Программирование на языке C++

Глава 1

1.1. Принципы модульного программирования на языке C++

В языке C++ очень бедные средства модульного программирования, поэтому для достижения модульности программ, следует придерживаться определенных принципов.

Роль программного интерфейса модуля играет h-файл, а cpp-файл — роль реализации этого модуля. Внутрь h-файла включаются h-файлы других модулей, необходимые для компиляции интерфейсной части. Внутрь cpp-файла включаются h-файлы других модулей, необходимые для компиляции cpp- и h-файлов интерфейсной части модуля.

Очевидно, что программисту при включении h-файла другого модуля предоставляется выбор: подключить его в h-файле модуля или в cpp-файле. В данном случае предпочтение следует отдавать части реализации модуля (cpp-файл).

При подключении h-файла следует придерживаться следующей схемы: предположим, что наш модуль называется SysModule и состоит из двух частей: SysModule.h и SysModule.cpp. Рекомендуется следующая схема подключения:

SysModule.h:

```
#include "Config.h" // наш файл конфигурации
                    // подключается первым во всех h-файлах
                    // всех наших проектов
#include "Другой стандартный модуль"
#include "Другой наш модуль"
```

SysModule.cpp:

```
#include "Файл предкомпилированных заголовков"
#include "Еще один наш модуль"
#include "Другой стандартный модуль"
#include "SysModule.h" // подключается последним
```

Поскольку один и тот же h-файл может одновременно включатся в другие h-файлы и несколько раз подключаться при компиляции одного и того же cpp-файла, его следует защищать от повторной компиляции. Для этого в начале любого h-файла вставляются следующие директивы компилятора:

```
#ifndef __SysModule_h__
#define __SysModule_h__
...
#endif //__SysModule_h__
```

Таким образом, в том случае, когда файл подключается несколько раз, скомпилируется он только один раз.

Внимание! Согласно стандарту ISO, любой h- и cpp-файл в языке C++ должен заканчиваться символом перевода строки.

1.2. Пространства имен

В больших проектах наблюдается серьезная проблема — конфликт идентификаторов. Она решается с помощью пространства имен.

```
namespace Sys
{
  int  var;
  void Proc();
}
```

Внутри пространства имен обращение к определенным внутри переменным и подпрограммам можно осуществлять, используя неполную форму записи:

```
var = 10;
Proc();
```

за пределами – надо использовать полную форму записи:

```
Sys::var = 10;
Sys::Proc();
```

Для того чтобы избежать возможного конфликта идентификаторов, все определения внутри модуля следует помещать в пространство имен. Следует давать небольшой буквенный идентификатор, который будет соответствовать префиксу файла.

Существует возможность открыть пространство имен таким образом, чтобы можно было использовать неполную форму записи. Для этого надо написать строку:

```
using namespace Sys;
```

Но следует отметить, что данная конструкция является причиной многих ошибок, поэтому так писать не стоит.

Существует второй способ открыть пространство имен — это открыть его для конкретного определения:

```
using Sys::Proc();
...
Proc();
...
```

Но рекомендуется использовать Sys::Proc();

Идентификаторы, объявленные вне пространства имен, относятся к так называемому глобальному пространству имен, доступ к которым осуществляется с помощью оператора **::**

```
::Funk();
```

Для того чтобы была возможность закрыть доступ к данным и подпрограммам внутри данного пространства существует пространство имен без имени:

```
namespace
{
  ...
}
```

Пространства имен могут быть вложенными:

```
namespace Sys
{
  namespace Local
  {
    int var;
    ...
  }
  ...
}
Sys::Local::var = 10;
```

Замечание! Когда возникает желание объявить переменный тип данных или подпрограмму внутри пространства имен, а реализовать за пределами (или наоборот), следует поступать так:

SysModul.h:

```
namespace Sys
{
  int Proc();
}
```

SysModul.cpp:

```
namespace Sys
{
  int Proc();
   {
      ...
   };
 }
```

1.3. Перегрузка идентификаторов

В языке C++ можно определить несколько функций с одним и тем же именем. Это явление называется перегрузкой имени — **overloading**.

Эти функции должны отличаться по количеству и типу параметров:

```
void print(int);
void print(const char *);
void print(double);
void print(long);
void print(char);
```

Процесс поиска подходящей функции из множества перегруженных осуществляется путем проверки набора критериев в следующем порядке:

- полное соответствие типов;
- соответствие, достигнутое продвижением скалярных типов данных:

```
bool   - int
char   - int
short  - int
float  - double
double - long double
```

- соответствия, достигнутые путем стандартных преобразований:

```
int    - double
double - int
int    - unsigned int;
```

- соответствия, достигнутые за счет преобразований, определенных пользователем (перегрузка операторов в преобразовании типов);
- соответствия за счет многоточия в объявлении функции.

Если соответствие может быть достигнуто двумя способами на одном и том же уровне, то вызов функции неоднозначен и компилятор выдаст ошибку.

Пример:

```
void TestPrint(char c, int i, short s, float f)
{
   print(c);   // char
   print(i);   // int
   print(s);   // int
   print(f);   // double
   print('a'); // char
   print(49);  // int
   print(0);   // int
   print("a"); // const char*
}
```

Замечание! Перегрузку следует использовать в исключительных случаях. Ее следует использовать по типу параметров, а не по их смыслу или количеству. Так же алгоритм всех перегруженных функций должен быть идентичным (идентичная семантика).

1.4. Переопределенные элементы подпрограмм

Один или несколько последних параметров в заголовке функции могут содержать стандартные значения:

```
void print(int value, int base = 10); // base - система счисления
void print(..., int base)
{
  ...
}
```

При этом функция может вызываться либо print(100, 10) либо print(100).

При вызове данной функции компилятор автоматически подставляет значения для опущенных параметров.

Глава 2

2.1. Классы в языке C++

Классы в языке C++ определяются с помощью одного из ключевых слов: **class** или **struct.**

```
class TTextReader
{
  ... // private
};
```

```
struct TTextReader
{
  ... // public
};
```

В языке C++ доступны атрибуты доступа в классах:

- **public**
- **protected**
- **private**

В Delphi данные секции принято употреблять в порядке: **private...protected...public**. В языке C++ их можно чередовать.

В работе секций protected и private в Delphi и C++ есть различия:

- В Delphi классы внутри одного модуля могут обращаться к данным и подпрограммам друг друга без ограничений. А действие секций protected и private распространяется только за пределами данного модуля.
- В языке C++ действие этих секций распространяется на любые два класса. Но установленные ограничения можно обойти с помощью специального оператора **friend**:

```
class TTextReader
{
  friend class TList;
};
```

После этого объект класса TList может обращаться к полям из секций private и protected класса TTextReader.

