

ГЛАВА 21. НАСЛЕДОВАНИЕ

21.1. Введение в наследование

Процесс **наследования** позволяет создать иерархию классов таким образом, что один класс может приобретать свойства другого класса и добавлять к ним свои собственные свойства.

Когда один класс наследуется другим, тот класс, который наследуется, называется **базовым классом**. Наследующий класс называется **производным классом**. Процесс наследования начинается с задания базового класса, который определяет все те качества, которые будут общими для всех производных от него классов. Базовый класс часто называют *предком*, производный – *потомком*.

Синтаксис наследования базового класса производным:

```
class имя_производного_класса: спецификатор_доступа
имя_базового_класса
{
    // объявление производного класса
};
```

Спецификатор доступа (рис. 21.1) определяет, как элементы базового класса наследуются производным классом.

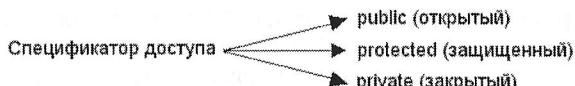


Рис. 21.1. Спецификаторы доступа при наследовании классов

Пример 21.1

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
// Базовый класс B:
class B { int i;
public:
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};
// Производный класс D будет наследовать все компоненты класса B
// таким образом, что открытые элементы B будут открытыми элементами
// D. Но к закрытым элементам B доступа из D не будет.
class D: public B
{ int j;
public:
    void set_j(int n) { j = n; }
```

```
int done() { if (j % 2 == 0) return j * get_i();
             else return j + get_i();
         }
};

int _tmain(int argc, _TCHAR* argv[])
{ D ob;
ob.set_i(10);           // загрузка i в классе B
ob.set_j(5);            // загрузка j в классе D
cout << ob.done();      // будет выведено число 15
_getch();
return 0;
}
```

Производный класс может иметь доступ к открытым элементам базового класса, в том числе вызывать его открытые функции-элементы.

Функция `get_i()`, являющаяся элементом базового класса B, вызывается из производного класса D без указания конкретного объекта, так как открытые элементы B становятся открытыми элементами D. Но для обращения к закрытой переменной `i` класса B необходимо обращение к открытой функции-элементу класса B, поскольку закрытые элементы базового класса остаются закрытыми для производных классов.

21.2. Простое наследование

Под простым наследованием понимают такое наследование, при котором один или несколько производных классов наследуют свойства одного базового класса (рис. 21.2).

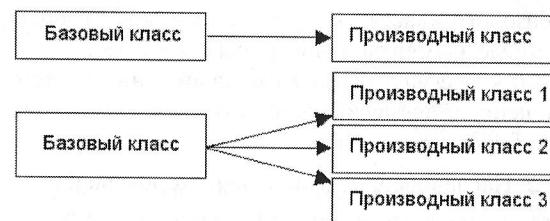


Рис. 21.2. Простое наследование

При простом наследовании используется стандартный синтаксис.

21.3. Режимы доступа к элементам базового класса

Режим доступа к элементам базового класса из производного определяется спецификатором доступа, используемым при наследовании (табл. 21.1).

Если при наследовании используется спецификатор доступа `public`,

то все открытые элементы базового класса становятся открытыми эле-

ГЛАВА 21. НАСЛЕДОВАНИЕ

21.1. Введение в наследование

Процесс **наследования** позволяет создать иерархию классов таким образом, что один класс может приобретать свойства другого класса и добавлять к ним свои собственные свойства.

Когда один класс наследуется другим, тот класс, который наследуется, называется **базовым классом**. Наследующий класс называется **производным классом**. Процесс наследования начинается с задания базового класса, который определяет все те качества, которые будут общими для всех производных от него классов. Базовый класс часто называют *предком*, производный – *потомком*.

Синтаксис наследования базового класса производным:

```
class имя_производного_класса: спецификатор_доступа
имя_базового_класса
{
    // объявление производного класса
};
```

Спецификатор доступа (рис. 21.1) определяет, как элементы базового класса наследуются производным классом.

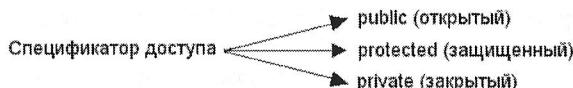


Рис. 21.1. Спецификаторы доступа при наследовании классов

Пример 21.1

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
// Базовый класс B:
class B { int i;
public:
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

// Производный класс D будет наследовать все компоненты класса B
// таким образом, что открытые элементы B будут открытами элементами
// D. Но к закрытым элементам B доступа из D не будет.
class D: public B
{ int j;
public:
    void set_j(int n) { j = n; }
```

```
int done() { if (j % 2 == 0) return j * get_i();
             else return j + get_i();
         }
};

int _tmain(int argc, _TCHAR* argv[])
{ D ob;
ob.set_i(10);           // загрузка i в классе B
ob.set_j(5);            // загрузка j в классе D
cout << ob.done();      // будет выведено число 15
_getch();
return 0;
}
```

Производный класс может иметь доступ к открытым элементам базового класса, в том числе вызывать его открытые функции-элементы.

Функция `get_i()`, являющаяся элементом базового класса B, вызывается из производного класса D без указания конкретного объекта, так как открытые элементы B становятся открытыми элементами D. Но для обращения к закрытой переменной `i` класса B необходимо обращение к открытой функции-элементу класса B, поскольку закрытые элементы базового класса остаются закрытыми для производных классов.

21.2. Простое наследование

Под простым наследованием понимают такое наследование, при котором один или несколько производных классов наследуют свойства одного базового класса (рис. 21.2).

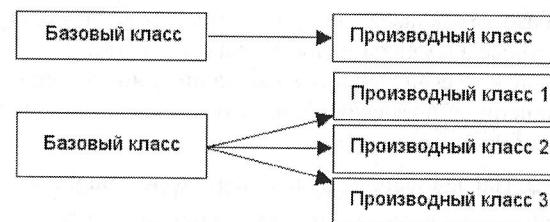


Рис. 21.2. Простое наследование

При простом наследовании используется стандартный синтаксис.

21.3. Режимы доступа к элементам базового класса

Режим доступа к элементам базового класса из производного определяется спецификатором доступа, используемым при наследовании (табл. 21.1).

Если при наследовании используется спецификатор доступа `public`, то все открытые элементы базового класса становятся открытыми эле-

ментами производного класса. При этом все закрытые элементы базового класса остаются закрытыми и недоступны из производного класса.

Таблица 21.1. Спецификаторы доступа

Доступ в базовом классе	Спецификатор доступа при наследовании	Доступ в производном классе
public protected private	public	public (не изменяется) protected (не изменяется) private (не изменяется)
public protected private	protected	protected protected (не изменяется) private (не изменяется)
public protected private	private	private private private (не изменяется)

При использовании спецификатора `private` все открытые элементы базового класса становятся закрытыми элементами производного класса, но они по-прежнему доступны для функций-членов производного класса. Все закрытые элементы базового класса остаются закрытыми и недоступны из производного класса.

Бывает необходимо, чтобы элементы базового класса, оставаясь закрытыми, были доступны для производного класса. Тогда используют спецификатор доступа `protected` (зашитченный). Спецификатор доступа `protected` эквивалентен `private` с единственным исключением: защищенные элементы базового класса доступны для элементов всех производных классов этого базового класса. За пределами базового или производных классов защищенные элементы недоступны (рис. 21.3).

Если базовый класс наследуется как защищенный (`protected`), то открытые и защищенные элементы базового класса становятся защищенными элементами производного класса.

Пример 21.2. Наследование со спецификатором `public`.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base { int i;
public:
    void set_i(int n) { i = n; }
    void show_i() { cout << i << endl; }
};

class Derived: public Base
{ int j;
public:
    void set_j(int n) { j = n; }
    void show_j() { cout << j << endl; }
};
```

```
};

int _tmain(int argc, _TCHAR* argv[])
{Derived ob; // создаем объект ob класса Derived
ob.set_i(10); // доступ к элементу класса Base
ob.set_j(20); // доступ к элементу класса Derived
ob.show_i(); // доступ к элементу класса Base
ob.show_j(); // доступ к элементу класса Derived
_getch();
return 0;
}
```

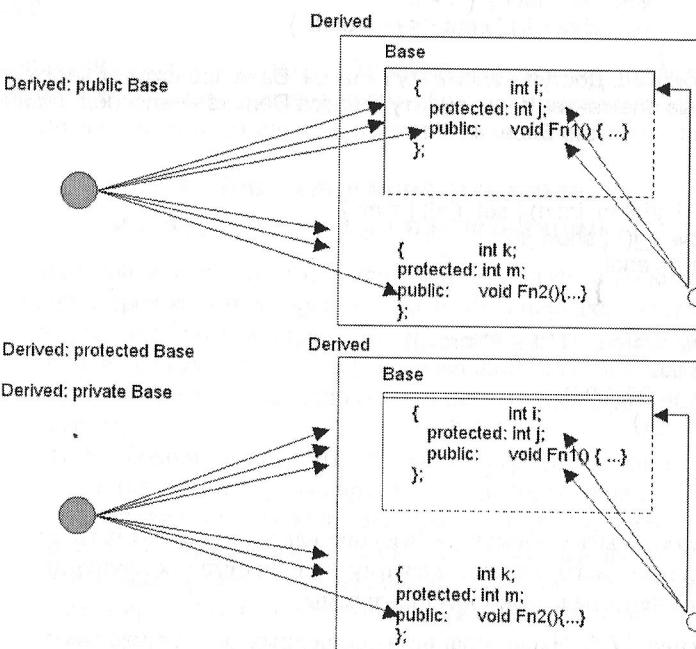


Рис. 21.3. Доступ к элементам базового и производного классов при наследовании с различными спецификаторами

Класс `Derived` наследует класс `Base` со спецификатором `public`, поэтому открытые элементы класса `Base` остаются открытыми и в классе `Derived` и доступными из любого места программы. Но при попытке непосредственно обратиться к закрытому элементу `i` класса `Base` возникает ошибка трансляции.

Пример 21.3

```
class Derived: public Base
{ int j;
public:
    void set_j(int n) { j = n; }
```

```
void show_i_j() { cout << i << ' ' << j << endl; } // ошибка доступа к i
};
```

Пример 21.4. Наследование со спецификатором private.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base { int i;
public:
    void set_i(int n) { i = n; }
    void show_i() { cout << i << endl; }
};

// Класс Derived. Доступ к элементу i класса Base происходит через его
// открытые элементы, а к элементу j класса Derived - непосредственно.
class Derived: private Base
{ int j;
public:
    void set_i_j(int m, int n) { set_i(n); j = m; }
    void show_i_j() { show_i(); }
    cout << j << endl;
};

int _tmain(int argc, _TCHAR* argv[])
{ Derived ob;
    ob.set_i_j(100, 200);
    ob.show_i_j();
    _getch();
    return 0;
}
```

Здесь открытые элементы базового класса стали закрытыми элементами производного класса, поэтому нет доступа к функциям set_i() и show_i() непосредственно из программы.

Пример 21.5. Наследование защищенных элементов класса как открытых.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base {protected:
    int i;
public:
    void set_i(int n) { i = n; }
};

// Создаем класс Derived. Доступ к защищенному элементу i
// класса Base происходит непосредственно из класса Derived.
class Derived: public Base
{ int j;
```

```
public:
    void set_j(int n) { j = n; }
    void show_i_j() { cout << i << ' ' << j << endl; }
};

int _tmain(int argc, _TCHAR* argv[])
{ Derived ob;
    ob.set_i(100);
    ob.set_j(200);
    ob.show_i_j();
    _getch();
    return 0;
}
```

Защищенный элемент i класса Base наследуется как открытый, поэтому он доступен для использования функциями-элементами класса Derived. Во всех остальных случаях он закрыт и недоступен.

21.4. Поведение конструкторов и деструкторов при наследовании

При наличии и у базового и у производного классов конструкторов и/или деструкторов конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке. То есть конструктор базового класса выполняется первым, конструктор производного класса – вторым; деструктор производного класса выполняется первым, деструктор базового класса – вторым.

При этом необходимо помнить, что ни конструкторы, ни деструкторы не наследуются (кроме случая виртуальных деструкторов).

Передача аргументов для конструктора производного или базового класса происходит следующим образом. При инициализации только в производном классе аргументы передаются как обычно. При необходимости передать аргумент конструктору базового класса отрабатывается следующий алгоритм:

- все необходимые аргументы базового и производного классов передаются конструктору производного класса;
- используя расширенную форму объявления конструктора производного класса, соответствующие аргументы передаются дальше в базовый класс.

Синтаксис передачи аргументов из производного класса в базовый имеет вид:

```
конструктор_производного_класса(аргументы) :
имя_базового_класса(аргументы)
{
    // определение конструктора производного класса
}
```

Базовый и производный классы могут использовать одни и те же аргументы. Кроме того, производный класс может не использовать полученные аргументы (все или часть), а просто передавать их в базовый класс.

Пример 21.6. Конструкторы производного и базового классов совместно используют один аргумент. При этом производный класс передает его в базовый.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base
{ int i;
public:
    Base(int n)
    { cout << "Конструктор базового класса" << endl;
        i = n;
    }
    ~Base() {cout << "Деструктор базового класса" << endl;}
    void show_i() { cout << i << endl; }
};
class Derived: public Base
{ int j;
public:
    Derived(int n): Base(n)
    { cout << "Конструктор производного класса\n";
        j = n;
    }
    ~Derived() {cout << "Деструктор производного класса\n";}
    void show_j() { cout << j << endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{ Derived ob(50);
    ob.show_i();
    ob.show_j();
    getch();
    return 0;
}
```

Обычно конструкторы базового и производного классов не используют один и тот же аргумент. Тогда конструктору производного класса передаются все аргументы, необходимые как для производного, так и для базового класса. Затем конструктор производного класса использует свои аргументы и передает базовому классу аргументы, предназначенные для него. При этом конструктору производного класса нет необходимости использовать аргументы, полученные им для передачи в базовый класс.

