Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

Ю.А. Луцик, А.М. Ковальчук, И.В. Лукьянова

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

УЧЕБНОЕ ПОСОБИЕ
по курсу «Объектно-ориентированное программирование»
для студентов специальности
«Вычислительные машины, системы и сети»
всех форм обучения

Библиотека БГУИР

Минск 2003

УДК 681.322 (075.8)
ББК 32.97 я 73
Л 86

Р е ц е н з е н т :
заведующий кафедрой математики и информатики ЕГУ, канд. техн. наук
В.И. Романов

Библиотека БГУИР

Луцик Ю.А.

Л 86 Объектно-ориентированное программирование на языке С++: Учеб. пособие по курсу «Объектно-ориентированное программирование» для студ. спец. «Вычислительные машины, системы и сети» всех форм обуч. / Ю.А. Луцик, А.М. Ковальчук, И.В. Лукьянова. – Мн.: БГУИР, 2003. – 203 с.:ил.

ISBN 985-444-562-3.

В учебном пособии рассмотрены приемы и правила объектно-ориентированного программирования с использованием языка С++. Изложены основные конструкции языка С++, а также общие принципы разработки объектно-ориентированных программ.

Пособие может быть использовано студентами всех специальностей, магистрантами и аспирантами.

УДК 681.322 (075.8)
ББК 32.97 я 73

ISBN 985-444-562-3

© Луцик Ю.А., Ковальчук А.М., Лукьянова И.В., 2003
© БГУИР, 2003

Содержание

Введение 4
Объектно-ориентированное программирование 4
  Абстрактные типы данных 4
  Базовые принципы объектно-ориентированного программирования 5
  Основные достоинства языка С++ 7
  Особенности языка С++ 7
    Ключевые слова 7
    Константы и переменные 7
    Операции 8
    Типы данных 8
    Передача аргументов функции по умолчанию 8
  Простейший ввод и вывод 9
    Объект cout 9
    Манипуляторы hex и oct 9
    Объект cin 12
  Операторы для динамического выделения и освобождения памяти (new и delete) 12
Базовые конструкции объектно-ориентированных 15
программ 15
  Объекты 15
  Понятие класса 16
  Конструктор explicit 28
  Встроенные функции (спецификатор inline) 30
  Организация внешнего доступа к локальным компонентам класса (спецификатор friend) 31
  Вложенные классы 35
  Static-члены (данные) класса 38
  Указатель this 40
  Компоненты-функции static и const 43
  Ссылки 45
    Параметры ссылки 47
    Независимые ссылки 48
  Наследование (производные классы) 48
    Конструкторы и деструкторы 53
  Виртуальные функции 57
  Абстрактные классы 63
  Множественное наследование 69
    Виртуальное наследование 72
  Перегрузка функций 75
  Перегрузка операторов 76
    Перегрузка бинарного оператора 77
    Перегрузка унарного оператора 80

Библиотека БГУИР

Дружественная функция operator ... 82
Особенности перегрузки операции присваивания ... 83
Перегрузка оператора [] ... 85
Перегрузка оператора () ... 87
Перегрузка оператора -> ... 89
Перегрузка операторов new и delete ... 90
Преобразование типа ... 94
Явные преобразования типов ... 94
Преобразования типов, определенных в программе ... 95
Шаблоны ... 96
Параметризированные классы ... 97
Передача в шаблон класса дополнительных параметров ... 99
Шаблоны функций ... 100
Совместное использование шаблонов и наследования ... 101
Некоторые примеры использования шаблона класса ... 103
Задание свойств класса ... 105
Пространства имен ... 108
Ключевое слово using как директива ... 109
Ключевое слово using как объявление ... 110
Псевдоним пространства имен ... 110
Организация ввода-вывода ... 111
Состояние потока ... 114
Строковые потоки ... 116
Организация работы с файлами ... 117
Организация файла последовательного доступа ... 120
Создание файла произвольного доступа ... 123
Основные функции классов ios, istream, ostream ... 126
Исключения в C++ ... 128
Основы обработки исключительных ситуаций ... 129
Перенаправление исключительных ситуаций ... 136
Исключительная ситуация, генерируемая оператором new ... 137
Генерация исключений в конструкторах ... 139
Задание собственной функции завершения ... 140
Спецификации исключительных ситуаций ... 141
Задание собственного неожиданного обработчика ... 141
Иерархия исключений стандартной библиотеки ... 143
Стандартная библиотека шаблонов (STL) ... 143
Общее понятие о контейнере ... 143
Общее понятие об итераторе ... 147
Категории итераторов ... 149
Основные итераторы ... 150
Вспомогательные итераторы ... 156
Операции с итераторами ... 159
Контейнерные классы ... 159

Библиотека БГУИР

Контейнеры последовательностей 159
Контейнер последовательностей vector 160
Контейнер последовательностей list........ 162
Контейнер последовательностей deque 165
Ассоциативные контейнеры 167
Ассоциативный контейнер multiset 167
Ассоциативный контейнер set........ 168
Ассоциативный контейнер multimap 169
Ассоциативный контейнер map 170
Адаптеры контейнеров........ 171
Адаптеры stack 171
Адаптеры queue........ 172
Адаптеры priority_queue........ 173
Пассивные и активные итераторы 173
Алгоритмы........ 175
Алгоритмы сортировки sort, partial_sort, sort_heap 176
Алгоритмы поиска find, find_if, find_end, binary_search........ 177
Алгоритмы fill, fill_n, generate и generate_n 178
Алгоритмы equal, mismatch и lexicographical_compare........ 179
Математические алгоритмы 180
Алгоритмы работы с множествами 182
Алгоритмы swap, iter_swap и swap_ranges........ 183
Алгоритмы copy, copy_backward, merge, unique и reverse........ 183
Примеры реализации контейнерных классов 184
Связанные списки 184
Реализация односвязного списка 184
Реализация двусвязного списка 184
Реализация двоичного дерева........ 190
Литература........ 198

Библиотека БГУИР

Введение

Одна из важнейших задач программирования – разработка алгоритма. Имеется два основных подхода к разработке программ. Первый из них называется процедурным программированием. Для создания программ на его основе необходимо следующее:

определить задачу, которую нужно решить;
продумать интерфейс программы с пользователем;
разбить программу на логически законченные этапы;
создать текст программы;
отладить программу;
тестировать программу.

Второй подход называется объектно-ориентированным программированием (ООП). Для разработки программ с использованием методики объектно-ориентированного программирования необходимо:

определить задачу;
определить уникальные объекты в области решаемой задачи;
определить взаимосвязь между объектами;
создать классы объектов, определяя переменные, представляющие всевозможные состояния, в которых может находиться объект;
определить сообщения, принимаемые каждым объектом, и коды функций, согласно которым объект будет реагировать на эти сообщения. Оформить их как функции-члены некоторых классов;
объявить объекты данных классов;
определить начальное состояние системы;
скомпилировать, скомпоновать систему.

Настоящее учебное пособие ориентировано на изучение особенностей языка C++, поддерживающих объектно-ориентированный подход в программировании. Для успешного освоения излагаемого материала необходимо знание основных конструкций языка C.

Материал пособия основывается на ряде изданий [1–4].

Объектно-ориентированное программирование

Абстрактные типы данных

Концепция **абстрактных типов** и **абстрактных типов данных** является ключевой в программировании. **Абстракция** подразумевает разделение и независимое рассмотрение **интерфейса** и **реализации**.

Интерфейс и **внутренняя реализация** являются определяющими свойствами объектов окружающего нас мира. Интерфейс – это средство взаимодействия с некоторым объектом. Реализация – это внутреннее свойство объекта. Наибольший интерес представляет эффективность реализации.

Модульность и **абстракция** дополняют друг друга. Модульность предпо-

Библиотека БГУИР

лагает скрытие деталей реализации, а абстракция позволяет специфицировать каждый модуль перед тем, как будет написана соответствующая программа.

Предположим, мы покупаем некоторый достаточно сложный бытовой прибор, имеющий развитый интерфейс с пользователем. При эксплуатации прибора мы редко задумываемся о физических процессах, происходящих в данном объекте, то есть его реализации. Чем совершеннее интерфейс объекта, тем он удобнее в эксплуатации. При приобретении объекта нас интересует его интерфейс, но не его реализация. Иначе мы возвращаемся к свойствам объекта: интерфейсу и реализации. Основная цель абстракции в программировании заключается в отделении интерфейса от реализации. Попытка усовершенствовать объект (его реализацию) пользователю, не являющемуся специалистом в этой области, приводит к отрицательному результату. В программировании запрет таких действий поддерживается механизмом **запрета доступа** или **скрытия** внутренних компонент. Принцип абстракции обязывает использовать механизмы скрытия, которые предотвращают умышленное или случайное изменение внутренней реализации.

Различают процедурную абстракцию и абстракцию данных.

Процедурная абстракция требует раздельного рассмотрения цели процедуры (например функции С/С++) и ее внутренней реализации.

Абстракция данных требует раздельного рассмотрения операций над данными и реализации этих операций. Достаточно знать, какие операции выполняет модуль, но не требуется знать, какие данные он при этом использует (они скрыты) и как в действительности выполняются эти операции.

Таким образом, абстракция позволяет отделить внешнее представление модуля от его внутренней структуры. Пользователя не интересует внутренняя структура модуля, а лишь то, что этот модуль может «делать». С точки зрения же разработчика качество модуля определяется его дешевизной и эффективностью.

Абстракция данных предполагает определение и рассмотрение абстрактных типов данных (АТД), или, иначе, новых типов данных, введенных пользователем. АТД – это совокупность данных вместе с множеством операций, которые можно выполнять над этими данными.

Базовые принципы объектно-ориентированного программирования

Объектно-ориентированное программирование основывается на трех основных концепциях: **инкапсуляции**, **полиморфизме** и **наследовании**.

Инкапсуляция (пакетирование) представляет собой механизм, связывающий вместе данные и код, обрабатывающий эти данные, и сохраняющий их от внешнего воздействия и ошибочного использования. Инкапсуляция позволяет создавать объект, являющийся логическим целым, включающим данные и код для работы с этими данными. Объект обеспечивает защиту против случайной или несанкционированной модификации частных (private) составляющих его членов.

Полиморфизм – принцип (подход), обеспечивающий возможность ис-

Библиотека БГУИР

пользования одного и того же кода для решения разных задач. Полиморфизм позволяет уменьшить сложность программы посредством использования одного и того же интерфейса для задания целого класса действий. Задача выбора специфического действия (метода) в зависимости от конкретной ситуации (количества и типа передаваемых аргументов) возлагается на компилятор.

Наследование представляет собой процесс, благодаря которому один объект может наследовать (приобретать) свойства от другого объекта. Объект, используя наследование, нуждается только в определении специфичных только для этого объекта свойств, отличающих его от других объектов этого класса.

Различают **полиморфные** и **мономорфные** языки. Для мономорфных языков характерно то, что используемые функции, процедуры и операторы имеют уникальный тип. Полиморфные языки поддерживают концепцию полиморфизма в теории типов, когда одно и то же имя может быть использовано для выражения различных действий. Поддержка полиморфизма осуществляется через виртуальные функции, механизм перегрузки функций и операторов.

Передача сообщений выражает основную методологию построения объектно-ориентированных программ. Программы представляются в виде набора объектов и передачи сообщений между ними.

При построении объектно-ориентированной программы одним из основных является вопрос **иерархии классов**. Пусть имеется некоторая иерархия (структура, взаимосвязь) классов. В этом случае можно:

определить объект для заданного класса;

построить новый класс, наследуя его из существующего класса;

изменить поведение нового класса (изменить существующие и добавить новые функции).

Построение нового класса, наследуя его из существующего, предполагает:

добавление в новый класс новых компонент-данных;

добавление в новый класс новых компонент-функций;

замену в новом классе наследуемых из старого класса компонент-функций;

Наследование может быть одиночным и множественным (рис. 1). При множественном наследовании наследуемый (новый) класс имеет более одного старого класса, из которых образуется новый класс. При этом новый класс наследует поведение этих классов.

Таким образом, объектно-ориентированное программирование – метод построения программ в виде множества взаимодействующих объектов, структура и поведение которых описаны соответствующими классами. Все эти классы образуют иерархию классов, выражающую отношение наследования.

При разработке объектно-ориентированных программ часто используются библиотеки классов. Библиотека может рассматриваться как заданная базовая иерархическая структура. Для разрабатываемой программы из библиотеки может быть выбрана некоторая подструктура и затем расширена новыми клас-

Библиотека БГУИР

сами с использованием принципов наследования.

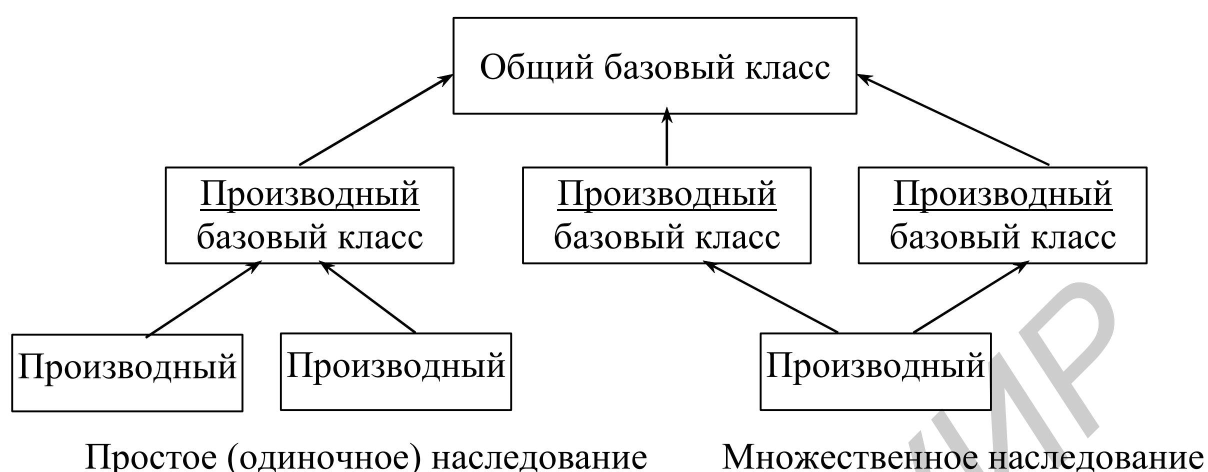

Рис. 1. Виды иерархии классов

Язык программирования называется объектно-ориентированным, если:

он поддерживает абстрактные типы данных (объекты с определенным интерфейсом и скрытым внутренним состоянием);

объекты имеют связанные с ними типы (классы);

поддерживается механизм наследования.

Основные достоинства языка С++

Язык С++ основывается на языке С, сохраняя большую часть возможностей языка С и расширяя их новыми, ориентированными на реализацию идей ООП. Язык С++ является легко переносимым языком. Получаемый программный код обладает высоким быстродействием и компактными размерами.

Особенности языка С++

Отметим некоторые дополнительные возможности языка С++. Далее в процессе рассмотрения материала мы более подробно остановимся на этих и других, не отмеченных здесь особенностях языка С++.

Необходимо четко представлять, что достоинство языка С++ состоит не в добавлении в С новых типов, операций и т.д., а в возможности поддержки объектно-ориентированного подхода к разработке программ.

Ключевые слова

Язык С++ расширяет множество ключевых слов принятых в языке С следующими ключевыми словами:

class	protected	friend	inline
private	new	operator	virtual
public	delete	this	template

Константы и переменные

В С++ односимвольные константы имеют тип char, в то же время в С++

Библиотека БГУИР

поддерживается возможность работы с двухсимвольными константами типа int:
’aB’ , ’\n\t’
При этом первый символ располагается в младшем байте, а второй – в старшем.

Операции

В языке C++ введены следующие новые операции:

:: – операция разрешения контекста;

.* и ->* – операции обращения через указатель к компоненте класса;

new и delete – операции динамического выделения и освобождения памяти.

Использование этих и других операций при разработке программ будет показано далее, при изучении соответствующего материала.

Типы данных

В C++ поддерживаются все типы данных, предопределенные в С. Кроме того, введены несколько новых типов данных: классы и ссылки.

Ссылки расширяют и упрощают передачу аргументов в функцию (по значению или по адресу).

Передача аргументов функции по умолчанию

В C++ поддерживается возможность задания некоторого числа аргументов по умолчанию. Это означает, что в заголовке функции некоторым параметрам при их описании присваиваются значения. При вызове данной функции число фактических параметров может быть меньше числа формальных параметров. В этом случае принимается умалчиваемое значение соответствующего параметра. Например:

```
#include "iostream.h"
int sm(int i1, int i2, int i3=0, int i4=0)
{ cout<<i1<<’ ’<<i2<<’ ’<<i3<<’ ’<<i4<<’ ’;
   return i1+i2+i3+i4; }
void main()
{ cout <<"сумма = "<<  sm(1,2) << endl;
  cout <<"сумма = "<<  sm(1,2,3) << endl;
  cout << "сумма = "<< sm(1,2,3,4) << endl;
}
```

Результатом работы программы будет:

```
1  2  0  0  сумма =   3
1  2  3  0  сумма =   6
1  2  3  4  сумма = 10
```

Описание параметров по умолчанию должно находиться в конце списка формальных параметров (в заголовке функции). Задание параметров по умолчанию может быть выполнено только в прототипе функции или при его отсутствии в заголовке функции.

Библиотека БГУИР

Простейший ввод и вывод

В C++ ввод-вывод данных производится потоками байт. Поток (последовательность байт) – это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода-вывода. При операциях ввода байты направляются от устройства в основную память. В операциях вывода – наоборот.

Имеется четыре потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле iostream.h:

cout – поток стандартного вывода;
cin – поток стандартного ввода;
cerr – поток стандартной ошибки;
clog – буферизируемый поток стандартных ошибок.

Объект cout

Объект cout позволяет выводить информацию на стандартное устройство вывода – экран. Формат записи cout имеет следующий вид:

cout << data [<< data];

data – это переменные, константы, выражения или комбинации всех трех типов.

Простейший пример применения cout – это вывод, например, символьной строки:

```
cout <<  ”объектно-ориентированное программирование”;
cout <<  ”программирование на C++”.
```

Надо помнить, что cout не выполняет автоматический переход на новую строку после вывода информации. Для перевода курсора на новую строку надо вставлять символ ’\n’ или манипулятор endl.

```
cout <<  ”объектно-ориентированное программирование \n”;
cout <<  ”программирование на C++”<<endl;
```

Для управления выводом информации используются манипуляторы.

Манипуляторы hex и oct

Манипуляторы hex и oct используются для вывода числовой информации в шестнадцатеричном или восьмеричном представлении. Применение их можно видеть на примере следующей простой программы:

```
#include "iostream.h"
void main()
{ int a=0x11, b=4,    //  целые числа шестнадцатеричное и десятичное
      c=051, d=8,     //    --//--            восьмеричное и десятичное
      i,j;
  i=a+b;
  j=c+d;
  cout << i  <<' ' <<hex << i <<' '<<oct << i <<' ' <<dec << i <<endl;
  cout <<hex << j <<' ' << j <<' '<<dec << j <<' ' << oct << j <<endl;
}
```

Библиотека БГУИР

В результате выполнения программы на экран будет выведена следующая информация:

```
21  15  25  21
31  31  49  61
```

Другие манипуляторы

Манипуляторы изменяют значение некоторых переменных в объекте cout.

Эти переменные называются флагами состояния. Когда объект посылает данные на экран, он проверяет эти флаги.

Рассмотрим манипуляторы, позволяющие выполнять форматирование выводимой на экран информации:

```
#include "iostream.h"
#include "iomanip.h"
void main()
{ int a=0x11;
  double d=12.362;
  cout << setw(4) << a << endl;
  cout << setw(10) << setfill('*') << a << endl;
  cout << setw(10 ) << setfill(' ') << setprecision(3) << d << endl;
}
```

Результат работы программы:

```
  17
********17
      12.4
```

В приведенной программе использованы манипуляторы setw(), setfill(' ') и setprecision(). Синтаксис их показывает, что это функции. На самом деле это *компоненты-функции* (рассмотрим позже), позволяющие изменять флаги состояния объекта cout. Для их использования необходим заголовочный файл iomanip.h. Функция setw(4) обеспечивает вывод значения переменной a в 4 позиции (по умолчанию выравнивание вправо). Функция setfil('символ') заполняет пустые позиции символом. Функция setprecision(n) обеспечивает вывод числа с плавающей запятой с точностью до n знаков после запятой (при необходимости производится округление дробной части). Таким образом, функции имеют следующий формат:

setw(количество_позиций_для_вывода_числа)
setfil(символ_для_заполнения_пустых_позиций)
setprecision (точность_при_выводе_дробного_числа)

Наряду с перечисленными выше манипуляторами в С++ используются также манипуляторы **setiosflags**() и **resetiosflags**() для установки определенных глобальных флагов, используемых при вводе и выводе информации. На эти флаги ссылаются как на *переменные состояния*. Функция setiosflags() устанавливает указанные в ней флаги, а resetiosflags() сбрасывает (очищает) их. В приведенной ниже таблице показаны аргументы для этих функций.

Библиотека БГУИР

Таблица 1

Значение	Результат, если значение установлено
ios::skipws	Игнорирует пустое пространство при вводе
ios::left	Вывод с выравниванием слева
ios::right	Вывод с выравниванием справа
ios::internal	Заполнять пространство после знака или основания с/с
ios::dec	Вывод в десятичном формате
ios::oct	Вывод в восьмеричном формате
ios::hex	Вывод в шестнадцатеричном формате
ios::boolalpha	Вывод булевых значений в виде TRUE и FALSE
ios::showbase	Выводить основание с/с
ios::showpoint	Выводить десятичную точку
ios::uppercase	Выводить шестнадцатеричные числа заглавными буквами
ios::showpos	Выводить + перед положительными целыми числами
ios::scientific	Использовать научную форму вывода чисел
ios::fixed	Использовать форму вывода с фиксированной запятой
ios::unitbuf	Сброс после каждой операции вывода
ios::sktdio	Сброс после каждого символа

Как видно из таблицы, флаги формата объявлены в классе **ios**.

Пример программы, в которой использованы манипуляторы:

```
#include "iostream.h"
#include "iomanip.h"
void main()
{ char s[]="БГУИР  факультет КСиС";
  cout << setw(30) << setiosflags(ios::right) << s << endl;
  cout << setw(30) << setiosflags(ios::left) << s << endl;
}
```

Результат работы программы:

```
         БГУИР  факультет КСиС
БГУИР  факультет КСиС
```

Наряду с манипуляторами setiosflags() и resetiosflags(), для того чтобы установить или сбросить некоторый флаг, могут быть использованы функции класса ios   **setf**() или **unsetf**(). Например:

```
#include <iostream.h>
#include <string.h>
int main()
{ char *s="Я изучаю  C++";
  cout.setf(ios::uppercase | ios::showbase | ios::hex);
  cout << 88 << endl;
  cout.unsetf(ios::uppercase);
  cout << 88 << endl;
```

Библиотека БГУИР

```
  cout.unsetf(ios::uppercase | ios::showbase | ios::hex);
  cout.setf(ios::dec);
  int len = 10 + strlen(s);
  cout.width(len);
  cout << s << endl;
  cout << 88 << " hello C++ " << 345.67 << endl;
  return 0;
}
```

Результат работы программы:

```
 0X58
 0x58
            Я изучаю  C++
88  hello C++  345.67
```

Объект cin

Для ввода информации с клавиатуры используется объект cin. Формат записи cin имеет следующий вид:

cin [>>имя_переменной];

Объект cin имеет некоторые недостатки. Необходимо, чтобы данные вводились в соответствии с форматом переменных, что не всегда может быть гарантировано.

Операторы для динамического выделения и освобождения памяти (new и delete)

Различают два типа памяти: статическую и динамическую. В статической памяти размещаются локальные и глобальные данные при их описании в функциях. Для временного хранения данных в памяти ЭВМ используется динамическая память, или heap. Размер этой памяти ограничен, и запрос на динамическое выделение памяти может быть выполнен далеко не всегда.

Для работы с динамической памятью в языке С использовались функции: calloc, malloc, realloc, free и другие. В С++ для выделения памяти можно также использовать встроенные операторы **new** и **delete**.

Оператор new имеет один операнд. Оператор имеет две формы записи:

[::] new [(список_аргументов)] имя_типа [(инициализирующее_значение)]
[::] new [(список_аргументов)] (имя_типа) [(инициализирующее_значение)]

В простейшем виде оператор new можно записать:

new имя_типа   или   new (имя_типа)

Оператор new возвращает указатель на объект типа имя_типа, для которого выполняется выделение памяти. Например:

```
char *str;        // str – указатель на объект типа char
str=new char;    // выделение памяти под объект типа char
```

или

Библиотека БГУИР

```
str=new (char);
```

В качестве аргументов можно использовать как стандартные типы данных, так и определенные пользователем. В этом случае именем типа будет имя структуры или класса. Если память не может быть выделена, оператор new возвращает значение NULL.

Оператор new позволяет выделять память под массивы. Он возвращает указатель на первый элемент массива в квадратных скобках. Например:

```
int  *n;                   //   n – указатель на целое
n=new int[20];             //   выделение памяти для массива
```

При выделении памяти под многомерные массивы все размерности кроме крайней левой должны быть константами. Первая размерность может быть задана переменной, значение которой к моменту использования new известно пользователю, например:

```
k=3;
int *p[]=new int[k][5];     // ошибка  cannot convert from 'int (*)[5]' to 'int *[]'
int (*p)[]=new int[k][5];   //  верно
```

При выделении памяти под объект его значение будет неопределенным. Однако объекту можно присвоить начальное значение.

```
int *a = new int (10234);
```

Этот параметр нельзя использовать для инициализации массивов. Однако на место инициализирующего значения можно поместить через запятую список значений, передаваемых конструктору при выделении памяти под массив (массив новых объектов, заданных пользователем). Память под массив объектов может быть выделена только в том случае, если у соответствующего класса имеется конструктор, заданный по умолчанию.

```
class matr
{    int a;      float b;
  public:
     matr(){};                       // конструктор по умолчанию
     matr(int i,float j): a(i),b(j) {}
    ~matr(){};
};
void main()
{ matr mt(3,.5);
  matr *p1=new matr[2];            //  верно      p1 – указатель на 2 объекта
  matr *p2=new matr[2] (2,3.4) ;  //  неверно,  невозможна инициализация
  matr *p3=new matr  (2,3.4);      // верно p3 – инициализированный объект
}
```

Следует отметить, что конструктор по умолчанию в примере требуется при использовании оператора new matr[2].

Использование знака :: перед оператором new указывает на вызов глобальной версии оператора new. Оператор new вызывает функцию operator new(). Аргумент имя_типа используется для автоматического вычисления раз-

Библиотека БГУИР

мера памяти sizeof(имя_типа), то есть инструкция типа new имя_типа приводит к вызову функции:

operator new(sizeof(имя_типа));

Далее, в подразделе «Перегрузка операторов», будет рассмотрен случай использования доопределенного оператора new для некоторого класса. Доопределение оператора new позволяет расширить возможности выделения памяти для объектов (их компонент) данного класса.

Создание объекта с помощью операции new вызывает также выполнение конструктора для этого объекта. Если в new не указан список инициализации либо он пуст (только скобки), то выполняется конструктор по умолчанию (default), который будет рассмотрен ниже. Если имеется непустой список инициализации, то выполняется тот конструктор, для которого этот список соответствует списку аргументов.

При создании массива выполняется стандартный конструктор для каждого элемента.

Отметим преимущества использования оператора new перед использованием malloc():

оператор new автоматически вычисляет размер необходимой памяти. Не требуется использование оператора sizeof(). При этом он предотвращает выделение неверного объема памяти;

оператор new автоматически возвращает указатель требуемого типа (не требуется использование оператора преобразования типа);

имеется возможность инициализации объекта;

можно выполнить перегрузку оператора new (delete) глобально или по отношению к тому классу, в котором он используется.

Для разрушения объекта, созданного с помощью оператора new, необходимо использовать в программе оператор delete.

