базовом классе. Туда включаются только прототипы этих функций. Для чистой виртуальной функции используется такая основная форма:

virtual ***тип имя_функция(список_параметров) = 0;***

Ключевой частью этого объявления является приравнивание функции нулю. Это сообщает компилятору, что в базовом классе не существует тела функции. Если функция задается как чистая виртуальная, это предполагает, что она обязательно должна подменяться в *каждом* производном классе. Если этого нет, то при компиляции возникнет ошибка. Таким образом, создание чистых виртуальных функций — это путь, гарантирующий, что производные классы обеспечат их переопределение.

Если класс содержит хотя бы одну чистую виртуальную функцию, то о нём говорят как об *абстрактном классе (abstract class).* Поскольку в абстрактном классе содержится, по крайней мере, одна функция, у которой отсутствует тело функции, технически такой класс неполон, и ни одного объекта этого класса создать нельзя. Таким образом, абстрактные классы могут быть только наследуемыми. Они никогда не бывают изолированными. Важно понимать, однако, что по-прежнему можно создавать указатели абстрактного класса, благодаря которым достигается динамический полиморфизм. (Также допускаются и ссылки на абстрактный класс.)

Если виртуальная функция наследуется, то это соответствует ее виртуальной природе. Это означает, что если производный класс используется в качестве базового для другого производного класса, то виртуальная функция может подменяться в последнем производном классе (так же, как и в первом производном классе). Например, если базовый класс B содержит виртуальную функцию **f()**, и класс **D1** наследует класс **B,** а класс **D2** наследует класс **D1**, тогда функция **f()** может подменяться как в классе **D1,** так и в классе **D2.**

Примеры

1. Здесь представлена несколько усовершенствованная версия программы, казанной в примере 4 предыдущего раздела. В этой версии программы в зовом классе **area** функция **getarea()** объявляется как чистая виртуальная функция.

```
// Создание абстрактного класса
#include <iostream>
using namespace std;

class area {
  double dim1, dim2; // размеры фигуры
public:
  void setarea(double d1, double d2)
  {
    dim1 = d1;
    dim2 = d2;
  }
```

```
  void getdim(double &d1, double &d2)
  {
    d1 = dim1;
    d2 = dim2;
  }
  virtual double getarea() = 0; // чистая виртуальная функция
};

class rectangle : public area {
public:
  double getarea()
  {
    double d1, d2;
    getdim(d1, d2);
    return d1 * d2;
  }
};

class triangle : public area {
public:
  double getarea()
  {
    double d1, d2;
    getdim(d1, d2);
    return 0.5 * d1 * d2;
  }
};

int main()
{
  area *p;
  rectangle r;
  triangle t;
  r.setarea(3.3, 4.5);
  t.setarea(4.0, 5.0);
  p = &r;
  cout << "Площадь прямоугольника: " << p->getarea() << '\n';
  p = &t;
  cout << "Площадь треугольника: " << p->getarea() << '\n';
  return 0;
}
```

Теперь то, что функция **getarea()** является чистой виртуальной, гарантирует ее обязательную подмену в каждом производном классе.

2. В следующей программе показано, как при наследовании сохраняется виртуальная природа функции:

```
/* Виртуальная функция при наследовании сохраняет свою виртуальную природу
*/
#include <iostream>
using namespace std;

class base {
public:
   virtual void func()
   {
      cout << "Выполнение функции func() базового класса \n";
   }
};

class derived1 : public base {
public:
   void func()
   {
      cout << "Выполнение функции func() класса derived1\n";
   }
};

// Класс derived1 наследуется классом derived2
class derived2 : public derived1 {
public:
   void func()
   {
      cout << " Выполнение функции func() класса derived2\n";
   }
};
int main()
{
   base *p;
   base ob;
   derived1 d_ob1;
   derived2 d_ob2;
   p = &ob;
   p->func(); // функция func() базового класса
```

```
  p = &d_ob1;
  p->func(); // функция func() производного класса derived1
  p = &d_ob2;
  p->func(); // функция func() производного класса derived2
  return 0;
}
```

В этой программе виртуальная функция **func()** сначала наследуется классом **derived1,** в котором она подменяется. Далее класс **derived1** наследуется классом **derived2.** В классе **derived2** функция **func()** снова подменяется.

Поскольку виртуальные функции являются иерархическими, то если бы в 'классе **derived2** функция **func()** не подменялась, при доступе к объекту **d_ob2** использовалась бы переопределенная в классе **derived1** версия функции **func().** Если бы функция **func()** не подменялась ни в классе **derived1,** ни в классе **derived2,** то все ссылки на функцию **func()** относились бы к ее определению в классе **base.**

10.4. Применение полиморфизма

Теперь, когда вы знаете, как использовать виртуальные функции для реализации динамического полиморфизма, самое время рассмотреть, зачем это нужно. Как уже много раз в этой книге отмечалось, полиморфизм является процессом, благодаря которому общий интерфейс применяется к двум или более схожим (но технически разным) ситуациям, т. е. реализуется философия "один интерфейс, множество методов". Полиморфизм важен потому, что может сильно упростить сложные системы. Один хорошо определенный интерфейс годится для доступа к некоторому числу разных, но связанных по смыслу действий, и таким образом устраняется искусственная сложность. Уточним: полиморфизм позволяет сделать очевидной логическую близость схожих действий; поэтому программа становится легче для понимания и сопровождения. Если связанные действия реализуются через общий интерфейс, вам нужно гораздо меньше помнить.