Пример 21.7. Передача одного аргумента в базовый класс через производный. Другой аргумент используется непосредственно конструктором производного класса.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base
{ int i;
public:
    Base(int n)
    { cout << "Конструктор базового класса" << endl;
        i = n;
    }
    ~Base()
    { cout << "Деструктор базового класса" << endl;}
    void show_i() { cout << i << endl; }
};
class Derived: public Base
{ int j;
public:
    Derived(int n, int m): Base(m)
    { cout << "Конструктор производного класса\n";
        j = n;
    }
    ~Derived()
    {cout << "Деструктор производного класса\n";}
    void show_j() { cout << j << endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{ Derived ob(50, 70);
    ob.show_i();
    ob.show_j();
    getch();
    return 0;
}
```

21.5. Множественное наследование

Существует два способа, позволяющие производному классу наследовать более чем один базовый класс.

21.5.1. Создание иерархии классов

Производный класс может использоваться в качестве базового класса для другого производного класса, создавая несколько уровней иерархии классов (рис. 21.4).

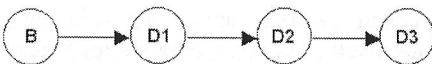


Рис. 21.4. Иерархия классов

В случае создания иерархической структуры конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке. При необходимости передачи аргументов каждый производный класс передает классу, являющемуся для него базовым, аргументы, предназначенные для него и для всех предшествующих классов.

Пример 21.8. Иерархия классов (рис. 21.5).

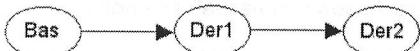


Рис. 21.5. Иерархия трех классов

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Bas { int i;
public:
    Bas(int n) { i = n; }
    int get_i() { return i; }
};

class Der1: public Bas
{ int j;
public:
    Der1(int n, int m): Bas(m) { j = n; }
    int get_j() { return j; }
};

class Der2: public Der1
{ int k;
public:
    Der2(int n, int m, int l): Der1(m, l) { k = n; }
    void show()
    { cout << get_i() << ' ' << get_j() << ' ' << k << endl; }
};

int _tmain(int argc, _TCHAR* argv[])
{ Der2 ob(50, 70, 90);
ob.show(); // на экран будет выведено: 90 70 50
_getch();
return 0;
}
  
```

21.5.2. Прямое наследование нескольких базовых классов

Производный класс может прямо наследовать более одного базового класса (рис. 21.6).

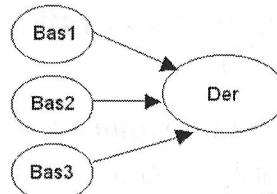


Рис. 21.6. Множественное наследование

В этом случае используется расширенное объявление:

```

class имя_производного_класса :
    спецификатор_доступа имя_базового_класса_1,
    спецификатор_доступа имя_базового_класса_2,
    ...
    спецификатор_доступа имя_базового_класса_N
{
    // определение класса
}
  
```

Спецификатор доступа может быть разным для каждого класса. Когда наследуется множество базовых классов, конструкторы выполняются слева направо, в том порядке, как заданы в объявлении производного класса. Деструкторы – в обратном порядке.

Передача параметров происходит через производный класс с использованием расширенной формы объявления конструктора производного класса:

```

конструктор_производного_класса(аргументы) :
    имя_базового_класса_1(аргументы),
    имя_базового_класса_2(аргументы),
    ...
    имя_базового_класса_N(аргументы)
{
    // определение конструктора производного класса
}
  
```

Пример 21.9. Непосредственное наследование сразу нескольких (двух) базовых классов (рис. 21.7).

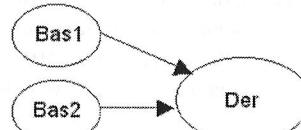


Рис. 21.7. Множественное наследование двух классов

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Bas1 { int i;
  
```

```

public:
    Bas1(int n) { i = n; }
    int get_i() { return i; }
};

class Bas2 { int j;
public:
    Bas2(int n) { j = n; }
    int get_j() { return j; }
};

class Der: public Bas1, public Bas2
{ int k;
public:
// здесь I и m передаются в Bas1 и Bas2 отдельно.
Der(int n, int m, int l): Bas1(l), Bas2(m)
{ k = n; }
void show()
{ cout << get_i() << ' ' << get_j() << ' ' << k << endl; }
};

int _tmain(int argc, TCHAR* argv[])
{ Der ob(50, 70, 90);
ob.show(); // на экран будет выведено: 90 70 50
_getch();
return 0;
}

```

Контрольные вопросы

1. В чем заключаются различия между наследованием со спецификаторами `public`, `protected`, `private`?
2. Что такое простое наследование?
3. Что такое множественное наследование?
4. Как реализовать иерархию классов?

ГЛАВА 22. ВИРТУАЛЬНЫЕ ФУНКЦИИ

22.1. Понятие о виртуальных функциях

Виртуальной называется функция, вызов которой зависит от типа объекта. Виртуальные функции пишутся так, чтобы объект определял, какую функцию необходимо вызвать во время выполнения программы.

Виртуальная функция является членом класса. Она объявляется внутри базового класса и переопределяется в производном классе. Для объявления функции как виртуальной используется ключевое слово `virtual`.

Деструкторы могут быть виртуальными, конструкторы – не могут.

Виртуальные функции необходимы для поддержки динамического полиморфизма (run-time polymorphism). В C++ полиморфизм поддерживается двумя механизмами:

- при компиляции – посредством перегрузки операторов и функций;
- во время выполнения программы – с помощью виртуальных функций.

Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция может переопределяться.

Каждое переопределение виртуальной функции в производном классе определяет ее реализацию, т. е. создает *конкретный метод*. При переопределении виртуальной функции в производном классе, ключевое слово `virtual` не требуется. Но несмотря на это, некоторые программисты явно объявляют функции виртуальными на каждом уровне иерархии, чтобы обеспечить ясность программы.

Обычно компиляторы при обработке виртуальных функций добавляют к каждому объекту скрытый элемент, который содержит указатель на массив адресов функций, называемый *таблицей виртуальных функций* или *vtbl*. Таблица содержит адреса виртуальных функций, объявленных для объектов этого класса.

Виртуальные функции могут вызываться как функции-элементы или через указатели. Виртуальная функция может объявляться с параметрами и иметь возвращаемое значение. В классе можно объявлять несколько виртуальных функций. Виртуальные функции могут быть открытыми, закрытыми и защищенными членами класса.

22.2. Указатели на базовый и производные классы

Указатель на базовый класс может ссылаться на объект этого класса или на объект любого класса, производного от этого базового. Указатель на производный класс может ссылаться на объекты производного класса и не может на объекты базового класса (рис. 22.1).

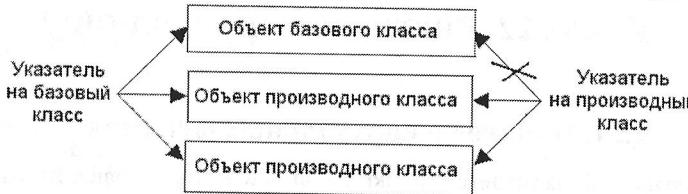


Рис. 22.1. Указатели на базовый и производный классы

Пример 22.1

```

base *p; // указатель базового класса
base b_ob; // объект базового класса
derived d_ob; // объект производного класса
...
p = &b_ob; // указатель p указывает на базовый класс –
// естественные действия
p = &d_ob; // указатель p указывает на производный класс –
// допустимые действия, при этом ошибка несоответствия
// типов не возникает
    
```

Для указания на объект производного класса можно воспользоваться указателем базового класса, но при этом доступ возможен лишь к тем элементам производного класса, которые были унаследованы от базового, так как указатель на базовый класс имеет данные только о базовом классе.

Указатель производного класса нельзя использовать для доступа к объектам базового класса.

Арифметика указателей связана с типом данных (т. е. классом), который задан при объявлении указателя. То есть если указатель базового класса указывает на объект производного класса, а затем инкрементируется, то он уже не будет указывать на следующий объект производного класса.

Пример 22.2

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base
{
public:
    virtual void show() { cout << "Base" << endl;}
};
class Derv1: public Base
{
public:
    void show() { cout << "Derv1" << endl;}
};
    
```

```

class Derv2: public Base
{ public:
    void show() { cout << "Derv2" << endl;}
};
int _tmain(int argc, _TCHAR* argv[])
{ Derv1 dv1;
    Derv2 dv2;
    Base *ptr; // указатель на базовый класс
    ptr = &dv1; // Derv1
    ptr->show(); // Derv1
    ptr = &dv2; // Derv2
    ptr->show(); // Derv2
    getch();
    return 0;
}
    
```

22.3. Виртуальные функции и наследование

Если несколько различных классов являются производными от базового, содержащего виртуальную функцию, то, если указатель базового класса ссылается на разные объекты этих производных классов, выполняются разные версии виртуальной функции. Этот процесс является *реализацией динамического полиморфизма*, а класс, содержащий виртуальную функцию, называется *полиморфным классом*.

Виртуальная функция может быть переопределена внутри производного класса. Это переопределение похоже на перегрузку функций. Но эти два процесса имеют принципиальные различия:

- перегружаемые функции должны различаться типом и/или количеством параметров, а переопределяемая виртуальная функция должна иметь одинаковую сигнатуру и возвращаемое значение для всех версий;
- виртуальная функция должна быть членом класса, а перегружаемая функция – не обязательно;
- конкретная версия перегружаемой функции выбирается на этапе компиляции, а конкретная версия виртуальной функции – на этапе выполнения.

Выбор версии виртуальной функции происходит на основе типа объекта, который передается ей в качестве аргумента – указателя this. Будучи однажды объявлена как виртуальная, функция сохраняет это свойство во всех переопределениях в производных классах.

Виртуальные функции имеют иерархический порядок наследования. Если виртуальная функция не подменяется (не переопределяется) в производном классе, то используется та версия функции, которая определена в базовом классе. То есть если в производном классе отсутствует описание виртуальной функции, то этот производный класс наследует описание виртуальной функции непосредственно из базового класса.

Пример 22.3

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Base
{ public:
    int i;
    Base(int x) {i = x;}
    virtual void virt_fn()
    { cout << "Execution virt_fn() of Base" << endl;
      cout << i << endl;
    }
};
class Derv1: public Base
{ public:
    Derv1(int x): Base(x) {}
    void virt_fn()
    { cout << "Execution virt_fn() of Derv1" << endl;
      cout << 2 * i * i * i << endl;
    }
};
class Derv2: public Base
{ public:
    // в классе Derv2 функция virt_fn()
    // не переопределяется:
    Derv2(int x): Base(x) {};
};
int _tmain(int argc, _TCHAR* argv[])
{ Base *B_Ptr;           // указатель на базовый класс
  Base ob(10);           // объект базового класса
  Derv1 dv1(10);         // объект производного класса
  Derv2 dv2(10);         // объект производного класса
  B_Ptr = &ob;
  B_Ptr -> virt_fn();    // virt_fn() базового класса напечатает 10
  B_Ptr = &dv1;
  B_Ptr -> virt_fn();    // virt_fn() производного класса напечатает 2000
  B_Ptr = &dv2;
  B_Ptr -> virt_fn();    // virt_fn() базового класса напечатает 10
  _getch();
  return 0;
}
```

22.4. Абстрактные классы и чисто виртуальные функции

Встречаются ситуации, когда в базовом классе законченный тип данных точно не определен. Вместо этого в нем может содержаться только базовый набор функций-элементов и переменных, для которых в про-

водном классе определяются все недостающие компоненты. Виртуальные функции в таких классах являются фиктивными и не выполняют никаких действий. Они имеют пустое тело в базовом классе, но в производных классах такие функции обязательно должны быть переопределены. В C++ для таких случаев введено понятие чисто виртуальных функций.

Чисто виртуальная функция – это виртуальная функция, тело которой не определено. Она используется для того, чтобы отложить выбор реализации функции. Это называется *отложенным методом*. Чисто виртуальная функция объявляется внутри базового класса с помощью синтаксической конструкции:

```
virtual тип имя_функции (список_параметров) = 0;
```

Класс, имеющий хотя бы одну чисто виртуальную функцию, называется *абстрактным классом*. Такой класс должен задавать основные общие свойства для своих производных классов, но сам не может использоваться для объявления объектов. Но он может применяться для объявления указателей, которым разрешен доступ к объектам, производным от абстрактного класса. Благодаря этому достигается динамический полиморфизм. Можно также создавать ссылки на абстрактный класс.

Если существует класс, производный от абстрактного класса и чисто виртуальная функция в нем не определена, то эта функция остается чисто виртуальной и в производном классе, а сам класс также является абстрактным. Бывают случаи, когда абстрактные классы образуют несколько верхних уровней иерархии. Такую структуру имеет, например, иерархия форм.

Пример 22.4. Создание и использование абстрактного класса.

```
#include "stdafx.h"
#include <iostream>
#include <math.h>
#include <conio.h>
using namespace std;
class area
{ double dim1, dim2;           // размеры фигуры
public:
  void setdim(double d1, double d2) { dim1 = d1; dim2 = d2; }
  void getdim(double &d1, double &d2) { d1 = dim1; d2 = dim2; }
  virtual double getarea()=0; //чисто виртуальная функция
};

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

```

};

class ring: public area
{ public:
    double getarea()
    { const double Pi = 3.14159;
      double d1, d2;
      getdim(d1, d2);
      return Pi * fabs (d1 * d1 - d2 * d2);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    area *Ptr;
    rectangle r;
    ring rn;
    r.setdim(3.5, 4.7);
    rn.setdim(4.0, 5.0);
    Ptr = &r;
    cout << "Square of rectangle = " << Ptr->getarea() << endl;
    Ptr = &rn;
    cout << "Square of ring = " << Ptr->getarea() << endl;
    _getch();
    return 0;
}

```

22.5. Виртуальные деструкторы

В отличие от конструкторов деструкторы могут быть виртуальными. Если объект удаляется явным образом, с использованием операции `delete` над указателем базового класса на объект, то вызывается деструктор базового класса данного объекта.

Обявление деструктора базового класса виртуальным приводит к тому, что все деструкторы производных классов становятся виртуальными, даже если они имеют имена, отличные от имени деструктора базового класса. Деструктор базового класса автоматически выполняется после деструктора производного класса.