Оператор delete имеет две формы записи:

[::] delete переменная_указатель // для указателя на один элемент
[::] delete [] переменная_указатель // для указателя на массив

Единственный операнд в операторе delete должен быть указатель, возвращенный оператором new. Если оператор delete применить к указателю, полученному не посредством оператора new, то результат будет непредсказуем.

Использование оператора delete вместо delete[] по отношению к указателю на массив может привести к логическим ошибкам. Таким образом, освобождать память, выделенную для массива, необходимо оператором delete [], а для отдельного элемента – оператором delete.

```
#include <iostream.h>
class A
{   int i;                // компонента-данное класса A
 public:
    A(){}                 // конструктор класса A
```

Библиотека БГУИР

```
   ~A(){}                // деструктор  класса A
};
void main()
{  A *a,*b;             // описание указателей на объект класса A
   float *c,*d;         // описание указателей на элементы типа float
   a=new A;             // выделение памяти для одного объекта класса A
   b=new A[3];          // выделение памяти для массива объектов класса A
   c=new float;         // выделение памяти для одного элемента типа float
   d=new float[4];      // выделение памяти для массива элементов типа float
   delete a;            // освобождение памяти, занимаемой одним объектом
   delete [] b;         // освобождение памяти, занимаемой массивом объектов
   delete c;            // освобождение памяти одного элемента типа float
   delete [] d;         // освобождение памяти массива элементов типа float
}
```

При удалении объекта оператором delete вначале вызывается деструктор этого объекта, а потом освобождается память. При удалении массива объектов с помощью операции delete[] деструктор вызывается для каждого элемента массива.

Базовые конструкции объектно-ориентированных программ

Объекты

Базовыми блоками ООП являются **объект** и **класс**. Объект С++ – абстрактное описание некоторой сущности, например, запись о человеке. Формально объект определить достаточно сложно. Grady Booch определил объект через свойства: **состояние** и **поведение,** которые однозначно **идентифицируют объект.** Класс – это множество объектов, имеющих общую структуру и поведение.

Рассмотрим некоторые конструкции языка С.

```
int a,b,c;
```

Здесь заданы три статических объекта целого типа, имеющих общую структуру, которые только могут получать значения и о поведении которых ничего не известно. Таким образом, о типе int можно говорить как об имени класса – некоторой абстракции, используемой для описания общей структуры и поведения множества объектов.

Рассмотрим пример описания структуры в языке С:

```
struct str{  char a[10];

             double b;
          };
 str s1,s2;
```

Структура позволяет описать АТД (тип, определенный программистом), являющийся абстракцией, так как в памяти при этом место под структуру вы-

делено не будет. Таким образом, str – это **класс**.

В то же время объекты a, b, c и s1, s2 – это уже реальность, существующая в пространстве (памяти ЭВМ) и во времени. Таким образом, класс можно определить как общее описание для множества объектов.

Определим теперь понятия **состояние**, **поведение** и **идентификация** объекта. Состояние объекта объединяет все его поля данных (статическая компонента) и текущие значения каждого из этих полей (динамическая компонента). Поведение объекта определяет, как объект изменяет свои состояния и взаимодействует с другими объектами. Идентификация объекта позволяет выделить объект из числа других объектов.

Процедурный подход к программированию предполагает разработку взаимодействующих подпрограмм, реализующих некоторые алгоритмы. Объектно-ориентированный подход представляет программы в виде взаимодействующих объектов. Взаимодействие объектов осуществляется посредством сообщений. Под передачей сообщения объекту понимается вызов некоторой функции (компонента этого объекта).

Говоря об объекте, можно выделить две его характеристики: интерфейс и реализацию.

Интерфейс показывает, как объект общается с внешней средой. Он может быть ассоциирован с окном, через которое можно заглянуть внутрь объекта и получить доступ к функциям и данным объекта.

Все данные делятся на локальные и глобальные. ***Локальные*** данные недоступны (через окно). Доступ к ним и их модификация возможна только из компонент-функций этого объекта. ***Глобальные*** данные видны и модифицируемы через окно (извне). Для активизации объекта (чтобы он что-то выполнил) ему посылается сообщение. ***Сообщение*** проходит через окно и активизирует некоторую глобальную функцию. Тип сообщения определяется именем функции и значениями передаваемых аргументов. Локальные функции (за некоторым исключением) доступны только внутри класса.

Говоря о **реализации** объекта, мы подразумеваем особенности реализации функций соответствующего класса (особенности алгоритмов и кода функций).

Объект включает в себя все данные, необходимые, чтобы описать сущность, и функции или методы, которые манипулируют этими данными.

Понятие класса

Одним из основных, базовых понятий объектно-ориентированного программирования является понятие класса.

Основная идея класса как абстрактного типа заключается в разделении интерфейса и реализации.

Интерфейс показывает, как можно использовать класс. При этом совершенно не важно, каким образом соответствующие функции реализованы внутри класса.

Реализация – внутренняя особенность класса. Одно из требований к реа-

Библиотека БГУИР

лизации – критерий изоляции кода функции от воздействия извне. Это достигается использованием локальных компонент данных и функций.

Интерфейс и реализация должны быть максимально независимы друг от друга, то есть изменение кода функций класса не должно изменять интерфейс.

По синтаксису класс аналогичен структуре. Класс определяет новый тип данных, объединяющих данные и код (функции обработки данных), и используется для описания объекта. Реализация механизма сокрытия информации (данные и функциональная часть) в C++ обеспечивается механизмом, позволяющим содержать публичные (public), частные (private) и защищенные (protected) части. Защищенные (protected) компоненты класса можно пока рассматривать как синоним частных (private) компонент, но реальное их назначение связано с наследованием. По умолчанию все части класса являются частными. Все переменные, объявленные с использованием ключевого слова public, являются доступными для всех функций в программе. Частные (private) данные доступны только в функциях, описанных в данном классе. Этим обеспечивается принцип инкапсуляции (пакетирования). В общем случае доступ к объекту из остальной части программы осуществляется с помощью функций со спецификатором доступа public. В ряде случаев лучше ограничить использование переменных, объявив их как частные, и контролировать доступ к ним через функции, имеющие спецификатор доступа public.

Сокрытие данных – важная компонента ООП, позволяющая создавать легче отлаживаемый и сопровождаемый код, так как ошибки и модификации локализованы в нем. Для реализации этого компоненты-данные желательно помещать в private-секцию.

```
#include <iostream.h>
class kls
{  int sm;            // по умолчанию предполагается   private
   int m[5];
public:
   void inpt(int i)  // функция ввода данных в компоненту  m  класса
   { cin >> m[i]; }
   int summ();       // прототип функции summ
};
int kls::summ()     // описание функции summ
{  sm=0;             // инициализация компоненты  sm  класса
  for(int i=0; i<5; i++) sm+=m[i];
  return sm;
}

void main()
{ kls k1,k2;           //  объявление объектов  k1 и k2 класса kls
   int i;
  cout<< ”Вводите элементы массива ПЕРВОГО объекта  : ”;
  for(i=0;i<5; k1.inpt(i++));     // ввод данных в первый объект
```

Библиотека БГУИР

```
  cout<< "Вводите элементы массива ВТОРОГО объекта  : ";
  for(i=0;i<5; k2.inpt(i++));     //                    во второй объект
  cout << "\n Сумма элементов первого объекта (k1) = " << k1.summ();
  cout << "\n Сумма элементов второго объекта (k2) = " << k2.summ();
}
```

Результат работы программы:
Вводите элементы массива ПЕРВОГО объекта : 2 4 1 3 5
Вводите элементы массива ВТОРОГО объекта : 2 4 6 4 9
Сумма элементов первого объекта (k1) = 15
Сумма элементов второго объекта (k2) = 25

В приведенном примере описан класс, для которого задан пустой список объектов. В main() функции объявлены два объекта описанного класса. При описании класса в него включаются прототипы функций для обработки данных класса. Текст самой функции может быть записан как внутри описания класса (функция inpt()), так и вне его (функция summ()).

Знак :: называется областью действия оператора. Он используется для информирования компилятора о том, что описываемая функция (в примере это summ) принадлежит классу, имя которого расположено слева от знака ::.

Если при описании класса некоторые функции объявлены со спецификатором public, а часть со спецификатором private, то доступ к последним из функции main() возможен только через функции этого же класса. Например:

```
#include <iostream.h>
class kls
{    int max,a,b,c;
public:
     void init(void);
     void out();
private:
     int f_max(int,int,int);
};
void kls::init(void)          // инициализация компонент  a, b, c  класса
{ cout << "Введите 3 числа  ";
   cin >> a >> b >> c;
   max=f_max(a,b,c);     // обращение к функции, описанной как private
}
void kls::out(void)
{ cout <<"\n MAX из 3  чисел  = " << max <<' endl;
}
int kls::f_max(int i,int j, int k) );    // функция нахождения наибольшего из
{ int kk;                                 //  трех чисел
   if(i>j) if(i>k) return i;
```

Библиотека БГУИР

```
            else return k;
   else if(j>k) return j;
            else return k;
}
void main()
{ kls k;         // объявление объекта класса kls
   k.init();     // обращение к public функциям ( init и out)
   k.out();
}
```

Отметим ошибочность использования инструкции

```
k.max=f_max(k.a,k.b,k.c);
```

в main функции, так как данные max, a, b, c и функция f_max класса kls являются частными и недоступны через префикс k из функции main(), не принадлежащей классу kls.

В примере для работы с объектом k класса kls выполнялась инициализация полей a, b и c путем присвоения им некоторых начальных значений. Для этого использована функция init. В языке C++ имеется возможность одновременно с описанием (созданием) объекта выполнять и его инициализацию. Эти действия выполняются специальной функцией, принадлежащей этому классу. Эта функция носит специальное название: ***конструктор***. Название этой функции всегда должно совпадать с именем класса, которому она принадлежит. При использовании конструктора функцию init в описании класса можно заменить на

```
kls(int A, int B, int C) {a=A; b=B; c=C;}     // функция конструктор
```

Конструктор представляет собой обычную функцию, имя которой совпадает с именем класса, в котором он объявлен и используется. Он никогда не должен возвращать никаких значений. Количество и имена фактических параметров в описании функции конструктора зависят от числа полей, которые будут инициализированы при объявлении объекта (экземпляра) данного класса. Кроме отмеченной формы записи конструктора в программах на C++ можно встретить и форму записи конструктора в следующем виде:

```
kls(int A, int B, int C) :  a(A), b(B), c(C) { }
```

В этом случае после двоеточия перечисляются инициализируемые данные и в скобках – инициализирующие их значения (точнее, через запятую перечисляются конструкторы объектов соответствующих типов). Возможна комбинация отмеченных форм.

Наряду с перечисленными выше формами записи конструктора существует конструктор, либо не имеющий параметров, либо все аргументы которого заданы по умолчанию – *конструктор по умолчанию*:

```
kls(){ }     это, для примера выше,  аналогично      kls() :  a(), b(), c() { }
kls(int=0, int=0, int=0){ }     это аналогично      kls() :  a(0), b(0), c(0) { }
```

Каждый класс может иметь только один конструктор по умолчанию. Бо-

Библиотека БГУИР

лее того, если при объявлении класса в нем отсутствует явно описанный конструктор, то компилятором автоматически генерируется конструктор по умолчанию. Конструктор по умолчанию используется при создании объекта без инициализации его, а также незаменим при создании массива объектов. Если при этом конструкторы с параметрами в классе есть, а конструктора по умолчанию нет, то компилятор зафиксирует синтаксическую ошибку.

Существует еще один особый вид конструкторов – *конструктор копирования*, но о нем разговор будет идти несколько позже.

Противоположным по отношению к конструктору является ***деструктор*** – функция, приводящая к разрушению объекта соответствующего класса и возвращающая системе область памяти, выделенную конструктором. Деструктор имеет имя, аналогичное имени конструктора, но перед ним ставится знак ~:

```
~kls(void){}    или    ~kls(){}              // функция-деструктор
```

Рассмотрим использование конструктора и деструктора на примере программы подсчета числа встреч некоторой буквы в символьной строке.

```
#include <iostream.h>
#include <string.h>
#define n 10
class stroka
{   int m;
    char st[20];
  public:
     stroka(char *st);              //  конструктор
    ~stroka(void);                  //  деструктор
    void out(char);
    int poisk(char);
};
stroka::stroka(char *s)
{ cout << "\n работает конструктор";
   strcpy(st,s);
}
stroka::~stroka(void)
{cout << "\n работает деструктор";}


void stroka::out(char c)
{ cout << "\n символ " << c << " найден в строке "<< st<<m<<" раз";
}
int stroka::poisk(char c)
{ m=0;
  for(int i=0;st[i]!='\0';i++)
```

Библиотека БГУИР

```
  if (st[i]==c) m++;
  return m;
}
void main()
{ char c;                      // символ для поиска его в строке
  cout << "введите символ для поиска его в строке ";
  cin >> c;
  stroka str("abcadbsaf");    // объявление объекта str  и вызов конструктора
  if (str.poisk(c))            //  подсчет числа вхождений буквы c в строку
    str.out(c);               //  вывод результата
  else cout << "\n буква"<<c<<" не найдена"<<endl;
}
```

В функции main(), при объявлении объекта str класса stroka, происходит вызов функции конструктора, осуществляющего инициализацию поля st этого объекта символьной строкой "abcadbsaf":

```
stroka str("abcadbsaf");
```

Все функции-компоненты класса stroka объявлены со спецификатором public и, следовательно, являются глобальными и могут быть вызваны из функции main() . Вызов функции осуществляется с использованием префикса str.

```
str.poisk(c);
str.out(c);
```

Поля данных класса stroka объявлены со спецификатором private по умолчанию и, следовательно, являются локальными по отношению к классу stroka. Обращение внутри любой функции-компоненты класса к полям этого же класса производится без использования префикса. За пределами видимости класса эти поля недоступны. Таким образом, обращение к ним из функции main(), например, str.st[1] или str.m, являются ошибочными.

Необходимо также отметить, что количество конструкторов класса может быть более одного. Это возможно, если в шаблоне класса имеется несколько полей данных и при определении нескольких объектов этого класса необходимо инициализировать некоторые (определенные для каждого объекта) поля (группы полей). Рассмотренный выше пример программы может быть изменен следующим образом:

```
#include <iostream.h>
#include <string.h>
#define n 10
class stroka
{ public:
      int m;
      char st[20];
       stroka(){}                  // конструктор по умолчанию
      stroka(char *);             // конструктор 1
      stroka(int M) : m(M)        // конструктор 2
```

Библиотека БГУИР

```
      { cout << "работает конструктор 2"<<endl; }
      ~stroka(void);                    //  деструктор
      void out(char);
       int poisk(int);
};
void stroka::stroka(char *s)
{ cout << "работает конструктор 1"<<endl;
  strcpy(st,s);
}
void stroka::~stroka(void)
{cout << "работает деструктор"<<endl;}
void stroka::out(char c)
{ cout << "символ " << c << " встречен в строке "<<st<< ' '<<m<<" раз\n";
}
int stroka::poisk(int k)
{ m=0;
 for(int i=0;st[i]!='\0';i++)
 if (st[i]==st[k]) m++;
 return m;
}
void main()
{ int i;
 cout << "введите номер (0-8) символа для поиска в строке"<<endl;
 cin >> i;
 stroka str1("abcadbsaf");      // описание и инициализация объекта str1
 stroka str2(i);               // описание и инициализация объекта str2
 if (str1.poisk(str2.m))           // вызов функции поиска символа в строке
    str1.out(str1.st[str2.m]);     // вызов функции вывода результата
 else cout << "символ не встречен в строке "<<str1.st<<" ни разу"<<endl;
}
```

Как и любая функция, конструктор может иметь как параметры по умолчанию, так и явно описанные.

```
#include <iostream.h>
class kls
{  int n1,n2;
 public:
  kls(int,int=2);
};
kls::kls(int i1,int i2) : n1(i1),n2(i2)
{ cout<<n1<<' '<<n2<<endl;}
```

Библиотека БГУИР

```
void main()
{ kls k(1);
  .   .   .
}
```

Следует отметить, что компоненты-данные класса желательно описывать в private секции, что соответствует принципу инкапсуляции и запрещает несанкционированный доступ к данным класса (объекта).

Отметим основные свойства и правила использования конструкторов:

– конструктор – функция, имя которой совпадает с именем класса, в котором он объявлен;

– конструктор предназначен для создания объекта (массива объектов) и инициализации его компонент-данных;

– конструктор вызывается, если в описании используется связанный с ним тип:

```
class cls{ . . . };
main()
{ cls aa(2,.3);    // вызывает cls :: cls(int,double)
  extern cls bb;  // объявление, но не описание, конструктор не вызывается
}
```

– конструктор по умолчанию не требует никаких параметров;

– если класс имеет члены, тип которых требует конструкторов, то он может иметь их определенными после списка параметров для собственного конструктора. После двоеточия конструктор имеет список обращений к конструкторам типов, перечисленным через запятую;

– если конструктор объявлен в private-секции, то он не может быть явно вызван (из main функции) для создания объекта класса.

Далее выделим основные правила использования деструкторов:

– имя деструктора совпадает с именем класса, в котором он объявлен с префиксом ~;

– деструктор не возвращает значения (даже типа void);

– деструктор не наследуется в производных классах;

– деструктор не имеет параметров (аргументов);

– в классе может быть только один деструктор;

– деструктор может быть виртуальным (виртуальной функцией);

– невозможно получить указатель на деструктор (его адрес);

– если деструктор отсутствует в описании класса, то он автоматически генерируется компилятором (с атрибутом public);

– библиотечная функция exit вызывает деструкторы только глобальных объектов;

– библиотечная функция abort не вызывает никакие деструкторы.

В С++ для описания объектов можно использовать не только ключевое слово class , но также применять ключевые слова struct и union. Различие между АТД class и struct, union состоит в том, что все компоненты struct и union по умолчанию имеют атрибут доступа public. Ниже приведен пример использова-

Библиотека БГУИР

ния union и struct.

```
#include<iostream.h>
#include<string.h>
union my_union              // объявления объединения
{  char str[14];              // компонента – символьный массив
   struct my_struct           // компонета-структура
  {  char str1[5];            // первое поле структуры
     char str2[8];            // второе поле структуры
  } my_s;                     // объект типа my_struct
   my_union(char *s);     // прототип конструктора
   void print(void);
};
my_union::my_union(char* s)  // описание функции конструктора
{  strcpy (str,s);}
void my_union::print(void)
{  cout<<my_s.str2<<"\n";      // вывод второго поля объекта my_s
   my_s.str2[0]=0;                // вставка ноль-символа между полями
   cout<<my_s.str1<<"\n";      // вывод первого поля (до ноль-символа)
}
void main(void)
{   my_union obj ("MinskBelarus");
   obj.print();
}
```

Результат работы программы:

Belarus

Minsk

Деструктор генерируется автоматически компилятором. Для размещения данных объекта obj выделяется область памяти размером максимального поля union (14 байт) и инициализируется символьной строкой. В функции print() информация, занесенная в эту область, выводится на экран в виде двух слов, при этом используется вторая компонента объединения – структура (точнее два ее поля).

Перечислим основные свойства и правила использования структур и объединений:

– в C++ struct и union можно использовать так же, как класс;

– все компоненты struct и union имеют атрибут public;

– можно описать структуру без имени struct{…} s1,s2,…;

– объединение без имени и без списка описываемых объектов называется анонимным объединением;

– доступ к компонентам анонимного объединения осуществляется по их имени;

– глобальные анонимные объединения должны быть объявлены статическими;

Библиотека БГУИР

скими;

– анонимные объединения не могут иметь компонент-функций;

– компоненты объединения нельзя специфицировать как private, public или protected, они всегда имеют атрибут public;

– union можно инициализировать, но всегда значение будет присвоено первой объявленной компоненте.

Ниже приведен текст программы с разработанным классом. В программе выполняются операции помещения символьных строк на вершину стека и чтение их с вершины стека. Более подробно механизм работы со списками будет рассмотрен в разделах «Шаблоны» и «Контейнерные классы».

```
#include <iostream.h>
#include <string.h>
class stack              //  класс СТЕК
 {   char *inf;            // компонента-данное (симв. строка)
     stack *nx;            // компонента-данное (указатель на элемент стека)
   public:
     stack(){};            // конструктор
    ~stack(){};            // деструктор
     stack *push(stack *,char *);  // занесение информации на вершину стека
     char *pop(stack **);          // чтение информации с вершины стека
 };
// помещаем информацию (строку) на вершину стека
 // возвращаем указатель на вершину стека
 stack * stack::push(stack *head,char *a)
 { stack *PTR;
   if(!(PTR=new stack))
   { cout << "\nНет памяти"; return NULL;}
   if(!(PTR->inf=new char[strlen(a)]))
   { cout << "\nНет памяти"; return NULL;}
   strcpy(PTR->inf,a); // инициализация созданной вершины
   PTR->nx=head;
   return PTR;             // PTR – новая вершина стека
 }
 // pop удаляет информацию (строку) с вершины стека и возвращает
 // удаленную строку. Изменяет указатель на вершину стека
 char * stack::pop(stack **head)
 { stack *PTR;
char *a;
   if(!(*head)) return '\0';     // если стек пуст, возвращаем \0
   PTR=*head;                    // в PTR – адрес вершины стека
   a=PTR->inf;                   // чтение информации с вершины стека
  *head=PTR->nx;        // изменяем адрес вершины стека (nex==PTR->next)
   delete PTR;           // освобождение памяти
```

Библиотека БГУИР

```
    return a;
  }
  void main(void)
  { stack *st=NULL;
char l,ll[80];
   while(1)
   { cout <<"\n выберите режим работы:\n  1- занесение в стек"
          <<"\n  2- извлечь из стека\n  0- завершить работу"<<endl;
    cin >>l;
    switch(l)
    { case '0': return; break;
      case '1': cin >> ll;
                  if(!(st=st->push(st,ll))) return;
                  break;
      case '2': cout << st->pop(&st); break;
      default: cout << "    error     " << endl;
    }
   }
  }
```

В данной реализации в main() создается указатель st на объект класса stack. Далее методами класса stack выполняется модификация указателя st. При этом создается множество взаимосвязанных объектов класса stack, образующих список (стек). Подход, более отвечающий принципам объектно-ориентированного программирования, предполагает создание одного или нескольких объектов, каждый из которых уже сам является, например, списком. Дальнейшее преобразование списка осуществляется внутри каждого конкретного объекта. Это может быть продемонстрировано на примере программы, реализующей некоторые функции бинарного дерева.

```
#include <string.h>
#include <iostream.h>
#define N 20
class tree           // класс бинарное дерево
{  struct node      // структура – узел бинарного дерева
   { char *inf;       // информационное поле
     int n;           // число встреч информационного поля в бинарном дереве
     node *l,*r;
   };
   node *dr;         // указатель на корень дерева
 public:
   tree(){ dr=NULL;}
   void see(node *);                // просмотр бинарного дерева
   int sozd();                      // создание+дополнение бинарного дерева
   node *root(){return dr;}         // функция возвращает указатель на корень
```

Библиотека БГУИР

```
};
void main(void)
{ tree t;
  int i;
  while(1)
  { cout<<"вид операции: 1 – создать дерево"<<endl;
    cout<<"             2 – рекурсивный вывод содержимого дерева"<<endl;
    cout<<"             3 – нерекурсивный вывод содержимого дерева"<<endl;
    cout<<"             4 – добавление элементов в дерево"<<endl;
    cout<<"             5 – удаление любого элемента из дерева"<<endl;
    cout<<"             6 – выход"<<endl;
    cin>>i;
    switch(i)
    { case 1: t.sozd(); break;
      case 2: t.see(t.root());  break;
      case 6: return;
    }
  }
}
int tree::sozd()  // функция создания бинарного дерева
{ node *u1,*u2;
   if(!(u1=new node))
   { cout<<"Нет свободной памяти"<<endl;
     return 0;
   }
   cout<<"Введите информацию в узел дерева ";
   u1->inf=new char[N];
   cin>>u1->inf;
   u1->n=1;          // число повторов информации в дереве
   u1->l=NULL;    // ссылка на левый узел
   u1->r=NULL;    // ссылка на правый узел
   if (!dr)
   dr=u1;
   else
   { u2=dr;
     int k,ind=0;            // ind=1 – признак выхода из цикла поиска
     do
     { if(!(k=strcmp(u1->inf,u2->inf)))
       { u2->n++;          // увеличение числа встреч информации узла
         ind=1;              // для выхода из цикла do ... while
       }
       else
       {  if (k<0)           // введ. строка < строки в анализируемом узле
```

Библиотека БГУИР

```
        { if (u2->l!=NULL) u2=u2->l; // считываем новый узел дерева
          else ind=1;     // выход из цикла do ... while
        }
        else                  // введ. строка > строки в анализируемом узле
        { if (u2->r!=NULL) u2=u2->r; // считываем новый узел дерева
          else ind=1;     // выход из цикла do ... while
        }
      }
    } while(!ind);
    if (k)                  // не найден узел с аналогичной информацией
    { if (k<0) u2->l=u1;   // ссылка в dr1 налево
      else u2->r=u1;         //  ссылка в dr1 направо
    }
  }
  return 1;
}
void tree::see(node *u)  // функция рекурсивного вывода бинарного дерева
{ if(u)
  { cout<<"узел содержит: "<<u->inf<<" число встреч "<<u->n<<endl;
    if (u->l) see(u->l);     // вывод левой ветви дерева
    if (u->r) see(u->r);     // вывод правой ветви дерева
  }
}
```

При небольшой модификации в функции main() можно создать несколько объектов класса tree и, например, используя указатель на объект класса tree, вызывать методы класса для работы с каждым из созданных объектов (бинарных деревьев). Это преобразование предлагается выполнить самостоятельно.

Конструктор explicit

В C++ компилятор для конструктора с одним аргументом может автоматически выполнять *неявные преобразования*. В результате этого тип, получаемый конструктором, преобразуется в объект класса, для которого определен данный конструктор.

```
#include "iostream.h"
class array       //  класс массив целых чисел
{  int size;        //  размерность массива
   int *ms;         //  указатель на массив
 public:
   array(int = 1);
   ~array();
   friend void print(const array&);
};
array::array(int kl) : size(kl)
```

Библиотека БГУИР

```
{ cout<<"работает конструктор"<<endl;
  ms=new int[size];                    // выделение памяти для массива
  for(int i=0; i<size; i++) ms[i]=0; // инициализация
}
array::~array()
{ cout<<"работает деструктор"<<endl;
  delete [] ms;
}
void print(const array& obj)
{ cout<<"выводится массив размерностью"<<obj.size<<endl;
  for(int i=0; i<obj.size; i++)
  cout<<obj.ms[i];
  cout<<endl;
}
void main()
{ array obj(10);
  print(obj);      // вывод содержимого объекта obj
  print(5);        // преобразование 5 в array и вывод
}
```

В результате выполнения программы получим:

```
работает конструктор
выводится массив размерностью 10
0 0 0 0 0 0 0 0 0 0
работает конструктор
выводится массив размерностью 5
0 0 0 0 0
работает деструктор
работает деструктор
```

В данном примере в инструкции:

```
array obj(10)
```

определяется объект obj и для его создания (и инициализации) вызывается конструктор array(int). Далее в инструкции:

```
print(obj);      // вывод содержимого объекта obj
```

выводится содержимое объекта obj, используя friend-функцию print(). При выполнении инструкции:

```
print(5);        // преобразование 5 в array и вывод
```

компилятором не находится функция print(int) и выполняется проверка на наличие в классе array конструктора, способного выполнить преобразование в объект класса array. Так как в классе имеется конструктор array(int), то такое преобразование возможно (создается временный объект, содержащий массив из пяти чисел).