2.2. Наследование

Наследование класса выполняется следующим образом:

```
class TDelimitedReader: public TTextReader
{
  ...
};
```

При наследовании указываются атрибуты доступа к элементам базового класса (public, protected, private). Для того чтобы понять смысл атрибута доступа к базовому классу, базовый класс следует рассматривать, как поле производного класса.

2.3. Конструкторы и деструкторы

Конструктор создает объект и инициализирует память для него (деструктор — наоборот).

```
class TTextReader
{
  public:
     TTextReader();
    ~TTextReader();
};
```

В Delphi стандартный деструктор является виртуальным. В языке C++ это определяет программист, если планируется создавать объекты в динамической памяти (по ссылке), деструктор необходимо делать виртуальным:

```
virtual ~TTextReader();
```

Создание объектов:

- по значению (на стеке):

```
TTextReader reader;
```

- по ссылке (в динамической памяти):

```
TTextReader*reader = new TTextReader(); //оператор new служит для размещения
объекта в динамической памяти
```

- с помощью оператора new:

```
TTextReader*reader = new (адрес ) TTextReader;
```

Таким способом объект создается по ссылке по указанному адресу памяти.

Разрушение объектов:

- если объект создан по значению (на стеке), его разрушение выполняется автоматически при выходе переменной за область видимости

```
{
  TTextReader reader;
  ...
} // здесь происходит разрушение объекта reader при автоматическом вызове
деструктора
```

- если объект создан в динамической памяти (по ссылке), он должен быть уничтожен с помощью оператора **delete**:

```
delete reader;
```

  При этом сначала происходит вызов деструктора, а затем — освобождение динамической памяти.

Так выглядит динамическое создание и разрушение объектов:

```
new:
  malloc();
  TTextReader();

delete:
  ~TTextReader();
  free();
```

2.4. Стандартные конструкторы

Если программист не определяет в классе конструкторы, то компилятор создает автоматически два конструктора:

- конструктор без параметров
- конструктор копирования

Пример:

```
class TTextReader
{
  public:
    TTextReader(); // конструктор без параметров
    TTextReader(const TTextReader&R); // конструктор копирования
}
```

Внимание! Если программист определил хотя бы один конструктор в класс — компилятор не создаст никаких стандартных конструкторов.

Конструктор без параметров создается для того, чтобы можно было написать:

```
TTextReader R;
```

Конструктор копирования нужен для следующей записи:

```
TTextReader R1 = R2; // означает TTextReader R1(R2);
```

Конструктор копирования вызывается в том случае, когда создаваемый по значению объект создается путем копирования другого уже существующего объекта.

Следует отметить, что запись:

```
TTextReader R1 = R2;
```

и два оператора:

```
TTextReader R1;
R1 = R2;
```

имеют схожий синтаксис, но разную семантику: в первом случае объект создается конструктором копирования, во втором — конструктором без параметров, а затем с помощью оператора ‘=’ выполняется присваивание одного объекта другому (данный вариант требует перегрузки оператора ‘=’ для класса TTextReader)

Работа стандартного конструктора копирования, создаваемого компилятором, заключается в том, чтобы выполнить полное копирование памяти с помощью функции **memcpy**.

2.5. Реализация методов класса

Метод класса может быть реализован по месту или отдельно от класса:

```
class TTextReader  // по месту
{
  public:
     TTextReader() { ... }
    ~TTextReader() { ... }
};

TTextReader::TTextReader() // отдельно от класса
{
  ...
}

TTextReader::~TTextReader()
{
  ...
}
```

Если класс описан в интерфейсной части модуля, его методы рекомендуется реализовывать отдельно от класса в cpp-файле. В том случае, когда некоторый класс надо сделать **inline**-методом, следует писать так:

```
class TTextReader
{
  public:
     TTextReader();
     ~TTextReader();
     int ItemCount();
};

inline int TTextReader::ItemCount()
{
  ...
}
```

2.6. Порядок конструирования и разрушения объектов

По причине автоматичности конструкторов и деструкторов в языке C++ существует определенный порядок конструирования базовых и агрегированных объектов.

```
class TTextReader
{
  public:
    TTextReader();
    ~TTextReader();
}

class  TDelimitedReader: public TTextReader
{
  public:
    TDelimitedReader();
    ~TDelimitedReader();
}

TDelimitedReader::TDelimitedReader()
{
  ...
}

TDelimitedReader::~TDelimitedReader()
{
  ...
}
```

В конструкторе производного класса конструктор базового класса вызывается автоматически до выполнения первого оператора в теле конструктора.

В деструкторе производного класса деструктор базового класса вызывается автоматически после последнего оператора в теле деструктора.

Если базовый класс содержит конструктор с параметрами или несколько конструкторов, то возникает неопределенность в том, какой конструктор базового класса будет вызван. Эту неопределенность можно устранить следующим образом:

```
TDelimitedReader::TDelimitedReader() : TTextReader(...)
// в скобках записываются параметры для вызова конструктора
{
  ...
}
```

После двоеточия разрешена запись списка операторов, разделенных запятыми. Эти операторы называются списком инициализации.

В языке C++ поддерживается множественное наследование. В этом случае конструктор базовых классов вызывается автоматически в порядке их упоминания в описании класса. Деструктор же базовых классов вызывается строго в обратном порядке.

Каждый конструктор перед началом своей работы инициализирует указатель **vtable** (в Delphi он называется **VMT**). Конструктор базового класса тоже инициализирует этот указатель. В результате этого объект как бы "рождается", сначала становясь экземпляром базового класса, а затем производного. Деструкторы выполняют противоположную операцию.

В результате этого в конструкторах и деструкторах виртуальные методы работают как невиртуальные.

2.7. Агрегированные объекты

В языке C++ объекты могут агрегироваться по ссылке и по значению (агрегирование по ссылке похоже на агрегирование в Delphi).

Агрегирование по значению:

```
class TDelimitedReader
{
  public:
    ...
  private:
    std::string m_FileName; // std::string - стандартный класс
                            // для представления строк
};
```

Агрегированные по значению объекты конструируются автоматически в порядке объявления после вызова конструктора базового класса (если не указан другой способ инициализации).

Стандартный способ инициализации можно переопределить до открывающей фигурной скобки конструктора:

```
TTextReader::TDelimitedReader() : TTextReader(), m_FileName("c:/myfile.txt")
{
  ...
}
```

Следует отметить, что данная запись отличается от следующей записи:

```
TDelimitedReader::TDelimitedReader() : TTextReader()
{
  m_FileName = "c:/myfile.txt";
}
```

Во втором случае строка вначале создается пустой, а в теле конструктора переприсваивается.