Пример 22.5

```

#include "stdafx.h"
#include <iostream>
#include <math.h>
#include <conio.h>
#include <string>
using namespace std;
class Base
{ char *str1;
public:
    Base(const char *s) {str1 = _strup(s);}
    ~Base() { delete str1;
              cout << "Base удален" << endl;
    }
};

```

```

}

class Derived: public Base
{ char *str2;
public:
    Derived(const char *s1, const char *s2):Base(s1)
    {str2 = _strup(s2);}
    ~Derived() { delete str2;
                cout << "Derived удален" << endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "Russian");
    Base *pBase;
    pBase = new Derived("Дождь бросает монетки - ",
                        "он играет в чет-нечет.");
    delete pBase;
    _getch();
    return 0;
}

```

В результате работы программы на экран будет выведено
Base удален

т. е. деструктор производного класса вообще не вызывается. Для правильной работы программы следует объявить деструктор виртуальным, т. е. написать:

```

virtual ~Base() { delete str1;
                  cout << "Base удален" << endl;
}

```

Тогда в результате работы программы получим:

Derived удален
Base удален

т. е. порожденный класс удален корректно (пример 22.6).

Пример 22.6:

```

#include "stdafx.h"
#include <iostream>
#include <math.h>
#include <conio.h>
#include <string>
using namespace std;
class Base
{ char *str1;
public:
    Base(const char *s) {str1 = _strup(s);}
    virtual ~Base()
    { delete str1;
      cout << "Base удален" << endl;
    }
};

```

```

};

class Derived: public Base
{ char *str2;
public:
    Derived(const char *s1, const char *s2):Base(s1) {str2 = _strup(s2);}
    ~Derived() { delete str2;
        cout << "Derived удален" << endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian");
    Base *pBase;
    pBase = new Derived("Дождь бросает монетки - ",
                        "он играет в чет-нечет.");
    delete pBase;
    _getch();
    return 0;
}

```

В общем случае, чтобы быть уверенным в том, что объекты порожденных классов удаляются корректно, следует объявлять деструкторы в базовых классах виртуальными.

22.6. Виртуальные базовые классы

Базовые классы и классы, производные от них, могут образовывать довольно сложные иерархии, работа с которыми может привести к возникновению неопределенных ситуаций (рис. 22.2).

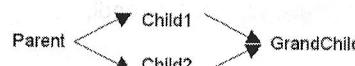


Рис. 22.2. Иерархия классов

В данной иерархии могут возникнуть проблемы, связанные с неоднозначностью, если класс GrandChild захочет получить доступ к членам класса Parent.

Пример 22.7

```

class Parent
{ protected:
    int dataParent;
};

class Child1: public Parent { };
class Child2: public Parent { };
class GrandChild: public Child1, public Child2
{public:
    int getdata()
    {return dataParent;} // ошибка, связанная с неоднозначностью
};

```

Неоднозначность возникает вследствие того, что каждый из порожденных классов Child1 и Child2 создает собственную копию класса Parent. Поэтому при обращении производного класса GrandChild к элементам базового класса непонятно, к какой именно копии будет происходить обращение.

Если объявить базовый класс виртуальным, неоднозначность устраняется, поскольку теперь копии объекта не создаются.

```

class Parent
{ protected:
    int dataParent;
};

class Child1: virtual public Parent { };
class Child2: virtual public Parent { };
class GrandChild: public Child1, public Child2
{ public:
    int getdata() { return dataParent; }
};

```

Теперь ошибок, связанных с неоднозначностью, нет.

Контрольные вопросы

- Что называется виртуальной функцией? Где и каким образом она объявляется?
- Для чего необходимы виртуальные функции?
- Что представляет собой динамический полиморфизм (run-time polymorphism), статический полиморфизм? С помощью каких механизмов они реализуются?
- Как создаются указатели на базовый и производные классы? Перечислите их основные свойства.
- Какую роль играют виртуальные функции при наследовании?
- Дайте определение абстрактных классов и чисто виртуальных функций.
- Как объявляется чисто виртуальная функция?
- Когда необходимы виртуальные базовые классы?

Практические задания

22

- Разработать программу с использованием наследования классов, реализующую классы:
 - графический объект;
 - круг (найти площадь);
 - квадрат (найти площадь).
 Используя виртуальные функции, не зная, с объектом какого класса вы работаете, выведите на экран его название и площадь.
- Разработать программу с использованием наследования классов, реализующую классы:

- точка;
- линия (длина);
- круг (площадь).

Используя виртуальные функции, не зная, с объектом какого класса вы работаете, выведите на экран его название и соответствующий размер.

3. Разработать программу с использованием наследования классов, реализующую классы:

- воин;
- пехотинец (винтовка);
- матрос (кортик).

Используя виртуальные функции, не зная, с объектом какого класса вы работаете, выведите на экран имя, возраст, вид оружия.

4. Разработать программу с использованием наследования классов, реализующую классы:

- человек;
- школьник (номер школы, класс);
- студент (название института, курс, факультет);
- инженер (название фирмы, должность, зарплата).

Используя виртуальные функции, не зная, с объектом какого класса вы работаете, выведите на экран полную информацию об объекте.

ГЛАВА 23. ОБЪЕКТЫ И ФАЙЛОВЫЕ ПОТОКИ. ТЕКСТОВЫЕ ФАЙЛЫ

23.1. Потоковый ввод-вывод дисковых файлов

Операции ввода-вывода в языке C++ осуществляются через потоки. Поток – это логическое устройство, выдающее и принимающее информацию [25]. Клавиатура и экран дисплея называются *стандартными потоками ввода-вывода* и интерпретируются как текстовые файлы.

Для работы с дисковыми файлами необходимо подключение заголовочного файла `<fstream>`, содержащего наборы специальных классов:

- `ifstream` – для ввода;
- `ofstream` – для вывода;
- `fstream` – для чтения и записи данных в один и тот же файл.

Чтобы получить возможность работать с дисковым файлом, нужно открыть его с указанием режима доступа (табл. 23.1), который определяется значением константы `open_mode` класса `ios`.

Таблица 23.1. Режимы доступа к элементам файлов

Режим доступа	Стандарт	Действие
<code>app</code>	Нет	Открывает файл для дозаписи
<code>ate(atend)</code>	Да	При открытии файла устанавливает файловый указатель на конец файла
<code>binary(bin)</code>	Да	Открыть файл в двоичном представлении
<code>in</code>	Да	Открыть файл для чтения (ввода)
<code>nocreate</code>	Нет	Если файл не существует, то новый файл не создается
<code>noreplace</code>	Нет	Если файл уже существует, файл не перезаписывается
<code>out</code>	Да	Открыть файл для записи (вывода)
<code>trunc</code>	Нет	Открывает и усекает существующий файл. Новая информация замещает существующую

Иерархия классов стандартных потоков показана на рис. 23.1.

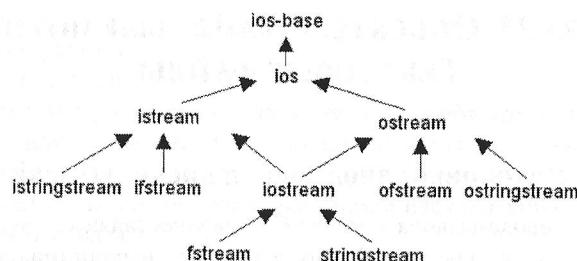


Рис. 23.1. Иерархия классов стандартных потоков [22]

Базовым классом иерархии является класс `ios`, в котором определены основные понятия потокового ввода-вывода.

23.2. Текстовые файлы

23.2.1. Создание и запись

Для создания текстового файла определяют объект класса `ofstream` и передают конструктору класса имя дискового файла в качестве первого параметра и режим доступа в качестве второго параметра:

```
ofstream out_file("Out.txt", ios::out);
```

Можно объявить константу, определяющую режим открытия файла, например:

```
const ios::open_mode=open_mode=ios::out | ios::app;
```

После того, как предпринималась попытка открыть файл, следует убедиться в том, что файл открыт и готов для записи (или перезаписи):

```
if (!out_file) { cerr << "Error: unable to write to Out.txt" << endl;
    exit(1);
}
```

Все сказанное верно и для файлов, открываемых для чтения (или входных файлов):

```
ifstream in_file("Input.txt", ios::in);
if (!in_file) { cerr << "Error: unable to open Input.txt" << endl;
    exit(1);
}
```

При работе с текстовыми файлами наиболее часто встречаются 4 действия:

- посимвольное чтение,
- посимвольная запись,

- построчное чтение,
- построчная запись.

23.2.2. Посимвольное чтение файла

Функция `get()`, которая является методом `istream`, применяется для посимвольного чтения текстового файла.

Пример 23.1. Посимвольное чтение файла и вывод его на экран.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{char sym;
ifstream in_file ("d:\\Input.txt", ios::in);
if (!in_file) { cerr << "Error input file" << endl;
    exit(1);
}
while (in_file) { in_file.get(sym);
    cout << sym;
}
cout << endl;
_getch();
return 0;
}
```

23.2.3. Посимвольная запись файла

Функция `put()`, которая является методом `ostream`, позволяет осуществлять посимвольную запись данных в текстовый файл.

Пример 23.2. Посимвольная запись данных в текстовый файл.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
string quote = "Зорко одно лишь сердце. Самого главного
глазами не увидишь. А. де Сент-Экзюпери";
/* в ранних версиях объявление строки quote выглядит так:
char *quote = "Зорко одно лишь сердце. Самого главного гла-зами не уви-
дишь. А. де Сент-Экзюпери"; */
ofstream out_file ("Out_file.txt", ios::out);
```

```

if (!out_file) { cerr << "Error output file"
    << endl;
    exit(1);
}
for (unsigned int i = 0; i < quote.size(); i++) out_file.put(quote[i]);
/* в ранних версиях длина строки quote определяется по-другому:
for (int i = 0; i < strlen(quote) + 1; i++) out_file.put(quote[i]); */
cout << "Конец записи" << endl;
_getch();
return 0;
}

```

23.2.4. Построчное чтение файла

Обычно построчное чтение и запись файлов работают быстрее символьных действий. Для чтения строки из файла воспользуемся функцией `getline()`, которая является методом класса `ifstream`. Функция читает строку (в том числе и разделители), пока не встретит символ новой строки `\n`, помещая ее в буфер. Имя буфера служит первым аргументом функции. Максимальный размер буфера задается как второй аргумент функции.

Пример 23.3. Построчное чтение файла.

```

#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <string>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{const int LEN = 80;
char BUF[LEN];
ifstream in_file("d:\\Input.txt", ios::in);
if (!in_file) { cerr << "Error input file" << endl;
    exit(1);
}
while (in_file)
{ in_file.getline(BUF,LEN);
    cout << BUF << endl;
}
_getch();
return 0;
}

```

23.2.5. Построчная запись файла

Пример 23.4. Построчная запись текста в файл.

```

#include "stdafx.h"
#include <fstream>
#include <iostream>

```

```

#include <string>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
ofstream out_file ("Out_file.txt", ios::out);
if (!out_file) { cerr << "Error output file" << endl;
    exit(1);
}
out_file << "Я не знаю, где встретиться\n";
out_file << "Нам придется с тобой,\n";
out_file << "Глобус крутится-вертится,\n";
out_file << "Словно шар голубой\n";
_getch();
return 0;
}

```

Записываемые строки являются не объектами класса `string`, а строками типа `*char`, которые завершаются символом `\n`.

23.3. Признак конца файла

Признак конца файла обычно приходится искать в файлах, открытых для чтения. Этот признак устанавливается в тот момент, когда в файле не осталось больше данных, которые можно из него считать.

Признак конца файла анализируется в выражении вида

```
while (!in_file.eof()) { ... }
```

Для этой цели нельзя пользоваться циклом `do { ... } while (!in_file.eof())`, поскольку файл может оказаться пустым, а цикл с последующим условием выполняется по крайней мере один раз. Этого единственного прохода цикла при чтении из пустого файла достаточно, чтобы вызвать ошибку выполнения программы.

Однако проверка на конец файла не анализирует ошибок, которые могут встретиться в процессе чтения файла.

Для проверки как признака конца файла, так и наличия ошибок при его чтении пользуются оператором цикла с предварительным условием, в котором условие выхода из цикла выглядит следующим образом:

```
while (in_file.good()) { ... }
```

Оператор цикла

```
while (in_file) { ... }
```

выполняется до тех пор, пока нет ошибок, в том числе и признака конца файла (EOF), который в этом случае тоже интерпретируется как ошибка чтения из файла.

Контрольные вопросы

1. Что такое поток?
2. Что представляет собой файловый указатель?
3. Перечислить режимы доступа к файлу.
4. Как открыть и как закрыть файл?

Практические задания 23

1. Дан символьный файл f. Подсчитать число вхождений в файл каждой из букв a, b, c, d, e, f. Результат вывести в файл g в виде таблицы с комментариями.
2. Дан файл f, компоненты которого являются целыми числами. Записать в файл g все четные числа исходного файла, в файл h – все нечетные. Порядок следования чисел сохраняется. Записать в файл g и h комментарии.
3. Дан текстовый файл, содержащий программу на языке С. Проверить эту программу на соответствие числа открывающих и закрывающих фигурных скобок.
4. Дан символьный файл f. Найти и записать в файл g самое длинное слово файла f, снабдив его комментарием.
5. Дан файл f. Создать два файла, записав в первый из них все четные числа, расположив их в порядке возрастания, а во второй – все нечетные, расположив их в порядке убывания.
6. Дан текстовый файл f. Переформатировать исходный файл, разделяя его на строки так, чтобы каждая строка содержала столько символов, сколько содержит самая короткая строка исходного файла.
7. Дан текстовый файл f. Определить, являются ли первые два символа цифрами, и если да, то четно ли это число. Записать его в файл g, если оно четно и в h, если оно нечетно.
8. Дан текстовый файл f. Создать новый файл g и переписать в него исходный в обратном порядке, разделив элементы его пробелами.

ГЛАВА 24. ОБЪЕКТЫ И ДВОИЧНЫЕ ФАЙЛЫ

24.1. Сохранение данных в двоичных файлах

24.1.1. Сохранение в двоичных файлах данных стандартных типов

Двоичные файлы более компактны и в некоторых случаях более удобны для обработки.

Для того чтобы открыть двоичный файл, необходимо задать режим доступа ios::binary (в некоторых компиляторах C++ – ios::bin).

Для создания выходного файла создают объект

```
ofstream out_fil("Outfil.dat", ios::out | ios::binary);
if (!out_fil) { cerr << "Error: Outfil.dat" << endl;
    exit(1);
}
```

Для того, чтобы открыть существующий двоичный файл для чтения, нужно создать объект

```
ifstream in_fil("Infil.dat", ios::in | ios::binary);
if (!in_fil) { cerr << "Error: Infil.dat" << endl;
    exit(2);
}
```

К сожалению, созданные объекты in_fil и out_fil не слишком приспособлены для работы с двоичными файлами и требуют некоторых дополнительных действий, необходимых для корректной работы.