В некоторых случаях такие преобразования являются нежелательными или, возможно, приводящими к ошибке. В C++ имеется ключевое слово explicit

Библиотека БГУИР

для подавления неявных преобразований. Конструктор, объявленный как explicit:

explicit array(int = 1);

не может быть использован для неявного преобразования. В этом случае компилятором (в частности Microosft C++) будет выдано сообщение об ошибке:

```
Compiling...
  error C2664: 'print' : cannot convert parameter 1 from 'const int' to 'const class array &'
  Reason: cannot convert from 'const int' to 'const class array'
  No constructor could take the source type, or constructor overload resolution was ambiguous
```

Если необходимо при использовании explicit-конструктора все же создать массив и передать его в функцию print, то надо использовать инструкцию

print(array(5));

Встроенные функции (спецификатор inline)

В ряде случаев в качестве компонент класса используются достаточно простые функции. Вызов функции означает переход к памяти, в которой расположен выполняемый код функции. Команда перехода занимает память и требует времени на ее выполнение, что иногда существенно превышает затраты памяти на хранение кода самой функции. Для таких функций целесообразно поместить код функции вместо выполнения перехода к функции. В этом случае при выполнении функции (обращении к ней) выполняется сразу ее код. Функции с такими свойствами называются встроенными. Если описание компоненты функции включается в класс, то такая функция называется встроенной. Например:

```
class stroka
{   char str[100];
  public:
          ……..
     int size()
    { for(int i=0; *(str+i); i++);
       return i;
    }
};
```

В описанном примере функция size() – встроенная. Если функция, объявленная в классе, а описанная за его пределами, должна быть встроенной, то она описывается со спецификатором inline:

```
class  stroka
{   char str[100];
  public:
     ……..
     int size();
};
inline int stroka ::size()
```

Библиотека БГУИР

```
{ for(int i=0; str[i]; i++);
  return i;
}
```

Спецификация inline может быть проигнорирована компилятором, поскольку иногда введение встроенных функций оказывается невозможным или нежелательным.

Организация внешнего доступа к локальным компонентам класса (спецификатор friend)

В С++ одна функция не может быть компонентой двух различных классов. В то же время иногда возникает необходимость организации доступа к локальным данным нескольких классов из одной функции. Для реализации этого в С++ введен спецификатор friend. Если некоторая функция определена как friend функция для некоторого класса, то она:

– не является компонентой-функцией этого класса;

– имеет доступ ко всем компонентам этого класса (private, public и protected).

```
#include <iostream.h>
class kls
{    int i,j;
 public:
    kls(int i,int J) : i(I),j(J) {}         // конструктор
    int max() {return   i>j? i : j;}    // функция-компонента класса kls
    friend double fun(int, kls&);    // friend-объявление внешней функции fun
};
double fun(int i, kls &x)
{  return (double)i/x.i;
}
main()
{ kls obj(2,3);
  cout  << obj.max() << endl;

  cout << fun(3,obj) << endl;
  return 1;
}
```

Функции со спецификатором friend, не являясь компонентами класса, не имеют и, следовательно, не могут использовать this указатель (будет рассмотрен несколько позже). Следует также отметить ошибочность следующей заголовочной записи функции double kls :: fun(int i,int j), так как fun не является компонентой-функцией класса kls.

В общем случае friend-функция является глобальной независимо от секции, в которой она объявлена (public, protected, private), при условии, что она не объявлена ни в одном другом классе без спецификатора friend. Функция friend,

Библиотека БГУИР

объявленная в классе, может рассматриваться как часть интерфейса класса с внешней средой.

Вызов компоненты-функции класса осуществляется с использованием операции доступа к компоненте (.) или(->). Вызов же friend-функции производится по ее имени (так как friend-функции не являются его компонентами). Следовательно, как это будет показано далее, в fried-функции не передается this-указатель и доступ к компонентам класса выполняется либо явно (.), либо косвенно (->).

Компонента-функция одного класса может быть объявлена со спецификатором friend для другого класса:

```
class X{   ……….
          void fun (….);
        };
class Y{   ……….
          friend void X:: fun (….);
        };
```

В приведенном фрагменте функция fun() имеет доступ к локальным компонентам класса Y. Запись вида friend void X:: fun (….) говорит о том, что функция fun принадлежит классу X, а спецификатор friend разрешает доступ к локальным компонентам класса Y (так как она объявлена со спецификатором в классе Y).

Ниже приведен пример программы расчета суммы двух налогов на зарплату.

```
#include "iostream.h"
#include "string.h"
#include "iomanip.h"
class nalogi;              // неполное объявление класса nalogi
class work
{  char s[20];             // фамилия работника
   int zp;                 // зарплата
public:
   float raschet(nalogi);   // компонента-функция класса work
   void inpt()
   { cout << "вводите фамилию и зарплату" << endl;
     cin >> s >>zp;
   }
   work(){}
   ~work(){};
};
class nalogi
{  float pd,                // подоходный налог
   st;                      // налог на соцстрахование
   friend float work::raschet(nalogi); // friend-функция класса nalogi
 public:
```

Библиотека БГУИР

```
  nalogi(float f1,float f2) : pd(f1),st(f2){};
 ~nalogi(void){};
};

float work::raschet(nalogi nl)
{ cout << s << setw(6) << zp <<endl;         // доступ к данным класса work
  cout << nl.pd << setw(8) << nl.st <<endl; // доступ к данным класса nalogi
  return zp*nl.pd/100+zp*nl.st/100;
}

int main()
{ nalogi nlg((float)12,(float)2.3);    // описание и инициализация объекта
  work wr[2];                          // описание массива объектов
  for(int i=0;i<2;i++) wr[i].inpt();   // инициализация массива объктов
 cout<<setiosflags(ios::fixed)<<setprecision(3)<<endl;
 cout << wr[0].raschet(nlg) << endl; // расчет налога для объекта wr[0]
 cout << wr[1].raschet(nlg) << endl; // расчет налога для объекта wr[1]
 return 0;
}
```

Следует отметить необходимость выполнения неполного (предварительного) объявления класса nalogi, так как в прототипе функции raschet класса work используется объект класса nalogi, объявляемого далее. В то же время полное объявление класса nalogi не может быть выполнено ранее (до объявления класса work), так как в нем содержится friend-функция, описание которой должно быть выполнено до объявления friend-функции. В противном случае компилятор выдаст ошибку.

Если функция raschet в классе work также будет использоваться со спецификатором friend, то приведенная выше программа будет выглядеть следующим образом:

```
#include "iostream.h"
#include "string.h"
#include "iomanip.h"
class nalogi;              // неполное объявление класса nalogi
class work
{  char s[20];             // фамилия работника
   int zp;                 // зарплата
 public:
   friend float raschet(work,nalogi); // friend-функция класса work
   void inpt()
   { cout << "вводите фамилию и зарплату" << endl;
     cin >> s >>zp;
   }
   work(){}                // конструктор по умолчанию
   ~work(){}               // деструктор по умолчанию
```

Библиотека БГУИР

```
};
class nalogi
{   float pd,              // подоходный налог
    st;                    // налог на соцстрахование
    friend float raschet(work,nalogi); // friend-функция класса nalogi
  public:
    nalogi(float f1,float f2) : pd(f1),st(f2){};
    ~nalogi(void){};
};

float raschet(work wr,nalogi nl)
{ cout << wr.s << setw(6) << wr.zp <<endl;// доступ к данным класса work
  cout << nl.pd << setw(8) << nl.st <<endl;// доступ к данным класса nalogi
  return wr.zp*nl.pd/100+wr.zp*nl.st/100;
}
int main()
{ nalogi nlg((float)12,(float)2.3);  // описание и инициализация объекта
  work wr[2];
  for(int i=0;i<2;i++) wr[i].inpt();  // инициализация массива объктов
  cout<<setiosflags(ios::fixed)<<setprecision(3)<<endl;
  cout << raschet(wr[0],nlg) << endl; // расчет налога для объекта wr[0]
  cout << raschet(wr[1],nlg) << endl; // расчет налога для объекта wr[1]
  return 0;
}
```

Все функции одного класса можно объявить со спецификатором friend по отношению к другому классу следующим образом:

```
class X{ ………..
         friend class Y;
         ………..
       };
```

В этом случае все компоненты-функции класса Y имеют спецификатор friend для класса X (имеют доступ к локальным компонентам класса X).

```
#include <iostream.h>
class A
{  int i;              // компонента-данное класса A
 public:
   friend class B;     // объявление класса B другом класса A
   A():i(1){}          // конструктор
   ~A(){}              // деструктор
   void f1_A(B &); // метод, оперирующий данными обоих классов
};
class B
{   int j;              // компонента-данное класса B
```

Библиотека БГУИР

```
  public:
    friend A;           // объявление класса A другом класса B
    B():j(2){}          // конструктор
    ~B(){}              // деструктор
    void f1_B(A &a){ cout<<a.i+j<<endl;}  // метод, оперирующий
                                          // данными обоих классов
};
void A :: f1_A(B &b)
{ cout<<i<<' '<<b.j<<endl;}
void main()
{ A aa;
  B bb;
  aa.f1_A(bb);
  bb.f1_B(aa);
}
```

Результат выполнения программы:

1 2

3

В объявлении класса A содержится инструкция friend class B, являющаяся и предварительным объявлением класса B, и объявлением класса B дружественным классу A. Отметим также необходимость описания функции f1_A после явного объявления класса B (в противном случае не может быть создан объект b еще не объявленного типа).

Отметим основные свойства и правила использования спецификатора friend:

– friend-функции не являются компонентами класса, но имеют доступ ко всем его компонентам независимо от их атрибута доступа;

– friend-функции не имеют указателя this;

– friend-функции не наследуются в производных классах;

– отношение friend не является *ни симметричным* (то есть если класс A есть friend классу B, то это не означает, что B также является friend классу A), *ни транзитивным* (то есть если A friend B и B friend C, то не следует, что A friend C);

– друзьями класса можно определить перегруженные функции. Каждая перегруженная функция, используемая как friend для некоторого класса, должна быть явно объявлена в классе со спецификатором friend.

Вложенные классы

Один класс может быть объявлен в другом классе, в этом случае внутренний класс называется вложенным:

```
class ext_class
{  class int_cls
```

Библиотека БГУИР

```
    {   • • •
    };
      public:
       • • •
};
```

Класс int_class является вложенным по отношению к классу ext_class (внешний).

Доступ к компонентам вложенного класса, имеющим атрибут private, возможен только из функций внешнего класса и из функций внешнего класса, объявленных со спецификатором friend.

```
#include "iostream.h"
class cls1                      //  внешний класс
{  class cls2                   //  вложенный класс
   {  int b;                    // все компоненты private для cls1 и cls2
      cls2(int bb) : b(bb){}    // конструктор класса cls2
   };
   public:                      // public секция для cls1
        int a;
   class cls3                   // вложенный класс
   {  int c;             // private для cls1 и cls3
    public:              // public-секция для класса cls3
      cls3(int cc): c(cc) {}   // конструктор класса cls3
   };
   cls1(int aa): a(aa) {}      // конструктор класса  cls
};
void main()
{ cls1 aa(1980);             // верно
  cls1::cls2 bb(456);        // ошибка cls2 cannot access private
  cls1::cls3 cc(789);        // верно
  cout << aa.a << endl;   // верно
  cout << cc.c << endl;  // ошибка 'c' : cannot access private member
                          //                  declared in class 'cls1 :: cls3'
}
```

В приведенном тексте программы инструкция cls1 :: cls2 bb(456) является ошибочной, так как объявление класса cls2 и его компонента находятся в секции private. Для устранения ошибки при описании объекта bb необходимо изменить атрибуты доступа для класса cls2 следующим образом:

```
public:
  class cls2
  {    int b;           // private-компонента
    public:
       cls2(int bb) : b(bb){}
  };
```

Пример доступа к private-компонентам вложенного класса из функций

Библиотека БГУИР

внешнего класса, объявленных со спецификатором friend, приводится ниже.

```
#include "iostream.h"
class cls1                         // внешний класс
{     int a;
  public:
     cls1(int aa): a(aa) {}
     class cls3;                   // неполное объявление класса
     void fun(cls1::cls3);         // функция получает объект класса cls3
     class cls3                    // вложенный класс
     {   int c;
       public:
         cls3(int cc) : c(cc) {}
         friend void cls1::fun(cls1::cls3);  // ф-ция, дружественная классу cls1
         int pp(){return c;}
     };
};
void cls1::fun(cls1::cls3 dd) {cout << dd.c << endl << a;}
void main()
{  cls1 aa(123);
   cls1:: cls3 cc(789);
   aa.fun(cc);
}
```

Внешний класс cls1 содержит public-функцию fun(cls1 :: cls3 dd), где dd есть объект, соответствующий классу cls3, вложенному в класс cls1. В свою очередь в классе cls3 имеется friend-функция friend void cls1 :: fun(cls1 :: cls3 dd), обеспечивающая доступ функции fun класса cls1 к локальной компоненте **c** класса cls3.

```
#include "iostream.h"
#include "string.h"
class ext_cls                 // ВНЕШНИЙ класс
{   int gd;                   // год рождения
 double zp;                   // з/плата
    class int_cls1            // ПЕРВЫЙ вложенный класс
    {   char *s;              // фамилия
      public:
        int_cls1(char *ss)    // конструктор 1-го вложенного класса
        { s=new char[20];
          strcpy(s,ss);
        }
       ~int_cls1()            // деструктор 1-го вложенного класса
        { delete [] s;}
       void disp_int1() {cout << s<<endl;}
    };
```

Библиотека БГУИР

```
 public:
   class int_cls2              // ВТОРОЙ вложенный класс
   {    char *s;               // фамилия
     public:
        int_cls2(char *ss)     // конструктор 2-го вложенного класса
      { s=new char[20];
        strcpy(s,ss);
      }
      ~int_cls2()              // деструктор 2-го вложенного класса
      { delete [] s;}
       void disp_int2() {cout << s << endl;}
   };
        // public функции класса ext_cls
    ext_cls(int god,double zpl):gd(god),zp(zpl){}// конструктор внешнего
                                                 // класса
    int_cls1 *addr(int_cls1 &obj){return &obj;} // возврат адреса объекта obj
    void disp_ext()
    {  int_cls1 ob("Иванов");      // создание объекта вложенного класса
      addr(ob)->disp_int1();       // вывод на экран содержимого объекта ob
      cout <<gd<<' '<< zp<<endl;
    }
};
void main()
{  ext_cls ext(1980,230.5);
   //  ext_cls::int_cls1 in1("Петров");   // неверно, т.к. int_cls1 имеет
                                          // атрибут private
   ext_cls::int_cls2 in2("Сидоров");
   in2.disp_int2();
   ext.disp_ext();                  //
}
```

В результате выполнения программы получим:

Сидоров
Иванов
1980 230.5

В строке in2.disp_int2() (main функции) для объекта вложенного класса вызывается компонента-функция ext_cls :: int_cls2 :: disp_int2() для вывода содержимого объекта in2 на экран. В следующей строке ext.disp_ext() вызывается функция ext_cls::disp_ext(), в которой создается объект вложенного класса int_cls1. Затем, используя косвенную адресацию, вызывается функция ext_cls :: int_cls1 :: disp_int1(). Далее выводятся данные, содержащиеся в объекте внешнего класса.

***Static*-члены (данные) класса**

Библиотека БГУИР

Компоненты-данные могут быть объявлены с модификатором класса памяти **static**. Класс, содержащий static компоненты-данные, объявляется как глобальный (локальные классы не могут иметь статических членов). Static-компонента совместно используется всеми объектами этого класса и хранится в одном месте. Статическая компонента глобального класса должна быть явно определена в контексте файла. Использование статических компонент-данных класса продемонстрируем на примере программы, выполняющей поиск введенного символа в строке.

```
#include "string.h"
#include "iostream.h"
enum boolean {fls,tru};
class cls
{    char *s;
   public:
      static int k;           // объявление static-члена в объявлении класса
      static boolean ind;
      void inpt(char *,char);
      void print(char);
};
  int cls::k=0;            // явное определение static-члена в контексте файла
  boolean cls::ind;
void cls::inpt(char *ss,char c)
{ int kl;                  // длина строки
 cin >> kl;
 ss=new char[kl];      // выделение блока памяти под строку
 cout << "введите строку\n";
 cin >> ss;
 for (int i=0; *(ss+i);i++)
 if(*(ss+i)==c) k++;    // подсчет числа встреч буквы в строке
 if (k) ind=tru;          // ind==tru – признак того, что буква есть в строке
 delete [] ss;             // освобождение указателя на строку
}
void cls::print(char c)
{ cout << "\n число символов "<< c <<"строки   " << k;
}
void main()
{ cls c1,c2;
 char c;
 char *s;
 cls::ind=fls;
 cout << "введите символ для поиска в строках";
 cin >> c;
```

Библиотека БГУИР

```
  c1.inpt(s,c);
  c2.inpt(s,c);
  if(cls::ind) c1.print(c);
  else cout << "\n символ не найден";
}
```

Объявление статических компонент-данных задает их имена и тип, но не инициализирует значениями. Присваивание им некоторых значений выполняется в программе.

В функции main() использована возможная форма обращения к static-компоненте

cls::ind   (имя класса :: идентификатор),

которая обеспечивается тем, что идентификатор имеет видимость public. Это дальнейшее использование оператора разрешения контекста "::".

Отметим основные правила использования статических компонент:

– статические компоненты будут одними для всех объектов данного класса. То есть ими используется одна область памяти;

– статические компоненты не являются частью объектов класса;

– объявление статических компонент-данных в классе не является их описанием. Они должны быть явно описаны в контексте файла;

– локальный класс не может иметь статических компонент;

– к статической компоненте st класса cls можно обращаться cls::st, независимо от объектов этого класса, а также используя операторы . и -> при использовании объектов этого класса;

– статическая компонента существует даже при отсутствии объектов этого класса;

– статические компоненты можно инициализировать, как и другие глобальные объекты, только в файле, в котором они объявлены.

Указатель this

Как отмечалось выше, если некоторая функция является компонентой объекта, то при вызове этой функции к компонентам-данным этого объекта можно обращаться по имени (опуская имя объекта). Например, пусть имеется объявление двух объектов:

```
my_class ob1,ob2;
```

Вызовы компонент-функций имеют вид:

```
ob1.fun_1();
ob2.fun_2();
```

Пусть в обеих функциях содержится инструкция:

```
cout << str;
```

При объявлении двух объектов создаются две компоненты-данные str. Возникает вопрос, откуда каждая из двух функций узнает, с какой из компонент ей работать (точнее, где она расположена). Ответ состоит в следующем. В памяти для каждого располагаемого объекта создается ***скрытый указатель***, адресующий начало выделенной под объект области памяти. Получить значение

Библиотека БГУИР

этого указателя в компонентах-функциях можно посредством ключевого слова this. Для любой функции, принадлежащей классу my_class, указатель this неявно объявлен так:

my_class *const this;

Таким образом, при объявлении объектов ob1 и ob2 создаются два this-указателя на эти объекты. Следовательно, любая функция, являющаяся компонентой некоторого объекта, при вызове получает this-указатель на этот объект. И приведенная выше инструкция в функции воспринимается как

cout << this->str;

Однако эта форма записи избыточна. С другой стороны, явное использование указателя this эффективно при решении некоторых задач.

Рассмотрим пример использования this-указателя на примере упорядочивания чисел в массиве.

```
#include<stdio.h>
#include<iostream.h>
class m_cl
{    int a[3];
  public:
     m_cl  srt();            // функция упорядочивания информации в массиве
     m_cl *inpt();           // функция ввода чисел в массив
     void out();             //  вывод информации о результате  сортировки
};
 m_cl m_cl::srt()        // функция сортировки
{ for(int i=0;i<2;i++)
   for(int j=i;j<3;j++)
   if (a[i]>a[j]) {a[i]=a[i]+a[j]; a[j]=a[i]-a[j]; a[i]=a[i]-a[j];}
   return *this;              // возврат содержимого объекта, на который
}                              // указывает указатель this
 m_cl * m_cl::inpt()        // функция ввода
{ for(int i=0;i<3;i++)
   cin >> a[i];
   return this;                 // возврат скрытого указателя this
}                                // (адреса начала объекта)
void m_cl::out()
{ cout << '\n';
   for(int i=0;i<3;i++)
   cout << a[i] << ' ';
}
main()
{  m_cl o1,o2;               // описание двух объектов класса m_cl
   o1.inpt()->srt().out();    // вызов компонент-функций первого объекта
   o2.inpt()->srt().out();    // вызов компонент-функций второго объекта
   return 1;
```

Библиотека БГУИР

}

Вызов компонент-функций для каждого из созданных объектов осуществляется:

o1.inpt()->srt().out;

Приведенная инструкция интерпретируется следующим образом:

сначала вызывается функция inpt для ввода информации в массив данных объекта o1;

функция inpt возвращает адрес памяти, где расположен объект o1;

далее вызывается функция сортировки информации в массиве, возвращающая содержимое объекта o1;

после этого вызывается функция вывода информации.

Ниже приведен текст еще одной программы, использующей указатель this.

```
#include<iostream.h>
#include<string.h>
class B;                 // предварительное объявление  класса B
class A
{   char *s;
  public:
     A(char *S)
    { s=new char[50];
      strcpy(s,S);
    }
    ~A(){delete [] s;}
    void pos_str(char *);
};
class B
{   char *ss;
  public:
     B(char *SS)
    { ss=new char[20];
       strcpy(ss,SS);
    }
    ~B(){delete [] ss;}
    char *f_this(void){return this->ss;}
};
void A::pos_str(char *ss)
{ char *dd;
   int ii;
   dd=new char[strlen(s)+strlen(ss)+2];
   strcpy(dd,s);
   ii=strlen(s);
```

Библиотека БГУИР

```
  for(int i=0;*(ss+i);i++)
  *(dd+ii+i)=*(ss+i);
  *(dd+ii+i)=0;
  delete s;
  s=new char[strlen(s)+strlen(ss)];
  strcpy(s,dd);
  delete [] dd;
  cout << s << endl;
}
void main(void)
{ A a1("aa bcc dd ee");
  B b1("bd");
  a1.pos_str(b1.f_this());
}
```

Отметим основные правила использования this-указателей:

– каждому объявляемому объекту соответствует свой скрытый this- указатель;

– this-указатель может быть использован только для нестатической функции;

– this указывает на начало своего объекта в памяти;

– this не надо дополнительно объявлять;

– this передается как скрытый аргумент во все нестатические (не имеющие спецификатора static) компоненты-функции своего объекта;

– указатель this – локальная переменная и недоступна за пределами объекта;

– обращаться к скрытому указателю можно this или *this.

Компоненты-функции static и const

В С++ компоненты-функции могут использоваться с модификатором static и const. Обычная компонента-функция, вызываемая

object . function(a,b);

имеет явный список параметров a и b и неявный список параметров, состоящий из компонент данных переменной object. Неявные параметры можно представить как список параметров, доступных через указатель this. Статическая (***static***) компонента-функция не может обращаться к любой из компонент посредством указателя this. Компонента-функция ***const*** не может изменять неявные параметры.

```
#include "iostream.h"
class cls
{   int kl;                // количество изделий
    double zp;           // зарплата на производство 1 изделия
    double nl1,nl2;     // два налога на з/пл
    double sr;            // кол-во сырья на изделие
```

Библиотека БГУИР

```
   static double cs;   // цена сырья на 1 изделие
 public:
   cls(){}             // конструктор по умолчанию
   ~cls(){}            // деструктор
   void inpt(int);
   static void vvod_cn(double);
   double seb() const;
};
double cls::cs;         // явное определение static-члена в контексте файла
void cls::inpt(int k)
{ kl=k;
  cout << "Введите з/пл и 2 налога";
  cin >> nl1 >> nl2 >> zp;
}
void cls::vvod_cn(double c)
{ cs=c;      // можно обращаться в функции только к static-компонентам;
}
double cls::seb() const
{return kl*(zp+zp*nl1+zp*nl2+sr*cs);//в функции нельзя изменить ни один
}                                   // неявный параметр (kl zp nl1 nl2 sr)
 void main()
{ cls c1,c2;
   c1.inpt(100);            //  инициализация первого объекта
   c2.inpt(200);            //  инициализация второго объекта
   cls::vvod_cn(500.);      //
   cout << "\nc1" << c1.seb() << "\nc2" << c2.seb() << endl;
}
```

Ключевое слово static не должно быть включено в описание объекта статической компоненты класса. Так, в описании функции отсутствует ключевое слово static. В противном случае возможно противоречие между static- компонентами класса и внешними static-функциями и переменными.

```
void cls::vvod_cn(double c)
{ cs=c; }
```

Функции класса, объявленные со спецификатором const, могут быть вызваны для объекта со спецификатором const, а функции без спецификатора const – не могут.

```
const cls c1;
cls c2;
c1.inpt(100);        //  неверный вызов
c2.inpt(100);        //  правильный вызов функции
c1.seb();            //  правильный вызов функции
```

Для функций со спецификатором const указатель this имеет следующий

Библиотека БГУИР

тип:

const имя_класса * const this;

Следовательно, нельзя изменить значение компоненты объекта через указатель this без явной записи. Рассмотрим это на примере функции seb.

```
double cls::seb() const
{ ((cls *)this)->zp--;         //  возможная модификация неявного параметра
                               //  zp посредством явной записи this-указателя
  return kl*(zp+zp*nl1+zp*nl2+sr*cs);
}
```

Если функция, объявленная в классе, описывается отдельно (вне класса), то спецификатор const должен присутствовать как в объявлении, так и в описании этой функции.

Основные свойства и правила использования static- и const- функций:

– статические компоненты-функции не имеют указателя this, поэтому обращаться к нестатическим компонентам класса можно только с использованием . или ->;

– не могут быть объявлены две одинаковые функции с одинаковыми именами и типами аргументов, при этом одна статическая, а другая нет;

– статические компоненты-функции не могут быть виртуальными.

Ссылки

При передаче параметров в функцию они помещаются в стековую память. В отличие от стандартных типов данных (char, int, float и др.) объекты обычно требуют много больше памяти, при этом стековая память может существенно увеличиться. Для уменьшения объема передаваемой через стек информации в С(С++) используются указатели. В языке С++ наряду с использованием механизма указателей имеется возможность использовать *неявные указатели* (ссылки). Ссылка, по существу, является не чем иным, как вторым именем некоторого объекта.

Формат объявления ссылки имеет вид:

тип & имя_ссылки = инициализатор.

Ссылку нельзя объявить без ее инициализации. Ниже приведен пример использования ссылки.

```
#include "iostream.h"
#include "string"
class A
{  char s[80];
   int i;
 public :
   A(char *S,int I):i(I)  { strcpy(s,S);}
    ~A(){}
    void see()  {cout<<s<<"  "<<i<<endl;}
};
```

Библиотека БГУИР

```
void main()
{  A a("aaaaa",3),aa("bbbb",7);
   A &b=a;    //  ссылка на объект класса A инициализирована значением  a
   cout<<"компоненты объекта :"; a.see();
   cout<<"компоненты ссылки  :"; b.see();
   cout <<"адрес a="<<&a << "  адрес &b= "<< &b << endl;
   b=aa;            // присвоение значений объекта aa ссылке b (и объекту a)
   cout<<"компоненты объекта :"; a.see();
   cout<<"компоненты ссылки  :"; b.see();
   int i=4,j=2;
   int &ii=i;     // ссылка на переменную i типа int
   cout << "значение i= "<<i<<" значение ii= "<<ii<<endl;
   ii++;           // увеличение значения переменной i
   cout << "значение i= "<<i<<" значение ii= "<<ii<<endl;
   ii=j;           // инициализация переменной i значением j
   cout << "значение i= "<<i<<" значение ii= "<<ii<<endl;
}
```

В результате выполнения программы получим:

```
компоненты объекта : aaaaa  3
компоненты ссылки  : aaaaa  3
адрес a= 0x________   адрес &b= 0x________
компоненты объекта : bbbbb  7
компоненты ссылки  : bbbbb  7
значение i= 4 значение ii= 4
значение i= 5 значение ii= 5
значение i= 2 значение ii= 2
```

Из примера следует, что переменная и ссылка на нее имеют один и тот же адрес в памяти. Изменение значения по ссылке приводит к изменению значения переменной и наоборот.