Объекты, агрегированные по ссылке, нужно создавать вручную с помощью оператора **new**, а удалять — с помощью оператора **delete**:

```
class TDelimitedReader : public TTextReader
{
  ...
  private:
    std::string m_FileName;
    TItems *m_Items;
};

TDelimitedReader::TDelimitedReader() : TTextReader(), m_FileName("c:/myfile.txt")
{
  m_Items = new TItems;
}

TDelimitedReader::~TDelimitedReader()
{
  delete m_Items;
}
```

Правило конструирования агрегированных объектов:

- объекты, агрегированные по значению, и константные ссылки инициализируются до тела конструктора.
- объекты, агрегированные по ссылке, инициализируются в теле конструктора.

2.8. Вложенные определения класса

В C++ внутри класса можно определить другой класс:

```
class TTextReader
{
  public:
    class TItems
    {
      ...
    };
  ...
};
```

Эта запись по смыслу соответствует следующей записи:

```
class TTextReader::TItems
{
  ...
};

class TTextReader
{
  friend class TTextReader::TItems; // см. ниже "Друзья класса"
};
```

Таким образом, определения классов и других типов данных внутри класса означает использование имени внешнего класса как пространства имен.

2.9. Друзья класса

Для того, чтобы объекты некоторого класса могли получить доступ в private и protected полям другого класса, используется оператор **friend**, который разрешает доступ ко всем записям и методам класса для того класса, который указан в операторе. Данный оператор используется внутри класса, с его помощью нельзя разрешать доступ к членам класса извне, иначе это нарушит принцип сокрытия данных.

2.10. Статические члены класса

Поля и методы класса могут быть объявлены при помощи слова **static**:

```
class TTextReader
{
  public:
    ...
    static char*ClassName();
    ...
  private:
    static int m_ObjectCount;
    ...
};
```

По смыслу данный код эквивалентен следующему:

```
class TTextReader
{
  friend char*TTextReader::ClassName();
  ...
};

class TTextReader::ClassName()
{
  ...
};

int TTextReader::m_ObjectCount;
```

Если поле объявлено с ключевым словом **static**, то это — обычная глобальная переменная, для которой имя класса используется как пространство имен.

Если метод объявляется с этим словом, то это — обычная глобальная функция, которая является другом класса. Такая функция не имеет псевдо-параметра **this.**

2.11. Множественное наследование

В языке C++ множественное наследование подразумевает, что у одного класса может быть несколько базовых классов:

```
class TDelimitedReader : public TTextReader, public TStringList
{
  ...
};
```

Объект класса TDelimitedReader содержит все поля и методы базовых классов TTextReader и TStringList. При этом в классе TDelimitedReader можно переопределять виртуальные методы каждого базового класса.

Множественное наследование имеет ряд проблем:

- отсутствие эффективной реализации (неэффективность скрыта от программиста);
- неоднозначность, возникающая из-за того, что в базовых классах могут быть одноименные поля, а также методы с одинаковой сигнатурой;
- повторяющийся базовый класс в иерархии классов.

Неоднозначность при множественном наследовании:

```
class TTextReader
{
  virtual void NextLine();
  ...
};

class TStringList
{
  public:
    virtual void NextLine();
  ...
};

class TDelimitedReader: public TTextReader, public TStringList
{
  ...
};

TDelimitedReader*Reader;
Reader->NextLine(); // Ошибка. Неоднозначность.
```

Неоднозначность возникает потому, что в классе TDelimitedReader существуют две таблицы виртуальных методов и неизвестно, к какой из них надо обращаться за методом NextLine(). Поэтому последний оператор должен быть скорректирован на следующий:

```
Reader->TTextReader::NextLine();
```

или:

```
Reader->TStringList::NextLine();
```

В языке C++ для классов поддерживается столько таблиц виртуальных методов, сколько у него базовых классов. При перекрытии общего виртуального метода, существующего в нескольких базовых классах, происходит замещение адреса во всех таблицах виртуальных методов.

Перегрузка функций по типам аргументов не приводит к разрешению неоднозначности.

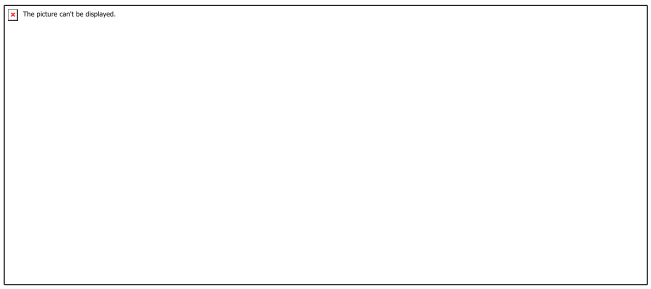
Если функция NextLine() была объявлена с различной сигнатурой в различных классах, то неоднозначность тоже остается.

В некоторых случаях наличие в базовых классах функций с одинаковыми именами (но различным количеством параметров или различными типами параметров) является преднамеренным решением. Чтобы в производном классе открыть нужную функцию нужного базового класса, применяется оператор **using**:

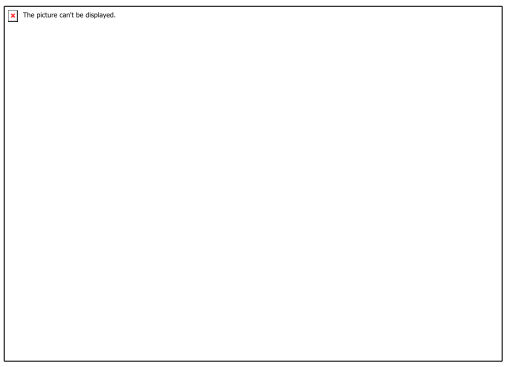
```
class TTextReader
{
  public:
    virtual void NextLine();
  ...
};

class TStringList
{
  public:
    virtual void NextLine(int);
  ...
};

class TDelimitedReader : public TTextReader, public TStringList
{
  public:
    using TStringList::NextLine;
    virtual void NextLine(int);
  ...
};
```

2.12. Проблема повторяющихся базовых классов

Классы TStringList и TTextReader в нашем примере могут иметь одинаковый базовый класс, например, TObject. Это приводит к следующей иерархии классов:



В этом случае объект класса TDelimitedReader имеет две копии полей класса TObject.

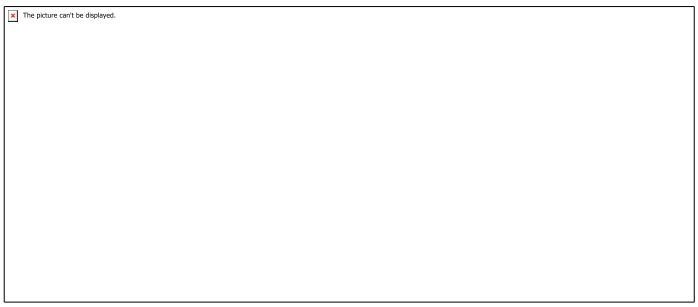


Из-за дублирования полей возникает неоднозначность при обращении к полю класса TObject из метода класса TDelimitedReader. Проблема решается с помощью уточненного имени:

```
TTextReader::m_Field;
TStringList::m_Field;
```

Однако главная проблема состоит в том, что одна сущность дублируется внутри базового класса.