Пример 24.1. Запись значения типа double в двоичный файл.

```
#include "stdafx.h"
#include <iostream>
#include <iostream>
#include <conio.h>
using namespace std;
class bin_outstream: public ostream
{public:
    bin_outstream(const char *fn): ostream(fn, ios::out | ios::binary) {}
    void writeOurDate(const void*, int);
    ostream &operator<<(double d)
    { writeOurDate(&d, sizeof(d));
        return *this;
    }
};
int _tmain(int argc, _TCHAR* argv[])
{bin_outstream bin_out("B_out.dat");
if (!bin_out)
    { cerr << "Unable to write to B_out.dat" << endl;
```

```

    exit(1);
}
double d = 5.252;
bin_out << d;
bin_out << d*d;
d = 5.2E-5;
bin_out << d;
_getch();
return 0;
}
void bin_outstream::writeOurDate(const void *Ptr, int len)
{
if (!Ptr) return;
if (len <= 0) return;
write((char*)Ptr, len);
}

```

Пример 24.2. Чтение значений типа double из двоичного файла.

```

#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <conio.h>
using namespace std;
class bin_instream: public ifstream
public:
bin_instream(const char *fn): ifstream(fn, ios::in | ios::binary) {}
void readOurDate(void*, int);
bin_instream &operator>>(double &d)
{ readOurDate(&d, sizeof(d));
return *this;
}
int _tmain(int argc, _TCHAR* argv[])
{bin_instream bin_in("B_in.dat");
if (!bin_in) { cerr << "Unable to open B_in.dat" << endl;
exit(1); }
double d;
long count = 0;
bin_in >> d;
while (!bin_in.eof())
{ cout << ++count << ":" << d << endl;
bin_in >> d;
}
_getch();
return 0;
}
void bin_instream:: readOurDate(void *p, int len)
{ if (!p) return;
if (len <= 0) return;
read((char*)p, len);
}

```

Для работы с файловыми потоками любого из стандартных типов нужно перегрузить операторы ввода и вывода под требуемый тип данных или воспользоваться шаблоном класса, задаваемым с помощью ключевого слова template. Речь о шаблонах классов пойдет в гл. 25.

24.1.2. Сохранение в двоичных файлах данных, имеющих тип, создаваемый пользователем

Иногда возникает необходимость сохранить в файле данные, структура которых задается программистом. В этих случаях задают класс, содержащий данные и функции, перегружающие операторы ввода и вывода под эти данные.

Пример 24.3. Объявим структуру

```

struct mountine {char name[20]; //название горы
int altitude; //высота над уровнем моря
int complicate; //сложность
};
mountine mount;

```

Для сохранения такой структуры в текстовом файле необходимо выполнить следующие действия:

```

ofstream fil_out("mountines.txt", ios_base::app);
fil_out << mount.name << " " << mount.altitude << ' ' << mount.complicate << "\n";

```

Для сохранения той же информации в двоичном файле выполняют следующее:

```

ofstream fil_out("mountines.dat", ios_base::app | ios_base::binary);
fil_out.write((char *) &mount, sizeof(mountine));

```

Метод write копирует указанное число байтов (в данном случае – sizeof(mountine)) в файл из памяти ЭВМ. Несмотря на то что сохранение данных происходит в двоичном файле, адрес переменной преобразуется к указателю на тип char.

Для чтения данных из двоичного файла используют метод read:

```

ifstream fil_in("mountines.dat", ios_base::binary);
fil_in.read((char *) &mount, sizeof(mountine));

```

При записи или чтении классов, не содержащих виртуальных функций, можно использовать тот же самый подход. Чтобы сделать класс потоковым, нужно перегрузить операторы << и >>:

```

friend ostream &operator<<(ostream &, AnyClass &);
friend istream &operator>>(istream &, AnyClass &);

```

24.2. Произвольный доступ к элементам двоичных файлов

24.2.1. Файловый указатель

Каждый файл имеет два связанных с ним значения: *указатель чтения* и *указатель записи*, по-другому называемые *файловым указателем* или *текущей позицией*.

При последовательном доступе к элементам файлов перемещение файлового указателя происходит автоматически. Но иногда бывает нужно контролировать его состояние. Для этого используются следующие функции:

- seekg() – установить текущий указатель чтения;
- tellg() – проверить текущий указатель чтения;
- seekp() – установить текущий указатель записи;
- tellp() – проверить текущий указатель записи.

24.2.2. Организация доступа к элементам двоичных файлов

Благодаря наличию файлового указателя в двоичных файлах допустим произвольный доступ к их элементам, который можно реализовать с помощью перегруженных функций-элементов, унаследованных из класса istream:

istream &seekg(streampos) или
istream &seekg(streamoff, ios::seek_dir);

Типы данных streampos и streamoff эквивалентны значениям типа long, но использовать long в явном виде не рекомендуется из-за неоднозначности работы различных компиляторов. Поэтому их определяют как

```
typedef long streampos;
typedef long streamoff;
```

Первая из перегруженных форм функции seekg позиционирует входной поток на заданном байте, вторая – на смещении относительно одной из трех позиций, определенных значением константы ios::seek_dir (табл. 24.1).

Таблица 24.1. Константы позиционирования файлового указателя

Константа	Значение	Описание
beg	0	Поиск от начала файла
cur	1	Поиск от текущей позиции файла
end	2	Поиск от конца файла

Для позиционирования внутреннего указателя файла для выходных потоков используют перегруженные функции выходных файловых потоков, унаследованных из класса ostream:

```
ostream &seekp(streampos);
ostream &seekp(streamoff, ios::seek_dir);
```

Пример 24.4. В двоичном файле, содержащем целые числа, заменить максимальное значение файла суммой его четных элементов.

```
#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <time.h>
#include <conio.h>
using namespace std;
class bin_stream: public fstream
{public:
    bin_stream(const char *fn): fstream(fn, ios::out | ios::in | ios::binary){}
    void doneOurDate(const void*, int, int);
    bin_stream &operator<<(int d)
    { doneOurDate(&d, sizeof(d),0);
        return *this;
    }
    bin_stream &operator>>(int &d)
    { doneOurDate(&d, sizeof(d),1);
        return *this;
    }
};
class bin_outstream: public ofstream
{public:
    bin_outstream(const char *fn): ofstream(fn, ios::out | ios::binary) {}
    void writeOurDate(const void*, int);
    ofstream &operator<<(int d)
    { writeOurDate(&d, sizeof(d));
        return *this;
    }
};
int _tmain(int argc, _TCHAR* argv[])
{int i, d, max, i_max=0, sum_even = 0;
time_t t;
srand((int)time(&t));
bin_outstream bin_out("Bin.dat"); // создание файла
if (!bin_out)
{ cerr << "Unable to write to Bin.dat" << endl;
    exit(1);
}
for (i = 0; i < 10; i++)
{ d = rand() % 100;
    bin_out << d;
    if (d % 2 == 0) sum_even += d;
}
cout<<endl;
cout<<sum_even<<endl;
```

```

bin_out.close();
bin_stream bin("Bin.dat"); // обработка файла
if (!bin)
{   cerr << "Unable to write to Bin.dat" << endl;
    exit(1);
}
bin >> max;
i_max = 0;
for (i = 1; i < 10; i++)
{ bin >> d;
    if (d > max) { max = d; i_max = i; }
}
cout<<endl;
bin.seekp(sizeof(int) * i_max, ios::beg);
bin << sum_even;
bin.seekp(0, ios::beg);
for (i = 0; i < 10; i++)
{ bin >> d;
    cout << d << ' ';
}
bin.close();
_getch();
return 0;
}

void bin_stream:: doneOurDate(const void *Ptr, int len, int sign)
{ if (!Ptr) return;
if (len <= 0) return;
if (sign==0) write((char*)Ptr, len);
    else read((char*)Ptr, len);
}

void bin_outstream :: writeOurDate(const void *Ptr, int len)
{ if (!Ptr) return;
if (len <= 0) return;
write((char*)Ptr, len);
}

```

Контрольные вопросы

1. Что такое поток?
2. Особенности работы с двоичными файлами.
3. Что представляет собой файловый указатель?
4. Как организовать доступ к произвольному месту двоичного файла?

Практические задания 24

1. В двоичном файле целого типа заменить максимальный элемент суммой предыдущих элементов, минимальный – суммой последующих элементов.
2. В конец двоичного файла целого типа записать четные элементы этого файла.

3. В начало двоичного файла целого типа дописать нечетные элементы этого файла.
4. В середину двоичного файла целого типа поместить элементы этого файла, кратные пяти.
5. В двоичном файле целого типа поменять местами элементы, стоящие на четных местах с элементами, стоящими на нечетных местах.
6. В начало двоичного файла целого типа дописать его минимальное значение, в середину – максимальное.
7. В начало двоичного файла целого типа дописать элементы, являющиеся делителями максимального элемента этого файла.
8. В середину двоичного файла целого типа записать элементы этого файла, меньшие числа, введенного с клавиатуры.
9. Даны двоичные файлы f и g целого типа. Записать в начало файла f положительные компоненты файла g, а в конец файла g – отрицательные компоненты файла f с сохранением порядка их следования.
10. Дан двоичный файл с целыми числами. Удалить из него число, записанное после первого нуля (принять, что нули в файле имеются). Результат записать в другой файл.
11. Дан двоичный файл с положительными и отрицательными целыми числами. Записать в другой файл сначала отрицательные элементы, а затем положительные.

ГЛАВА 25. ШАБЛОНЫ. РОДОВЫЕ ФУНКЦИИ.

РОДОВЫЕ КЛАССЫ

25.1. Понятие шаблона и родовой функции

В новых версиях компилятора C++ реализовано новое средство программирования – *шаблон* (*template*). С помощью шаблонов можно создавать родовые (*genetic*) функции и классы, в которых тип задается в качестве параметра. То есть одна и та же функция или класс могут использоваться с несколькими различными типами данных без явного переписывания версии для каждого конкретного типа данных.

Шаблоны позволяют создавать программы на основе обобщенного типа, который определяется впоследствии. Такое программирование называется *обобщенным*.

Шаблоны обеспечивают возможность определять при помощи одного фрагмента программного кода целый набор перегруженных функций, называемых *шаблонными функциями* или *родовыми функциями*. Шаблонная (родовая) функция определяет базовый набор операций (алгоритм), которые будут применяться к данным различных типов. Такой способ перегрузки функций очень удобен, когда алгоритм обработки данных, реализуемый функцией, одинаков для любого типа данных.

Родовая функция (шаблон) создается с помощью ключевого слова *template*:

```
template <class Ttype> возвращаемое_значение имя_функции (спи-
сок_параметров)
{ // тело функции
}
```

Здесь на месте *Ttype* указывается тип данных, используемых функцией. Это имя можно употреблять внутри определения функции, но это имя – фиктивное, компилятор автоматически заменит его именем реального типа данных при создании конкретной версии функции.

Пример 25.1. Создание родовой функции, которая меняет значения двух переменных, передаваемых ей в качестве параметров.

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>
using namespace std;
template <class X>
void swaping(X &a, X &b) // это шаблон
{
    a = a + b;
    b = a - b;
    a = a - b;
}
```

```
}
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 10, j = 20;
    char sym1 = '^', sym2 = '#';
    cout << "i = " << i << " j = " << j << endl;
    cout << "sym1 = " << sym1 << " sym2 = " << sym2 << endl;
    swaping(i,j); // обмен целых
    swaping(sym1,sym2); // обмен символов
    cout << "i = " << i << " j = " << j << endl;
    cout << " sym1 = " << sym1 << " sym2 = " << sym2 << endl;
    _getch();
    return 0;
}
```

Здесь X – родовой тип. Функция *swap()* использует X в качестве фиктивного типа ее параметров. В функции *main()* функция *swap()* оперирует с двумя разными типами данных: целым и символьным; *swap()* – родовая функция, поэтому компилятор автоматически создает две ее версии под разные типы параметров. Процесс создания конкретной версии (экземпляра) родовой функции называется *созданием порожденной функции*. То есть порожденная функция представляет собой конкретный экземпляр функции-шаблона.

Пример 25.2. Создание родовой функции, которая меняет значения трех переменных таким образом, чтобы оказалось $a \geq b \geq c$.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
template <class X> void swap_3(X &a, X &b, X &c)
{
    X r;
    if (a < b) { r = a; a = b; b = r;}
    if (a < c) { r = a; a = c; c = r;}
    if (b < c) { r = b; b = c; c = r;}
}
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 10, j = 20, k = 30;
    char sym1 = '^', sym2 = '#', sym3 = '@';
    double x = 2.5, y = 5.8, z = 4.3;
    cout << "i = " << i << " j = " << j << " k = " << k << endl;
    cout << "sym1 = " << sym1 << " sym2 = " << sym2
        << " sym3 = " << sym3 << endl;
    cout << "x = " << x << " y = " << y << " z = " << z << endl;
    swap_3(i, j, k); // обмен целых
    swap_3(sym1, sym2, sym3); // обмен символов
    swap_3(x, y, z); // обмен действительных
    cout << "i = " << i << " j = " << j << " k = " << k << endl;
    cout << "sym1 = " << sym1 << " sym2 = " << sym2
        << " sym3 = " << sym3 << endl;
```

```

cout << "x = " << x << " y = " << y << " z = " << z << endl;
_getch();
return 0;
}

```

Если посмотреть на листинг кода программы, содержащей шаблоны, то сам шаблон увидеть не удастся. Вместо него код программы будет содержать несколько версий данной функции в таком виде, будто они написаны явно для данных разного типа.

25.2. Определение более чем одного родового типа данных

С помощью оператора template можно определить более чем один родовой тип данных.

Пример 25.3

```

#include "stdafx.h"
#include <iostream>
#include <iostream>
#include <time.h>
#include <conio.h>
using namespace std;
// допускается писать в две строки
template <class Type1, class Type2>
void MyFunc(Type1 v1, Type2 v2)
{ cout << v1 << ' ' << v2 << endl; }
int _tmain(int argc, _TCHAR* argv[])
{ MyFunc("Hallo, ", "World !");
  MyFunc(10, " hi ");
  MyFunc(0.25, 10L);
  getch();
  return 0;
}

```

Можно написать функцию, у которой часть аргументов является шаблонами, а часть имеет базовый тип.