Ссылка может также указывать на константу, в этом случае создается временный объект, инициализируемый значением константы.

```
const int &j=4;       // j инициализируется const-значением 4
j++;                  // ошибка l-value specifies const object
int k=j;              // переменная k инициализируется значением
                      // временного объекта
```

Если тип инициализатора не совпадает с типом ссылки, то могут возникнуть проблемы с преобразованием данных одного типа к другому, например:

```
double f=2.5;
int &n=(int &)f;
n=3;
```

Адрес переменной f и ссылки n совпадают, но значения различаются, так как структуры данных плавающего и целочисленного типов различны.

Можно создать ссылку на ссылку, например:

Библиотека БГУИР

```
int k=1;
int &n=k;              // n – ссылка на k (равно k)
n++;                   // значение k равно 2
int &nn=n;             // nn – ссылка на ссылку n (переменную k)
nn++;                  // значение k равно 3
```

Значения адреса переменной k, ссылок n и nn совпадают, следовательно, для ссылки nn не создается временный объект.

На применение переменных ссылочного типа накладываются некоторые ограничения:

– ссылки не являются указателями;
– можно взять ссылку от переменной ссылочного типа;
– можно создать указатель на ссылку;
– нельзя создать массив ссылок;
– ссылки на битовые поля не допускаются.

Параметры ссылки

Если требуется предоставить возможность функции изменять значения передаваемых в нее параметров, то в языке С они должны быть объявлены либо глобально, либо работа с ними в функции осуществляется через передаваемые в нее указатели на эти переменные. В С++ аргументы в функцию можно передавать также и через ссылку. Для этого при объявлении функции перед параметром ставится знак &.

```
#include <iostream.h>
void fun1(int,int);
void fun2(int &,int &);
void main()
{ int i=1,j=2;                    // i и j – локальные параметры
  cout << "\n адрес переменных в main()  i = "<<&i<<"   j = "<<&j;
  cout << "\n      i = "<<i<<"   j = "<<j;
  fun1(i,j);
  cout << "\n    значение   i = "<<i<<"   j = "<<j;
  fun2(i,j);
  cout << "\n    значение   i = "<<i<<"   j = "<<j;
}

void fun1(int i,int j)
{  cout << "\n адрес переменных в fun1()  i = "<<&i<<"   j = "<<&j;
   int a;                      //  при вызове fun1 i и j из main() копируются
   a=i; i=j; j=a;              //  в стек в переменные i и j при возврате в main()
}                              //  они просто теряются

void fun2(int &i,int &j)
{  cout << "\n адрес переменных в fun2()  i = "<<&i<<"   j = "<<&j;
   int a;               //  здесь используются ссылки на переменные i и j
```

Библиотека БГУИР

```
    a=i; i=j; j=a;        //   из main()   (вторые их имена)  и т.о. действия в
                         //  функции производятся с теми же переменными i и j
}
```

В функции fun2 в инструкции

a=i;

не используется операция *. При объявлении параметра-ссылки компилятор C++ определяет его как неявный указатель (ссылку) и обрабатывает его соответствующим образом. При вызове функции fun2 ей автоматически передаются адреса переменных i и j. Таким образом, в функцию передаются не значения переменных, а их адреса, благодаря чему функция может модифицировать значения этих переменных. При вызове функции fun2 знак & перед переменными i и j ставить **нельзя**.

Независимые ссылки

В языке C++ ссылки могут быть использованы не только для реализации механизма передачи параметров в функцию. Они могут быть объявлены в программе наряду с обычными переменными, например:

```
#include <iostream.h>
void main()
{ int i=1;
  int &j=i;                  //  j – ссылка (второе имя) переменной i
  cout << "\n  адрес переменных  i = "<<&i<<"   j = "<<&j;
  cout << "\n  значение    i = "<<i<<"   j = "<<j;
  j=5;                       //
  cout << "\n  адрес переменных  i = "<<&i<<"   j = "<<&j;
  cout << "\n   значение   i = "<<i<<"   j = "<<j;
}
```

В результате работы программы будет получено:

адрес переменных i = 0х**адрес1**   j = 0х**адрес2**

значение   i = 1   j = 1

адрес переменных i = 0х**адрес1**   j = 0х**адрес2**

значение   i =5   j = 5

В этом случае компилятор создает временный объект j, которому присваивается адрес ранее созданного объекта i. Далее j может быть использовано как второе имя i.

Наследование (производные классы)

Наследование – это одна из главных особенностей ООП. Наследование заключается в том, что один класс наследует некоторые свойства другого. Этот принцип предполагает использование *базового класса*, описывающего наиболее общие свойства ряда объектов. *Производные классы* включают в себя все черты базового класса, а также добавляют новые, характерные только для объектов данного класса. Спецификация описания производного класса имеет следую-

Библиотека БГУИР

щий синтаксис:

class имя_производного_класса : [атрибут] имя_базового_класса {тело_произв_класса} [список объектов];

Двоеточие отделяет производный класс от базового. Как отмечалось ранее, ключевое слово class может быть заменено на слово struct. При этом все компоненты будут иметь атрибут public. Следует отметить, что объединение (union) не может быть ни базовым, ни производным классом.

Одна из особенностей порожденного класса – видимость унаследованных компонент базового класса. Для определения доступности компонент базового класса из компонент производного класса используются ключевые слова private, protected и public (атрибуты базового класса). Например:

```
class base
{ private :        private-компоненты;
  public :         public-компоненты;
  protected :      protected-компоненты;
};
class proizv_priv : private base { любые компоненты};
class proizv_publ : public base { любые компоненты};
class proizv_prot :  protected base { любые компоненты};
```

Производный класс наследует атрибуты компонент базового класса в зависимости от атрибутов базового класса следующим образом:

если базовый класс имеет атрибут public, то компоненты public и protected базового класса наследуются с атрибутами public и protected в производном классе. Компоненты private остаются private-компонентами базового класса;

если базовый класс имеет атрибут protected, то компоненты public и protected базового класса наследуются с атрибутом protected в производном классе. Компоненты private остаются private-компонентами базового класса;

если базовый класс имеет атрибут private, то компоненты public и protected базового класса наследуются с атрибутами private в производном классе. Компоненты private остаются private-компонентами базового класса.

Отмеченные типы наследования называются: внешним, защищенным и внутренним.

Из этого видно, что использование атрибутов private и protected ограничивает права доступа к компонентам базового класса через производный от базового класс.

Доступ к данным базового класса из производного осуществляется по имени (опуская префикс).

```
#include <iostream.h>
#include <string.h>
#define n 10
class book                              // БАЗОВЫЙ класс book
{ protected:
```

Библиотека БГУИР

```
      char naz[20];                 // название книги
       int kl;                      // количество страниц
  public:
       book(char *,int);            // конструктор класса book
      ~book();                      // деструктор класса book
};
class avt : public book                     // ПРОИЗВОДНЫЙ класс
 {     char fm[10];                 // фамилия автора
  public:
       avt(char *,int,char *);      // конструктор класса avt
       ~avt();                      // деструктор класса avt
       void see();
};
enum razd {teh,hyd,uch};
class rzd : public book                     // ПРОИЗВОДНЫЙ класс
{      razd rz;                     // раздел каталога
  public:
       rzd(char *, int, razd);      // конструктор класса rzd
       ~rzd();                      // деструктор класса rzd
       void see();
};

book::book(char *s1,int i)
{ cout << "\n работает конструктор класса   book";
  strcpy(naz,s1);
  kl=i;
}

book::~book()
{cout << "\n работает деструктор класса    book";}

avt::avt(char *s1,int i,char *s2) : book(s1,i)
{ cout << "\n работает конструктор класса    avt";
  strcpy(fm,s2);
}

avt::~avt()
{cout << "\n работает деструктор класса    avt";}

void avt::see()
{ cout<<"\nназвание : "<<naz<<"\nстраниц : "<<kl;
}

rzd::rzd(char *s1,int i,razd tp) : book(s1,i)
```

Библиотека БГУИР

```
{ cout << "\n работает конструктор класса    rzd";
  rz=tp;
}

rzd::~rzd()
{cout << "\n работает деструктор класса    rzd";}

void rzd::see()
{ switch(rz)
  { case teh : cout << "\nраздел технической литературы"; break;
    case hyd : cout << "\ nраздел художественной литературы "; break;
    case uch : cout << "\ nраздел учебной литературы "; break;
  }
}
void main()
{avt av("Книга 1",123," автор1");//вызов конструкторов классов book и avt
 rzd rz("Книга  1",123,teh);        //вызов конструкторов классов book и rzd
 av.see();
 rz.see();
}
```

На приведенном ниже примере показаны различные способы доступа к компонентам классов иерархической структуры, в которой классы A, B, C – базовые для класса D, а класс D, в свою очередь, является базовым для класса E.

```
#include "iostream.h"
class A
{ private:    a_1(){cout<<"private-функция a_1"<< endl;}
  protected: a_2(){cout<<"protected-функция a_2"<< endl;}
  public:     a_3(){cout<<"public-функция a_3"<< endl;}
};

class B
{ private:    b_1(){cout<<"private-функция b_1"<< endl;}
  protected: b_2(){cout<<"protected-функция b_2"<< endl;}
  public:     b_3(){cout<<"public-функция b_3"<< endl;}
};

class C
{ private:    c_1(){cout<<"private-функция c_1"<< endl;}
  protected: c_2(){cout<<"protected-функция c_2"<< endl;}
  public:     c_3(){cout<<"public-функция c_3"<< endl;}
};

class D : public A, protected B, private C
```

Библиотека БГУИР

```
{ private:     d_1(){cout<<"private-функция d_1"<< endl;}
  protected: d_2(){cout<<"protected-функция d_2"<< endl;}
  public:      d_3();
};
 D:: d_3()
{     d_1();
      d_2();
//     a_1();    //'a_1' cannot access private member declared in class 'A'
      a_2();
      a_3();
//      b_1();    //'b_1' cannot access private member declared in class 'B'
       b_2();
       b_3();
//     c_1();    //'c_1' cannot access private member declared in class 'C'
      c_2();
      c_3();
return 0;
 }
```

class E : public D

```
{ public:   e_1();
};
E:: e_1()
{//   a_1();    //'a_1' cannot access private member declared in class 'A'
      a_2();
      a_3();
//     b_1();    // b_1 cannot access private member declared in class 'B'
      b_2();
      b_3();
//    c_1();    // c_1 cannot access private member declared in class 'C'
//    c_2();    // c_2 cannot access private member declared in class 'C'
//    c_3();    // c_3 cannot access private member declared in class 'C'
return 0;
}

int main()
{  A a;
   a.a_3();
   B b;
   b.b_3();
   C c;
   c.c_3();
   D d;
   d.a_3();
```

Библиотека БГУИР

```
//  d.b_3();  // b_3 cannot access public member declared in class 'B'
//  d.c_3();  // c_3 cannot access public member declared in class 'C'
    E e;
    e.d_3();
    e.e_1();
  return 0;
}
```

Конструкторы и деструкторы

Инструкция avt av("книга 1",123," автор 1") в примере программы предыдущего пункта приводит к формированию объекта av и вызова конструктора avt производного класса и конструктора book базового класса (предыдущая программа):

void avt::avt(char *s1,int i,char *s2) : book(s1,i)

При этом вначале вызывается конструктор базового класса book (выполняется инициализация компонент-данных naz и kl), затем конструктор производного класса avt (инициализация компоненты fm). Поскольку базовый класс ничего не знает про производные от него классы, его инициализация (вызов его конструктора) производится перед инициализацией (активизацией конструктора) производного класса.

В противоположность этому деструктор производного класса вызывается перед вызовом деструктора базового класса. Это объясняется тем, что уничтожение объекта базового класса влечет за собой уничтожение и объекта производного класса, следовательно, деструктор производного класса должен выполняться перед деструктором базового класса.

Ниже приведена программа вычисления суммы двух налогов, в которой использована следующая форма записи конструктора базового и производного классов.

имя_конструктора(тип переменной_1 имя_переменной_1,…,
тип переменной_n имя_переменной_n) :
имя_конструктора_базового_класса(имя_переменной_1,…, имя_переменной_k),
компонента_данное_1(имя_переменной_m),…,
компонента_данное_n(имя_переменной_n);

```
#include "iostream.h"
class A                          // БАЗОВЫЙ класс
{   protected:double pr1,pr2;    // protected для видимости pr в классе B
  public:
    A(double prc1,double prc2): pr1(prc1),pr2(prc2) {};
    void a_prnt(){cout << "% налога = " << pr1 << " и " << pr2 << endl;}
};
class B : public A               // ПРОИЗВОДНЫЙ класс
{    int  sm;
  public:
```

Библиотека БГУИР

```
        B(double prc1,double prc2,int sum): A(prc1,prc2),sm(sum) {};
        void b_prnt()
        { cout << " налоги на сумму = " << sm << endl;
          cout << "первый = " << pr1 <<"\n втрой = " << pr2 << endl;
        }
        double rashet() {return pr1*sm/100+pr2*sm/100;}
    };
    void main()
    { A aa(9,5.3);             // описание объекта aa (базового класса) и инициа-
                               // лизация его компонент с использованием
                               //  конструктора A()
      B bb(7.5,5,25000);   // описание объекта bb (производного класса)
                           // и инициализация его компонент (вызов конструк-
                           // тора  B() и конструктора A() (первым))
     aa.a_prnt();
     bb.b_prnt();
     cout << "Сумма налога = " << bb.rashet() << endl;
    }
```

В приведенном примере использованы функции-конструкторы следующего вида:

```
public: A(double prc1,double prc2): pr1(prc1),pr2(prc2) {};
public: B(double prc1,double prc2,int sum): A(prc1,prc2),sm(sum) {};
```

Конструктор A считывает из стека 2 double значения prc1 и prc2, которые далее используются для инициализации компонент класса A **pr1**(prc1),**pr2**(prc2). Аналогично конструктор B считывает из стека 2 double значения prc1 и prc2 и одно значение int, после чего вызывается конструктор класса A(prc1,prc2), затем выполняется инициализация компоненты sm класса B.

Производный класс может служить базовым классом для создания следующего производного класса. При этом, как отмечалось выше, конструкторы выполняются в порядке наследования, а деструкторы в обратном порядке. Если при наследовании переопределена хотя бы одна перегруженная функция, то все остальные варианты этой функции будут скрыты. Если необходимо, чтобы они оставались доступными в производном классе, все их надо переопределить. Если в производном классе создается функция с тем же именем, типом возвращаемого значения и сигнатурой, как и у функции–компоненты базового класса, но с новой реализацией, то эта функция считается ***переопределенной***. Под сигнатурой понимают имя функции со списком параметров, заданных в ее прототипе, а также слово const. Типы возвращаемых значений могут различаться. Распространенная ошибка: пытаясь переопределить функцию базового класса, забывают включить слово const, в результате чего функция оказывается скрыта, а не переопределена.

Ниже приведен текст еще одной программы, в которой также используется наследование. В программе выполняется преобразование арифметического выражения из обычной (инфиксной) формы в форму польской записи. Для это-

Библиотека БГУИР

го используется стек, в который заносятся арифметические операции. Алгоритм преобразования выражения здесь не рассматривается.

```
#include<iostream.h>
#include<stdlib.h>
class st                           // описание класса – элемент стека операций
{ public :
 char c;
     st *next;
 public:
    st(){}              //  конструктор
   ~st(){}              //  деструктор
};
class cl : public st
{  char *a;                         //   исходная строка (для анализа)
   char outstr[80];                 //   выходная строка
 public :
   cl() : st() {}       //  конструктор
   ~cl(){}             //  деструктор
   st *push(st *,char);             //  занесение символа в стек
   char pop(st **);                 //  извлечение символа из стека
   int PRIOR(char);                 //  определение приоритета операции
   char *ANALIZ(char *);            //  преобразование в польскую запись
};

char * cl::ANALIZ(char *aa)
{ st *OPERS;                        //
  OPERS=NULL;                       //  стек операций пуст
  int k,p;
  a=aa;
  k=p=0;
  while(a[k]!='\0'&&a[k]!='=')        // пока не дойдем до символа     '='
  { if(a[k]==')')                    // если очередной символ    ')'
    { while((c=pop(&OPERS))!='(') // считываем из стека в выходную
      outstr[p++]=c;                 // строку все знаки операций до символа
                                    // ‘(‘  и удаляем из стека ‘(‘
    }
   if(a[k]>='a'&&a[k]<='z')          // если символ – буква, то
   outstr[p++]=a[k];                 // заносим ее в выходную строку
   if(a[k]=='(')                     //  если очередной символ  '('  , то
     OPERS=push(OPERS,'(');          //  помещаем его в стек
   if(a[k]=='+'||a[k]=='-'||a[k]=='/'||a[k]=='*')
   {                             // если следующий символ – знак операции, то
     while((OPERS!=NULL)&&(PRIOR(c)>=PRIOR(a[k])))
```

```
    outstr[p++]=pop(&OPERS); // переписываем в выходную строку все
                             //  находящиеся в стеке операции с большим
                             //  или равным приоритетом
    OPERS=push(OPERS,a[k]);  // записываем в стек очередную операцию
  }
  k++;                // переход к следующему символу выходной строки
 }
 while(OPERS!=NULL)           // после анализа всего выражения
 outstr[p++]=pop(&OPERS);     // переписываем операции из стека
 outstr[p]='\0';              // в выходную строку
 return outstr;
}

st *cl::push(st *head,char a)  // функция записи символа в стек и возврата
{ st *PTR;                     // указателя на вершину стека
 if(!(PTR=new st))
 { cout << "\n недостаточно памяти для элемента стека";  exit(-1);}
 PTR->c=a;            // инициализация элемента стека
 PTR->next=head;
 return PTR;          // PTR – вершина стека
}

char cl::pop(st **head)  // функция удаления символа с вершины стека
{ st *PTR;                // возвращает символ (с вершины стека) и коррек-
                          // тирует указатель на вершину стека
 char a;
 if(!(*head)) return '\0';  // если стек пуст, то возвращается  '\0'
 PTR=*head;               // адрес вершины стека
 a=PTR->c;                //  считывается содержимое с вершины стека
 *head=PTR->next;  // изменяем адрес вершины стека (nex==PTR->next)
 delete PTR;
 return a;
}

int cl::PRIOR(char a)     // функция возвращает приоритет операции
{ switch(a)
 { case '*':case '/':return 3;
  case '-':case '+':return 2;
  case '(':return 1;
 } return 0;
}

void main()
{ char a[80];       //  исходная строка
```

```
  cl cls;
  cout << "\nВведите выражение  (в конце символ  '='):  ";
  cin >> a;
  cout << cls.ANALIZ(a) << endl;
}
```

В результате работы программы получим:

Введите выражение (в конце символ '=') : (a+b)-c*d=
ab+cd*-

Виртуальные функции

Один из основных принципов объектно-ориентированного программирования предполагает использование идеи «один интерфейс – множество методов реализации». Эта идея заключается также в том, что базовый класс обеспечивает все элементы, которые производные классы могут непосредственно использовать, плюс набор функций, которые производные классы должны реализовать путем их переопределения. Наряду с механизмом перегрузки функций это достигается использованием виртуальных (virtual) функций. *Виртуальная* функция – это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или нескольких производных от этого классах. При вызове объекта базового или производных классов динамически (во время выполнения программы) определяется, какую из функций требуется вызвать, основываясь на типе объекта.

Рассмотрим пример использования виртуальной функции.

```
#include "iostream.h"
#include "iomanip.h"
#include "string.h"
class grup                       // БАЗОВЫЙ класс
{ protected:
     char *fak;            //  наименование факультета
     long gr;               //  номер группы
 public:
      grup(char *FAK,long GR) : gr(GR)
      {  if (!(fak=new char[20]))
          { cout<<"ошибка выделения памяти"<<endl;
             return;
          }
          strcpy(fak,FAK);
       }
       ~grup()
       { cout << "деструктор класса grup " << endl;
         delete fak;
    }
      virtual void see(void);          // объявление виртуальной функции
```

Библиотека БГУИР

```
};
class stud : public grup              // ПРОИЗВОДНЫЙ класс
{     char *fam;                  //  фамилия
      int oc[4];                  //  массив оценок
 public:
      stud(char *FAK,long GR,char *FAM,int OC[]): grup(FAK,GR)
      { if (!(fam=new char[20]))
         { cout<<"ошибка выделения памяти"<<endl;
           return;
         }
         strcpy(fam,FAM);
         for(int i=0;i<4;oc[i]=OC[i++]);
      }
      ~stud()
      { cout << "деструктор класса stud " << endl;
        delete fam;
      }
      void see(void);
};
void grup::see(void)      // описание виртуальной функции
{ cout << fak << gr << endl;}
void stud::see(void)      //
{ grup ::see();               // вызов функции базового класса
  cout <<setw(10) << fam << "  ";
  for(int i=0; i<4; cout << oc[i++]<<' ');
  cout << endl;
}
int main()
{ int OC[]={4,5,5,3};
  grup gr1("факультет 1",123456), gr2("факультет 2",345678), *p;
  stud st("факультет 2",150502,"Иванов",OC);
  p=&gr1;          // указатель на объект базового класса
  p->see();        // вызов функции базового класса объекта gr1
  (&gr2)->see();  // вызов функции базового класса объекта gr2
  p=&st;           // указатель на объект производного класса
  p->see();        // вызов функции производного класса объекта st
  return 0;
}
```

Результат работы программы:

```
факультет 1   123456
факультет 2   345678
факультет 2",150502
      Иванов   4 5 5 3
```

Объявление virtual void see(void) говорит о том, что функция see может

Библиотека БГУИР

быть различной для базового и производных классов. Тип виртуальной функции не может быть переопределен в производных классах. Исключением является случай, когда возвращаемый тип виртуальной функции является указателем или ссылкой на порожденный класс, а виртуальная функция основного класса – указателем или ссылкой на базовый класс. В производных классах функция может иметь список параметров, отличный от параметров виртуальной функции базового класса. В этом случае эта функция будет не виртуальной, а перегруженной. Вызов функций должен производиться с учетом списка параметров.

Механизм вызова виртуальных функций можно пояснить следующим образом. При создании нового объекта для него выделяется память. Для виртуальных функций (и только для них) создается указатель на таблицу функций, из которой выбирается требуемая в процессе выполнения. Доступ к виртуальной функции осуществляется через этот указатель и соответствующую таблицу (то есть выполняется косвенный вызов функции).

Если функция вызывается с использованием ее полного имени grup::see(), то виртуальный механизм игнорируется. Игнорирование этого может привести к серьезной ошибке:

```
void stud::see(void)
{ see();
   . . . .  }
```

В этом случае инструкция see() приводит к бесконечному рекурсивному вызову функции see().

Заметим, что деструктор может быть виртуальным, а конструктор нет.

Замечание. В чем разница между виртуальными функциями (методами) и переопределением функции?

Что изменилось, если бы функция see() не была бы описана как виртуальная? В этом случае решение о том, какая именно из функций see() должна быть выполнена, будет принято при ее компиляции.

Свойство виртуальности проявляется только тогда, когда обращение к функции идет через указатель или ссылку на объект. Указатель или ссылка могут указывать как на объект базового, так и на объект производного классов. Если в программе имеется сам объект, то уже на стадии компиляции известен его тип и, следовательно, механизм виртуальности не используется. Например:

```
func(cls obj)
{
    obj.vvod();    // вызов компоненты-функции obj::vvod
}
func1(cls &obj)
{
    obj.vvod();    // вызов компоненты-функции в соответствии
}                  // с типом объекта, на который  ссылается obj
```

Виртуальные функции позволяют принимать решение в процессе выполнения.

```
#include "iostream.h"
#include "iomanip.h"
#include "string.h"
#define N 5
class base                    // БАЗОВЫЙ класс
{ public:
    virtual char *name(){ return " noname ";}
    virtual double area(){ return 0;}
};
class rect : public base   // ПРОИЗВОДНЫЙ класс (прямоугольник)
{    int h,s;
  public:
    virtual char *name(){ return " прямоугольника ";}
     rect(int H,int S): h(H),s(S) {}
    double area(){ return h*s;}
};
class circl : public base   // ПРОИЗВОДНЫЙ класс (окружность)

{    int r;
  public:
    virtual char *name(){ return " круга ";}
    circl(int R): r(R){}
    double area(){ return 3.14*r*r;}
};

int main()
{ base *p[N],a;
  double s_area=0;
  rect b(2,3);
  circl c(4);
  for(int i=0;i<N;i++) p[i]=&a;
  p[0]=&b;
  p[1]=&c;
  for(i=0;i<N;i++)
  cout << "плошадь" << p[i]->name() << p[i]->area() << endl;
  return 0;
}
```

Массив указателей p хранит адреса объектов базового и производных классов и необходим для вызова виртуальных функций этих классов. Виртуальная функция базового класса необходима тогда, когда она явно вызывается

Библиотека БГУИР

для базового класса или не определена (не переопределена) для некоторого производного класса.

Если функция была объявлена как виртуальная в некотором классе (базовом классе), то она остается виртуальной независимо от количества уровней в иерархии классов, через которые она прошла.

```
class A
{  . . .
   public:  virtual void fun() {}
};
class B : public A
{  . . .
   public:  void fun() {}
};

class C : public B
{  . . .
   public:
   . . .    // в объявлении класса C отсутствует описание функции fun()
};
main()
{ A a,*p=&a;
  B b;
  C c;
  p->fun();  // вызов версии виртуальной функции fun для класса A
  p=&b;
  p->fun();  // вызов версии виртуальной функции fun для класса B
  p=&c;
  p->fun();  // вызов версии виртуальной функции fun для класса C (из A)
}
```

Если в производном классе виртуальная функция не переопределяется, то используется ее версия из базового класса.

```
class A
{  . . .
   public:  virtual void fun() {}
};
class B : public A
{  . . .
   public:  void fun() {}
};

class C : public B
{  . . .
   public:  void fun() {}
```

Библиотека БГУИР

```
};
main()
{ A a,*p=&a;
  B b;
  C c;
  p->fun();    // вызов версии виртуальной функции fun для класса A
  p=&b;
  p->fun();    // вызов версии виртуальной функции fun для класса B
  p=&c;
  p->fun();    // вызов версии виртуальной функции fun для класса A
}
```

Основное достоинство данной иерархии состоит в том, что код не нуждается в глобальном изменении, даже при добавлении вычислений площадей новых фигур. Изменения вносятся локально и благодаря полиморфному характеру кода распространяются автоматически.

Рассмотрим механизм вызова виртуальной функции базового и производного классов из компонент-функций этих классов, вызываемых через указатель на базовый класс.

```
class A
{ public :
     virtual void f()
     { return; }
     void fn()
     { f(); }        // вызов функции f
};
class B : public A
{ public:
    void f()
     { return;  }
    void fn()
     { f(); }        // вызов функции f
};

main()
{ A a,*pa=&a;
  B b,*pb=&b;
  pa = &b;
  pa->fn();  //  вызов виртуальной функции f класса B через  A::fn()
  pb->fn();  //  вызов виртуальной функции f класса B через  B::fn()
}
```

В инструкции pa->fn() выполняется вызов функции fn() базового класса A, так как указатель **pa** – указатель на базовый класс и компилятор выполняет вызов функции базового класса. Далее из функции fn() выполняется вызов вир-

туальной функции f() класса B, так как указатель **pa** инициализирован адресом объекта B.

Перечислим основные свойства и правила использования виртуальных функций:

– виртуальный механизм поддерживает полиморфизм на этапе выполнения программы. Это значит, что требуемая версия программы выбирается на этапе выполнения программы, а не компиляции;

– класс, содержащий хотя бы одну виртуальную функцию, называется полиморфным;

– виртуальные функции можно объявить только в классах (class) и структурах (struct);

– виртуальными функциями могут быть только нестатические функции (без спецификатора static), так как характеристика virtual унаследуется. Функция порожденного класса автоматически становится virtual;

– виртуальные функции можно объявить со спецификатором friend для другого класса;

– виртуальными функциями могут быть только неглобальные функции (то есть компоненты класса);

– если виртуальная функция объявлена в производном классе со спецификатором virtual, то можно рассматривать новые версии этой функции в классах, наследуемых из этого производного класса. Если спецификатор virtual опущен, то новые версии функции далее не будут рассматриваться;

– для вызова виртуальной функции требуется больше времени, чем для невиртуальной. При этом также требуется дополнительная память для хранения виртуальной таблицы;

– при использовании полного имени при вызове некоторой виртуальной функции (например, grup ::see();), виртуальный механизм не применяется.