На практике обычно требуется получать следующий результат:

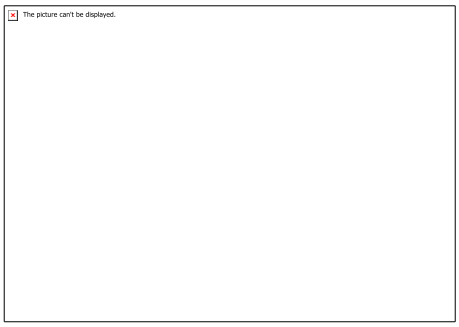


Такой результат достигается при применении виртуальных базовых классов:

```
class TDelimitedReader: public TTextReader, public TStringList
{
  ...
};

class TTextReader: public TObject
{
  ...
};

class TStringList: virtual public TObject
{
  ...
};
```

Обычное наследование соответствует агрегации всех полей базового класса. Виртуальное наследование соответствует агрегации ссылки на поля базового класса.

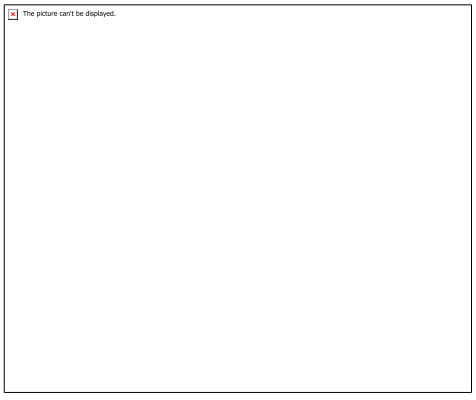
В данном случае структура полей в памяти будет следующей:



Если же при объявлении класса TTextReader мы запишем следующее:

```
class TTextReader: virtual public TObject
{
  ...
};
```

то структура полей будет такой:



Таким образом, множественное наследование таит следующую проблему: заранее неизвестно от каких классов программист захочет унаследовать свой класс. Однако при создании класса использовать виртуальное наследование неэффективно, если наследуются поля, так как доступ к полям всегда будет осуществляться через дополнительный указатель.

Вывод: одинарное наследование в стиле Java, C++, Delphi допустимо только от классов, множественное — от интерфейсов. Иначе можно осуществлять множественное наследование лишь от классов, в которых отсутствуют поля.

Глава 3

3.1. Виртуальные методы

В языке C++ виртуальные методы определяются при помощи ключевого слова **virtual**:

```
class TTextReader: virtual public TObject
{
  ...
};
```

При перекрытии виртуального метода ключевое слово **virtual** можно записать, а можно и опустить. Синтаксис перекрытия виртуальных методов не предусматривает такие проблемы, как версионность и рефакторинг кода (упрощение программного кода с сохранением функциональности).

Если метод виртуальный следует всегда писать ключевое слово **virtual**.

3.2. Абстрактные классы

В C++ абстрактный класс объявляется следующим образом:

```
class TTextReader
{
  protected:
    virtual void NextLine() = 0;
  ...
};
```

Такой метод называется абстрактным и класс, содержащий данный метод, тоже называется абстрактным.

Виртуальные методы следует объявлять в секции **protected**.

3.3. Подстановочные функции

В языке C существует способ оптимизировать вызов функций с помощью макросов (#**define**).

В языке C++ использование этого ключевого слово должно быть минимизировано, так как макросы обрабатываются препроцессором и в большинстве компиляторов символические имена макросов не видны в отладчике. Кроме того, макросы сильно затрудняют отладку программы, так как отладить макрос невозможно.

Вместо макросов в языке C++ используются подстановочные функции. Они определяются с помощью ключевого слова **inline**:

```
inline int Min(int x, int y)
{
  if (x < y)
    return x;
  else
    return y;
};
```

Тело подстановочной функции в большинстве случаев подставляется в код программы вместо ее вызова. Она должна быть целиком определена в h-файле. Типичной ошибкой программиста является размещение этой функции в cpp-файле и вынос ее прототипа в h-файл.

Если записана директива **inline**, это еще не означает, что компилятор подставляет тело функции в место ее вызова — он сам решает, что будет более удобным в данном случае.

3.4. Операторы преобразования типа

Существует четыре оператора преобразования типа в языке C++:

```
reinterpret_cast<тип>(переменная)
static_cast<тип>(переменная)
const_cast<тип>(переменная)
dynamic_cast<тип>(переменная)
```

Первый оператор (**reinterpret_cast**) позволяет отключить контроль типов данных на уровне компилятора, с помощью него любой указатель может быть интерпретирован, как любой другой указатель, а также любая память или переменная может быть интерпретирована иначе.

В программах этот оператор преобразования типа использовать не следует, так как он нарушает переносимость программ. Его наличие свидетельствует о том, что программа является кроссплатформенной. Пример использования:

```
int i;
char *p;
p = reinterpret_cast<char>(&i);
```

Второй оператор (**static_cast**) используется вместо преобразования тип(переменная), (тип)переменная и (тип)(переменная) при работе с классами, структурами и указателями на них.

Оператор static_cast был задуман по причине того, что в языке C++ выражение тип(переменная) может оказаться вызовом конструктора. Если в программе требуется преобразовать тип, а не вызвать конструктор типа, используется данный оператор. Кроме того, оператор (тип)переменная или (тип)(переменная) может в некоторых случаях оказаться преобразованием reinterpret_cast<тип>(переменная), а при разработке кроссплатформенных программ оператор reinterpret_cast всегда содержит потенциальную опасность неправильной работы программы на другой платформе. Поэтому вместо операторов тип(переменная), (тип)переменная и (тип)(переменная) следует использовать операторы reinterpret_cast и static_cast, которые убирают не явность из преобразования.

Так как оператор static_cast является громоздким, то для простых типов данных допустимо использование форм: (тип)переменная и (тип)(переменная). Форма тип(переменная) не должна использоваться для преобразования типа.

Третий оператор (**const_cast**) используется для приведения не константных указателей к константным и наоборот:

```
void f2(char *s);
void f1(const char *s)
{
  ...
  f1(const char *s);
  ...
  f2(const_cast<char>(s))
  ...
};
```

При объявлении переменных и параметров функций в описании типа может быть указано ключевое слово **const.**

Объявление f(const char *s); означает, что символы, адресуемые указателем s, изменять нельзя.