Пример 25.4. Написать функцию, которая возвращает индекс максимального элемента массива.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
template <class X> int max(X *arr, const int size)
{ int i_max = 0;
  X m = arr[0];
  for (int i = 0; i < size; i++)
    if (m < arr[i]) { m = arr[i];
}

```

```

      i_max = i;
    }
    return i_max;
}
char chArr[]={'a','m','r','s','t','q','n','h'};
const int chRang=sizeof(chArr)/sizeof(chArr[0]);
int iArr[] = {3, 67, 34, 95, 8, 9, 58, 24, 92};
const int iRang=sizeof(iArr)/sizeof(iArr[0]);
double fArr[]={8.3,6.7,3.4,5.7,4.8,9.3,5.8,2.4};
const int fRang=sizeof(fArr)/sizeof(fArr[0]);
int _tmain(int argc, _TCHAR* argv[])
{cout<<"Max element of chArr=" << max(chArr, chRang) << endl;
 cout<<"Max element of iArr=" << max(iArr, iRang) << endl;
 cout<<"Max element of fArr=" << max(fArr, fRang) << endl;
 _getch();
return 0;
}

```

Родовые функции похожи на перегружаемые функции, но являются более ограниченными, так как при перегрузке функции тело каждой из ее версий может содержать различные действия, а шаблоны всегда выполняют один и тот же алгоритм применительно к данным разных типов.

По умолчанию компилятор автоматически создает перегруженные версии функции-шаблона, но при необходимости можно перегрузить ее явным образом. При явной перегрузке компилятор не создает автоматическую версию функции для параметров указанных типов.

Пример 25.5. Написать функцию сложения ее аргументов.

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
const int MAX_LEN = 80; // максимальная длина строки
template <class X> X summa(X a, X b)
{ X sum;
  sum = a + b;
  return sum;
}
// Переопределение родовой функции summa()
char * summa(char s1[], char s2[])
{ // функция объединения строки s1 и строки s2:
  strcat_s(s1, MAX_LEN, s2);
  return s1;
}
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
 char s1[MAX_LEN]={"Блестит над ледником холодная звезда,";
 char s2[] = " но нет теплее на земле картины";

```

```

int i = 10, j = 20;
double x = 5.5, y = 2.7;
cout << summa(i, j) << endl;
cout << summa(x, y) << endl;
cout << summa(s1, s2) << endl;
_getch();
return 0;
}

```

Если алгоритм обработки данных различных типов одинаков, используют родовые функции, если же нужно несколько различающихся между собой функций для обработки данных разных типов, лучше использовать не шаблоны, а перегруженные функции.

25.3. Родовые классы (шаблоны классов)

Принцип построения шаблонов применим и к классам. Обычно это происходит в случае, когда класс является хранилищем данных и имеет общую логику. Например, с помощью шаблонов классов удобно создавать классы, реализующие самоссылочные структуры (очереди, стеки, деки и т. д.) для любых типов данных. Компилятор автоматически генерирует программный код на основе типа, задаваемого при создании объекта.

Фактические типы обрабатываемых данных при этом задаются в качестве параметров при создании объектов этого класса.

Синтаксис шаблона класса:

```

template <class Ttype> class имя_класса
{
    // описание класса
}

```

Здесь Ttype – фиктивное имя типа, который будет определен (задан) при создании экземпляра класса. После определения шаблона класса можно создать конкретный экземпляр этого класса:

имя_класса <type> объект;

где type – имя типа данных, с которым будет оперировать класс.

Функции-элементы родового класса автоматически становятся родовыми. Для них не обязательно указывать ключевое слово template.

Пример 25.6. Реализация класса stack в виде шаблона для хранения объектов любого типа.

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
#define SIZE 10
// создание шаблона класса stack

```

```

template <class StackType> class stack
{
    StackType stck[SIZE]; // массив, содержащий стек
    int tos; // индекс вершины стека
public:
    void init() {tos = 0;} // инициализация стека
    void push(StackType ob); // поместить объект в стек
    StackType pop(); // извлечь объект из стека
};

template <class StackType>
void stack <StackType>::push(StackType ob)
{
    if (tos == SIZE)
        { cout << "Stack is full" << endl;
        return;
    }
    stck[tos] = ob;
    tos++;
}

template <class StackType>
StackType stack < StackType>::pop()
{
    if (tos == 0)
        { cout << "Stack is empty" << endl;
        return 0;
    }
    tos--;
    return stck[tos];
}

int _tmain(int argc, _TCHAR* argv[])
{
    int i;
    char sym = 'a';
    // символьный стек:
    stack <char> st_ch;
    st_ch.init();
    for (i = 0; i < 7; i++) { st_ch.push(sym); sym++; }
    for (i = 0; i < 7; i++) cout << st_ch.pop() << ' ';
    cout << endl;
    // стек типа int:
    stack <int> st_int;
    st_int.init();
    for (i = 0; i < 8; i++) st_int.push(i * 10);
    for (i = 0; i < 8; i++) cout << st_int.pop() << ' ';
    cout << endl;
    // стек типа double:
    stack <double> st_fl;
    st_fl.init();
    for (i = 0; i < 5; i++) st_fl.push(i * 2.5);
    for (i = 0; i < 5; i++) cout << st_fl.pop() << ' ';
    cout << endl;
    _getch();
    return 0;
}

```

Пример 25.7. Создание шаблона класса, реализующего односвязный список.

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
template <class DatList> class list
{ public:
    DatList info;
    list *next;
    list(DatList data)
    { info = data;
      next = 0;
    }
    void insert(list *node) // поместить объект в список
    { node->next = this;
      next = 0;
    }
    list *get_next() // получить значение поля next
    { return next; }
    DatList get_info() // получить значение поля info
    { return info; }
};
int _tmain(int argc, _TCHAR* argv[])
{ list<char> *head, *tail, *p;
  int i;
  head = new list<char>('a');
  head->info = 'a';
  tail = head;
  for (i = 1; i < 26; i++)
  { p = new list<char> ('a' + i);
    p->insert(tail);
    tail = p;
  }
  p = head;
  while (p) { cout << p->get_info();
    p = p->get_next(); }
  getch();
  return 0;
}
```

Класс-шаблон, как и родовая функция, может иметь больше чем один родовой тип данных.

Пример 25.8. Создание класса-шаблона, имеющего два родовых типа данных.

```
#include "stdafx.h"
#include <iostream>
#include <string>
```

```
#include <conio.h>
using namespace std;
template <class Type1, class Type2> class MyClass
{ Type1 i;
  Type2 j;
public:
  MyClass(Type1 a, Type2 b) { i = a; j = b; }
  void Show() { cout << i << ' ' << j << endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
  MyClass <int, double> ob1(100, 25.5);
  MyClass <char *, char *>
  ob2("Интересно, ", "почему звезды светятся?");
  ob1.Show();
  ob2.Show();
  getch();
  return 0;
}
```

В программе объявлены два типа объектов. В обоих случаях компилятор автоматически генерирует необходимые данные и функции для обработки этих данных.

Контрольные вопросы

1. Что такое родовая функция? Как создается шаблон?
2. Что называют созданием порожденной функции?
3. Как написать функцию, у которой часть аргументов является шаблонами, а часть имеет базовый тип?
4. Каким образом можно определить более чем один родовой тип данных?
5. Как объявляют родовые классы (шаблоны классов)?

Практические задания 25

1. Создать шаблон класса, реализующего матрицу.
2. Создать шаблон класса, реализующего двусвязный список.

ГЛАВА 26. ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Исключения, или исключительные ситуации, – это ошибки, которые возникают во время работы программы. Исключение генерируется в результате нестандартных ситуаций, возникающих при выполнении программы. С помощью исключений можно передавать управление от одной части программы к другой.

Наиболее часто встречаются следующие исключения:

- недостаток памяти,
- выход индекса за границы массива,
- арифметическое переполнение,
- деление на нуль,
- недопустимые параметры функций.

Обработка исключений не предназначена для реакции на внешние события, такие, как щелчок мыши, завершение дисковых операций ввода-вывода и т. д., которые лучше взаимодействуют со средствами обработки прерываний.

Синтаксис операторов обработки исключений:

```
try { // блок try }
catch (type1 arg) { // блок catch }
catch (type2 arg) { // блок catch }
catch (type3 arg) { // блок catch }
.
.
.
catch (typeN arg) { // блок catch }
```

Обработка исключительных ситуаций состоит из трех этапов:

- генерации исключения,
- перехвата исключения обработчиком исключений,
- использования блока try.

Оператор генерации исключений фактически выполняет переход на блок, осуществляющий переход и обработку исключений. Операторы программы, во время выполнения которых необходимо обеспечить обработку исключительных ситуаций, располагаются в блоке try. Если исключительная ситуация (ошибка) имеет место внутри блока try, она генерируется с помощью ключевого слова throw. Переход и обработка исключений происходит в блоке catch. Ключевое слово catch служит меткой, обозначающей точку в программе, в которую передается управление в случае перехвата исключения.

Если с блоком try связано более одного оператора catch, то использование конкретного оператора зависит от типа исключительной ситуации. Если тип данных, указанный в операторе catch, соответствует типу исключительной ситуации, то выполняется данный оператор catch, а все

другие операторы блока try пропускаются. Если исключение перехвачено, аргумент arg получает его значение. Можно перехватывать любые типы данных, в том числе и классы.

Синтаксис оператора throw:

```
throw исключительная_ситуация;
```

При генерации исключительной ситуации, для которой нет соответствующего оператора catch, происходит аварийное завершение программы.

Пример 26.1. Исключительная ситуация вызывается из оператора, не входящего в блок try, но входящего в функцию, находящуюся в блоке try.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
void NoZeroTest(int test)
{ cout << "test = " << test << endl;
  if (test) throw test;
}
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); // подключение русификатора
try {  NoZeroTest(0);          // исключение не генерируется
      NoZeroTest(1);          // исключение генерируется
      NoZeroTest(2);          // а сюда программа не попадает
    }
catch (int i)
{ cout << "Перехвачена ошибка: " << i << endl;}
_getch();
return 0;
}
```

Если блок try расположен внутри функции, то при каждом входе в функцию обработчик исключений устанавливается в состояние готовности.

Пример 26.2

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
void NoZeroTest(int test)
{ try { if (test) throw test;
  }
catch (int i)
{ cout << "Перехвачена ошибка: " << i << endl;
  }
}

int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian");
```

```
NoZeroTest(1);           // Перехвачена ошибка: 1
NoZeroTest(2);           // Перехвачена ошибка: 2
NoZeroTest(0);           // исключение не генерируется
NoZeroTest(3);           // Перехвачена ошибка: 3
_getch();
return 0;
}
```

В приведенных примерах можно обойтись без использования механизма обработки исключений. Однако проблема поиска ошибок встает более остро при использовании классов, так как ошибки могут возникать и при отсутствии явных вызовов функций. Примером может служить ошибка в конструкторе класса, который вызывается неявным образом и не имеет возвращаемого значения.

Ситуация еще более усложняется, когда в приложении используются библиотеки классов.

Пример 26.3. Создание стека-массива, использующего исключения для отслеживания его верхней и нижней границы.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <conio.h>
using namespace std;
const int RANG = 10;           // максимальный размер стека
class Stack
{
    int stk[RANG];           // индекс вершины стека
    int top;                  // класс исключений для класса Stack,
public:                         // тело класса пусто
    class Range
    {
    };
    Stack() { top = -1; }      // конструктор
    void push(int variable)   // функция помещения данных в стек
    { if (top >= RANG - 1)    // если стек заполнен,
        throw Range();         // генерировать исключение
        stk[++top] = variable; // иначе – протолкнуть число
    }
    int pop()                 // функция извлечения данных из стека
    { if (top < 0)             // если стек пуст,
        throw Range();         // генерировать исключение
        return stk[top--];     // иначе – извлечь число
    }
};                            // class Stack

int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); // подключение русификатора
Stack s_int;
```

```
int INP;
try { for (int i = 0; i < RANG; i++)
    { cout << "Input value";
        cin >> INP;
        s_int.push(INP);
    }
    s_int.push(INP);           // генерация исключения - стек полон
    for (int i = 0; i < RANG; i++)
        cout << i + 1 << ':' << s_int.pop() << endl;
    cout << "11: "             // генерация исключения - стек пуст
    << s_int.pop() << endl;
    }
    // end try
catch (Stack:: Range)       // обработчик исключений
{ cout << "Исключение: нарушение границы стека" << endl;
}
cout << "Завершение работы программы" << endl;
_getch();
return 0;
}
```

Контрольные вопросы

1. Что такое исключения (исключительные ситуации)? Когда они возникают?
2. Перечислите этапы обработки исключительных ситуаций.
3. Почему проблема поиска ошибок более актуальна при использовании классов?

Практические задания

1. Написать программу обработки исключения, возникающего при введении нецелого числа в отрицательную степень.
2. Написать программу обработки исключения, возникающего при попытке вычисления логарифма числа, меньшего или равного нулю.
3. Написать программу обработки исключения, возникающего при попытке вычисления корня из отрицательного числа.

ГЛАВА 27. ДИНАМИЧЕСКАЯ ИДЕНТИФИКАЦИЯ ТИПА

Динамическое определение типа позволяет получать информацию о классе объекта, а также изменять этот класс во время работы программы.

Эти возможности становятся необходимыми, когда имеет место большая и сложная иерархия классов, порожденных от одного базового класса. Существующее в C++ понятие полиморфизма реализуется через иерархию классов. При этом указатель базового класса может применяться либо для указания на объект базового класса, либо для указания на объект любого класса, производного от этого базового. Поэтому невозможно заранее узнать тип объекта, на который будет указывать указатель класса. Значит, определение типа объекта должно происходить в процессе выполнения программы, т. е. динамически.

Динамическая идентификация классов (Run-Time Type Information, RTTI) в языке C++ поддерживается с помощью трех компонентов:

- оператора `dynamic_cast`,
- оператора `typeid`,
- оператора `type_info`.

Библиотека RTTI подключается по умолчанию в системе Borland C++ Builder и требует подключения вручную в системе Microsoft Visual C++. Для этого в программу нужно включить заголовочный файл `typeinfo`.

Библиотека RTTI функционирует только с классами, которые имеют в качестве своих элементов виртуальные функции.

27.1. Оператор `dynamic_cast`

Оператор приведения типов в динамическом режиме `dynamic_cast` имеет два операнда: тип, заключенный в угловые скобки, и указатель (или ссылка), заключенный в круглые скобки:

`dynamic_cast <целевой_тип> (выражение)`

где **целевой_тип** – это тип, к которому нужно привести **выражение**. Целевой_тип должен быть указателем или ссылкой, следовательно, результат выполнения выражения тоже должен стать указателем или ссылкой.

То есть оператор `dynamic_cast` приводит тип одного указателя к типу другого указателя или тип одной ссылки к типу другой ссылки. Приведение `dynamic_cast` используется в тех случаях, когда правильность преобразования не может быть определена компилятором.

В основном оператор `dynamic_cast` предназначен для реализации операции приведения полиморфных типов.

Обычно оператор `dynamic_cast` выполняется успешно, если указатель (или ссылка) после приведения типов становится указателем (или ссылкой) на объект целевого типа или на объект типа, производного от целевого.

Пример 27.1