Имеются два термина, которые часто ассоциируются с объектно-ориентированным профаммированием вообще и с C++ в частности. Этими терминами являются *раннее связывание (early binding)* и *позднее связывание (late binding).* Важно понимать, что означают указанные термины. Раннее связывание относится к событиям, о которых можно узнать в процессе компиляции. Особенно это касается вызовов функций, которые настраиваются при компиляции. Функции раннего связывания — это "нормальные" функции, перегружаемые функции, невиртуальные функции-члены и дружественные функции. При компиляции функций этих типов известна вся необходимая для их вызова адресная информация. Главным преимуществом раннего связывания (и доводом в пользу его широкого использования) является то, что оно обеспечивает высокое быстродействие программ. Определение нужной версии вызываемой функции во время компиляции программы — это самый быстрый метод вызова функций. Главный недостаток — потеря гибкости.

Позднее связывание относится к событиям, которые происходят в процессе выполнения программы. Вызов функции позднего связывания — это вызов, при котором адрес вызываемой функции до запуска программы неизвестен. В C++ виртуальная функция является объектом позднего связывания. Если доступ к виртуальной функции

осуществляется через указатель базового класса, то в процессе работы программа должна определить, на какой тип объекта он ссылается, а затем выбрать, какую версию подменяемой функции выполнить. Главным преимуществом позднего связывания является гибкость во время работы программы. Ваша программа может легко реагировать на случайные события. Его основным недостатком является то, что требуется больше действий для; вызова функции. Это обычно делает такие вызовы медленнее, чем вызовы функций раннего связывания.

В зависимости от нужной эффективности, следует принимать решение, когда лучше использовать раннее связывание, а когда — позднее.

1. Ниже представлена программа, которая иллюстрирует принцип "один интерфейс, множество методов". В ней определен исходный базовый класс связанного списка целых. Интерфейс списка определяется с помощью чистых виртуальных функций **store() и retrieve().** Для хранения значения в списке вызывается функция **store().** Для выборки значения из списка вызывается функция **retrieve().** В базовом классе **list** для выполнения этих действий никакого встроенного метода не задается. Вместо этого в каждом производном классе явно определяется, какой тип списка будет поддерживаться. В программе реализованы списки двух типов: очередь и стек. Хотя способы работы с этими двумя списками совершенно различны, для доступа к каждому из них применяется один и тот же интерфейс. Вам следует тщательно изучить эту программу.

```
// Демонстрация виртуальных функций
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class list {
public:
   list *head; // указатель на начало списка
   list *tail; // указатель на конец списка
   list *next; // указатель на следующий элемент списка
   int num;    // число для хранения

   list() { head = tail = next = NULL; }
   virtual void store(int i) = 0;
   virtual int retrieve() = 0;
};

// Создание списка типа очередь
class queue : public list {
public:
   void store(int i);
   int retrieve();
};

void queue::store(int i)
{
   list *item;

   item = new queue;
   if(!item) {
```

```
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item->num = i;

    // добавление элемента в конец списка
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
    if(!head) head = tail;
}

int queue::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
        return 0;
    }

    // удаление элемента из начала списка
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

// Создание списка типа стек
class stack : public list {
public:
    void store(int i);
    int retrieve();
};

void stack::store(int i)
{
    list *item;

    item = new stack;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item->num = i;

    // добавление элемента в начало списка
    if(head) item->next = head;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
```

```
    list *p;

    if(!head) {
      cout << "Список пуст\n";
      return 0;
    }

    // удаление элемента из начала списка
    i = head->num;
    p = head;
    head = head->next;
    delete p;
    return i;
  }

  int main()
  {
    list *p;

    // демонстрация очереди
    queue q_ob;
    p = &q_ob; // указывает на очередь
    p->store(1);
    p->store(2);
    p->store(3);
    cout << "Очередь : ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();
    cout << '\n';
    // демонстрация стека
    stack s_ob;
    p = &s_ob; // указывает на стек
    p->store(1);
    p->store(2);
    p->store(3);
    cout << "Стек: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();
    cout << '\n';
    return 0;
}
```

2. Функция **main()** в программе со списками только иллюстрирует работу классов. Однако для изучения динамического полиморфизма попробуйте использовать в предыдущей программе следующую функцию **main():**

```
int main (){
list *p;
stack s_ob;
queue q_ob;
char ch;
int i;
```

```
    for(i=0; i<10; i++) (
    cout < "Стек или Очередь? (С/О): ";
    cin > ch;
    ch = tolower(ch);
    if(ch=='o') p = &q_ob;
    else p = &s_ob; .
    p->store(i);
    }
    cout < "Введите К для завершения работы\п";
    for(;;) (
    cout < "Извлечь элемент из Стека или Очереди? (С/О):
    cin > ch;
    ch = tolower(chit-
    if (ch=='K') break;
    if(ch=='o') p = &q_ob;
    else p = &s_ob;
    cout < p->reteieve() < '\n';
}
    cout < ' \n';
return 0;
```

Функция **main()** показывает, как случайные события, возникающие при выполнении программы, могут быть легко обработаны, если использовать виртуальные функции и динамический полиморфизм. В программе выполняется цикл **for** от 0 до 9. В течение каждой итерации предлагается выбор типа списка — стек или очередь. В соответствии с ответом, базовый указатель **p** устанавливается на выбранный объект (очередь или стек), и это значение запоминается. После завершения цикла начинается другой цикл, в котором предлагается выбрать список для извлечения запомненного значения. В соответствии с ответом пользователя выбирается число из указанного списка.

Несмотря на то, что этот пример достаточно прост, он позволяет, понять, как полиморфизм может упростить программу, которой необходимо реагировать на случайные события. Например, операционная система Windows взаимодействует с программой посредством сообщений. Эти сообщения генерируются как случайные, и ваша программа должна как-то реагировать на каждое получаемое сообщение. Одним из возможных способов обработки этих сообщений и является использование чистых виртуальных функций.