Объявление f(char const *s) означает, что указатель s изменять нельзя.

Так же можно сделать объявление: f(const char const *s), которое будет означать, что ни указатель, ни переменную изменять нельзя.

Если в программе объект объявлен с помощью модификатора **const**, то у него можно вызывать лишь те методы, которые объявлены с этим же модификатором:

```
class TTextReader
{
  public:
    int ItemCount() const;
  ...
};
```

Наличие константных объектов порождает проблему — огромная избыточность программного кода. Заранее программист не знает, будет ли пользователь (другой программист) его класса создавать константные объекты. Вследствие того, что это не исключено, программист начинает записывать слово **const** в объявление всех методов, в которых его можно записать. Многие методы являются виртуальными или вызывают виртуальные методы. Случается, что в производных классах виртуальные методы, вызванные константными методами, модифицируют поля объектов (это требуется по условию задачи). Это приводит к логической проблеме, которая решается либо за счет применения оператора **const_cast** к указателю **this** в производных классах, либо за счет объявления полей в производных классах с модификатором **mutable** (записывается при описании полей класса в том случае, если они должны модифицироваться константными методами). Пример:

```
mutable int m_RefCount;
```

Так же решить проблему можно при помощи перегрузки метода класса без модификатора **const**:

```
class TTextReader
{
  public:
   int ItemCount() const;
   int ItemCount();
   int ItemCount() const volatile;
   int ItemCount() volatile;
  ...
};
```

Варианты объявления:

```
volatile TTextReader r;
const volatile TTextReader r;
const TTextReader r;
TTextReader r;
```

Ключевое слово **volatile** запрещает кэшировать значение переменной. Если в программе происходит считывание значения переменной, объявленной с этим ключевым словом, то значение считывается из памяти, а не из регистров, а запись всегда производится в память, в которой размещается данная переменная.

Если ключевое слово **volatile** не указано, то оптимизатор C++ имеет право выполнять регистровые операции (оптимизации) при чтении и записи переменных, а также размещать их в регистрах.

Четвертый оператор (**dynamic_cast**) соответствует оператору **as** в Delphi. Для работы этого оператора нужно в опциях компилятора включить опцию RTTI. Если это выполнено, то оператор dynamic_cast работает, как static_cast.

Оператор dynamic_cast работает по-разному в зависимости от того, применяется он к ссылке на объект (&) или указателю на объект (*). Если оператор применяется к ссылке на объект, то преобразование не может быть выполнено и возникает исключительная ситуация. Если он применяется к указателю на объект и преобразование не может быть выполнено, оператор возвращает NULL.

3.5. Размещающий оператор new

Обычно оператор **new** размещает объекты в динамической памяти (**heap**).

```
TDelimitedReader *R = new TDelimitedReader();
```

В данном случае оператор new имеет следующий вид:

```
void *operator new(size_t size);
```

Существует вид оператора new, который позволяет расположить объект по заданному адресу:

```
TDelimitedReader *R = new (Buffer)TDelimitedReader();
```

В этом же случае оператор new имеет следующий вид:

```
void *operator new(size_t size, void *p)
{
  return p;
}
```

Внимание! Комбинировать различные способы выделения и освобождения памяти не рекомендуется.

3.6. Ссылки

Ссылка является альтернативным именем объекта и объявляется следующим образом:

```
int i;
int &r = i;
```

Использование ссылки r эквивалентно использованию переменной i.

Основное применение ссылок — передача параметров в функцию и возврат значения.

В случае, когда ссылка используется в качестве параметра функции, она объявляется неинициализированной:

```
void f(int &i);
```

Во всех остальных случаях ссылка должна инициализироваться при объявлении, как показано ранее.

Если ссылка является полем класса, она должна инициализироваться в конструкторе класса в списке инициализации до тела конструктора.

При использовании в качестве параметров функций ссылки соответствуют **var**-параметрам в языке Delphi:

```
procedure P(var I: Integer)
begin
  ...
end;
```

Константные ссылки соответствуют **const**-параметрам в языке Delphi:

```
procedure P(const I: Integer)
begin
  ...
end;
```

При передаче ссылочного параметра в стек заносится адрес переменной, а не ее копия.

Ссылку следует рассматривать, как псевдоним переменной, которой она инициализирована. Ссылки отличаются от указателей тем, что позволяют компилятору лучше оптимизировать программный код.

Для возврата значений из процедур (через параметры) предпочтение следует отдавать указателям, а не ссылкам. Ссылки следует использовать лишь в тех случаях, когда известно, что возвращаемый объект должен создаваться не в динамической памяти, а на стеке, то есть ссылки применяют при возврате value-type-объектов.

При работе со ссылками существует типовая ошибка — возврат через ссылку переменной, созданной на стеке. Пример ошибочной записи:

```
void f(int *p)
{
  int i;
  p = &i;
}
```

Следующая запись тоже будет ошибочной:

```
void f(int &r)
{
  int i;
  r = i;
}
```

Следующий пример тоже ошибочен, так как нельзя возвращать адрес объекта, созданного на стеке:

```
std::string& GetName(Object* Obj)
{
  const char* str = Obj->GetName();
  return std::string(str);
}
```

3.7. Обработка исключительных ситуаций

В языке C++ отсутствует аналог блока **try…finally…end**.

На платформе Windows благодаря структурной обработке ОС существуют следующий блок:

```
__try
{
  ...
}
__finally
{
  ...
}
```

Но следует отметить, что для переносимых программ он не подходит.

В языке C++ существует аналог блока **try…except…end**:

```
try
{
  ...
}
catch(std::ios_base::failure)
{
  ...
}
catch(std::exception)
{
  ...
}
catch(...)
{
  ...
}
```

Распознавание исключительных ситуаций происходит последовательно блоками **catch**, поэтому их последовательность должна быть от частного к общему.

Последний блок **catch** в примере выше ловит любую исключительную ситуацию.

Создание исключительных ситуаций выполняется с помощью оператора **throw** (аналог **raise** в Delphi):

```
throw std::exception("Ошибка");
```

Внутри блока **catch** оператор **throw** возобновляет исключительную ситуацию, как и **raise** в Delphi.