```
Base *bp;
Derived *dp;
...
dp = dynamic_cast<Derived *>(bp);
```

При неудачной попытке приведения типов с помощью оператора `dynamic_cast` результатом будет 0 (NULL), если в операторе использовались указатели. В случае использования ссылок генерируется исключительная ситуация `bad_cast`.

Пример 27.2. Определение типов указателей с помощью `dynamic_cast`.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <math.h>
#include <time.h>
using namespace std;
class area
{
    double dim1, dim2;           // размеры фигуры
public:
    void setdim(double d1, double d2) { dim1 = d1; dim2 = d2; }
    void getdim(double &d1, double &d2) { d1 = dim1; d2 = dim2; }
    virtual void my_name() { cout << "I am an area" << endl; }
    virtual double getarea() { return 1; }
};
class rectangle: public area
{ public:
    double getarea() { double d1, d2;
        getdim(d1, d2);
        return d1 * d2;
    }
    virtual void my_name() { cout << "I am a rectangle" << endl; }
};
class ring: public area
{ public:
    double getarea()
    { const double Pi = 3.14159;
        double d1, d2;
        getdim(d1, d2);
        return Pi * fabs (d1 * d1 - d2 * d2);
    }
    virtual void my_name() { cout << "I am a ring" << endl; }
};
class cylinder: public area
{ double h;
public:
    void set_h(double h1) { h = h1; }
    double getarea()
```

```

{ const double Pi = 3.14159;
double d1, d2;
getdim(d1, d2);
return Pi * fabs (d1 * d1 - d2 * d2) * h;
}
virtual void my_name() { cout << "I am a cylinder" << endl; }
};

area *GetObj();
int _tmain(int argc, _TCHAR* argv[])
{ area *Ptr;
cylinder *cPtr;
double d1, d2, h;
for (int i = 0; i < 4; i++)
{ Ptr = GetObj();
cPtr = dynamic_cast<cylinder *>(Ptr);
Ptr->my_name();
if (Ptr != NULL)
{ cout << "Input d1 and d2";
cin >> d1 >> d2;
Ptr->setdim(d1, d2);
if (cPtr != NULL)
{ cout << "Input h";
cin >> h;
cPtr->set_h(h);
}
cout << "Square of figure = " << Ptr->getarea() << endl;
}
}
 getch();
return 0;
}
area *GetObj()
{ area *p;
time_t t;
srand((int)time (&t));
switch (rand() % 4)
{ case 0: p = new area;
break;
case 1: p = new rectangle;
break;
case 2: p = new ring;
break;
case 3: p = new cylinder;
break;
}
return p;
}

```

Результаты работы программы

I am an area

Input d1 and d2 4 7

Square of figure = 1

I am a ring

Input d1 and d2 8 3

Square of figure = 172.787

I am a cylinder

Input d1 and d2 9 6

Input h5

Square of figure = 706.858

I am a ring

Input d1 and d2 8 3

Square of figure = 172.787

27.2. Оператор typeid

Для определения типа объекта служит оператор typeid, синтаксис которого имеет вид:

typeid (объект)

где **объект** – тот объект, тип которого нужно определить. Для использования оператора typeid в программу включают заголовочный файл typeinfo.

С помощью оператора typeid во время выполнения программы можно определить тип объекта, на который указывает указатель базового класса или ссылка на базовый класс.

Оператор typeid, у которого в качестве аргумента указано имя типа:

typeid (имя_типа)

позволяет получить объект типа type_info, который можно использовать при сравнении типов.

Оператор typeid чаще всего применяют к разыменованным указателям. В случае нулевого указателя возникает исключительная ситуация bad_typeid, которая возбуждается оператором typeid.

Если в качестве аргумента оператора typeid задать указатель полиморфного базового класса, оператор будет автоматически возвращать тип реального объекта, на который указывает указатель. Этим объектом может быть как объект базового класса, так и объект любого класса, производного от этого базового класса. Полиморфным называется класс, содержащий виртуальную функцию.

Когда оператор typeid применяют к неполиморфному классу, получают указатель или ссылку на базовый тип.

Пример 27.3. Определение типа объекта с помощью оператора typeid.

```
#include "stdafx.h"
#include <iostream>
```

```
#include <cstdlib>
#include <typeinfo>
#include <conio.h>
#include <math.h>
#include <time.h>
using namespace std;
class area { double m1, m2; // размеры фигуры
public:
    void setarea(double d1, double d2)
    { m1 = d1; m2 = d2; }
    void getm(double &d1, double &d2)
    { d1 = m1; d2 = m2; }
    virtual double getarea() = 0;
};
class rectangle: public area
{public:
    double getarea() { double d1, d2;
        getm(d1, d2);
        return d1 * d2;
    }
};
class triangle: public area
{public:
    double getarea() { double d1, d2;
        getm(d1, d2);
        return 0.5 * d1 * d2;
    }
};
class cylinder: public area
{public:
    double getarea()
    { const double Pi = 3.14159;
        double d1, d2; // d1 – радиус, d2 - высота
        getm(d1, d2);
        return 2 * Pi * d1 * (d2 + d1);
    }
};
class ring: public area
{public:
    double getarea()
    { const double Pi = 3.14159;
        double d1, d2;
        getm(d1, d2);
        return Pi * fabs(d1 * d1 - d2 * d2);
    }
};
// создание объектов, производных от класса area:
area *generator()
{ switch (rand() % 4)
    { case 0: return new rectangle;
```

```
        break;
    case 1: return new triangle;
        break;
    case 2: return new cylinder;
        break;
    case 3: return new ring;
        break;
    }
    return NULL;
}
int _tmain(int argc, _TCHAR* argv[])
{ int i;
    area *p;
    for (i = 0; i < 10; i++)
    { p = generator(); // создание объекта
        cout << typeid(*p).name() << endl;
        p->getarea();
    }
    getch();
    return 0;
}
```

Результаты работы программы

```
class triangle
class ring
class cylinder
class rectangle
class triangle
class rectangle
class cylinder
class cylinder
class rectangle
```

Оператор typeid может работать с классами-шаблонами.

Пример 27.4. Определение типа объекта, созданного с использованием шаблона, с помощью оператора typeid.

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <typeinfo>
#include <conio.h>
#include <math.h>
#include <time.h>
using namespace std;
template <class T> class area
{ T m1, m2; // размеры фигуры
```

```

public:
    void setarea(T d1, T d2) { m1 = d1; m2 = d2; }
    void getm(T &d1, T &d2) { d1 = m1; d2 = m2; }
    virtual T getarea() = 0;
};

template <class T> class rectangle: public area<T>
{public:
    T getarea() { T d1, d2;
        getm(d1, d2);
        return d1 * d2;
    }
};

template <class T> class triangle: public area<T>
{public:
    T getarea() { T d1, d2;
        getm(d1, d2);
        return 0.5 * d1 * d2;
    }
};

// создание объектов, производных от класса area:
area<double> *generator()
{
    switch (rand() % 2)
    {
        case 0: return new rectangle<double>;
        break;
        case 1: return new triangle<double>;
        break;
    }
    return NULL;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int i;
    area<double> *p;
    for (i = 0; i < 5; i++)
    {
        p = generator(); // создание объекта
        cout << typeid(*p).name() << endl;
        p->getarea();
    }
    _getch();
    return 0;
}

```

Результаты работы программы

```

class triangle<double>
class triangle<double>
class rectangle<double>
class rectangle<double>
class triangle<double>

```

27.3. Структура класса type_info

Структура класса type_info содержит информацию об определенном типе. В классе type_info определены следующие открытые элементы:

```

bool operator == (const type_info &объект);
bool operator != (const type_info &объект);
bool before (const type_info &объект);
const char *name();

```

Сравнение типов возможно благодаря перегруженным операторам == и !=. Функция before() возвращает значение true, если вызывающий объект расположен раньше объекта, заданного в качестве параметра, в смысле сортировки. Функция name() возвращает указатель на имя типа.

Реализации класса type_info могут варьироваться, но все они содержат функцию-элемент name(), которая возвращает имя класса.

27.4. Операторы приведения типа

C++ поддерживает способы приведения типов, принятые в языке С. Кроме того, в C++ появились новые операторы: dynamic_cast, const_cast, reinterpret_cast, static_cast.

Оператор *dynamic_cast*, рассмотренный выше, используется наиболее часто.

Оператор *const_cast* применяется для явной подмены атрибутов const (постоянный) и/или volatile (изменяемый внешним устройством или фоновым процессом).

Синтаксис оператора:

const_cast<целевой_тип> (*выражение*)

Целевой_тип должен совпадать с исходным типом за исключением атрибутов const или volatile.

Снятие атрибута const в процессе выполнения программы может оказаться опасным, поэтому использовать оператор const_cast следует с большой осторожностью.

Оператор *reinterpret_cast* позволяет преобразовать указатель одного типа в указатель другого типа. С его помощью можно также приводить указатель к целочисленному типу и целочисленный тип – к указателю.

Синтаксис оператора:

reinterpret_cast<целевой_тип> (*выражение*)

Однако, преобразование reinterpret_cast является системно-зависимым, поэтому его использование нежелательно.

Оператор *static_cast* предназначен для выполнения операций приведения типов над объектами неполиморфных классов. Преобразования производятся в статическом режиме.

Пример 27.5. Использование операторов приведения типов.

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <typeinfo>
#include <conio.h>
#include <time.h>
using namespace std;
class Base { protected:
    int b;
public:
    Base() { b = 0; }
    Base(int m) { b = m; }
    virtual void virtFn() {}
    void show() { cout << "Base: b = " << b << endl; }
};
class Derived: public Base
{
    int d;
public:
    Derived(int m, int n) { d = n; b = m; }
    void show() { cout << "Derived: b = " << b << " d = " << d << endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{
    Base *pBase = new Base(10);
    Derived *pDerived = new Derived(21, 22);
    // приведение к базовому типу:
    pBase = static_cast<Base *>(pDerived);
    pBase->show();           // Base: b = 21
    pBase = new Derived(31, 32);
    // приведение к производному типу:
    pDerived = reinterpret_cast<Derived *>(pBase);
    pDerived->show();        // Derived: b = 31 d = 32
    getch();
    return 0;
}
```

Результаты работы программы

```
Base: b = 21
Derived: b = 31 d = 32
```

Контрольные вопросы

- Что такое динамическая идентификация типов? В каких случаях она используется?
- Какие действия выполняет оператор `dynamic_cast`?
- В чем состоят различия использования операторов `typeid` (**имя_типа**) и `typeid` (**объект**)?
- Что описывает и какие элементы содержит структура класса `type_info`?
- Перечислите операторы приведения типа, используемые в C и в C++.

ГЛАВА 28. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ. КОНТЕЙНЕРЫ, ИТЕРАТОРЫ, АЛГОРИТМЫ

28.1. Основные определения

Библиотека стандартных шаблонов (Standard Template Library, STL) – наиболее совершенный инструмент языка программирования C++.

Ядро библиотеки стандартных шаблонов состоит из четырех основных элементов:

- контейнеров,
- итераторов,
- алгоритмов,
- функциональных объектов.

Контейнеры – это объекты, предназначенные для хранения других объектов. Контейнеры бывают различных типов – массивы, очереди, списки и т. д. Кроме базовых контейнеров, в библиотеке стандартных шаблонов определены ассоциативные контейнеры, позволяющие с помощью ключей быстро получать хранящиеся в них значения. В каждом классе-контейнере определен набор функций для работы с этим контейнером.

Алгоритмы – выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска или замены содержимого контейнеров.

Итераторы – это объекты, которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера так, как указатель позволяет получить доступ к элементу массива.

Функциональные объекты – это объекты, действующие как функции. Они могут быть объектами класса или указателями на функции, включающими в себя имя функции, поскольку именно оно работает как указатель. Библиотека STL позволяет построить множество контейнеров, включая массивы, очереди и списки, и выполнить над ними различные операции: заполнение случайным образом, поиск, сортировку и т. д.

Контейнерный класс – это абстрактный тип данных (АТД). Он обеспечивает абстрактные методы хранения данных. Например, стек – это АТД с функциями для запоминания и обработки данных по принципу «Последним вошел – первым вышел». Он представляет собой просто метод программирования, который не зависит от конкретных деталей реализации способа хранения данных в нем.

Фундаментальная структура данных (ФСД) – это любая структура, которая может использоваться для реализации АТД. Например, класс

стека (АТД) можно реализовать с помощью связного списка (ФСД), или массива (ФСД), или другого фундаментального типа.

28.2. Контейнеры

В каждом контейнерном классе сочетаются один АТД и одна ФСД.

Контейнеры могут содержать простые объекты (целые, вещественные, символьные и т. д.), структурные данные (массивы, строки, структуры), объекты классов. В контейнерах можно хранить сами объекты или указатели на них.

К контейнерам применимы следующие правила [29, 30]:

1. Контейнер может содержать любые типы данных, кроме ссылок.
2. Конструктор копирования помещает объект в контейнер. При этом программист получает полный контроль над процессом помещения объекта в контейнер.
3. Контейнеры автоматически выделяют и освобождают память, необходимую для хранения объекта.
4. Вектор может автоматически увеличивать свой размер для размещения новых данных, но он не может автоматически уменьшаться в размерах. Другие контейнеры (списки, деки, множества) могут увеличиваться и уменьшаться по мере необходимости.
5. При уничтожении контейнера программой, он сначала вызывает деструктор, удаляющий его объекты, поэтому нет необходимости в предварительном удалении содержимого контейнера, кроме случаев, когда контейнер содержит указатели на объекты. В этом случае программист сам отвечает за удаление содержимого контейнера или за освобождение памяти.