Абстрактные классы

Базовый класс иерархии типа обычно содержит ряд виртуальных функций, обеспечивающих динамическую типизацию. Часто в базовом классе эти виртуальные функции фиктивны и имеют пустое тело. Эти функции существуют как некоторая абстракция, конкретное значение им придается в производных классах. Такие функции называются **чисто виртуальными функциями,** то есть такими, тело которых, как правило, не определено. Общая форма записи абстрактной функции имеет вид:

virtual прототип функции = 0;

Чисто виртуальная функция используется для того, чтобы отложить решение о реализации функции. То, что функция объявлена чисто виртуальной, требует, чтобы эта функция была определена во всех производных классах от класса, содержащего эту функцию. Если класс имеет хотя бы одну чисто виртуальную функцию, то он называется **абстрактным**. Для абстрактного класса нельзя создать объекты и он используется только как базовый класс для других

Библиотека БГУИР

классов. Если base – абстрактный класс, то для инструкций

```
base a;
base *p= new base;
```

компилятор выдаст сообщение об ошибке. В то же время вполне можно использовать инструкции вида

```
rect b;
base *p=&b;
base &p=b;
```

Чисто виртуальную функцию, как и просто виртуальную функцию, не обязательно переопределять в производных классах. При этом если в производном классе она не переопределена, то этот класс тоже будет абстрактным, и при попытке создать объект этого класса компилятор выдаст ошибку. Таким образом, забыть переопределить чисто виртуальную функцию невозможно. Абстрактный базовый класс навязывает определенный интерфейс всем производным от него классам. Главное назначение абстрактных классов – в определении интерфейса для некоторой иерархии классов.

Класс можно сделать абстрактным, даже если все его функции определены. Это можно сделать, например, чтобы быть уверенным, что объект этого класса создан не будет. Обычно для этих целей выбирается деструктор.

```
class base
{   компоненты-данные
    public:
       virtual ~base() = 0;
       компоненты-функции
}
base ::~base()
{реализация деструктора}
```

Объект класса base создать невозможно, в то же время деструктор его определен и будет вызван при разрушении объектов производных классов.

Для иерархии типа полезно иметь базовый абстрактный класс. Он содержит общие свойства порожденных объектов и используется для объявления указателей, которые могут обращаться к объектам классов, порожденным от базового. Рассмотрим это на примере программы экологического моделирования. В примере мир будет иметь различные формы взаимодействия жизни с использованием абстрактного базового класса living. Его интерфейс унаследован различными формами жизни. Создадим fox (лис) – хищника, rabbit (кролик) – жертву и grass – (траву).

```
#include "iostream.h"
#include "conio.h"
// моделирование хищник – жертва с использованием
// иерархии классов
const int N=6,                   // размер квадратной площади (мира)
        STATES=4,                // кол-во видов жизни
```

Библиотека БГУИР

```
        DRAB=5,DFOX=5,     // кол-во циклов жизни кролика и лиса
        CYCLES=10;             // общее число циклов моделирования мира
enum state{EMPTY,GRASS,RABBIT,FOX};
class living;                          // forvard  объявление
typedef living *world[N][N];         // world- модель мира
void init(world);
void gener(world);
void update(world,world);
void dele(world);

class living
{protected:
     int  row,col;                         // местоположение в модели
     void sums(world w,int sm[]);        //
 public:
     living(int r,int c):row(r),col(c){}
     virtual state who() = 0;               // идентификация состояний
     virtual living *next(world w)=0;    // расчет next
     virtual void print()=0;                // вывод содержимого поля модели
};

void living::sums(world w,int sm[])
{ int i,j;
  sm[EMPTY]=sm[GRASS]=sm[RABBIT]=sm[FOX]=0;
  int i1=-1,i2=1,j1=-1,j2=1;
  if(row==0) i1=0;                     // координаты внешних клеток модели
  if(row==N) i2=0;
  if(col==0) j1=0;
  if(col==N) j2=0;
  for(i=i1;i<=i2;++i)
    for(j=j1;j<=j2;++j)
    sm[w[row+i][col+j]->who()]++;
}
```

В базовом классе living объявлены две чисто виртуальные функции who() и next() и одна обычная функция sums(). Моделирование имеет правила для решения о том, кто продолжает жить в следующем цикле. Они основаны на соседствующих популяциях в некотором квадрате. Глубина иерархии наследования – один уровень.

```
// текущий класс – только хищники
class fox:public living
{ protected:
     int age; // используется для принятия решения о смерти лиса
  public:
```

Библиотека БГУИР

```
    fox(int r,int c,int a=0):living(r,c),age(a){}
    state who() {return FOX;}            // отложенный метод для foxes
    living *next(world w);               // отложенный метод для foxes
    void print(){cout << " ли ";}
};

// текущий класс – только жертвы
class rabbit:public living
{ protected:
    int age; // используется для принятия решения о смерти кролика
  public:
    rabbit(int r,int c,int a=0):living(r,c),age(a){}
    state who() {return RABBIT;}         // отложенный метод для rabbit
    living *next(world w);               // отложенный метод для rabbit
    void print(){cout << " кр ";}
};

// текущий класс – только растения
class grass:public living
{ public:
     grass(int r,int c):living(r,c){}
     state who() {return GRASS;}          // отложенный метод для grass
     living *next(world w);               // отложенный метод для grass
     void print(){cout << " тр ";}
};

// жизнь отсутствует
class empty : public living
{ public:
    empty(int r,int c):living(r,c){}
    state who() {return EMPTY;}          // отложенный метод для empty
    living *next(world w);               // отложенный метод для empty
    void print(){cout << "    ";}
};
```

Характеристика поведения каждой формы жизни фиксируется в версии next(). Если в окрестности имеется больше grass, чем rabbit, grass остается, иначе grass будет съедена.

```
living *grass::next(world w)
{ int sum[STATES];
  sums(w,sum);
  if(sum[GRASS]>sum[RABBIT])             // кролик ест траву
    return (new grass(row,col));
  else
    return(new empty(row,col));
```

```
}
```

Если возраст rabbit превышает определенное значение DRAB, он умирает либо, если поблизости много лис, он может быть съеден.

```
living *rabbit::next(world w)
{ int sum[STATES];
  sums(w,sum);
  if(sum[FOX]>=sum[RABBIT])            // лис ест кролика
    return (new empty(row,col));
  else if(age>DRAB)                    // кролик слишком старый
         return(new empty(row,col));
      else
         return(new rabbit(row,col,age+1));  // кролик постарел
}
```

Fox тоже умирает от старости.

```
living *fox::next(world w)
{ int sum[STATES];
  sums(w,sum);
  if(sum[FOX]>5)                       // слишком много лис
    return (new empty(row,col));
  else if(age>DFOX)                    // лис слишком старый
         return(new empty(row,col));
       else
         return(new fox(row,col,age+1));  // лис постарел
}

// заполнение пустой площади
living *empty::next(world w)
{ int sum[STATES];
  sums(w,sum);
  if(sum[FOX]>1)                       // первыми добавляются лисы
    return (new fox(row,col));
  else if(sum[RABBIT]>1)               // вторыми добавляются кролики
         return (new rabbit(row,col));
       else if(sum[GRASS])             // третьими добавляются растения
              return (new grass(row,col));
            else return (new empty(row,col));  // иначе пусто
}
```

Массив world представляет собой контейнер для жизненных форм. Он должен иметь в собственности объекты living, чтобы распределять новые и удалять старые.

```
// world полностью пуст
void init(world w)
{ int i,j;
  for(i=0;i<N;++i)
```

Библиотека БГУИР

```
    for(j=0;j<N;++j)
    w[i][j]=new empty(i,j);
}

// генерация исходной модели мира
void gener(world w)
{ int i,j;
  for(i=0;i<N;++i)
    for(j=0;j<N;++j)
    { if(i%2==0 && j%3==0) w[i][j]=new fox(i,j);
            else if(i%3==0 && j%2==0) w[i][j]=new rabbit(i,j);
                else if(i%5==0) w[i][j]=new grass(i,j);
                    else w[i][j]=new empty(i,j);
    }
}

// вывод содержимого модели мира на экран
void pr_state(world w)
{ int i,j;
  for(i=0;i<N;++i)
  { cout<<endl;
    for(j=0;j<N;++j)
   w[i][j]->print();
  }
  cout << endl;
}

// новый world w_new рассчитывается из старого world w_old
void update(world w_new, world w_old)
{ int i,j;
  for(i=0;i<N;++i)
   for(j=0;j<N;++j)
   w_new[i][j]=w_old[i][j]->next(w_old);
}

// очистка мира
void dele(world w)
{ int i,j;
  for(i=1;i<N-1;++i)
  for(j=1;j<N-1;++j) delete(w[i][j]);
}
```

Модель имеет odd и even мир. Их смена является основой для расчета последующего цикла.

Библиотека БГУИР

```
int main()
{ world odd,even;
  int i;
  init(odd);
  init(even);
  gener(even);                  // генерация начального мира
  pr_state(even);               // выводит типы состояний gener
  for(i=0;i<CYCLES;++i)   // моделирование
  { //getch();
    if(i%2)
    { update(even,odd);
      pr_state(even);
      dele(odd);
    }
    else
   { update(odd,even);
     pr_state(odd);
    dele(even);
  }
 }
 return 1;
}
```

Множественное наследование

В языке С++ имеется возможность образовывать производный класс от нескольких базовых классов. Общая форма множественного наследования имеет вид:

class имя_произв_класса : имя_базового_кл 1,…,имя_базового_кл N
{ содержимое класса
}

Иерархическая структура, в которой производный класс наследует от несколько базовых классов, называется множественным наследованием. В этом случае производный класс, имея собственные компоненты, имеет доступ к protected- и public-компонентам базовых классов.

Конструкторы базовых классов при создании объекта производного класса вызываются в том порядке, в котором они указаны в списке при объявлении производного класса.

При применении множественного наследования возможно возникновение нескольких конфликтных ситуаций. ***Первая*** – конфликт имен методов или атрибутов нескольких базовых классов:

```
class A
{public: void fun(){}
};
```

Библиотека БГУИР

```
class B
{public: void fun(){}
};
class C : public A, public B
{ };
main()
{ C *c=new C;
  c->fun();        //  error C::f' is ambiguous
  return 0;
}
```

При таком вызове функции fun() компилятор не может определить, к какой из двух функций классов A и B выполняется обращение. Неоднозначность можно устранить, явно указав, какому из базовых классов принадлежит вызываемая функция:

c->A:: fun();   или   c->B::fun();

Вторая проблема возникает при многократном включении некоторого базового класса:

```
#include "iostream.h"
#include "string.h"
class A                          // БАЗОВЫЙ класс I уровня
{    char naz[20];         //  название фирмы
 public:
    A(char *NAZ) {strcmp(naz,NAZ);}
     ~A() {cout << "деструктор класса A" << endl;}
 void a_prnt() {cout << naz << endl;}
};
class B1 : public A              // ПРОИЗВОДНЫЙ класс (1 Базовый II уровня)
{ protected:
       long tn;
       int nom;
  public:
      B1(char *NAZ,long TN,int NOM): A(NAZ),tn(TN),nom(NOM) {};
      ~B1() {cout << "деструктор класса B1" << endl;}
     void b1_prnt()
     {  A::a_prnt();
        cout << " таб. N " << tn <<"  подразделение = " << nom <<endl;
 }
};
class B2 : public A              // ПРОИЗВОДНЫЙ класс (2 Базовый II уровня)
{ protected:
      double  zp;
 public:
```

Библиотека БГУИР

```
    B2(char *NAZ,double ZP): A(NAZ),zp(ZP) {};
    ~B2(){cout << "деструктор класса B2" << endl;}
     void b2_prnt()
     { A::a_prnt();
       cout << " зар/плата = " << zp << endl;
     }
};
class C : public B1, public B2   // ПРОИЗВОДНЫЙ класс ( III уровня)
{       char *fam;
  public:
     C(char *FAM,char *NAZ,long TN,int NOM,double ZP) :
     B1(NAZ,TN,NOM), B2(NAZ,ZP)
     {  fam = new char[strlen(FAM)+1]
        strcpy(fam,FAM);
     };
      ~C() {cout << "деструктор класса C" << endl;}
      void c_prnt()
     { B1::b1_prnt();
       B2::b2_prnt();
       cout << " фамилия " << fam<<endl;
     }
};

void main()
{ C cc("Иванов","мастра",1234,2,555.6),*pt=&cc;
//  cc.a_prnt();        ошибка    'C::a_prnt' is ambiguous
//  pt->a_prnt();

  cc.b1_prnt();
  pt->b1_prnt();

  cc.b2_prnt();
  pt->b2_prnt();

  cc.c_prnt();
  pt->c_prnt();
}
```

В приведенном примере производный класс C имеет по цепочке два одинаковых базовых класса A (A<-B1<-C и A<-B2<-C), для каждого базового класса A строится свой объект (рис. 2, 3). Таким образом, вызов функции

cc.a_prnt();

pt->a_prnt();

некорректен, так как неизвестно, какую из двух функций (какого из двух классов A) требуется вызвать.

Библиотека БГУИР

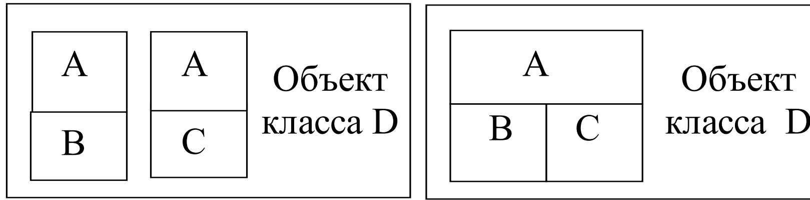

Рис. 2. Структура объекта

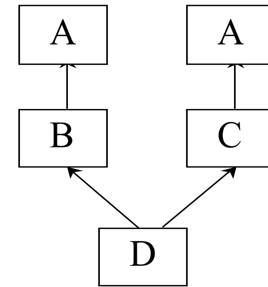

Рис. 3. Иерархия классов

Виртуальное наследование

Если базовый класс (в приведенном выше примере это класс A) является виртуальным, то будет построен единственный объект этого класса (см. рис. 2).

```
#include "iostream"
using namespace std;

class A                          // БАЗОВЫЙ виртуальный класс
{    int aa;
 public: A() {}
      A(int AA) : aa(AA) {}
       ~A() {}
};
class B : virtual public A     // ПРОИЗВОДНЫЙ класс (1 Базовый)
{   char bb;
 public: B() {}
      B(int AA,char BB): A(AA),bb(BB) {}
     ~B() {}
};
class C : virtual public A     // ПРОИЗВОДНЫЙ класс (2 Базовый)
{    float cc;
 public: C() {}
      C(int AA,float CC): A(AA),cc(CC) {}
      ~C() {}
};
class D : public B,public C  // ПРОИЗВОДНЫЙ класс (2 Базовый II уровня)
{     int dd;
public: D() {}
       D(int AA,char BB,float CC,int DD) :
       A(AA),B(AA,BB),C(AA,CC),dd(DD) {}
       ~D() {}
};

void main()
```

Библиотека БГУИР

```
{ D d(1,'a',2.3,4);
  D dd;
}
```

Виртуальный базовый класс всегда инициализируется только один раз. В примере при создании объектов d и dd конструктор класса A вызывается из конструктора класса D первым и только один раз, затем – конструкторы классов B и C, в том порядке, в котором они описаны в строке наследования классов:

class D : public B, public C .

В одно и то же время класс может иметь виртуальный и невиртуальный базовые классы, например:

```
class A{ … };
class B1: virtual public A{ … };
class B2: virtual public A{ … };
class B3: public A{ … };
class C:  public B1, public B2, public B3 { … };
```

В этом случае класс C имеет два подобъекта класса A, один наследуемый через классы B1 и B2 (общий для этих классов) и второй через класс B3 (рис. 4, 5).

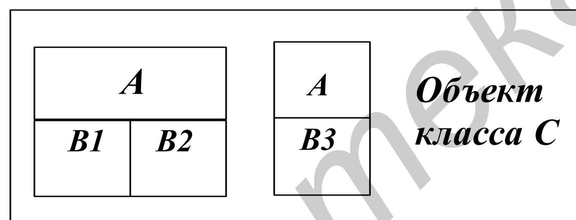

Рис. 4. Структура объекта при виртуальном наследовании

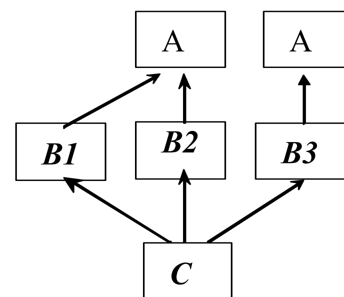

Рис. 5. Иерархия классов при виртуальном наследовании

```
#include "iostream.h"
#include "string.h"

class A                    //  БАЗОВЫЙ класс  I уровня
{   char *naz;         //  название фирмы
  public:
   A(char *NAZ)
   { naz=new char[strlen(NAZ)+1];
     strcpy(naz,NAZ);
   }
    ~A()
   { delete naz;
     cout << "деструктор класса A" << endl;
    }
```

Библиотека БГУИР

```
 void a_prnt(){cout <<"марка а/м "<< naz << endl;}
};
class B1 : virtual public A //  ПРОИЗВОДНЫЙ класс (1 Базовый II уровня)
{ protected:
     char *cv;        // цвет а/м
      int  kol;        // количество дверей
  public:
      B1(char *NAZ,char *CV,int KOL): A(NAZ),kol(KOL)
      { cv=new char[strlen(CV)+1];
        strcpy(cv,CV);
      }
      ~B1()
      { delete cv; ;
        cout << "деструктор класса B1" << endl;
   }
      void b1_prnt()
      { A::a_prnt();
        cout << "цвет а/м" << cv <<" кол-во дверей " << kol <<endl;
      }
};
class B2 : virtual public A  // ПРОИЗВОДНЫЙ класс (2 Базовый II уровня)
{ protected:
     int pow;         // мощность а/м
     double  rs;       // расход топлива
   public:
     B2(char *NAZ,int POW,double RS): A(NAZ),pow(POW),rs(RS) {};
    ~B2(){cout << "деструктор класса B2" << endl;}
    void b2_prnt()
    { A::a_prnt();
      cout <<"мощность двигателя "<<pow<<" расход топлива "<<rs;
      cout<<endl;
    }
};
class C : public B1,public B2//ПРОИЗВОДНЫЙ класс (2 Базовый II уровня)
{   char *mag;         // название магазина
  public: C(char *NAZ,char *CV,int KOL,int POW,double RS,char *MAG):
    B1(NAZ,CV,KOL),B2(NAZ,POW,RS),A(NAZ)
    { mag =new char[strlen(MAG)];
      strcpy(mag,MAG);
    }
    ~C()
   { delete mag;
     cout << "деструктор класса C" << endl;
   }
```

Библиотека БГУИР

```
    void c_prnt()
   { A::a_prnt();
     B1::b1_prnt();
     B2::b2_prnt();
     cout << " название магазина" << mag <<endl;
   }
};

void main()
{ C cc("BMW","красный",100,4,8.5,"магазин 1"),*pt=&cc;
   cc.a_prnt();
   pt->a_prnt();

  cc.b1_prnt();
  pt->b1_prnt();

  cc.b2_prnt();
  pt->b2_prnt();

  cc.c_prnt();
  pt->c_prnt();
}
```

Перегрузка функций

Одним из подходов реализации принципа полиморфизма в языке C++ является использование *перегрузки* функций. В C++ две и более функций могут иметь одно и то же имя. Компилятор C++ оперирует не исходными именами функций, а их внутренним представлением, которое учитывает количество и тип принимаемых аргументов. В то же время тип возвращаемого функцией значения не учитывается. Поэтому для компилятора функции с различным списком аргументов – это разные функции, а с одинаковым списком аргументов, но с разными типами возвращаемого значения – одинаковые. Для корректной работы программ последнего следует избегать. Функции, имеющие одинаковые имена, но разные списки аргументов, называются перегруженными. Рассмотрим простой пример перегрузки функции sum, выполняющей сложение нескольких чисел различного типа.

```
#include "iostream.h"
class cls
{   int n;
    double f;
  public:
        cls(int N,float F) : n(N),f(F) {}
        int sum(int k)              // функция sum с целочисленнным аргументом
```

Библиотека БГУИР

```
	{ n+=k;
	  return n;
	}
	double sum(double k)  // функция sum с дробным аргументом
	{ f+=k;
	  return f;
	}
	void see()
	{cout <<n<<' '<<f<<endl;}
};

void main()
{ cls obj(1,2.3);
  obj.see();                    // вывод содержимого объекта
  cout <<obj.sum(1)<<endl;  // вызов функции sum с целочисл. аргументом
  cout <<obj.sum(1.)<<endl; // вызов функции sum с дробным аргументом
}
```

Результат работы программы:

```
1  2.3
2
3.3
```

Перегрузка операторов

Имеется ограничение в языках С и С++, накладываемое на операции над известными типами данных (классами) char, int, float и т.д.:

```
char c,
int i,j;
double d,k;
```

В С и С++ определены множества операций над объектами c,i,j,d,k этих классов, выражаемых через операторы: i+j, j/k, d*(i+j). Большинство операторов (операций) в С++ может быть перегружено (переопределено), в результате чего расширяется диапазон применения этих операций. Когда оператор перегружен, ни одно из его начальных значений не теряет смысла. Просто для некоторого класса объектов определен новый оператор (операция). Для перегрузки (доопределения) оператора разрабатываются функции, являющиеся либо компонентами, либо friend-функциями того класса, для которого они используются. Остановимся на перегрузке пока только с использованием компонент- функций класса.

Для того чтобы перегрузить оператор, требуется определить действие этого оператора внутри класса. Общая форма записи функции-оператора, являющейся компонентой класса, имеет вид:

Возвращ_тип имя_класса :: operator #(список аргументов)
{   действия, выполняемые применительно к классу

Библиотека БГУИР

}

Вместо символа # ставится значок перегружаемого оператора.

Следует отметить, что нельзя перегрузить триадный оператор ”?:.”, оператор ”sizeof” и оператор разрешения контекста ”::”.

Функция operator должна быть либо компонентой класса, либо иметь хотя бы один аргумент типа ”объект класса” (за исключением при перегрузке операторов new и delete).Это позволит доопределить свойства операции, а не изменить их. При этом новое значение оператора будет иметь силу только для типов данных, определенных пользователем; для других выражений, использующих операнды стандартных типов, значение оператора останется прежним.

Выражение **a#b,** имеющее первый операнд **a** стандартного типа данных, не может быть переопределено функцией operator, являющейся компонентом класса. Например, выражение а-4 может быть представлено как a.operator-(4), где а – объект некоторого типа. Выражение вида 4-а нельзя представить в виде 4.operator(a). Это может быть реализовано с использованием *глобальных функций* **operator**.

Функция **operator** может быть вызвана так же, как и любая другая функция. Использование операции – лишь сокращенная форма вызова функции. Например, запись вида a=в-с; эквивалентна a=operator-(b,c).

Перегрузка бинарного оператора

Функция operator для перегрузки (доопределения) бинарных операторов может быть описана двумя способами:

как компонента-функция класса с одним аргументом;

как глобальная функция (функция, описанная вне класса) с двумя аргументами.

При перегрузке бинарного оператора # выражение **a#b** может быть представлено при первом способе как **a.operator#(b)** или как **operator #(a,b)** при втором способе перегрузки.

Рассмотрим простой пример переопределения операторов *, = , > и == по отношению к объекту, содержащему декартовы координаты точки на плоскости. В примере использован первый способ перегрузки.

```
#include "iostream.h"

class dek_koord
{   int x,y;     //  декартовы координаты точки
  public:
    dek_koord(){};
    dek_koord(int X,int Y): x(X),y(Y) {}
    dek_koord operator*(const dek_koord );
    dek_koord operator=(const dek_koord);
    dek_koord operator>(const dek_koord);
    void see();
```

Библиотека БГУИР

```
};
dek_koord dek_koord::operator*(const dek_koord a)
{ dek_koord tmp;            // перегрузка операции *
  tmp.x=x*a.x;
  tmp.y=y*a.y;
  return tmp;
}
dek_koord dek_koord::operator =(const dek_koord a)
{ x=a.x;                    // перегрузка операции =
  y=a.y;
  return *this;
}
dek_koord dek_koord::operator >(const dek_koord a)
{ if (x<a.x) x=a.x;         // перегрузка операции >
  if (y<a.y) y=a.y;
  return *this;
}
int dek_koord::operator ==(const dek_koord a)
{ if (x-a.x==0 && y-a.y==0) return 0;  // перегрузка операции ==
  if (x-a.x>0 && y-a.y>0) return 1;
  if (x-a.x<0 && y-a.y<0) return -1;
  else return 2;   // неопределенность
}
void dek_koord::see()
{ cout << "координата x = " << x << endl;
 cout << "координата y = " << y << endl;
}
void main()
{ dek_koord A(1,2), B(3,4), C;
  int i;
  A.see();
 B.see();
 C=A*B;
 C.see();
 C=A>B;    // компоненты объекта С принимают значение max от A и B
 C.see();
 i=A==B;
 cout << A==B << endl;        // ошибка
   // error  binary '<<' : no operator defined which takes a right-hand operand
   // of type 'class dek_koord' (or there is no acceptable conversion)
 cout << (A==B) << endl;      // верно
```

Библиотека БГУИР

```
}
```

В приведенной выше программе функцию перегрузки оператора * можно изменить, например, следующим образом

```
dek_koord &dek_koord::operator*(const dek_koord &a)
{x*=a.x;
  y*=a.y;
  return *this;
}
```

В этом примере функция operator в качестве параметра получает ссылку на объект, стоящий в правой части выражения A*B, то есть на B. Ссылка – это второе имя (псевдоним) для одного и того же объекта. Более подробно ссылки будут рассмотрены позже. Функция при вызове получает скрытый указатель на объект A и модифицирует неявные параметры (компоненты-данные объекта A – x и y). Возвращается значение по адресу this, то есть объект A.

Следует отметить, что если в описании класса dek_koord присутствуют объявления двух функций перегрузки операции *:

```
class dek_koord
{   . . .
    dek_koord operator*(const dek_koord );
    dek_koord &operator*(const dek_koord &);
     . . .
};
```

то возникает ошибка.

Если возвращаемое значение функции operator является ссылкой, то в этом случае возвращаемое значение не может быть автоматической или статической локальной переменной.

Ниже приведен пример еще одной программы перегрузки оператора ”-“ для использования его при вычитании из одной строки другой.

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

class String
{   char str[80];                 //  локальная компонента
  public:                         //  глобальные компоненты
    void init (char *s);          // функция инициализации
    int operator – (String s_new);  // прототип функции  operator
} my_string1, my_string2;   // описание двух объектов класса String

void String::init (char *s)      // ф-ция обеспечивает копирование строки-
                               // аргумента(s)  в строку-компоненту (str)
{ strcpy(str,s);}                // класса String

int String::operator – (String s_new)   // перегрузка оператора – (вычитания
                                      // строк)
{  for (int i=0; str[i]==s_new.str[i]; i++)
```

Библиотека БГУИР

```
    if (!str[i]) return 0;
    return str[i] – s_new.str[i];
}
void main(void)
{   char s1[51], s2[51];
    cout <<"Введите первую строку не более 80 символов:" <<endl;
    cin >>s1;
    cout<<" Введите  вторую строку не более 80 символов "<<endl;
    cin>>s2;
    my_string1.init(s1);     //инициализация объекта my_string1
    my_string2.init(s2);     //инициализация объекта my_string2
    cout <<"\nString1 –  String2  = ";   // вывод на экран разности двух строк
    cout << my_string1 –  my_string2 << endl;
}
```

Результат работы программы:

```
Введите первую строку не более 80 символов:
overload
Введите вторую строку не более 80 символов
function
String1 – String2 = 9
```

При перегрузке бинарного оператора с использованием компоненты-функции ей передается в качестве параметра только один аргумент. Второй аргумент получается посредством использования неявного указателя this на объект, компоненты которого модифицируются.

Перегрузка унарного оператора

При перегрузке унарной операции функция operator не имеет параметров. Как и в предыдущем случае, модифицируемый объект передается в функцию-operator неявным образом, используя указатель this.

Унарный оператор, как и бинарный, может быть перегружен двумя способами:

как компонента-функция без аргументов;

как глобальная функция с одним аргументом.

Как известно, унарный оператор может быть префиксным и постфиксным. Для любого префиксного унарного оператора выражение **#a** может быть представлено при первом способе как **a.operator#()**, а при втором как **#operator(a)**.

При перегрузке унарного оператора, используемого в постфиксной форме, выражение вида **a#** может быть представлено при первом способе как **a.operator#(int)** или как **operator#(a,int)** при втором способе. При этом аргумент типа int не существует и используется для отличия префиксной и постфиксной форм при перегрузке.

Ниже приведен пример программы перегрузки оператора ++ и реализа-

Библиотека БГУИР

ции множественного присваивания. Для перегрузки унарного оператора ++, предшествующего оператору i++, вызывается функция operator ++(). В случае если оператор ++ следует за операндом, то вызывается функция
operator++(int x), где x принимает значение 0.