При создании исключительной ситуации при помощи оператора **throw** объект, описывающий исключительную ситуацию, может быть создан в динамической памяти:

```
throw new std::exception("Ошибка");
```

Если применяется такой способ создания исключительной ситуации, ее уничтожение должно происходить следующим образом:

```
try
{
  ...
  throw new std::exception("Ошибка");
}
catch(std::exception *e)
{
  delete e;
}
catch(...)
{
  ...
}
```

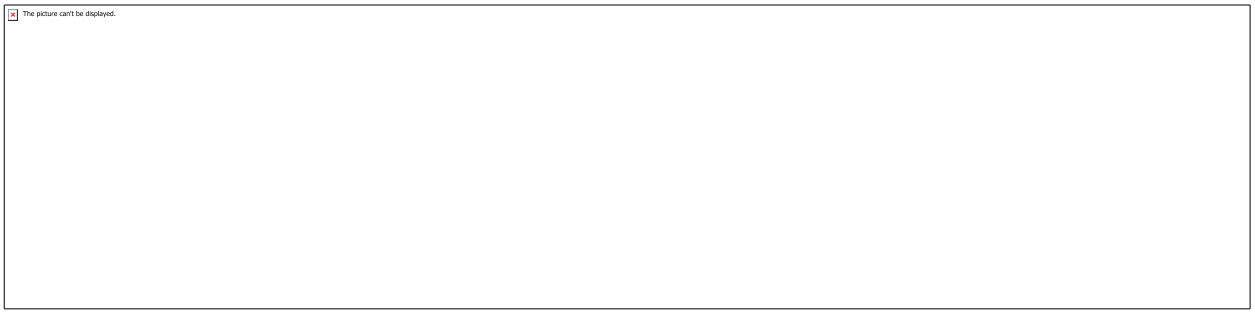
Если же записать так:

```
try
{
  ...
  throw new std::exception("Ошибка");
}
catch(...)
{
  ...
}
```

то возникнет утечка ресурсов из-за того, что объект std::exception, созданный в динамической памяти, не будет освобожден.

В языке C++ отсутствует общий базовый класс для исключительных ситуаций, поэтому на верхнем уровне работы программы нужно отлавливать все возможные базовые классы исключительных ситуаций. Это является препятствием на пути построения расширяемых систем.

Program



Program

3.8. Защита ресурсов от утечки

Поскольку в языке C+ отсутствует блок **try…finally**, его приходится эмулировать.

```
Object *p = new Object();
try
{
  ...
}
catch(...)
{
  delete p;
  throw;
}
delete p;
```

Данный код эквивалентен следующему:

```
Object *p = new Object();
__try
{
  ...
}
__finally
{
  delete p;
}
```

за исключение того, что второй пример не является переносимым.

Согласно стандарту языка C++ в деструкторах и операторах delete не должно быть исключительных ситуаций, если же исключительная ситуация произошла, то поведение программы не определено.

Если исключительная ситуация происходит в конструкторе объекта, объект считается не созданным и деструктор для этого объекта не вызывается, но память, выделенная для объекта, освобождается.

Если внутри объекта агрегированы другие объекты, то вызываются деструкторы лишь для тех объектов, которые были полностью созданы к моменту возникновения исключительной ситуации.

Если объект создается в динамической памяти и освобождается в той же самой процедуре, то для защиты от утечки ресурсов можно применять оболочечные объекты — **wrapper** (содержит указатель на динамический объект, который уничтожается в деструкторе оболочечного объекта). Оболочечный элемент создается на стеке, поэтому его деструктор вызывается автоматически, гарантируя тем самым уничтожение агрегированного динамического объекта.

Такие оболочечные объекты в библиотеках программирования называются **AutoPtr**, **SafePtr** и т.д.

```
class AutoPtr
{
  public:
    AutoPtr(int *arr);
    ~AutoPtr();
  private:
    int *m_arr;
};

AutoPtr::AutoPtr(int *arr)
{
  m_arr = arr;
}

AutoPtr::~AutoPtr()
{
  delete[] m_arr;
}

void Proc()
{
  int *arr = new int[100];
  AutoPtr autoc(arr);
  ...
}
```

3.9. Перегрузка операторов

Перегрузка операторов позволяет заменить смысл стандартных операторов (+, –, = и др.) для пользовательских типов данных.

В языке C++ разрешена перегрузка операторов, выраженных в виде символов, а также операторов:

```
new         delete
new[]       delete[]
```

Запрещена перегрузка следующих операторов:

```
::  .  .*  ?:
```

Перегрузка операторов таит угрозу: она резко усложняет понимание программы, поэтому ей пользоваться нужно очень осторожно. Для стандартных типов данных перегрузка запрещена, хотя бы один из операторов должен принадлежать пользовательскому типу данных.

Бинарные операторы

Бинарный оператор можно определить либо в виде нестатической функции членов с одним аргументом, либо в виде статической функции с двумя аргументами.

Для любого бинарного оператора @ выражение aa@bb интерпретируется как aa.operator@(bb) или operator@(aa, bb). Если определены оба варианта, то применяется механизм разрешения перегрузки функций.

Пример:

```
class X
{
  public:
    void operator +(int);
  X(int);
};

void operator +(X, X);
void operator +(X, double);

void Proc(X, a)
{
  a + 1;   // a.operator +(1)
  1 + a;   // ::operator +(X(1),a)
  a + 1.0; // ::operator +(a, 1.0)
}
```

Унарные операторы

Унарные операторы бывают префиксными и постфиксными.

Унарный оператор можно определить в виде метода класса без аргументов и в виде функции с одним аргументом. Аргумент функции — объект некоторого класса.

Для любого префиксного унарного оператора выражение @aa интерпретируется как:

```
aa.operator @();
operator @(aa);
```

Для любого постфиксного унарного оператора выражение aa@ интерпретируется, как:

```
aa.operator @(int);
operator @(aa, int);
```

Запрещено перегружать операторы, которые нарушают грамматику языка.

Существует три оператора, которые следует определить внутри класса в виде методов:

```
operator =
operator []
operator ->
```

Это гарантирует, что в левой части оператора будет записан **lvalue** (присваиваемое значение).

Операторы преобразования

В языке C++ существуют операторы преобразования типов. Это является хорошим способом использования конструктора для преобразования типа. Конструктор не может выполнять следующие преобразования:

- неявное преобразование из типа, определяемого пользователем в базовый тип. Это связано с тем, что базовые типы не являются классами.
- преобразование из нового класса в ранее определенный класс, не модифицируя объявление ранее определенного класса.

Оператор преобразования типа:

```
X::operator T() // определяет преобразования класса X в тип данных T (T — класс
или базовый тип)
Пример:
class Number
{
  public:
    operator int();
  operator Complex();
};

Number::operator int()
{
  ...
}

Number::operator Complex()
{
  ...
}
```

Оператор преобразования типа возвращает значение типа T, однако в сигнатуре оператора он не указывается. В этом смысле операторы преобразования типа похожи на конструкторы.

Хотя конструктор не может использоваться для неявного преобразования типа из класса в базовый тип, он может использоваться для неявного преобразования типа из класса в класс.