В настоящее время STL включает в себя 11 структур данных контейнеров, реализованных в виде шаблонов классов. Контейнеры, определенные в STL, представлены в табл. 28.1.

Таблица 28.1. Контейнеры, определенные в STL

Контейнер	Описание	Заголовок
bitset	Множество битов	<bitset>
deque	Двунаправленный список	<deque>
list	Линейный список	<list>
map	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано только одно значение	<map>
multimap	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано два или более значений	<map>
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>

Окончание табл. 28.1

Контейнер	Описание	Заголовок
priority queue	Очередь с приоритетом	<queue>
queue	Очередь	<queue>
set	Множество, в котором каждый элемент уникален	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

Имена типов элементов, входящих в объявление класса-шаблона, могут быть различными, поэтому в классах-контейнерах были введены некоторые согласованные имена типов, объявляемые с помощью ключевого слова typeid (табл. 28.2).

Таблица 28.2. Согласованные имена типов

Согласованное имя типа	Описание
size type	Интегральный тип, эквивалентный типу size_t
reference	Ссылка на элемент
const reference	Постоянная ссылка на элемент
iterator	Итератор
const iterator	Постоянный итератор
reverse iterator	Обратный итератор
const reverse iterator	Постоянный обратный итератор
value type	Тип хранящегося в контейнере значения
allocator type	Тип распределения памяти
key type	Тип ключа
key compare	Тип функции, которая сравнивает два ключа
value compare	Тип функции, которая сравнивает два значения

В контейнере может храниться не любой объект, а только такой, тип которого поддается конструированию с точки зрения копирования и присваивания. Этому требованию удовлетворяют базовые типы и классы, в которых конструктор копирования и/или оператор присваивания не являются закрытыми (private) или защищенными (protected).

Методы, общие для всех контейнеров, приведены в табл. 28.3.

Таблица 28.3. Методы, общие для всех контейнеров

Имя	Назначение
size()	Возвращает число элементов в контейнере
empty()	Возвращает true, если контейнер пуст
max_size()	Возвращает максимально допустимый размер контейнера
begin()	Возвращает итератор на начало контейнера

Окончание табл. 28.3

Имя	Назначение
end()	Возвращает итератор на конец контейнера
rbegin()	Возвращает реверсивный итератор на конец контейнера (итерации происходят в обратном направлении)
rend()	Возвращает реверсивный итератор на начало контейнера (итерации происходят в обратном направлении)

28.2.1. Векторы

Вектор представляет собой фактически динамический массив и предоставляет непосредственный доступ к любому элементу данных, используя операцию индексирования.

Шаблон для класса vector имеет вид:

```
template <class T, class Allocator = allocator<T>> class vector;
```

где T – тип данных, хранящихся в контейнере ключевое слово Allocator задает способ распределения памяти (по умолчанию – стандартный).

Для класса vector определена операция индексирования [], а также операции сравнения ==, !=, <, <=, >, >=.

Наиболее важные функции-элементы класса vector представлены в табл. 28.4.

Таблица 28.4. Функции-элементы класса vector

Функция	Описание
size()	Возвращает текущий размер вектора. Поскольку размер вектора может изменяться динамически, важно иметь возможность определять его размер в процессе работы программы, а не во время компиляции
begin()	Возвращает итератор начала вектора, являющийся фактически указателем на его начало
end()	Возвращает итератор конца вектора, являющийся фактически указателем на его конец
push_back()	Помещает значение в конец вектора. Если необходимо, размер вектора при этом автоматически увеличивается
insert()	Помещает значение в середину вектора
erase()	Удаляет элементы из вектора

Пример 28.1. Основные операции, производимые над векторами.

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ vector<int> vect; // создание вектора 0 длины
```

```
/* vector<int> vect(5, 7); //создание вектора,
// состоящего из семерок, длина которого равна 5 */
unsigned int i;
cout << "Length of vector:" << vect.size() << endl; // 0
for (i = 0; i < 5; i++) vect.push_back(i * i);
cout << "Length of vector:" << vect.size() << endl; // 5
for (i = 0; i < vect.size(); i++) cout << vect[i] << ',';
cout << endl;
for (i = 0; i < vect.size(); i++) vect[i] = vect[i] * vect[i];
// можно организовать вывод вектора с помощью итератора:
vector<int>:: iterator ip = vect.begin();
while (ip != vect.end())
{ cout << *ip << ',';
  ip++;
}
ip = vect.begin();
ip += 3; // установили итератор на 4 элемент
// вставить 5 элементов, каждый из которых равен 13,
// начиная с того места, куда указывает итератор:
vect.insert(ip, 5, 13);
for (i = 0; i < vect.size(); i++) cout << vect[i] << ',';
cout << endl;
ip = vect.begin();
ip += 2; // установили итератор на 3 элемент
// удалить 3 элемента, начиная с того места,
// куда указывает итератор
vect.erase(ip, ip + 3);
for (i = 0; i < vect.size(); i++) cout << vect[i] << ',';
cout << endl;
_getch();
return 0;
}
```

Результат работы программы:

```
Length of vector: 0
Length of vector: 5
0 1 4 9 16
0 1 16 81 256 0 1 16 13 13 13 13 13 81 256
0 1 13 13 13 81 256
```

Пример 28.2. Использование векторов для хранения объектов.

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;
class MyVect
{ int iVal;
public:
  MyVect() { iVal = 0; }
```

```

MyVect( int n) { iVal = n; }
MyVect &operator =(int n)
{ iVal = n;
  return *this;
}
int get_Val() { return iVal; }
// В случае необходимости можно перегрузить
// соответствующие операторы, например:
// friend bool operator <(MyVect ob1, MyVect ob2);
// friend bool operator ==(MyVect ob1, MyVect ob2);
};

/* bool operator <(MyVect ob1, MyVect ob2)
{ return ob1.get_Val() < ob2.get_Val(); }
bool operator ==(MyVect ob1, MyVect ob2)
{ return ob1.get_Val() == ob2.get_Val(); }

int _tmain(int argc, _TCHAR* argv[])
{
  vector<MyVect> vect;
  unsigned int i;
  for (i = 0; i < 5; i++) vect.push_back(i * i);
  for (i = 0; i < vect.size(); i++) cout << vect[i].get_Val() << ' ';
  cout << endl;
  for (i = 0; i < vect.size(); i++) vect[i] = vect[i].get_Val() * 2;
  for (i = 0; i < vect.size(); i++) cout << vect[i].get_Val() << ' ';
  cout << endl;
  getch();
  return 0;
}

```

Результат работы программы:

```

0 1 4 9 16
0 2 8 18 32

```

28.2.2. Списки

Списки представлены последовательными наборами двусвязных элементов и поддерживаются классом `list`.

Шаблон для класса `list` имеет вид:

```
template <class T, class Allocator = allocator<T>> class list;
```

где `T` – тип данных, хранящихся в контейнере, ключевое слово `Allocator` задает способ распределения памяти (по умолчанию – стандартный).

Для класса `list` определены операции сравнения `==`, `!=`, `<`, `<=`, `>`, `>=`.

Наиболее важные функции-элементы класса `list` представлены в табл. 28.5.

Таблица 28.5. Функции-элементы класса `list`

Функция	Описание
<code>push_back()</code>	Помещает элемент в конец списка
<code>push_front()</code>	Помещает элемент в начало списка
<code>insert()</code>	Помещает элемент в середину списка
<code>merge()</code>	Выполняет слияние двух упорядоченных списков
<code>pop_back()</code>	Удаляет последний элемент списка
<code>pop_front()</code>	Удаляет первый элемент списка

Для любого типа данных, которые необходимо хранить в списке, должен быть определен конструктор по умолчанию.

Пример 28.3. Сортировка списка.

```

#include "stdafx.h"
#include <iostream>
#include <list>
#include <cstdlib>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); // подключение русификатора
list<int> l_int;
int i;
for(i = 0; i < 10; i++) l_int.push_back(rand() % 100);
cout << "Исходный список:" << endl;
list<int>:: iterator ip = l_int.begin();
while (ip != l_int.end())
{ cout << *ip << ' ';
  ip++;
}
cout << endl;
// сортировка
l_int.sort();
cout << "Отсортированный список:" << endl;
ip = l_int.begin();
while (ip != l_int.end())
{ cout << *ip << ' ';
  ip++;
}
cout << endl;
getch();
return 0;
}

```

Результат работы программы:

Исходный список:

```

41 67 34 0 69 24 78 58 62 64

```

Отсортированный список:

```

0 24 34 41 58 62 64 67 69 78

```

28.3. Итераторы

Итераторы предоставляют метод доступа к объектам контейнера. Итераторы есть у абстрактных контейнеров (таких, как массивы и очереди), а также у таких фундаментальных структур, как списки и векторы. Итераторы позволяют перемещаться по объектам в структуре без их удаления.

При использовании итератора в выражениях следует помнить о том, что он возвращает целое значение, поскольку в классах-итераторах оператор преобразования перегружается и преобразуется в тип int.

Итераторы можно устанавливать в начальное состояние путем обращения к его функции Restart:

```
iterator.Restart(); // установить итератор на первый объект
string *sPtr; // объявить указатель на строку
```

После этого можно написать цикл для перемещения по контейнеру:

```
while ((sPtr = iterator++) != 0)
    cout << *sPtr << endl;
```

Все итераторы имеют по крайней мере один конструктор, функцию Restart и перегруженные операторы int и ++. Некоторые итераторы имеют, кроме того, характерные только для них специальные функции.

28.4. Алгоритмы

Алгоритмы предназначены для обработки контейнеров. Каждый контейнер имеет собственный базовый набор операций, но стандартные алгоритмы обеспечивают более широкий спектр действий. Кроме того, они позволяют одновременно работать с двумя контейнерами, имеющими разные типы. Для обеспечения доступа к алгоритмам STL нужно подключить заголовочный файл `<algorithm>`.

Алгоритмы имеют два основных компонента:

- шаблоны для возможности обработки данных любого типа;
- итераторы для реализации доступа к данным в контейнере.

В STL существуют 4 группы алгоритмов:

- операции, не изменяющие последовательность элементов в контейнере;
- операции, изменяющие последовательность элементов в контейнере;
- операции сортировки и подобные им;
- обобщенные числовые операции.

Операции, не изменяющие последовательность, выполняются над каждым элементом и не изменяют контейнер. К ним относится, например, функция поиска `find()`.

Операции, изменяющие последовательность, также выполняются над каждым элементом и изменяют содержимое контейнера, т. е. изменяется

либо значение элементов контейнера, либо порядок их следования. К функциям, реализующим такие операции, относятся функции копирования `copy()`, присваивания с помощью генератора случайных чисел `random_shuffle()` и т. д.

Сортировка и связанные с ней операции включают функцию сортировки `sort()` и ряд других функций, в том числе средства работы с множествами.

Числовые операции включают функции суммирования содержимого диапазона, определения внутреннего произведения двух контейнеров и т. д. Основные алгоритмы STL приведены в табл. 28.6.

Таблица 28.6. Основные алгоритмы STL

Алгоритм	Действие
find	Возвращает первый элемент с указанным значением
count	Считает количество элементов, имеющих указанное значение
equal	Сравнивает содержимое двух контейнеров и возвращает true, если все соответствующие элементы эквивалентны
search	Ищет последовательность значений в одном контейнере, которая соответствует такой же последовательности в другом
copy	Копирует последовательность значений из одного контейнера в другой или в другое место того же контейнера
swap	Обменивает значения, хранящиеся в разных местах
sort	Сортирует значения в указанном порядке
merge	Комбинирует два сортированных диапазона значений для получения возможно большего диапазона
accumulate	Возвращает сумму элементов в заданном диапазоне
for_each	Выполняет указанную функцию для каждого элемента контейнера
reverse	Меняет на обратный порядок расположения элементов в контейнере

Пример 28.4. Алгоритм `reverse()` меняет на обратный порядок расположения элементов класса `vector` в диапазоне, заданном итераторами `begin` (начало) и `end` (окончание), а алгоритм `sort()` сортирует элементы вектора. Алгоритм `for_each` применяет функцию `multiple()` к каждому элементу вектора.

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // для greater<int>()
#include <conio.h>
#include <time.h>
```

```

using namespace std;
void multiple(int variable) {cout << variable * 10 << ' ';}
int mult_trans(int variable) { return (variable * 10); }
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); // подключение русификатора
vector<int> vect;
time_t t;
int i;
strand((int)time(&t));
cout << "Формирование списка" << endl;
for (i = 0; i < 5; i++) vect.push_back(rand() % 30);
for (i = 0; i < 5; i++) cout << vect[i] << ' ';
cout << endl;
cout << "Реверс списка" << endl;
reverse(vect.begin(), vect.end());
for (i = 0; i < 5; i++) cout << vect[i] << ' ';
cout << endl;
cout << "Сортировка списка по возрастанию" << endl;
sort(vect.begin(), vect.end()); //сорт. по возр.
for (i = 0; i < 5; i++) cout << vect[i] << ' ';
cout << endl;
cout << "Сортировка списка по убыванию" << endl;
// сортировка по убыванию:
sort(vect.begin(), vect.end(), greater<int>());
for (i = 0; i < 5; i++) cout << vect[i] << ' ';
cout << endl;
cout << "Работа метода for_each" << endl;
for_each(vect.begin(), vect.end(), multiple);
cout << endl;
for (i = 0; i < 5; i++) cout << vect[i] << ' ';
cout << endl;
cout << "Работа метода transform" << endl;
transform(vect.begin(), vect.end(), vect.begin(), mult_trans);
for (i = 0; i < 5; i++) cout << vect[i] << ' ';
cout << endl;
_getch();
return 0;
}

```

Результат работы программы:

Формирование списка

19 17 0 1 0

Реверс списка

0 1 0 17 19

Сортировка списка по возрастанию

0 0 1 17 19

Сортировка списка по убыванию

19 17 1 0 0

Работа метода for_each

190 170 10 0 0

19 17 1 0 0

Работа метода transform

190 170 10 0 0

Пример 28.5. Применение алгоритмов STL к целочисленному массиву.