```
#include "iostream.h"

    class dek_koord
    {     int x,y;      //  декартовы координаты точки
      public:
         dek_koord(){};
         dek_koord(int X,int Y): x(X),y(Y) {}
         void operator++();
         void operator++(int);
         dek_koord operator=(dek_koord);
         void see();
    };
void dek_koord::operator++()      // перегрузка операции ++A
    { x++;}
void dek_koord::operator++(int)  // перегрузка операции A++
    { y++;}
    dek_koord dek_koord::operator =(dek_koord a)
    { x=a.x;                              // перегрузка операции =
      y=a.y;
      return *this;
    }
    void dek_koord::see()
    { cout << "координата x = " << x << endl;
      cout << "координата y = " << y << endl;
    }
    void main()
    { dek_koord A(1,2), B, C;
      A.see();
      A++;                //  увеличение значения  компоненты x объекта A
      A.see();            //  просмотр содержимого объекта A
      ++A;                //  увеличение значения  компоненты y объекта A
      A.see();            //  просмотр содержимого объекта A
      C=B=A;              // множественное присваивание
      B.see();
      C.see();
    }
```

Результат работы программы:
координата x = 1

Библиотека БГУИР

координата y = 2
координата x = 1
координата y = 3
координата x = 2
координата y = 3
координата x = 2
координата y = 3
координата x = 2
координата y = 3

Дружественная функция operator

Функция operator может быть не только членом класса, но и friend-функцией этого класса. Как было отмечено ранее, friend-функции, не являясь компонентами класса, не имеют неявного указателя this. Следовательно, при перегрузке унарных операторов в функцию operator передается один, а бинарных – два аргумента. *Необходимо отметить, что операторы: =, (), [] и -> не могут быть перегружены с помощью friend-функции operator.*

```
#include "iostream.h"

class dek_koord
{    int x,y;      //  декартовы координаты точки
  public:
     dek_koord(){};
     dek_koord(int X,int Y): x(X),y(Y) {}
     friend dek_koord operator*(int,dek_koord);
     friend dek_koord operator*(dek_koord,int);
     dek_koord operator=(dek_koord);
     void see();
};
dek_koord operator*(int k,dek_koord dk)  // перегрузка операции A++
{ dk.x*=k;
  dk.y*=k;
  return dk;
}
dek_koord operator*(dek_koord dk,int k)
{ dk.x*=k;
  dk.y*=k;
  return dk;
}
dek_koord dek_koord::operator=(dek_koord dk)
{ x=dk.x;
  y=dk.y;
  return *this;
```

Библиотека БГУИР

```
}
void dek_koord::see()
{ cout << "координата т.(x,y) = " << x<<'  ' <<y<< endl;
}
void main()
{ dek_koord A(1,2), B;
  A.see();
  B=A*2;              //  увеличение значения объекта A в 2 раза
  B.see();            //  просмотр содержимого объекта B
}
```

Результат работы программы:
координата т.(x ,y)= 1 2
координата т.(x ,y)= 2 4

Особенности перегрузки операции присваивания

Доопределение оператора = позволяет решить проблему присваивания, но не решает задачи инициализации, так как при инициализации должен вызываться соответствующий конструктор. В рассматриваемом случае это решается использованием **конструктора копирования**. Общий вид конструктора копирования имеет следующий вид:

имя_класса (const имя_класса &);

Конструктор выполняет все необходимые действия при вызове функции и копировании содержимого объекта в стек (из стека). Вызов конструктора копирования осуществляется при обращении к функции и передаче в нее в качестве параметра объекта (объектов), а также возврата значения (объекта) из функции.

```
#include "iostream.h"
#include "string.h"
class string
{  char *str;              //
   int size;
 public:
   string(){};             // конструктор по умолчанию
 string(int n,char *s)   // конструктор с параметрами
  { str=new char[size=n>=(strlen(s)+1)? n : strlen(s)+1];
    strcpy(str,s);
  }
string(const string &); // конструктор копирования
  ~string(){};             // деструктор
  friend string operator+(string, const string);
  string &operator=(const string &);
  void see();
```

Библиотека БГУИР

```
  };
string::string(const string &a)           // описание конструктора копирования
  { str=new char[a.size+1];      // выделяем память под this->str   (+1 под ’\0’)
    strcpy(str,a.str);                // копирование строки
    size=strlen(str);
  }
 string operator+(string s1, const string s2)    // перегрузка операции +
  { string ss;
    ss.str=new char[ss.size=strlen(s1.str)+strlen(s2.str)+1];
    for(int i=0; ss.str[i]=s1.str[i]; i++);           // перезапись  символа ’\0’
    ss.str[i]=' ';                                        //  удаление  ’\0’
    for(int j=0; ss.str[i+1]=s2.str[j]; i++,j++); // дозапись второй строки
    return ss;
  }
string &string::operator =(const string &st)  // перегрузка операции =
  { if(this!=&st)               // проверка, не копирование ли объекта в себя
   { delete  str;               // освобождаем память  старой строки
     str=new char[size=st.size];// выделяем память под новую строку
     strcpy(str,st.str);
   }
   return *this;
  }
  void string::see()
  { cout << this->str << endl;
  }
  void main()
  { string s1(10,"язык"), s2(30,"программирования"), s3(30,"   ");
    s1.see();
    s2.see();
    string s4=s1;       // это только вызов конструктора копирования
    s4.see();
    s1+s2;              // перегрузка + (вызов функции operator +)
    s1.see();
    s3=s2;              // перегрузка = (вызов функции operator =)
    s3.see();
    s3=s1+s2;            //перегрузка операции + , затем операции =
    s3.see();
  }
```

Результаты работы программы:
язык
программирования

Библиотека БГУИР

язык
язык
программирования
язык программирования

Инструкция string s4=s1 только вызывает конструктор копирования для объекта s4. Выполнение инструкции s1+s2 приводит к двум вызовам конструктора копирования (вначале для копирования в стек объекта s2, затем s1). После этого выполняется вызов функции operator+ для инструкции s1+s2. При выходе из функции (инструкция return ss) вновь выполняется вызов конструктора копирования. При выполнении инструкции s3=s2 конструктор копирования для копирования объекта s2 в стек не вызывался, так как параметр в operator= передан (и возвращен) по ссылке.

Если для класса конструктор копирования явно не описан, то компилятор сгенерирует его. При этом значения одной компоненты-данного будут скопированы в компоненту-данное другого объекта. Это допустимо для объектов простых классов и недопустимо для объектов, имеющих динамические компоненты-данные (конструируются с использованием операторов динамического выделения памяти).

***Перегрузка оператора* []**

Как было отмечено выше, функция operator может быть с успехом использована для доопределения операторов C++ (в основном арифметические, логические и операторы отношения). В то же время в C++ существуют некоторые операторы, не входящие в число перечисленных, но которые полезно перегружать. К ним относится оператор []. Его необходимо перегружать с помощью **компоненты-функции**, использование **friend-**функции **запрещено**. Общая форма функции operator[]() имеет вид:

Тип_возвр_значения имя_класса::operator [](int i)
{ тело функции}

Параметр функции необязательно должен иметь тип int , но он использован, так как operator[] в основном применяется для индексации. Рассмотрим пример программы, с использованием перегрузки операции []:

```
#include "iostream.h"
class massiv
{    float f[3];
  public:
     massiv(float i,float j,float k){f[0]=i; f[1]=j; f[2]=k;}
     float operator[](int i)
    { return f[i];} // перегрузка оператора []
};
void main()
{ massiv ff(1,2,3);
```

Библиотека БГУИР

```
    double f;
    int i;
    cout << "введите номер индекса  ";
    cin >> i;
    cout <<"f["<< i <<" ]= " << ff[i] << endl;
}
```

В примере перегруженная функция operator[]() возвращает величину элемента массива, индекс которого передан в функцию в качестве параметра. Данная программа при небольшой модификации может позволить использовать оператор[] как справа, так и слева от оператора присваивания. Для этого необходимо, чтобы функция operator[]() возвращала не элемент, а ссылку на него.

```
#include "iostream.h"
class massiv
{   float f[3];
  public:
    massiv(float i,float j,float k){f[0]=i; f[1]=j; f[2]=k;}
    float &operator[](int i)    // перегрузка оператора []
    { if(i<0 || i>2)                // проверка на выход за границы массива
      { cout << "Выход за пределы массива"<<endl;
        exit(1);
      }
      return f[i];
    }
};
void main()
{ massiv ff(1,2,3);
  double f;
  int i;
  cout << "введите номер индекса  ";
  cin >> i;
  cout <<"f["<< i <<" ]= " << ff[i] << endl;
  ff[i]=5;    // если функция operator не возвращает ссылку,то компилятор
              // выдает ошибку      =' : left operand must be l-value
  cout <<"f["<< i <<" ]= " << ff[i] << endl;
}
```

Приведем еще один пример программы, использующей перегрузку operator[].

```
// перегрузка функции operator[] на примере вычисления n!
#include "iostream.h"
#include "values.h"           // для определения константы MAXLONG
class fact
{   long l;
```

```
  public:
     long operator[](int);        // перегрузка оператора []
};

long fact::operator[](int n)
{ long l;
  for (int i=0; i<=n; i++)      //выбор буквы из уменьшаемой строки
  if(l>MAXLONG/i)
    cerr<<"ОШИБКА факториал числа "<<n<<" больше "<<MAXLONG;
  else l*=i;
  return l;
}

void main()
{ fact f;
  int i,k;
  cout << "введите число k для нахождения факториала"
  cin >> k;
  for (i=1; i<=k; i++)
  cout << i <<"! = " << f[i] << endl;
}
```

***Перегрузка оператора* ()**

Оператор вызова функции можно доопределить, как и любой другой оператор. Как и в предыдущем случае, оператор () необходимо перегружать только с помощью **компоненты-функции**, использование **friend-**функции **запрещено**. Общая форма функции operator()() имеет вид:

тип_возвр_значения имя_класса::operator ()(список_аргументов)
{ тело функции}

```
#include "iostream.h"

class matr
{       int **m,a,b;
  public:
        matr(int,int);
        ~matr();
        int operator()(int,int);   // перегрузка оператора ()
        int operator()(int);       // перегрузка оператора ()
};

matr::matr(int i,int j): a(i),b(j)  // конструктор
{  i=0;
   m=new int *[a];
  for(int k=0; k<a; k++)
  {  *(m+k)=new int[b];
```

Библиотека БГУИР

```
        for(int n=0; n<b; n++)
        *(*(m+k)+n)=i++;          // заполнение m числами 0, 1, 2, 3, …, a*b
     }
}
matr::~matr()                     // деструктор
{ for(int k=0; k<a; k++)
  delete [] m[k];                 // освобождение памяти для k-й строки
  delete [] m;                    // освобождение памяти для всего массива
}                                 // указателей m
int matr::operator()(int i,int j)
{ if (i<0 || i>=a || j<0 || j>=b)
    { cerr<<"выход за пределы матрицы ";
      return m[0][0];      // например, при этом возврат m[0][0]
    }
  return m[i][j];            // возврат требуемого элемента
}
int matr::operator()(int i)
{ if (i<0 || i>=a*b)
    { cerr<<"выход за пределы массива ";
      return **m;         // как и выше возврат m[0][0]
    }
  return m[i/b][i%b];  // возврат требуемого элемента
}
void main()
{ matr mt(3,5);
  cout << mt(2,3) << endl;
  cout << mt(3,2) << endl;    // попытка получить элемент из 3-й строки
  cout << mt(3) << endl;
}
```

Результаты работы программы:

```
13
выход за пределы массива  0
6
```

Конструктор класса matr динамически выделяет и инициализирует память двухмерного массива. Деструктор разрушает массив автоматически при завершении программы. В классе matr реализованы две функции operator(): первая получает два аргумента (индексы в матрице), вторая получает один аргумент (порядковый номер элемента в матрице). Обе функции возвращают либо требуемый элемент, либо элемент m[0][0] при попытке выхода за пределы матрицы.

Библиотека БГУИР

Перегрузка оператора ->

Оператор -> доступа к компонентам объекта через указатель на него определяется как унарный постфиксный.

Ниже приведен простой пример программы перегрузки оператора ->:

```
#include "iostream.h"
#include "iomanip.h"
#include "string.h"

class cls_A
{   char a[40];
 public:
    int b;
    cls_A(char *aa,int bb): b(bb)                   // конструктор
    {strcpy(a,aa);}
    char *put_A() {return a;}
};

class cls_B
{   cls_A *p;            // указатель на класс cls_A
 public:
    cls_B(char *aa,int bb) {p=new cls_A(aa,bb);}  // конструктор
    ~cls_B() {delete  p;}                          // деструктор
    cls_A *operator->(){return p;}   // функция перегрузки   ->
};

void main()
{ cls_B ptr("перегрузка оператора -> ",2);              // объект класса cls_B
  cout << ptr->put_A() << setw(6) << ptr->b <<endl;   // перегрузка ->
  cout << (ptr.operator->())->put_A() << setw(6)
           << (ptr.operator->())->b <<endl;
  cout << (*ptr.operator->()).put_A() << setw(6)
           << (*ptr.operator->()).b <<endl;
}
```

Результат работы программы:

```
перегрузка оператора ->     2
перегрузка оператора ->     2
перегрузка оператора ->     2
```

В приведенной программе инструкции ptr->put_A() и ptr->b приводят к перегрузке операции ->, то есть позволяют получить адрес указателя на компоненты класса cls_A для (из) объекта ptr. Таким образом, инструкция ptr->b соответствует инструкции (ptr.p)->b. Следующие далее две группы инструкций также верны, но не являются примером перегрузки оператора ->, а только приводят к явному вызову функции operator-> – компоненты класса cls_B.

В целом доопределение оператора -> позволяет использовать ptr, с одной

Библиотека БГУИР

стороны, как специальный указатель (в примере для класса cls_A), а с другой стороны, как объект (для класса cls_B).

Специальные указатели могут быть доопредены следующим образом:

cls_A &operator*(){return *p;}

cls_A &operator[](int index){return p[index];}

а также доопределение может быть выполнено по отношению к большинству рассмотренных ранее операций (+, ++ и др.).

Перегрузка операторов new и delete

В C++ имеются две возможности перегрузки операторов new и delete – *локально* (в пределах класса) и *глобально* (в пределах программы). Эти операторы имеют правила переопределения, отличные от рассмотренных выше правил переопределения других операторов. Одна из причин перегрузки операторов new и delete состоит в том, чтобы придать им новые свойства, например, выдачи диагностики или более высокой защищенности от ошибок. Кроме того, может быть реализована более эффективная схема распределения памяти по сравнению со схемой, обеспечиваемой системой.

Оператор **new** можно задать в следующих формах:

<::> new <аргументы> имя_типа <инициализирующее_выражение>
<::> new <аргументы> имя_типа [ ]

Параметр "аргументы" можно использовать либо для того, чтобы различить разные версии глобальных операторов new, либо для использования их в теле функции operator. Доопределенную функцию operator new можно объявить:

void *operator new(size_t t<список _аргументов>);
void *operator new[](size_t t<список _аргументов>);

Вторая форма используется для выделения памяти для массивов. Возвращаемое значение всегда должно иметь тип **void ***. Единственный обязательный аргумент функции operator всегда должен иметь тип **size_t**. При этом в функцию operator автоматически подставляется аргумент sizeof(t).

Ниже приведен пример программы, в которой использованы две глобальные перегруженные и одна локальная функции operator.

```
#include "iostream.h"
#include "string.h"

void *operator new(size_t tip,int kol)       //глобальная ф-ция operator new
{ cout << "глобальная функция 1" <<endl;  // с одним параметром
  return new char[tip*kol];
}
void *operator new(size_t tip,int n1,int n2)  //глобальная ф-ция operator new
{ cout << "глобальная функция 2" <<endl; // с двумя параметрами
  void *p=new char[tip*n1*n2];
  return p;
```

Библиотека БГУИР

```
    }
class cls
    {  char a[40];
     public:
      cls(char *aa){ strcpy(a,aa); }
      ~cls(){}
      void *operator new(size_t,int);
    };
    void *cls::operator new(size_t tp,int n)        // локальная функция operator
    { cout << "локальная  функция " <<endl;
      return new char[tp*n]; }
    void main()
    { cls obj("перегрузка оператора   new");
      float *ptr1;
      ptr1=new (5) float;              // вызов 1 глобальной функции operator new
      ptr1=new (2,3) float;            // вызов 2 глобальной функции operator new
      ptr1=new float;                  // вызов системной глобальной функции
      cls *ptr2=new (3) cls("aa");  // вызов локальной функции operator new
    }                                  // используя cls::cls("aa")
```

Результаты работы программы:
глобальная функция 1
глобальная функция 2
локальная функция

Первое обращение ptr1=new (5) float приводит к вызову глобальной функции operator с одним параметром, в результате выделяется память 5*sizeof(float) байт (это соответствует массиву из 5 элементов типа float) и адрес заносится в указатель ptr1. Второе обращение приводит к вызову функции operator с двумя параметрами. Следующая инструкция new float приводит к вызову системной функции new. Инструкция new (3) cls("aa") соответствует вызову функции operator, описанной в классе cls. В функцию в качестве имени типа передается тип созданного объекта класса cls. Таким образом, ptr2 получает адрес массива из 3 объектов класса cls.

Оператор **delete** разрешается доопределять только по *отношению к классу*. В то же время можно заменить системную версию реализации оператора delete на свою.

Доопределенную функцию operator delete можно объявить:

void *operator delete(void *p<,size_t t>);
void *operator delete[](void *p<,size_t t>);

Функция operator должна возвращать значение void и имеет один обязательный аргумент типа void * – указатель на память, которая должна быть освобождена. Ниже приведен пример программы с доопределением оператора

delete.

```
#include "iostream.h"
#include "string.h"
#include "stdlib.h"

void *operator new(size_t tip,int kol)        // глобальная функция operator
{ cout << "глобальная функция NEW" <<endl;
  return new char[tip*kol];
}

class cls
{   char a[40];

public:
    cls(char *aa)
    {   cout<<"работает конструктор"<<endl;
        strcpy(a,aa);
    }
    ~cls(){}
    void *operator new(size_t,int);
    void operator delete(void *);
};

void *cls::operator new(size_t tip,int n)      // локальная функция operator
{ cout << "локальная  функция " <<endl;
  return new char[tip*n];
}

void cls::operator delete(void *p)  // локальная функция operator
{ cout << "локальная функция DELETE" <<endl;
  delete p;            // вызов системной функции delete
}

void operator deletex[](void *p)       // глобальная функция operator
{ cout << "глобальная функция DELETE" <<endl;
  delete p;            // вызов системной функции delete
}

void main()
{ cls obj("перегрузка операторов NEW и DELETE");
  float *ptr1;
  ptr1=new (5) float;       // вызов глобальной ф-ции  доопр. оператора new
  delete [] ptr1;            // вызов глобальной ф-ции  доопр.оператора delete
  cls *ptr2=new (10) cls("aa");  // вызов локальной функции доопределения
                                 // оператора new (из класса cls)
  delete ptr2;                   // вызов локальной функции доопределения
                                 // оператора delete (из класса cls)
```

Библиотека БГУИР

```
}
```

Результаты работы программы:
глобальная функция NEW
работает конструктор
глобальная функция DELETE
локальная функция NEW
локальная функция DELETE

Инструкция cls *ptr2=new (10) cls("aa") выполняется следующим образом: вначале вызывается локальная функция operator для выделения памяти, равной 10*sizeof(cls), затем вызывается конструктор класса cls.

Необходимо отметить тот факт, что при реализации переопределения глобальной функции в ней не должен использоваться оператор delete [], так как это приведет к бесконечной рекурсии. При выполнении инструкции системный оператор delete ptr2 сначала вызывается локальная функция доопределения оператора delete для класса cls, а затем из нее глобальная функция переопределения delete.

Далее рассмотрим пример доопределения функций new и delete в одном из классов, содержащемся в некоторой иерархии классов.

```
#include "iostream.h"

class A
{ public:
    A(){}
    virtual ~A(){}
    void *operator new(size_t,int);
    void operator delete(void *,size_t);
};

class B : public A
{ public:
    B(){}
   ~B(){}
};

void *A::operator new(size_t tip,int n)
{ cout << "перегрузка  operator NEW" <<endl;
  return new char[tip*n];
}

void A::operator delete(void *p,size_t t)  //глобальная функция operator
{ cout << " перегрузка operator DELETE" <<endl;
  delete p;            // вызов глобальной (системной функции)
}

void main()
```

Библиотека БГУИР

```
{ A *ptr1=new(2) A; // вызов локальной функции, используя A::A()
  delete ptr1;
  A *ptr2=new(3) B; // вызов локальной функции, используя B::B()
  delete ptr2;
}
```

Результаты работы программы:
перегрузка operator NEW
перегрузка operator DELETE
перегрузка operator NEW
перегрузка operator DELETE

Преобразование типа

Выражения, содержащиеся в функциях и используемые для вычисления некоторого значения, записываются в большинстве случаев с учетом корректности по отношению к объектам этого выражения. В то же время, если используемая операция для типов величин, участвующих в выражении, явно не определена, то компилятор пытается выполнить такое *преобразование типов* в два этапа.

На первом этапе выполняется попытка использовать стандартные преобразования типов. Если это невозможно, то компилятор использует преобразования, определенные пользователем.

Явные преобразования типов

Если перед выражением указать имя типа в круглых скобках, то значение выражения будет преобразовано к данному типу.

```
int a = (int) b;
char *s=(char *) addr;
```

Недостатком их является полное отсутствие контроля, что приводит к ошибкам и путанице. Этого можно избежать в C++. Для преобразования с минимальным контролем можно использовать операцию staic_cast.

static_cast<тип> (выражение)

Она позволяет выполнять преобразования, не проверяя типы выражений во время выполнения, а основываясь на сведениях, полученных при компиляции. Операция static_cast позволяет выполнять преобразования не только указателя на базовый класс к указателю на производный, но и наоборот.

В то же время преобразование int к int* приведет к ошибке компиляции. Для преобразований не связанных между собой типов используется reinterpret_cast.

```
int  i;
void *addr= reinterpret_cast<void *> i;
```

Если же нужно выполнить преобразование неизменяемого типа к изменяемому, то можно использовать const_cast:

```
const char *s
```

Библиотека БГУИР

```
char ss=const_cast<char*> s;
```

Преобразования типов, определенных в программе

Конструктор с одним аргументом можно явно не вызывать.

```
#include "iostream.h"
    class my_class
    {     double x,y;    //
      public:
         my_class(double X){x=X; y=x/3;}
         double summa();
     };
    double my_class::summa() { return x+y; }
    void main()
    { double d;
      my_class my_obj1(15),     // создание объекта obj1 и инициализация его
      my_obj2=my_class(15),     // создание объекта obj2 и инициализация его
      my_obj3=15;               // создание объекта obj3 и инициализация его
      d=my_obj1;  // error no operator defined which takes a right-hand operand of
                  // type 'class my_class' (or there is no acceptable conversion)
      cout << my_obj1.summa() << endl;
      cout << my_obj2.summa() << endl;
      cout << my_obj3.summa() << endl;
    }
```

В рассматриваемом примере все три создаваемых объекта будут инициализированы числом 15 (первые два явным, третий неявным вызовом конструктора). Следовательно, значение базовой переменной (определенной в языке) может быть присвоено переменной типа, определенного пользователем.

Для выполнения обратных преобразований, то есть от переменных, имеющих тип, определенный пользователем к базовому типу, можно задать преобразования с помощью соответствующей функции operator(), например:

```
    class my_class
    {   double x,y;
     public:
        operator double() {return x;}
        my_class(double X){x=X; y=x/3;}
        double summa();
    };
```

Теперь в выражении d=my_obj1 не будет ошибки, так как мы задали прямое преобразование типа. При выполнении этой инструкции активизируется функция operator, преобразующая значение объекта к типу double и возвращающая значение компоненты объекта . Наряду с прямым преобразованием в C++ имеется подразумеваемое преобразование типа:

```
#include "iostream.h"
```

Библиотека БГУИР

```
class my_cl1
{     double x;                    //
    public:
       operator double(){return x;}
  my_cl1(double X) : x(X) {}
};

class my_cl2
{     double x;                    //
   public:
       operator double(){return x;}
       my_cl2(my_cl1 XX):  x(XX) {}
};

void fun(my_cl2 YY)
{ cout << YY <<endl; }

void main()
{ fun(12);                         // ERROR cannot convert parameter 1
                                  //  from 'const int' to 'class my_cl2'
   fun(my_cl2(12));              // подразумеваемое преобразование типа
}
```

В этом случае для инструкции fun(my_cl2(12)) выполняется следующее:

активизируется конструктор класса my_cl1 (x инициализируется значением 12);

при выполнении в конструкторе my_cl2 инструкции x(XX) вызывается функция operator (класса my_cl1), возвращающая значение переменной x, преобразованное к типу double;

далее при выполнении инструкции cout << YY вызывается функция operator() класса my_cl2, выполняющая преобразование значения объекта YY к типу double.

Разрешается выполнять только одноуровневые подразумеваемые преобразования. В приведенном выше примере инструкция fun(12) соответствует двухуровневому неявному преобразованию, где первый уровень – my_cl1(12) и второй – my_cl2(my_cl1(12))

В заключение отметим основные правила доопределения операторов:

– все операторы языка C++ за исключением . * :: ?: sizeof и символов # ## можно доопределять;

– при вызове функции operator используется механизм перегрузки функций;

– количество операндов, которыми оперируют операторы (унарные, бинарные), и приоритет операций сохраняются и для доопределенных операторов.

Шаблоны

Библиотека БГУИР

Параметризированные классы

Параметризированный класс – некоторый шаблон, на основе которого можно строить другие классы. Этот класс можно рассматривать как некоторое описание множества классов, отличающихся только типами их данных. В С++ используется ключевое слово template для обеспечения параметрического полиморфизма. Параметрический полиморфизм позволяет использовать один и тот же код относительно различных типов (параметров тела кода). Это наиболее полезно при определении контейнерных классов. Шаблоны определения класса и шаблоны определения функции позволяют многократно использовать код, корректно по отношению к различным типам, позволяя компилятору автоматизировать процесс реализации типа.

Шаблон класса определяет правила построения каждого отдельного класса из некоторого множества разрешенных классов.

Спецификация шаблона класса имеет вид:

template <список параметров>
class объявление класса

Список параметров класса-шаблона представляет собой идентификатор типа, подставляемого в объявление данного класса при его генерации. Рассмотрим пример шаблона класса работы с динамическим массивом и выполнением контроля за значениями индекса при обращении к его элементам.