В программе следует избегать любых неявных преобразований типов, так как это приводит к ошибкам.

С помощью ключевого слова **explicit** можно запретить неявное преобразования типа к конструкторам.

Пример:

```
class File
{
  public:
  explicit File(const char *name); // одновременно не могут быть определены, надо
выбирать один из них
  explicit File(const char *name, int mode = FILE_READ);
};
```

Слово **explicit** записывается лишь для тех конструкторов, которые могут вызываться лишь с одним параметром. Если же они вызываются с несколькими параметрами, то неявное преобразование типов невозможно.

Если объект создается на стеке, то неявное преобразование типа часто бывает необходимо, тогда слово **explicit** писать надо. Так же его надо писать, когда объект создается динамически.

При перегрузке операторов нужно быть внимательным к типу возвращаемого значения: для некоторых операторов объект возвращается по ссылке, для некоторых — по значению:

```
X operator; // по значению
X &operator; // по ссылке
```

Для некоторых операторов возможен и первый и второй вариант перегрузки, поэтому программисту следует определяться с вариантом перегрузки.

Замечание по поводу преобразования типа в тернарном операторе (c ? x : y).

```
class A { ... };
class B: public A { ... };
class C: public A { ... };
```

Запись

```
A* p = cond ? new B : new C;
```

вызовет ошибку компилятора, поскольку между типами выражений "new B" и "new C" выбирается общий тип, а такого нет. Ошибку следует устранить, выполнив преобразование "new B" или "new C" к общему типу, например:

```
A* p = cond ? (A*)new B : new C;
```

или

```
A* p = cond ? new B : (A*)new C;
```

или

```
A* p = cond ? (A*)new B : (A*)new C; // самый лучший вариант
```

Глава 4

4.1. Шаблоны

Шаблоны обеспечивают непосредственную поддержку обобщенного программирования. Они представляют собой параметризованные классы и параметризованные имена функций.

Шаблон определяется с помощью ключевого слова **template**:

```
template <class T>
class basic_string
{
  public:
    basic_string();
    basic_string(const T*);
    basic_string(const basic_string&);
  private:
    T*str;
};

typedef basic_string<char> string;
typedef basic_string<unsigned int> wstring;
```

Вместо слова **typename** часто записывают слово **class**, но параметром шаблона может быть любой тип данных. С точки зрения компилятора, шаблон является макроподстановкой, поэтому шаблонные классы определяются целиком в заголовках файлов (в h-файле, а не в cpp-файле).

Методы шаблона описываются следующим образом:

```
template <class T>
basic_string<T>::basic_string(const *T)
{
  ...
}
```

4.2. Шаблоны функций

Допускается применение шаблонов с целью реализации абстрактных алгоритмов, то есть шаблонов функций.

```
template <class T>
void sort(vector<T>& v);
```

При вызове шаблонных функций компилятор подставляет тип данных и создает новый вариант функции. Если один и тот же тип данных используется несколько раз, то на все типы данных используется несколько раз, то на все типы данных создается один шаблон функции.

При использовании шаблонов существует три больших недостатка:

- шаблоны невозможно отлаживать.

  Разработка шаблонов обычно ведется так:

  1. разрабатывается класс или функция конкретного типа данных;
  2. этот класс или функция параметризуются, то есть создается шаблон.

- существенно замедляется время компиляции. В больших проектах оно может доходить до 30-60 минут.

- очень быстро растут размеры объектных модулей и библиотек на диске.

Компиляция относительно большого проекта в отладочном режиме может потребовать до 10 ГБ.

4.3. Перегрузка шаблонов функций

Можно объявить несколько шаблонов функций с одним и тем же именем, а так же можно объявить комбинацию шаблонов и обычных функций с одинаковым именем:

```
template <class T> T sqrt(T x)

template <class T>
  complex<T> sqrt(complex<T> x);

template <class T>
  double sqrt(double x);

void Proc(complex<double> z)
{
  sqrt(2);   // sqrt<int>(int)
  sqrt(2.0); // sqrt(double)
  sqrt(z);   // sqrt<double>(comlex<double>)
}
```

Шаблонные функции могут вызываться с явным указанием параметра шаблона:

```
sqrt<int>(2);
```

или без него:

```
sqrt(2);
```

В этом случае применяется механизм разрешения перегрузки:

- ищется набор специализации шаблонов функций, которые примут участие в разрешении перегрузки;

- если могут быть вызваны два шаблона функций и один из них более специализирован, то только он и будет рассматриваться;

- разрешается перегрузка для этого набора функций и любых обычных функций. Если аргументы функции шаблона были определены путем выведения по фактическим аргументам шаблона, к ним нельзя применять “продвижение” типа, стандартные и определяемые пользователем преобразования.

- если и обычная функция, и специализация подходят одинаково хорошо, предпочтение отдается обычной функции;

- если ни одного соответствия не найдено, или существует несколько одинаково хорошо подходящих вариантов, то выдается ошибка.

В параметрах шаблонов допустимы стандартные значения, принимаемые по умолчанию:

```
class Allocator
{
  ... // malloc, free, calloc
};

template <class T, class A = Allocator>
class basic_string
{
  ...
};

basic_string<char, MyAllocator> mystr;
basic_string<char> commonstr;
```

4.4. Специализации шаблонов

Как правило, шаблон представляет единственное определение, которое применяется к различным аргументам шаблона. Это не всегда удобно, иногда существует необходимость использовать различные реализации в зависимости от типа.

Например, надо для всех указателей использовать особую реализацию шаблона, а для всех базовых типов данных — обычную реализацию. Это делается с помощью специализации шаблона:

```
template <class T>
class vector
{
  ...
};

template <>
class vector<void*>
{
   ...
};

void Proc()
{
  vector<int> vi;
  vector<void*> vp;
  ...
}
```

Также может применяться частичная специализация шаблонов:

```
template <class T>
class vector<T*> : private vector<void*>
{
  ...
};

template <>
class vector<void*>
{
  ...
};
```

Специализация шаблонов, как правило, используется для сокращения объема программного кода. Если шаблон создается для указателей на какие-то объекты и класс объекта не так важен, то при использовании обычных шаблонов без специализации возникает многократное дублирование одного и того же кода. Это связано с тем, что в машинных кодах работа со всеми указателями строится одинаково. Чтобы избежать дублирования кода в случае использования указателей следует создавать специализации шаблонов.

4.5. Использование шаблонов при создании новых типов данных

На основе шаблонов можно создать новый типа данных, использование которого позволит упростить программный код и избежать возможных ошибок.

Существует два способа создания нового типа данных на базе шаблона:

- можно воспользоваться оператором **typedef**:

```
typedef basic_string<char> string;
```

Далее типом данных string можно пользоваться, забыв, что это шаблон.