```

#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
int big_arr[] = {2, 3, 6, 9, 5, 1, 8, 4, 7, 0};
int small_arr[] = {9, 5, 1, 8};
int i;
int *aPtr;
for (i = 0; i < 10; i++) cout << big_arr[i] << ' ';
cout << endl;
for (i = 0; i < 4; i++) cout << small_arr[i] << ' ';
cout << endl;
aPtr = search(big_arr, big_arr + 10, small_arr, small_arr + 4);
if (aPtr == big_arr + 10)
cout << "Подмассив не найден" << endl;
else
cout << "Первое из значений подмассива в массиве равно "
<< *aPtr << "\nни находится по адресу " << aPtr << endl;
_getch();
return 0;
}

```

Результат работы программы:

2 3 6 9 5 1 8 4 7 0

9 5 1 8

Первое из значений подмассива в массиве равно 9
и находится по адресу 0012FF48

Контрольные вопросы

- Из чего состоит ядро библиотеки стандартных шаблонов?
- Дайте определения контейнеров, алгоритмов, итераторов, функциональных объектов.
- Что представляет собой контейнерный класс?
- Фундаментальная структура данных (ФСД) – это . . .
- Перечислите методы, общие для всех контейнеров.

Практические задания 28

- Из конца вектора целого типа удалить число, если оно кратно значению, введенному с клавиатуры.
- Из начала вектора целого типа удалить число, если оно не кратно значению, введенному с клавиатуры.
- Даны два вектора, упорядоченные по возрастанию. Объединить их в один вектор.
- Сформировать динамический список, считая, что длина списка (количество элементов) задана. Описать функцию, которая удаляет из списка за каждым вхождением элемента E, значение которого введено с клавиатуры, один элемент, если такой есть и он отличен от E.
- Сформировать динамический список, считая, что длина списка (количество элементов) задана. Описать функцию, которая формирует список M1 – копию списка M, и список M2, представляющий собой «перевернутый» список M.
- Сформировать динамический список, считая, что длина списка (количество элементов) задана. Описать функцию, которая включает в упорядоченный по убыванию список новое значение, введенное с клавиатуры, таким образом, чтобы не нарушать упорядоченность.

ГЛАВА 29. СТРОКОВЫЙ КЛАСС

В C++ встроенный строковый тип отсутствует, но возможности для работы со строками все же существуют:

- использование символьного массива

```
char str[80];
```

- применение указателя на символьный тип

```
char *str;
```

- использование класса string

```
string s1("Я буду рядом – "); // инициализация
```

```
string s2("всюду и нигде"); // строк
```

```
string s3; // пустая переменная класса string
```

```
string s4(s1); // инициализация строки s4 строкой s1
```

Для того чтобы можно было работать с классом string, необходимо подключить заголовочный файл <string>. В классе string наряду с функциями обработки строк используются перегруженные операции:

```
s3 = s1; // присвоить строке s3 значение строки s1
s3 += s2; // s3 = "Я буду рядом – всюду и нигде"
```

Стандартные строковые шаблоны именуются basic_string, и все строковые объекты конструируются на базе этого шаблона класса.

Индексный доступ к элементам строки, объявленной как объект класса string, осуществляется обычным образом.

Строковый класс поддерживает большое количество операций, осуществляющих обработку строк. Он является классом-контейнером, поэтому поддерживает все алгоритмы библиотеки стандартных шаблонов.

В классе string поддерживается несколько конструкторов, но наиболее часто используются следующие три:

<code>string();</code>	// создает пустой объект типа string
<code>string(const char *строка);</code>	// обеспечивает преобразование их оканчивающейся нулем строки в объект типа string
<code>string(const string &строка);</code>	// объект типа string создается из другого объекта типа string

29.1. Ввод и вывод строк

Ввод и вывод строк осуществляется с помощью перегруженных операций << и >>. Но такой способ позволяет вводить строку только до первого пробела. Метод getline() позволяет вводить строки, содержащие пробелы, а также сообщения, состоящие из нескольких строк. Однако этот метод работает только для ввода оканчивающейся нулем строки и не подходит для объектов класса string.

Пример 29.1. Ввод строк различного вида.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ const int MAX_LEN = 180; // максимальная длина строки
    char str[MAX_LEN];
    cout << "Input string without spaces : ";
    cin >> setw(79) >> str; // ограничение буфера ввода
    cin.get();
    cout << "Inputed string :" << str << endl;
    cout << "Input string with spaces : ";
    cin.get(str,MAX_LEN);
    cout << "Inputed string :" << str << endl;
    cin.get(); // удаляет \n из буфера ввода
    cout << "Input string : ";
    cin.getline(str,MAX_LEN);
    cout << "Inputed string :" << str << endl;
    cout << "Input string : ";
    cin.getline(str,MAX_LEN);
    cout << "Inputed string :" << str << endl;
    getch();
    return 0;
}
```

Методы работы со строками, заканчивающимися нулевым байтом, были подробно рассмотрены в гл. 6.

29.2. Поиск в строке

В классе `string` существует множество методов поиска подстрок и отдельных символов в строке (табл. 29.1).

Таблица 29.1. Методы поиска в строке

Метод	Описание
<code>find</code>	Находит первое вхождение подстроки начиная с начала строки
<code>rfind</code>	Находит первое вхождение подстроки начиная с конца строки
<code>find_first_of</code>	Находит индекс первого вхождения любого из символов подстроки
<code>find_last_of</code>	Находит индекс последнего вхождения любого из символов подстроки
<code>find_first_not_of</code>	Находит индекс первого из символов, не входящих в подстроку
<code>find_last_not_of</code>	Находит индекс последнего из символов, не входящих в подстроку

Пример 29.2. Методы поиска в строке.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ string ABC = "abcdefghijklmnopqrstuvwxyz";
    string substr = "klmnop";
    // поиск строковых литералов
    cout << ABC.find("rstuv") << endl; // 17
    cout << ABC.rfind("fgh") << endl; // 5
    cout << ABC.find("lkji") << endl; // -
    cout << ABC.rfind("lkji") << endl; // -
    // поиск строковых объектов
    cout << ABC.find(substr) << endl; // 10
    // с указанием начальной позиции поиска
    cout << ABC.find(substr, 5) << endl; // 10
    // поиск символов из представленного набора
    cout << ABC.find_first_of("aeiou") << endl; // 0
    cout << ABC.find_last_of("aeiou") << endl; // 20
    cout << ABC.find_first_not_of("aeiou") << endl; // 1
    cout << ABC.find_last_not_of("aeiou") << endl; // 25
    getch();
    return 0;
}
```

Результат работы программы

```
17
5
4294967295
4294967295
10
10
0
20
1
25
```

29.3. Модификация строки

Модификация строки может заключаться в удалении, вставке или замене ее фрагмента. Эти действия выполняются с помощью методов, которые представлены в табл. 29.2.

Таблица 29.2.Методы модификации строк

Метод	Описание
append	Добавляет подстроку в конец строки
erase	Удаляет фрагмент из строки
insert	Вставляет подстроку в указанное место строки
replace	Заменяет одну подстроку на другую

Пример 29.3. Методы модификации строк.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); // подключение русификатора
string s1 = "Розовые лотосы удивительно красивы";
string s2 = " можно";
string s3 = " есть ";
string s4 = ", кажется, ";
cout << s1 << endl;
s1.erase(14, 12); // удаляем слово"удивительно"
// (12 символов, начиная с 15)
// заменяем слово "красивы" на слово "можно"
// добавляем слово "есть"
// добавляем слово ",кажется,"
// после слов "Розовые лотосы"
cout << s1 << endl;
_getch();
return 0;
}
```

Результат работы программы

Розовые лотосы удивительно красивы
Розовые лотосы, кажется, можно есть

29.4. Сравнение строк

Объекты класса string можно сравнивать, используя перегруженные операции или метод compare(). Равными считаются полностью идентичные строки; меньшей считается та, что стоит раньше в алфавитном порядке.

Пример 29.4. Сравнение строк с помощью перегруженных операций.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
```

```
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
string str1, str2;
cout << "Введите строку str1: ";
cin >> str1;
cout << "Введите строку str2: ";
cin >> str2;
if (str1 == str2) cout << "Строки равны" << endl;
if (str1 != str2) cout << "Строки не равны: ";
if (str1 > str2) cout << "строка str1 больше строки str2" << endl;
if (str1 < str2) cout << "строка str1 меньше строки str2" << endl;
_getch();
return 0;
}
```

Результат работы программы

Введите строку str1: qwertyu
Введите строку str2: qwertyu
Строки равны

Введите строку str1: qscwdv
Введите строку str2: awezsr
Строки не равны: строка str1 больше строки str2
Введите строку str1: abcd
Введите строку str2: abcd12
Строки не равны: строка str1 меньше строки str2

Перегруженные операции сравнения вызывают метод compare() шаблона basic_string, поддерживающий дополнительные параметры, которые можно использовать для сравнения подстрок.

Пример 29.5. Сравнение строк методом compare().

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <string>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
string str1, str2;
int cmp_res, pos = 2, cnt = 5;
cout << "Введите строку str1: ";
cin >> str1;
cout << "Введите подстроку str2: ";
cin >> str2;
// сравнение символов строки str2
```

```
// и cnt символов str1, начиная с str1[pos]
cmp_res = str1.compare(pos, cnt, str2);
if (cmp_res == 0) cout << "Equal" << endl;
else cout << "Not Equal" << endl;
cout << "Введите строку str1: ";
cin >> str1;
cout << "Введите подстроку str2: ";
cin >> str2;
// сравнение символов строки str2
// и символов str1
cmp_res = str1.compare(str2);
if (cmp_res == 0) cout << "Equal" << endl;
else cout << "Not Equal" << endl;
_getch();
return 0;
}
```

Результат работы программы

Введите строку str1: 0123456789

Введите подстроку str2: 23456

Equal

Введите строку str1: abcdefgh

Введите подстроку str2: abcdefgh

Equal

Введите строку str1: 0123456789

Введите подстроку str2: 23456789

Not Equal

Введите строку str1: abcdef

Введите подстроку str2: ABCDEF

Not Equal

В аргументе метода compare() вместо str2 может стоять лiteralная строка.

29.5. Получение информации об объекте класса string

В табл. 29.3 представлены методы, позволяющие получить фактическую информацию об объекте типа string.

Таблица 29.3. Методы для работы с объектом типа string

Методы	Тип возвращаемого значения	Описание
size() length()	size_type size_type	Возвращает количество символов в строке
capacity()	size_type	Возвращает максимальный потенциальный размер строки, не требующий повторного выделения памяти

Окончание табл. 29.3

Методы	Тип возвращаемого значения	Описание
max_size()	size_type	Возвращает максимально возможный размер объекта класса string. В зависимости от операционной системы и модели памяти это значение может совпадать с размером доступной памяти или с ее наибольшим непрерывным блоком
resize()	void	Метод используется для увеличения или уменьшения строки. В случае необходимости метод позволяет заполнить строку некоторым символом
reserve()	void	Резервирует дополнительную память для строки
empty()	bool	Возвращает true, если строка пустая, и false – если нет

Контрольные вопросы

1. Как осуществляется ввод и вывод строк?
2. Перечислите:
 - методы поиска подстрок и отдельных символов в строке;
 - методы модификации строки;
 - методы сравнения строк;
 - методы, позволяющие получить фактическую информацию об объекте типа string.

Практические задания

29

1. Данна строка слов, разделенных пробелами. Сформируйте новую строку, вставив перед каждым вхождением слова «and» запятую. Определите, сколько в строке симметричных слов.
2. Данна строка слов. Сформируйте новую строку, удалив пробелы, с которых может начинаться строка, а каждую внутреннюю группу пробелов замените одним пробелом. Подсчитайте количество слов в данной строке и количество слов, у которых первая и последняя буквы совпадают.
3. Данна строка слов, разделенных пробелами. Определите количество слов, которые встречаются более одного раза. Сформируйте строку из неповторяющихся слов.
4. Данна строка символов и некоторый символ п. Сформируйте новую строку, вставив после каждого вхождения символа п запятую. Определите самое большое слово в строке.

5. Даны строка слов, разделенных пробелами и запятыми. Подсчитайте количество слов в строке и сформируйте новую строку из самых длинных слов подстрок (заключенных между запятыми).
6. Даны строка слов, разделенных пробелами. Сформируйте новую строку, заменив каждую группу внутренних пробелов одним пробелом. Оставьте в строке только первые вхождения слов. Определите самое короткое слово.
7. Даны строка слов. Сформируйте новую строку, вставив перед каждым из слов «а» и «но» запятую. Подсчитайте количество подстрок, разделенных запятыми. Сформируйте строку из слов, с которых начинаются подстроки.

ПРИЛОЖЕНИЕ А. СВЕДЕНИЯ О СИСТЕМАХ СЧИСЛЕНИЯ

Ячейка памяти типичной микроЭВМ представляет собой 16 двоичных разрядов, называемых *битами* (binary digit). Бит является единицей информации в вычислительной технике. Кроме того, существует понятие байта. *Байт* – это 8 бит, расположенных последовательно. Номер байта называется его *адресом*. Байт – наименьшая адресуемая область памяти, так как бит адреса не имеет. Машинное слово (ячейка памяти) состоит из 2 байт.

Система счисления – это код, в котором используют специальные символы для обозначения количества каких-либо объектов.

В повседневной жизни применяется десятичная система счисления. В ней используются символы 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Общее количество символов в десятичной системе равно 10, поэтому ее называют системой счисления с основанием 10.

Введем понятие *веса разряда*. Представим число 542 в виде

$$5 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0.$$

Степень числа 10 представляет собой вес разряда.

Любое десятичное число можно представить в виде

$$M \sum_{i=k}^n d_i \cdot 10^i, \quad (A-1)$$

где d_i – число, соответствующее i -му разряду; i – вес разряда; 10 – основание системы счисления.

Обобщенная формула представляется как

$$M \sum_{i=k}^n d_i \cdot b^i, \quad (A-2)$$

где b – основание системы счисления.

В вычислительной технике широко используется двоичная система, или система с основанием 2. В ней применяются всего два символа: 0 и 1, поэтому ими удобно кодировать сигналы, с помощью которых управляются ЭВМ.

A-1. Десятично-двоичные и двоично-десятичные преобразования

Рассмотрим преобразование чисел из десятичной системы счисления в двоичную и обратно.