```
#include "iostream.h"
#include "string.h"
template <class T>

class vector
{      T *ms;
       int size;
  public:
       vector() : size(0),ms(NULL) {}
       ~vector(){delete [] ms;}

   void inkrem(const T &t)  // увеличение размера массива на 1 элемент
      {  T *tmp = ms;
         ms=new T[size+1];      // ms – указатель на новый массив
         if(tmp) memcpy(ms,tmp,sizeof(T)*size); // перезапись tmp -> ms
         ms[size++]=t;              // добавление нового элемента
         if(tmp) delete [] tmp;     // удаление временного массива
      }

   void decrem(void)           // уменьшение размера массива на 1 элемент
      {  T *tmp = ms;
          if(size>1) ms=new T[--size];
          if(tmp)
          { memcpy(ms,tmp,sizeof(T)*size); // перезапись без посл.элемента
```

Библиотека БГУИР

```
            delete [] tmp;          // удаление временного массива
          }
        }
        T &operator[](int ind)  // определение обычного метода
        {  // if(ind<0 || (ind>=size)) throw IndexOutOfRange; // возбуждение
                                  // исключительной ситуации IndexOutOfRange
          return ms[ind];
        }
};
void main()
{  vector <int> VectInt;
   vector <double> VectDouble;
   VectInt. inkrem(3);
   VectInt. Inkrem(26);
   VectInt. inkrem(12);        // получен int-вектор из 3 атрибутов
   VectDouble. inkrem(1.2);
   VectDouble. inkrem(.26);//получен double-вектор из 2 атрибутов
   int a=VectInt[1];           // a = ms[1]
   cout << a << endl;
   int b=VectInt[4];           // будет возбуждена исключительная ситуация
   cout << b << endl;
   double d=VectDouble[0];
   cout << d << endl;
   VectInt[0]=1;
   VectDouble[1]=2.41;
}
```

Класс vector наряду с конструктором и деструктором имеет 2 функции: decrem – добавление в конец вектора нового элемента, inkrem – уменьшение числа элементов на единицу и операция [] обращения к i-му элементу вектора.

Параметр шаблона vector – любой тип, у которого определены операция присваивания и операция new. Например, при задании объекта типа vector <int> происходит генерация конкретного класса из шаблона и конструирование соответствующего объекта VectInt, при этом тип T получает значение типа int. Генерация конкретного класса означает, что генерируются все его компоненты-функции, что может привести к существенному увеличению кода программы.

Выполнение функций

```
VectInt.decrem(3);
VectInt.decrem(26);
VectInt.decrem(12);
```

приведет к созданию вектора (массива) из трех атрибутов (3, 26 и 12).

Сгенерировать конкретный класс из шаблона можно, явно записав:

template vector<int>;

При этом не будет создано никаких объектов типа vector<int>, но будет

сгенерирован класс со всеми его компонентами.

В некоторых случаях желательно описания некоторых компонент-функций шаблона класса выполнить вне тела шаблона, например:

```
#include "iostream.h"

template <class T1,class T2>
T1 sm1(T1 aa,T2 bb)                    //  описание шаблона глобальной
{ return (T1)(aa+bb);                  //  функции суммирования значений
}                                      //  двух аргументов

template <class T1,class T2>
class cls
{   T1  a;
    T2  b;
 public:
    cls(T1 A,T2 B) : a(A),b(B) {}
    ~cls(){}
     T1 sm1()                    // описание шаблона функции
     { return (T1)(a+b);         // суммирования компонент объекта obj_
     }
    T1 sm2(T1,T2);               // объявление шаблона функции
};

template <class T1,class T2>
T1 cls<T1,T2>::sm2(T1 aa,T2 bb)        // описание шаблона функции
{ return (T1)(aa+bb);                  // суммирования внешних данных
}

void main()
{ cls <int,int> obj1(3,4);
  cls <double,double> obj2(.3,.4);
  cout<<"функция суммирования компонент объекта 1        = "
       <<obj1.sm1()<<endl;
  cout<<"функция суммирования внешних данных (int,int)  = "
       <<obj1.sm2(4,6)<<endl;
  cout<<"вызов глобальной функции суммирования (int,int)    = "
       <<sm1(4,.6)<<endl;
  cout<<"функция суммирования компонент объекта 2          = "
       <<obj2.sm1()<<endl;
  cout<<"функция суммирования внешних данных (double,double)= "
       <<obj2.sm2(4.2,.1)<<endl;
}
```

Передача в шаблон класса дополнительных параметров

При создании экземпляра класса из шаблона в него могут быть переданы не только типы, но и переменные и константные выражения:

Библиотека БГУИР

```
#include "iostream.h"

template <class T1,int i=0,class T2>
class cls
{   T1  a;
    T2  b;
 public:
    cls(T1 A,T2 B) : a(A),b(B){}
    ~cls(){}
    T1 sm() //описание шаблона ф-ции суммирования компонент объекта
    { //  i+=3;   // error member function 'int __thiscall cls<int,2>::sm(void)'
      return (T1)(a+b+i);
    }
};

void main()
{ cls <int,1,int> obj1(3,2);        //  в шаблоне const i инициализируется  1
  cls <int,0,int> obj2(3,2,1);      //  error 'cls<int,0>::cls<int,0>':no overloaded
                                    //  function takes 3 parameter s
  cls <int,int,int> obj13(3,2,1); //  error 'cls' : invalid template argument for 'i',
                                    //  constant expression expected
  cout<<obj1.sm()<<endl;
}
```

Результатом работы программы будет выведенное на экран число 6.

В этой программе инструкция **template <class T1,int i=0,class T2>** говорит о том, что шаблон класса cls имеет три параметра, два из которых – имена типов (T1 и T2), а третий (int i=0) – целочисленная константа. Значение константы i может быть изменено при описании объекта cls <int,**1**,int> obj1(3,2).

Шаблоны функций

В C++, так же как и для класса, для функции (глобальной, то есть не являющейся компонентой-функцией) может быть описан шаблон. Это позволит снять достаточно жесткие ограничения, накладываемые механизмом формальных и фактических параметров при вызове функции. Рассмотрим это на примере функции, вычисляющей сумму нескольких аргументов.

```
#include "iostream.h"
#include "string.h"

template <class T1,class T2>
T1 sm(T1 a,T2 b)                  // описание шаблона
{ return (T1)(a+b);               // функции с 2 параметрами
}

template <class T1,class T2,class T3>
T1 sm(T1 a,T2 b,T3 c)           // описание шаблона функции
```

Библиотека БГУИР

```
{ return (T1)(a+b+c);           // функции с 3 параметрами
}
void main()
{cout<<"вызов ф-ции суммирования sm(int,int)       =   "<<sm(4,6)<<endl;
  cout<<"вызов ф-ции суммирования sm(int,int,int)   = "<<sm(4,6,1)<<endl;
  cout<<"вызов ф-ции суммирования sm(int,double)  = "<<sm(5,3)<<endl;
  cout<<"вызов ф-ции суммирования sm(double,int,short)= " <<
   sm(.4,6,(short)1)<<endl;
// cout<<sm("я изучаю","язык C++")<<endl;  error cannot add two pointers
}
```

В программе описана перегруженная функция sm(), первый экземпляр которой имеет 2, а второй 3 параметра. При этом тип формальных параметров функции определяется при вызове функции типом ее фактических параметров. Используемые типы T1, T2, T3 заданы как параметры для функции с помощью выражения template <class T1,class T2,class T3>. Это выражение предполагает использование типов T1, T2 и T3 в виде ее дополнительных параметров. Результат работы программы будет иметь вид:

```
вызов функции суммирования sm(int,int)           =  10
вызов функции суммирования sm(int,int,int)       =  11
вызов функции суммирования sm(int,double)        =   8
вызов функции суммирования sm(double,int,short)=  7.4
```

В случае попытки передачи в функцию sm() двух строк, то есть типов, для которых не определена данная операция, компилятор выдаст ошибку. Чтобы избежать этого, можно ограничить использование шаблона функции sm(), описав явным образом функцию sm() для некоторых конкретных типов данных. В нашем случае:

```
char *sm(char *a,char *b)            //  явное описание функции объединения
{ char *tmp=a;                       //  двух строк
  a=new char[strlen(a)+strlen(b)+1];
  strcpy(a,tmp);
  strcat(a,b);
  return a;
}
```

Добавление в main() инструкции, например,

```
cout<<sm("я изучаю"," язык C++")<<endl;
```

приведет к выводу, кроме указанных выше, сообщения:

я изучаю язык C++

Следует отметить, что шаблон функции не является ее экземпляром. Только при обращении к функции с аргументами конкретного типа происходит генерация конкретной функции.

Совместное использование шаблонов и наследования

Шаблонные классы, как и обычные, могут использоваться повторно.

Библиотека БГУИР

Шаблоны и наследование представляют собой механизмы повторного использования кода и могут включать полиморфизм. Шаблоны и наследования связаны между собой следующим образом:

– шаблон класса может быть порожден от обычного класса;

– шаблонный класс может быть производным от шаблонного класса;

– обычный класс может быть производным от шаблона класса.

Ниже приведен пример простой программы, демонстрирующей наследование шаблонного класса oper от шаблонного класса vect.

```
#include "iostream.h"
template <class T>
class vect                        // класс-вектор
{protected:
     T *ms;                       // массив-вектор
      int size;                   // размерность массива-вектора
 public:
      vect(int n) : size(n)               // конструктор
      { ms=new T[size];}
      ~vect(){delete [] ms;}              // деструктор
      T &operator[](const int ind)     // доопределение операции []
      { if((ind>0) && (ind<size)) return ms[ind];
         else return ms[0];
      }
};
template <class T>
class oper : public vect<T>       // класс операций над вектором
{ public:
      oper(int n): vect<T>(n) {}    // конструктор
      ~oper(){}                     // деструктор
      void print()                  // функция вывода содержимого вектора
      { for(int i=0;i<size;i++)
         cout<<ms[i]<<' ';
         cout<<endl;
      }
};
void main()
{ oper <int> v_i(4);                //   int-вектор
  oper <double> v_d(4);             //  double-вектор
  v_i[0]=5;     v_i[1]=3;     v_i[2]=2;    v_i[3]=4;     // инициализация int
  v_d[0]=1.3;  v_d[1]=5.1;  v_d[2]=.5;  v_d[3]=3.5;  // инициализация double
  cout<<"int вектор = ";
  v_i.print();
  cout<<"double вектор = ";
```

Библиотека БГУИР

```
  v_d.print();
}
```

Как следует из примера, реализация производного класса от класса-шаблона в основном ничем не отличается от обычного наследования.

Некоторые примеры использования шаблона класса

При использовании в программе указателя на объект, память для которого выделена с помощью оператора new, в случае если объект становится не нужен, то для его разрушения необходимо явно вызвать оператор delete. В то же время на один объект могут ссылаться множество указателей и, следовательно, нельзя однозначно сказать, нужен ли еще этот объект или он уже может быть уничтожен. Рассмотрим пример реализации класса, для которого происходит уничтожение объектов, в случае если уничтожается последняя ссылка на него. Это достигается тем, что наряду с указателем на объект хранится счетчик числа других указателей на этот же объект. Объект может быть уничтожен только в том случае, если счетчик ссылок станет равным нулю.

```
#include "iostream.h"
#include "string.h"

template <class T>
struct Status          // состояние указателя
{ T *RealPtr;        // указатель
  int Count;          // счетчик числа ссылок на указатель
};

template <class T>
class Point                     // класс-указатель
{   Status<T> *StatPtr;
 public:
   Point(T *ptr=0);          // конструктор
   Point(const Point &);    // копирующий конструктор
   ~Point();
   Point &operator=(const Point &);     // перегрузка
 //  Point &operator=(T *ptr);            // перегрузка
   T *operator->() const;
   T &operator*() const;
};
```

Приведенный ниже конструктор Point инициализирует объект указателем. Если указатель равен NULL, то указатель на структуру Status, содержащую указатель на объект и счетчик других указателей, устанавливается в NULL. В противном случае создается структура Status

```
template <class T>
Point<T>::Point(T *ptr)      // описание конструктора
{ if(!ptr) StatPtr=NULL;
```

Библиотека БГУИР

```
  else
  { StatPtr=new Status<T>;
    StatPtr->RealPtr=ptr;
    StatPtr->Count=1;
  }
}
```

Копирующий конструктор StatPtr не выполняет задачу копирования исходного объекта, а так как новый указатель ссылается на тот же объект, то мы лишь увеличиваем число ссылок на объект.

```
template <class T>         // описание конструктора копирования
Point<T>::Point(const Point &p):StatPtr(p.StatPtr)
{ if(StatPtr) StatPtr->Count++;    // увеличено число ссылок
}
```

Деструктор уменьшает число ссылок на объект на 1, и при достижении значения 0 объект уничтожается

```
template <class T>
Point<T>::~Point()          // описание деструктора
{ if(StatPtr)
  { StatPtr->Count--;                 // уменьшается число ссылок на объект
    if(StatPtr->Count<=0)            // если число ссылок на объект <=0,
    { delete StatPtr->RealPtr;        // то уничтожается объект
      delete StatPtr;
    }
  }
}

template <class T>
T *Point<T>::operator->() const
{ if(StatPtr) return StatPtr->RealPtr;
  else return NULL;
}

template <class T>
T &Point<T>::operator*() const           // доступ к StatPtr осуществляется
{ if(StatPtr) return *StatPtr->RealPtr;    // посредством this-указателя
  else throw bad_pointer;                 // исключительная ситуация
}
```

При выполнении присваивания вначале необходимо указателю слева от знака = отсоединиться от ”своего” объекта и присоединиться к объекту, на который указывает указатель справа от знака =.

```
template <class T>
Point<T> &Point<T>::operator=(const Point &p)
{                            // отсоединение объекта справа от = от указателя
 if(StatPtr)
```

Библиотека БГУИР

```
  { StatPtr->Count--;
    if(StatPtr->Count<=0)            // так же, как и в деструкторе
     { delete StatPtr->RealPtr;       // освобождение выделенной под объект
       delete StatPtr;                // динамической памяти
     }
  }
                              // присоединение к новому указателю
  StatPtr=p.StatPtr;
  if(StatPtr) StatPtr->Count++;
  return *this;
}
struct Str
{  int a;
   char c;
};
void main()
{ Point<Str> pt1(new Str); // генерация класса Point, конструирование
                           // объекта pt1, инициализируемого указателем
                           // на стр-ру Str, далее с объектом можно обра-
                           // щаться как с указателем
  Point<Str> pt2=pt1,pt3;  // для pt2 вызывается конструктор копирования,
                           // затем создается указатель pt3
  pt3=pt1;                 // pt3 переназначается на объект указателя pt1
  (*pt1).a=12;             // operator*() получает this указатель на pt1
  (*pt1).c='b';
  int X=pt1->a;            // operator->() получает this-указатель на pt1
  char C=pt1->c;
}
```

Задание свойств класса

Если в функцию сортировки числовой информации, принадлежащей некоторому классу, передавать символьные строки (char *), то желаемый результат (отсортированные строки) не будет получен. Как известно, в этом случае произойдет сравнение указателей, а не строк.

```
#include "iostream.h"
#include "string.h"
#include "typeinfo.h"

class CompareNumb
{ public:
    static bool sravn(int a, int b){return a<b;}
};
class CompareString
```

```
{ public:
   static bool sravn(char *a, char *b){return strcmp(a,b)<0;}
};
template <class T,class Compare>
class vect
{  T *ms;
   int size;
public:
  vect(int SIZE):size(SIZE)
  { ms=new T[size];
    const type_info & t=typeid(T); // получение ссылки t на
    const char* s=t.name();            // объект класса type_info
    for(int i=0;i<size;i++)             // в описании типа
    if(!strcmp(s,"char *"))
      cin >> (*(ms+i)=(T)new char[20]);  // ввод символьных строк
    else cin >> *(ms+i);                // ввод числовой информации
  }
  void sort_vec(vect<T,Compare> &);
};
template <class T,class Compare>
void vect<T,Compare>::sort_vec(vect<T,Compare> &vec)
{ for(int i=0;i<size-1;i++)
    for(int j=i;j<size;j++)
    if(Compare::sravn(ms[i],ms[j]))
    { T tmp=ms[i];
       ms[i]=ms[j];
       ms[j]=tmp;
    }
   for(i=0;i<size;i++) cout << *(ms+i) << endl;
};
```

Класс Compare должен содержать логическую функцию sravn(), сравнивающую два значения типа T.

```
void main()
{ vect<int,CompareNumb> vec1(3);
 vec1.sort_vec(vec1);
 vect<char *,CompareString> vec2(3);
 vec2.sort_vec(vec2);
}
```

Нетрудно заметить, что для всех типов, для которых операция меньше (<) имеет нужный смысл, можно написать следующий шаблон класса сравнения.

```
template<class T>
class Compare
{ public:
```

Библиотека БГУИР

```
  static bool sravn(T a, T b)
  { const type_info & t=typeid(T); // получение ссылки t на
    const char* s=t.name();        // объект класса type_info
    if(!strcmp(s,"char *"))
        return strcmp((char *)a,(char *)b)<0;
    else return a<b;
  }
};
template <class T,class Compare>
class vect
{  T *ms;
   int size;
public:
   vect(int SIZE):size(SIZE)
   { ms=new T[size];
     const type_info & t=typeid(T); // получение ссылки t на
     const char* s=t.name();        // объект класса type_info
     for(int i=0;i<size;i++)         // в описании типа
     if(!strcmp(s,"char *"))
     cin >> (*(ms+i)=(T)new char[20]);  // ввод символьных строк
     else cin >> *(ms+i);                 // ввод числовой информации
   }
   void sort_vec(vect<T,Compare> &);
};
```

Чтобы сделать запись класса более простой, воспользуемся возможностью задания значений некоторых параметров класса по умолчанию.

```
template <class T,class C = Compare<T> >
void vect<T,C>::sort_vec(vect<T,C> &vec)
{ for(int i=0;i<size-1;i++)
    for(int j=i;j<size;j++)
    if(C::sravn(ms[i],ms[j]))
    { T tmp=ms[i];
        ms[i]=ms[j];
        ms[j]=tmp;
    }
   for(i=0;i<size;i++) cout << *(ms+i) << endl;
};
void main()
{ vect<int,Compare<int> > vec1(3);
  vec1.sort_vec(vec1);
  vect<char *,Compare<char *> > vec2(3);
  vec2.sort_vec(vec2);
  vect<long,Compare<long> > vec3(3);
```

Библиотека БГУИР

```
  vec3.sort_vec(vec3);
}
```

В инструкции vect<int,Compare<int> > vec1(3) содержится пробел между угловыми скобками. При его отсутствии компилятор спутает >> с операцией >> (сдвига).

Пространства имен

При совпадении имен разных элементов в одной области действия часто возникает *конфликт имен*. Наиболее часто это возникает при использовании различных пакетов библиотек, содержащих, например, одноименные классы.

Пространства имен используются для разделения глобального пространства имен, что позволяет уменьшить количество конфликтов. Синтаксис пространства имен некоторым образом напоминает синтаксис структур и классов. После ключевого слова namespace следует необязательное имя пространства имен, затем описывается пространство имен, заключенное в фигурные скобки.

```
namespace NAME
{  int a;
    doudle b;
   char *fun(char *,int);
   class CLS
    {   . . .
       public:
         . . .
    }
}
```

Далее, если обращение к элементам пространства имен производится вне контекста, его имя должно быть полностью квалифицировано, используя ::

```
NAME::b=2;
NAME:: fun(str,NAME:: a);
```

Внутри пространства имен можно поместить группу объявлений классов, типов и функций. Реализация функций пространства имен должна находиться вне самого пространства имен. Это позволит не только отделить реализацию функций от их объявления, но и избежать загромождения пространства имен. По существу, namespace определяет область видимости.

Использование безымянного пространства имен (отсутствует имя пространства имен) позволяет определить уникальность объявленных в нем идентификаторов с областью видимости в пределах файла.

Контексты пространства имен могут быть вложены.

```
namespace NAME1
{  int a;
    namespace NAME2
    { int a;
      int fun1(){return NAME1:: a};   // возвращается значение первого a
      int fun2(){return  a};               // возвращается значение второго a
```

Библиотека БГУИР

```
    }
}
NAME!::NAME2::fun1();             // вызов функции
```

Если в каком-то месте программы интенсивно используется некоторый контекст и все имена уникальны по отношению к нему, то можно сократить полные имена, объявив контекст текущим с помощью оператора using.

Если элементы пространства имен будут интенсивно использоваться, то можно использовать ключевое слово using для упрощения доступа к ним. Ключевое слово using используется и как директива, и для объявления. Синтаксис слова using определяет, является ли оно директивой или объявлением.

Ключевое слово using как директива

Директива ***using namespace имя*** позволяет предоставить все имена, объявленные в пространстве имен, для доступа в текущей области действия. Это позволит обращаться к этим именам без указания их полного имени, включающего название пространства имен.

```
#include <iostream.h>
namespace NAME
{ int n1=1;
  int n2=2;
}
// int n1;    приводит к неоднозначности в main   для переменной  n1
int main()
{  NAME::n1=3;
  // n1=3;           // error   'n1' : undeclared identifier
  // n2=4;           // error   'n2' : undeclared identifier
  using namespace NAME;           // далее n1 и n2 доступны
  n2=4;
  cout << n1 <<"  "<< n2 << endl;  // результат 3 4
  { n1=5;
    n2=6;
    cout << n1 <<"  "<< n2 << endl;   // результат 5 6
  }
  return 0;
}
```

В результате выполнения программы получим:

3   4

5   6

Область действия директивы using распространяется на блок, в котором она использована, и на все вложенные блоки.

Если одно из имен относится к глобальной области, а другое объявлено внутри пространства имен, то возникает неоднозначность. Это проявится толь-

Библиотека БГУИР

ко при использовании этого имени, а не при объявлении.

Ключевое слово using как объявление

Объявление ***using имя::член*** подобно директиве, при этом оно обеспечивает более подробный уровень управления. Обычно using используется для объявления некоторого имени (из пространства имен) как принадлежащего текущей области действия (блоку).

```
#include <iostream.h>
namespace NAME
{ int n1=1;
  int n2=2;
}
int main()
{ NAME::n1=3;
 // n1=4;   error  'n1' надо указывать полностью  NAME::n1
 // n2=5;   error  'n2' : undeclared identifier
 // int n2;        следующая строка приводит к ошибке
 using NAME::n2;           // далее n2 доступно
 n2=6;
 cout <<NAME::n1<<"  "<< n2 << endl;  // результат  3  6
 { NAME::n1=7;
  n2=8;
  cout <<NAME::n1<<"  "<< n2 << endl;// результат  7  8
 }
 return 1;
}
```

В результате выполнения программы получим:

3   6
7   8

Объявление using добавляет определенное имя в текущую область действия. В примере к переменной n2 можно обращаться без указания принадлежности классу, а для n1 необходимо полное имя. *Объявление* using обеспечивает более подробное управление именами, переносимыми в пространство имен. Это и есть ее основное отличие от *директивы*, которая переносит все имена пространства имен.

Внесение в локальную область (блок) имени, для которого выполнено явное объявление (и наоборот), является серьезной ошибкой.

Псевдоним пространства имен

Псевдоним пространства имен существует для того, чтобы назначить другое имя именованному пространству имен.

```
namespace spisok_name_peremen   // пространство имен
```

Библиотека БГУИР

```
{ int n1=1;
  . . .
}
NAME = spisok_name_peremen;    // псевдоним пространства имен
```

Пространству имен spisok_name_peremen назначается псевдоним NAME. В этом случае результат выполнения инструкций:

```
cout <<NAME::n1<< endl;
cout<< spisok_name_peremen::n1<<endl;
```

будет одинаков.

Организация ввода-вывода

Системы ввода-вывода С и С++ основываются на понятии потока. Поток в С++ это абстрактное понятие, относящееся к переносу информации от источника к приемнику. В языке С++ реализованы 2 иерархии классов, обеспечивающих операции ввода-вывода, базовыми классами которых являются streambuf и ios. На рис.6 приведена диаграмма классов, базовым для которых является ios (рис. 6).

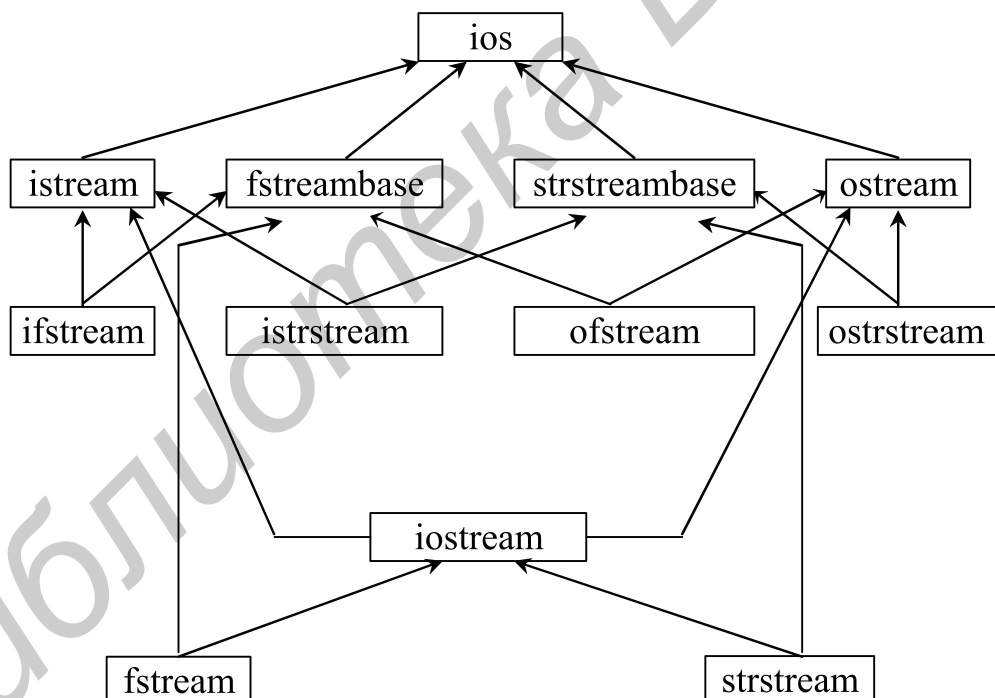

Рис. 6. Диаграмма наследования класса ios

В С++ используется достаточно гибкий способ выполнения операций ввода-вывода классов с помощью перегрузки операторов << (вывода) и >> (ввода). Операторы, перегружающие эти операции, обычно называют инсертером и экстрактором. Для обеспечения работы с потоками ввода-вывода необходимо включить файл iostream.h, содержащий класс iostream. Этот класс является производным от ряда классов, таких как ostream, обеспечивающего вывод

Библиотека БГУИР

данных в поток, и istream – соответственно чтения из потока. Приводимый ниже пример показывает, как можно перегрузить оператор ввода-вывода для произвольных классов.

```
#include "iostream.h"
class cls
{  char c;
   short i;
public :
  cls(char C,short I ) : c(C), i(I){}
  ~cls(){}
  friend ostream &operator<<(ostream &,const cls);
  friend istream &operator>>(istream &,cls &);
};
ostream &operator<<(ostream &out,const cls obj)
{ out << obj.c<<obj.i << endl;
  return out;
}
istream &operator>>(istream &in,cls &obj)
{ in >> obj.st>>obj.i;
  return in;
}
main()
{ cls s('a',10),ss(' ',0);
 out<<"abc"<<endl;
 cout<<s<<ss<<endl;
 cin >> ss;
 return 0;
}
```

Общая форма функции перегрузки оператора ввода-вывода имеет вид:

istream &operator>>(istream &поток,имя_класса &объект)
ostream &operator<<(ostream &поток,const имя_класса объект)

Как видно, функция принимает в качестве аргумента и возвращает ссылку на поток. В результате доопределенный оператор можно применить последовательно к нескольким операндам

cout <<s<<ss;

Это соответствует

(cout.operator<<(a)).operator<<(b);

В приведенном примере функция operator не является компонентом класса cls, так как левым аргументом (в списке параметров) такой функции должен быть this-указатель для объекта, инициирующего перегрузку. Доступ к private-данным класса cls осуществляется через friend-функцию operator этого класса.

Библиотека БГУИР

Рассмотрим использование перегрузки операторов << и >> для определения новых манипуляторов. Ранее мы рассмотрели использование стандартных манипуляторов форматирования выводимой информации. Однако можно определить и новые манипуляторы без изменения стандартных. В качестве примера рассмотрим переопределение **манипулятора с параметрами,** задающего для выводимого дробного числа ширину поля и точность.