На самом деле для компилятора оператор **typedef** является как бы макроподстановкой, то есть при использовании типа string компилятор всегда подставляет значение типа basic_string<char>.

Если шаблонные классы агрегируют другие шаблонные классы и порождаются от шаблонных классов, то длина программного идентификатора внутри объектного модуля может быть очень велика. Порой она превышает 4 КБ. Некоторые компиляторы на платформе Unix имеют ограничения на длину программного идентификатора внутри объектного модуля. Поэтому при использовании шаблонов (в особенности библиотеки **stl**) нередко возникает проблема с компиляцией.

Например, выражение:

```
string::iterator;
```

на самом деле в объектном модуле выглядит так:

```
basic_string<char>::iterator;
```

Оператор **typedef** по существу отличается от раздела type в Delphi. В C++ упрощение записи — макроподстановка.

- наследование

```
class string : public basic_string<char>
{
  ...
};
```

При наследовании создается именно новый тип данных, поэтому в объектный модуль в данном случае помещается идентификатор следующего вида:

```
string::iterator;
```

Таким образом, использование наследования позволяет решить проблему длины идентификаторов при работе с шаблонами.

4.6. Стандартная библиотека шаблонов

Перечислим, что содержится в стандартной библиотеке шаблонов:

- **Классы и шаблоны для организации потоков ввода/вывода**

В языке C++ вместо функций **printf** и **scanf** предлагается использовать объекты потоковых классов:

```
std::cout;
std::cin;
```

Вывод осуществляется с помощью оператора сдвига:

```
std::cout << "Hello!";
int n;
std::cin >> n;
```

Чтобы перевести строку надо сделать следующую запись:

```
std::cout << "Hello!" << std::endl; // или "\n"
```

Объекты **cout** и **cin** являются экземплярами классов **ostream** и **istream**. Существуют также классы **iostream** (класс для ввода/вывода) и **streambuf** (позволяет выполнить буферизованный ввод/вывод).

В программе не следует смешивать потоковый ввод/вывод с функциями **printf** и **scanf**. Если все же это происходит, то между блоками кода, использующими тот или иной подход, надо выполнять вызов функции **fflush** — сброс буферов.

- **Ссылки**

Для работы с ссылками подключается файл:

```
#include <string>
```

Расширение «.h» у файлов стандартной библиотеки отсутствует.

Среди строк наибольшей популярностью пользуются следующие классы:

```
std::string
std::wstring
```

- **Контейнеры**

В языке C++ существуют следующие контейнеры:

```
std::vector<T> // обычный массив
std::list<T>   // список, реализация не уточняется.
std::queve<T>  // FIFO
std::deque<T>  // двунаправленная очередь
std::stack<T>  // LIFO
std::map<K,T>  // список объектов T, индексированных ключами K
std::set<T>    // множество
std::bitset    // массив битов
```

Для использования любого из шаблонов надо подключить заголовочный файл, название которого совпадает с названием подключаемого шаблона.

В примере выше кроме параметра <T> есть еще, как правило, три параметра. Здесь они преднамеренно опущены и для них заданы стандартные значения.

Например, среди этих параметров есть **allocator** — это класс, который отвечает за создание или удаление элементов контейнера. Благодаря нему, можно создавать элементы в любой области памяти.

- **Итераторы**

```
#include <iterator>
```

Итератор — абстракция указателя на элемент контейнера.

Пример использования:

```
#include <iterator>
void strlen(const cahr *str)
{
  const char *p = str;
  while(*p != 0)
    {
      ...
      ++p;
    }
  ...
}
```

Указатель в строке — это итератор по строке.

Можно сказать, что в примере выше типы данных char* и const char* являются итераторами строки (обычной 0-терминированной).

С помощью оператора **typedef** можно определить такой тип данных и затем их использовать:

```
typedef char *iterator;
typedef const char *const_iterator;
```

Внутри каждого контейнера стандартной библиотеки C++ определены два типа данных: **iterator** и **const_iterator**, которые фактически являются указателями на элемент контейнера.

Работа с этими типами данных происходит следующим образом:

```
void Find(std::vector<MyObject*> &v, const char *s)
{
  typename std::vector<MyObject*>::iterator it = v.begin();
  while (it != v.end())
  {
    const char *szName = (*it).Name;
    if (strcmp(szName, s) == 0)
      return true;
    ++it;
  }
  return false;
}
```

В стандартном контейнере существуют функции begin() и end(), которые возвращают соответственно итераторы на первый и последний элементы контейнера. Физически функция end() возвращает **NULL**.

Если итератор адресует объект, то доступ к полям следует осуществлять с помощью оператора *. Допустимо использование и оператора ->, но он может быть переопределен и поэтому работа операторов *. и -> может отличаться. Переход к следующему элементу контейнера выполняется префиксным инкрементом ++it. Допустимо использование постфиксного оператора it++, но в последнем случае может возникнуть неоднозначность в том, что увеличивается на единицу — итератор или значение, возвращаемое итератором.

- **Алгоритмы**

```
#include <algorithm>
#include <cstdlib>
```

В стандартной библиотеке существует несколько стандартных алгоритмов, которые следует изучить до того, как разрабатывать собственные алгоритмы над контейнерами.

```
find()
sort()
*bsearch() // бинарный поиск
*qsort()   // быстрая сорировка
```

Недостатком **bsearch** является то, что не возвращается место вставки элемента.

- **Утилиты**

```
#include <utility>
#include <functional>
#include <memory>
#include <ctime>
```

В **utility** переопределен шаблон pair<F,S>.

В **fuctional** определены объекты функций — это такие объекты, которые могут использоваться как функции. Объекты функций применяются в алгоритмах. Например, выполняется обход контейнера и вызывается некоторая функция для каждого элемента контейнера. Вместо функции можно подставить объект-функцию. В этом и состоит смысл объекта-функции.

memory — распределение памяти для контейнера (здесь находятся стандартные алгоритмы).

В **ctime** находятся функции времени и даты.

- **Диагностика**

```
#include <stdexcept> // классы исключительных ситуаций
#include <cassert>   // макрос assert. Вместо него лучше использовать
                     // исключительные ситуации
#include <cerrno>    // файлы обработки ошибок в стиле C, то есть функции,
                     // которые возвращают коды ошибок
```

- **Локализация**

Поддержка различных языков:

```
#include <locale>
#include <clocale>
```

- **Поддержка языка программирования C++**

Крайние граничные значения типов данных:

```
#include <limits>
#include <climits>
#include <typeinfo>  // поддержка RTTI
#include <exception> // поддержка исключительных ситуаций
```

- **Работа с числами**

```
#include <complex>
#include <volarray>
#include <numeric>
#include <cmath>
#include <cstdlib>
```