```
#include<iostream.h>
class manip
{  int n,m;
   ostream & (*f)(ostream&,int,int) ;
 public:
   manip(ostream& (*F)(ostream&,int,int), int N, int M) :
           f(F), n(N), m(M) {}
   friend ostream& operator<<(ostream& s, manip& obj)
   {return obj.f(s,obj.n,obj.m);}
};
ostream& f_man(ostream & s,int n,int m)
{ s.width(n);
  s.flags(ios::fixed);
  s.precision(m);
  return s;
}
manip wp(int n,int m)
{ return manip(f_man,n,m);
}
void main(void)
{ cout<< 2.3456 << endl;
  cout<<wp(8,1)<<2.3456 << endl;
}
```

Компонента-функция put и вывод символов

Компонента-функция **put**() используется для вывода одиночного символа:

```
char c='a';
 .  .  .
cout.put(c);
```

Вызовы функции put() могут быть сцеплены:

```
cout.put(c).put('b').put('\n');
```

в этом случае на экран выведется буква **a**, затем **b** и далее символ новой строки.

Компоненты-функции get и getline для ввода символов.

Функция get может быть использована в нескольких вариантах.

Первый вариант – функция используется без аргументов. Вводит символ из соответствующего потока одиночный символ и возвращает его значение. Ес-

Библиотека БГУИР

ли из потока прочитан признак конца файла, то get возвращает EOF.

```
#include<iostream.h>
main(void)
{ char c;
  cout << in.eof()<< "вводите текст" << endl;
  while((c=cin.get())!=EOF)
  cout.put(c);
  cout << endl<<cin.eof();
  return 0; }
```

В программе считывается из потока cin очередной символ и выводится с помощью функции put. При считывании признака конца файла завершается цикл while. До и после цикла выводится значение cin.eof(), равное false (выводится 0) до начала цикла и true (выводится 1) после его окончания.

Второй вариант – когда функция get() используется с одним символьным аргументом. Функция возвращает false при считывании признака конца файла, иначе – ссылку на объект класса istream, для которого вызывалась функция get.

```
 . . .
while(cin.get(c))
cout.put(c);
    . . .
```

При ***третьем варианте*** функция get() принимает три параметра: символьный массив, максимальное число символов и ограничитель ввода (по умолчанию ’\n’). Ввод прекращается, когда считано число символов на один меньшее максимального или считан символ-ограничитель. При этом в вводимую строку добавляется нуль-символ. Символ-ограничитель из входного потока не удаляется, это при повторном вызове функции get приведет к формированию пустой строки.

```
char s[30];
 . . .
cin.get(s,20))          // аналогично cin.get(s,20,’\n’)
cout<<s<<endl;
    . . .
```

Функция getline() действует аналогично функции get с тремя параметрами с тем отличием, что символ-ограничитель удаляется из входного потока.

Ниже коротко рассмотрены другие функции-компоненты класса istream.

Состояние потока

Потоки iostream или ostream имеют связанное с ними состояние. В классе ios, базовом для классов iostream и ostream, имеется несколько public функций, позволяющих проверять и устанавливать состояние потока:

inline int ios::**bad**() const { return state & badbit; }
inline int ios::**eof**() const { return state & eofbit; }

Библиотека БГУИР

```
inline int  ios::fail() const { return state & (badbit | failbit); }
inline int  ios::good() const { return state == 0; }
```

Состояние потока фиксируется в элементах перечисления, объявленного в классе ios:

```
public:
   enum io_state {  goodbit = 0x00,
                    eofbit   = 0x01,
                    failbit  = 0x02,
                    badbit  = 0x04 };
```

Кроме отмеченных выше функций в классе ios есть еще несколько функций, позволяющих прочитать (rdstate) и очистить (clear) состояние потока:

```
inline int ios::rdstate() const { return state; }
inline void ios::clear(int _i=0){ lock(); state = _i; unlock(); }
```

Так, если было установлено состояние ошибки, то попытка выполнить ввод/вывод будет игнорироваться до тех пор, пока не будет устранена причина ошибки и биты ошибки не будут сброшены функцией clear().

```
#include "iostream.h"
main()
{ int i;
  int flags;
  do{ cin >> i;
       flags=cin.rdstate();  // чтение состояния потока  cin
       if(flags | ios::badbit)
       { cout << "error in stream"<< endl;
         cin.clear(0);       // сброс всех состояний потока
       }

   } while(flags);        // пока ошибка во входном потоке
   return 0;
}
```

В приведенном примере функция cin.clear(0) выполняет сброс всех установленных бит ошибки. Если требуется сбросить, например, только badbit, то clear(ios::badbit).

```
 . . .
 ifstream in("file");
 while(1)
 { state=in.rdstate();
   if (state)
   {  if(state&ios::badbit) cout<<"ошибка открытия файла"<<endl;
      else if(state&ios::eofbit) cout<<"в файле  больше нет данных"<<endl;
      break;
   }
   else in >> ss;
```

Библиотека БГУИР

```
 }
. . .
```

Необходимо также отметить, что в классе ios выполнена перегрузка операции !:

```
inline int ios::operator!() const { return state&(badbit|failbit); }
```

то есть операция ! возвращает ненулевое значение в случае установки одного из бит badbit или failbit, иначе нулевое значение, например:

```
if(!cin) cout << ”ошибка потока  cin”<<endl;       // проверка состояния
else cin>>i;                                        // входного потока
```

Строковые потоки

Особой разновидностью потоков являются строковые потоки, представленные классом *strstream*:

```
class strstream : public iostream
{ public:
    strstream();
    strstream(char *s, streamsize n,
              ios_base::openmode=ios_base::in | ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz=true);
    char *str();
    streamsize pcount() const;
};
```

Важное свойство класса strstream состоит в том, что в нем автоматически выделяется требуемый объем памяти для хранения строковых данных. Все операции со строковыми потоками происходят в памяти в специально выделенном для них буфере *strstreambyf*. Строковые потоки позволяют облегчить формирование данных в памяти. В примере демонстрируется ввод данных в буфер, копирование их в компоненту s класса string и их просмотр.

```
#include "strstrea.h"
class string
{ char s[80];
 public:
  string(char *S) {for(int i=0;s[i++]=*S++;);}
  void see(){cout<<s<<endl;}
};
void fun(const char *s)
{ strstream st;                 //  создание объекта st
  st << s << ends;             //  вывод данных в поток (в буфер)
  string obj(st.str());         //  создание объекта класса string
  st.rdbuf()->freeze(0);       //  освобождение памяти в буфере
  obj.see();                   // просмотр скопированной из буфера строки
}
```

Библиотека БГУИР

```
main()
{ fun("1234");
}
```

Вначале создается объект st типа strstream. Далее при выводе переданной в функцию символьной строки в поток добавлен манипулятор ends, который соответствует добавлению ’\0’. Функция str() класса strstream обращается непосредственно к хранимой в буфере строке. Следует отметить, что при обращении к функии str() объект st перестает контролировать эту память и при уничтожении объекта память не освобождается. Для ее освобождения требуется вызвать функцию st.rdbuf()–>freeze(0).

Далее в примере показывается проверка состояния потока, чтобы убедиться в правильности выполняемых операций. Это позволяет, например, легко определить переполнение буфера.

```
#include "strstrea.h"
void fun(const char *s,int n)
{ char *buf=new char[n];                  // входной буфер
  strstream st(buf,n,ios::in|ios::out);   // связывание потока st и буфера byf
  st << s << ends;                        // ввод информации в буфер
  if(st.good()) cout << buf<<endl;        // проверка состояния потока
  else cerr<<"error"<<endl;               // вывод сообщения об ошибке
}
main()
{ fun("123456789",5);
  fun("123456789",15);
}
```

Организация работы с файлами

В языке С++ для организации работы с файлами используются классы потоков **ifstream** (ввод), **ofstream** (вывод) и **fstream** (ввод и вывод) (рис. 7). Перечисленные классы являются производными от istream, ostream и iostream соответственно. Операции ввода-вывода выполняются так же, как и для других потоков, то есть компоненты-функции, операции и манипуляторы могут быть применены и к потокам файлов. Различие состоит в том, как создаются объекты и как они привязываются к требуемым файлам.

Библиотека БГУИР

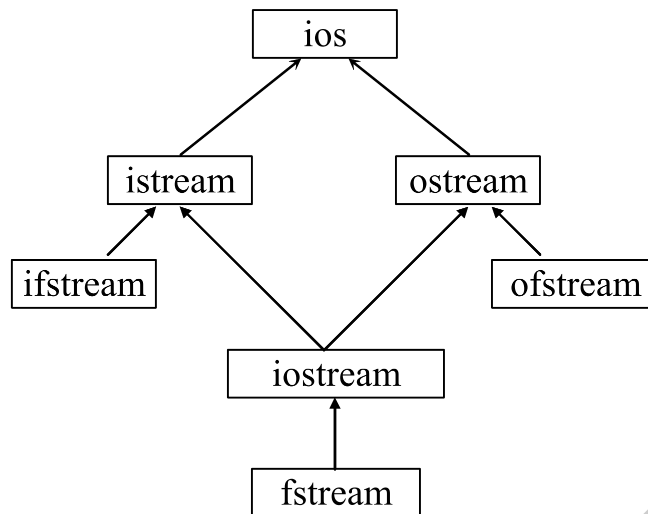

Рис. 7. Часть иерархии классов потоков ввода-вывода

В С++ файл открывается путем стыковки его с соответствующим потоком. Рассмотрим организацию связывания потока с некоторым файлом. Для этого используются конструкторы классов ifstream и ofsream:

ofstream(const char* Name, int nMode= ios::out, int nPot= filebuf::openprot);
ifstream(const char* Name, int nMode= ios::in, int nPot= filebuf::openprot);

Первый аргумент определяет имя файла (единственный обязательный параметр).

Второй аргумент задает режим для открытия файла и представляет битовое ИЛИ величин:

ios::app при записи данные добавляются в конец файла, даже если текущая позиция была перед этим изменена функцией **ostream::seekp**;

ios::ate указатель перемещается в конец файла. Данные записываются в текущую позицию (произвольное место) файла;

ios::in поток создается для ввода: если файл уже существует, то он сохраняется;

ios::out поток создается для вывода (по умолчанию для всех **ofstream** объектов);

ios::trunc если файл уже существует, его содержимое уничтожается (длина равна нулю). Этот режим действует по умолчанию, если **ios::out** установлен, а **ios::ate**, **ios::app** или **ios:in** не установлены;

ios::nocreate если файл не существует, функциональные сбои;

ios::noreplace если файл уже существует, функциональные сбои;

ios::binary ввод-вывод будет выполняться в двоичном виде (по умолчанию текстовой режим).

Третий аргумент – данное класса filebuf, используется для установки атрибутов доступа к открываемому файлу.

Возможные значения *nProt*:

filebuf::sh_compat овместно используют режим;

Библиотека БГУИР

filebuf::sh_none режим Exclusive: никакое совместное использование;

filebuf::sh_read совместно использующее чтение;

filebuf::sh_write совместно использующее запись.

Для комбинации атрибутов **filebuf::sh_read** and **filebuf::sh_write** используется операция логическое ИЛИ (||).

В отличие от рассмотренного подхода можно создать поток, используя конструктор без аргументов. Позже вызвать функцию open, имеющую те же аргументы, что и конструктор, при этом второй аргумент не может быть задан по умолчанию:

void open(const char* name, int mode, int prot=fileout::openprot);

Только после того как поток создан и соединен с некоторым файлом (используя либо конструктор с параметрами, либо функцию open), можно выполнять ввод информации в файл или вывод из файла.

```
#include "iostream.h"
#include "fstream.h"
#include "string.h"

class string
{  char *st;
   int size;
public :
  string(char *ST,int SIZE) : size(SIZE)
  { st=new char[size];
    strcpy(st,ST);
  }
  ~string() {delete [] st;}
  string(const string &s)    // копирующий конструктор необходим, так как
  { st=new char[s.size];     // при перегрузке << в функцию operator переда-
    strcpy(st,s.st);         // ется объект, содержащий указатель на строку, а
  }                          // в конце вызовется деструктор для объекта obj
  friend ostream &operator<<(ostream &,const string);
  friend istream &operator>>(istream &,string &);
};

ostream &operator<<(ostream &out,const string obj)
{ out << obj.st << endl;
  return out;
}

istream &operator>>(istream &in,string &obj)
{ in >> obj.st;
  return in;
}

main()
```

Библиотека БГУИР

```
{ string s("asgg",10),ss("aaa",10);
  int state;
  ofstream out("file");
  if (!out)
 { cout<<"ошибка открытия файла"<<endl;
   return 1;      // или exit(1)
 }
 out<<"123"<<endl;
 out<<s<<ss<<endl;      //  запись в файл
 ifstream in("file");
 while(in >> ss)            //  чтение из файла
 {cout<<ss<<endl;}
 in.close();
 out.close();
 return 0;
}
```

В приведенном примере в классе содержится копирующий конструктор, так как в функцию operator передается объект obj, компонентой которого является строка. В инструкции cout <<s<<ss копирующий конструктор вызывается вначале для объекта ss, затем для s, после этого выполняется перегрузка в порядке, показанном ранее. При завершении каждой из функций operator (вначале для s, затем для ss) будет вызван деструктор.

В операторе if (!out) вызывается (как и ранее для потоков) функция ios::operator! для определения, успешно ли открылся файл.

Условие в заголовке оператора while автоматически вызывает функцию класса ios перегрузки операции void *:

```
operator void *() const { if(state&(badbit|failbit) ) return 0;
                          return (void *)this; }
```

то есть выполняется проверка; если для потока устанавливается failbit или badbit, то возвращается 0.

Организация файла последовательного доступа

В С++ файлу не предписывается никакая структура. Для последовательного поиска данных в файле программа обычно начинает считывать данные с начала файла до тех пор, пока не будут считаны требуемые данные. При поиске новых данных этот процесс вновь повторяется.

Данные, содержащиеся в файле последовательного доступа, не могут быть модифицированы без риска разрушения других данных в этом файле. Например, если в файле содержится информация:

Коля 12 Александр 52

то при модификации имени Коля на имя Николай может получиться следующее:

НиколайАлександр 52

Библиотека БГУИР

Аналогично в последовательности целых чисел 12 –1 132 32554 7 для хранения каждого из них отводится sizeof(int) байт. А при форматированном выводе их в файл они занимают различное число байт. Следовательно, такая модель ввода-вывода неприменима для модификации информации на месте. Эта проблема может быть решена перезаписью (с одновременной модификацией) в новый файл информации из старого. Это сопряжено с проблемой обработки всей информации при модификации только одной записи.

Следующая программа выполняет перезапись информации из одного файла в два других, при этом в первый файл из исходного переписываются только числа, а во второй – вся остальная информация.

```
#include "fstream.h"
#include "stdlib.h"
#include "math.h"

void error(char *s1,char *s2="")  // вывод сообщения об ошибке
{ cerr<<s1<<"  "<<s2<<endl;      // при открытии потока для файла
  exit(1);
}

main(int argc,char **argv)
{ char *buf=new char[20];
  int i;
  ifstream f1;                     // входной поток
  ofstream f2,f3;                  // выходные потоки
  f1.open(argv[1],ios::in);        // открытие исходного файла
  if(!f1)                          // проверка состояния потока
  error("ошибка открытия файла",argv[1]);
  f2.open(argv[2],ios::out);       // открытие 1 выходного файла
  if(!f2) error("ошибка открытия файла",argv[1]);
  f3.open(argv[3],ios::out);       // открытие 2 выходного файла
  if(!f3) error("ошибка открытия файла",argv[1]);
  f1.seekg(0);                 // установить текущий указатель в начало потока
  while(f1.getline(buf,20,' '))  // считывание в буфер до 20 символов
  { if(int n=f1.gcount())        // число реально считанных символов
     buf[n-1]='\0';
                              // проверка на только цифровую строку
     for(i=0;*(buf+i)&&(*(buf+i)>='0' && *(buf+i)<='9');i++);
     if(!*(buf+i)) f2 <<::atoi(buf)<<' ';     // преобразование в число и запись
                                               // в файл f2
     else f3<<buf<<' ';                       // просто выгрузка буфера в файл f3
  }
 delete [] buf;
 f1.close();                       // закрытие файлов
 f2.close();
 f3.close();
```

}

В программе для ввода имен файлов использована командная строка, первый параметр – имя файла источника (входного), а два других – имена файлов приемников (выходных). Для работы с файлами использованы функции – open, close, seekg, getline и gcount. Более подробное описание функций приведено ниже.

Ниже приведена программа, выполняющая ввод символов в файл с их одновременным упорядочиванием (при вводе) по алфавиту. Использование функций seekg, tellg позволяет позиционировать текущую позицию в файле, то есть осуществлять прямой доступ в файл. Обмен информацией с файлом осуществляется посредством функций get и put.

```
#include "fstream.h"
#include "stdlib.h"

void error(char *s1,char *s2="")
{ cerr<<s1<<"  "<<s2<<endl;
  exit(1);
}

main()
{ char c,cc;
  int n;
  fstream f;                    // выходной поток
  streampos p,pp;
  f.open("aaaa",ios::in|ios::out); // открытие выходного файла
  if(!f) error("ошибка открытия файла","aaaa");
  f.seekp(0);                   // установить текущий указатель в начало потока
  while(1)
  { cin>>c;
    if (c=='q' || f.bad()) break;
    f.seekg(0,ios::beg);
    while(1)
    { if(((cc=f.get())>=c) || (f.eof()))
      { if(f.eof())
        { f.clear(0);
          p=f.tellg();
        }
        else
        { p=f.tellg()-1;
          f.seekg(-1,ios::end);
          pp=f.tellg();
          while(p<=pp)
          { cc=f.get();
            f.put(cc);
            f.seekg(--pp);
```

Библиотека БГУИР

```
            }
          }
         f.seekg(p);
         f.put(c);
         break;
       }
    }
   }
   f.close();
   return 1;
  }
```

Каждому объекту класса istream соответствует указатель get (указывающий на очередной вводимый из потока байт) и указатель put (соответственно на позицию для вывода байта в поток). Классы istream и ostream содержат по 2 перегруженные компоненты-функции для перемещения указателя в требуемую позицию в потоке (связанном с ним файле). Такими функциями являются seekg (переместить указатель для извлечения из потока) и seekp (переместить указатель для помещения в поток).

```
istream& seekg( streampos pos );
istream& seekg( streamoff off, ios::seek_dir dir );
```

Сказанное выше справедливо и для функций seekp. Первая функция перемещает указатель в позицию pos относительно начала потока. Вторая перемещает соответствующий указатель на off (целое число) байт в трех возможных направлениях: ios::beg (от начала потока), ios::cur (от текущей позиции) и ios::end (от конца потока). Кроме рассмотренных функций в этих классах имеются еще функции tellg и tellp, возвращающие текущее положение указателей get и put соответственно

```
streampos tellg();
```

Создание файла произвольного доступа

Организация хранения информации в файле прямого доступа предполагает доступ к ней не последовательно от начала файла по некоторому ключу, а непосредственно, например, по их порядковому номеру. Для этого требуется, чтобы все записи в файле были бы одинаковой длины.

Наиболее удобными для организации произвольного доступа при вводе-выводе информации являются следующие компоненты-функции:

```
istream& istream::read(E *s, streamsize n);
ostream& ostream::write(E *s, streamsize n);
```

при этом, так как функция write (read) ожидает первый аргумент типа const char * (char *), то для требуемого приведения типов используется оператор явного преобразования типов:

```
istream& istream::read(reinterpret_cast<char *>(&s), streamsize n);
```

Библиотека БГУИР

ostream& ostream::write(reinterpret_cast<const char *>(&s),
streamsize n);

Ниже приведен текст программы организации работы с файлом произвольного доступа на примере удаления и добавления в файл информации о банковских реквизитах клиента (структура inf).

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

struct inf
{ char cl[10];    // клиент
  int pk;           // пин-код
  double sm;      // сумма на счете
} cldata;

class File
{  char filename[80];       // имя файла
   fstream *fstr;               // указатель на поток
   int maxpos;                 // число записей в файле
 public:
   File(char *filename);
  ~File();
   int Open();
   const char* GetName();
   int Read(inf &);
   void Remote();
   void Add(inf);
   int Del(int pos);
   friend ostream& operator << (ostream &out, File &obj)
   { inf p;
     out << "File " << obj.GetName() << endl;
     obj.Remote();
     while(obj.Read(p))
     out<<"\nКлиент -> "<<p.cl<<' '<<p.pk<<' '<<p.sm;
     return out;
   }
};

File::File(char *_filename)             // конструктор
{  strncpy(filename,_filename,80);
   fstr = new fstream();
}

File::~File()                             // деструктор
```

Библиотека БГУИР

```
{  fstr->close(); }
int File::Open()   // функция открывает файл для ввода-вывода бинарный
{  fstr->open(filename, ios::in | ios::out | ios::binary);
   if (!fstr->is_open()) return -1;
   return 0;
}
int File::Read(inf &p)     // функция чтения из потока fstr в объект   p
{  if(!fstr->eof() &&              // если не конец файла
   fstr->read(reinterpret_cast<char*>(&p),sizeof(inf)))
   return 1;                        //
   fstr->clear();
   return 0;
}
void File::Remote()      // сдвиг указателей get и put в начало потока
{  fstr->seekg(0,ios_base::beg); // сдвиг указателя на get на начало
   fstr->seekp(0,ios_base::beg); // сдвиг указателя на put на начало
   fstr->clear();                      // чистка состояния потока
}
const char* File::GetName()      // функция возвращает имя файла
{  return this->filename;  }
void File::Add(inf cldata)
{  fstr->seekp(0,ios_base::end);
  fstr->write(reinterpret_cast<char*>(&cldata),sizeof(inf));
  fstr->flush();
}
int File::Del(int pos) // функция удаления из потока записи с номером pos
{  Remote();
  fstr->seekp(0,ios::end);  // для вычисления maxpos
  maxpos = fstr->tellp();   // позиция указателя put
  maxpos/=sizeof(inf);
  if(maxpos<pos) return -1;

  fstr->seekg(pos*sizeof(inf),ios::beg); // сдвиг на позицию,
                                         // следующую за pos
  while(pos<maxpos)
  { fstr->read(reinterpret_cast<char*>(&cldata),sizeof(inf));
    fstr->seekp(-2*sizeof(inf), ios::cur);
    fstr->write(reinterpret_cast<char*>(&cldata),sizeof(inf));
    fstr->seekg(sizeof(inf),ios::cur);
    pos++;
  }
```

Библиотека БГУИР

```
    strcpy(cldata.cl,"");  // для занесения пустой записи в
    cldata.pk=0;          // конец файла
    cldata.sm=0;
    fstr->seekp(-sizeof(inf), ios::end);
    fstr->write(reinterpret_cast<char*>(&cldata),sizeof(inf));
    fstr->flush();        // выгрузка выходного потока в файл
  }
  int main(void)
  { int n;
    File myfile("file");
    if(myfile.Open() == -1)
    { cout << "Can't open the file\n";
      return -1;
    }
    cin>>cldata.cl>>cldata.pk>>cldata.sm;
    myfile.Add(cldata);
    cout << myfile << endl;      // просмотр файла
    cout << "Введите номер клиента для удаления ";
    cin>>n;
    if(myfile.Del(n) == -1)
    cout << "Клиент с номеном "<<n<<" вне файла\n";
    cout << myfile << endl;      // просмотр файла
}
```

Основные функции классов ios, istream, ostream

Из диаграммы классов ввода-вывода следует, что классы ios, istream, ostream являются базовыми для большинства классов. Класс ios является типом-синонимом для шаблона класса basic_ios.

```
template <class E, class T = char_traits<E> >
class basic_ios : public ios_base
{public:
   iostate rdstate() const;              // возвращает текущее состояние потока
   void clear(iostate state = goodbit); // устанавливает состояние потока в
                                        // заданное значение. Состояние потока
                                        // можно изменить:  cin.clear(ios::eofbit)
   void setstate(iostate state);
   bool good() const;           //  true-значение при отсутствии ошибки
   bool eof() const;            //  true  при достижении конца файла, т.е. при
                                //  попытке выполнить в/в за пределами файла
   bool fail() const;           //  true  при ошибке в текущей операции
   bool bad() const;            //  true  при ошибке
   basic_ostream<E, T> *tie() const;
   basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
```

Библиотека БГУИР

```
  basic_streambuf<E, T> *rdbuf() const;   // возвращает указатель на буфер
                                          // ввода-вывода
  basic_streambuf<E, T> *rdbuf(basic_streambuf<E, T> *sb);
  basic_ios& copyfmt(const basic_ios& rhs);
  locale imbue(const locale& loc);
  E widen(char ch);
  char narrow(E ch, char dflt);
 protected:
  basic_ios();
  void init(basic_streambuf<E, T>* sb);
};
```

Далее приводятся прототипы некоторых функций классов istream и ostream.

```
template <class E, class T = char_traits<E> >
class basic_istream : virtual public basic_ios<E, T>
{public:
  bool ipfx(bool noskip = false);
  void isfx();
  streamsize gcount() const;
  int_type get();
  basic_istream& get(E& c);
  basic_istream& get(E *s, streamsize n);
  basic_istream& get(E *s, streamsize n, E delim);
  basic_istream& get(basic_streambuf<E, T> *sb);
  basic_istream& get(basic_streambuf<E, T> *sb, E delim);
  basic_istream& getline(E *s, streamsize n)E
  basic_istream& getline(E *s, streamsize n, E delim);
  basic_istream& ignore(streamsize n = 1,     // пропускает n символов или
                  int_type delim = T::eof()); // завершается при считывании
                                              // заданного ограничителя по
                                              // умолчанию – EOF
  int_type peek();      // читает символ из потока, не удаляя его из потока
  basic_istream& read(E *s, streamsize n);  // неформатированный ввод
  streamsize readsome(E *s, streamsize n);
  basic_istream& putback(E c);// возвращает обратно в поток предыдущий
                             // символ, полученный из потока ф-цией get
  basic_istream& unget();
  pos_type tellg();
  basic_istream& seekg(pos_type pos);
  basic_istream& seekg(off_type off, ios_base::seek_dir way);
  int sync();
};
```

Библиотека БГУИР

```
template <class E, class T = char_traits<E> >
class basic_ostream : virtual public basic_ios<E, T>
{public:
   bool opfx();
   void osfx();
   basic_ostream& put(E c);
   basic_ostream& write(E *s, streamsize n);  // неформатированный вывод
   basic_ostream& flush();
   pos_type tellp();
   basic_ostream& seekp(pos_type pos);
   basic_ostream& seekp(off_type off, ios_base::seek_dir way);
};
```

Исключения в C++

Исключения – возникновение непредвиденных ошибочных условий, например, деление на ноль, невозможность выделения памяти при создании нового объекта и т.д. Обычно эти условия завершают выполнение программы с системной ошибкой. C++ позволяет восстанавливать программу из этих условий и продолжать ее выполнение. В то же время исключение – это более общее, чем ошибка, понятие и может возникать и тогда, когда в программе нет ошибок.

Механизм обработки исключительных ситуаций на сегодняшний день является неотъемлемой частью языка C++. Этот механизм предоставляет программисту средство реагирования на нештатные события и позволяет преодолеть ряд принципиальных недостатков следующих традиционных методов обработки ошибок:

– возврат функцией кода ошибки;

– возврат значений ошибки через аргументы функций;

– использование глобальных переменных ошибки;

– использование оператора безусловного перехода goto или функций setjmp/longjmp;

– использование макроса assert.

Возврат функцией кода ошибки является самым обычным и широко применяемым методом. Однако этот метод имеет существенные недостатки. Во-первых, нужно помнить численные значения кодов ошибок. Эту проблему можно обойти, используя перечисляемые типы. Но в некоторых случаях функция может возвращать широкий диапазон допустимых (неошибочных) значений, и тогда сложно найти диапазон для возвращаемых кодов ошибки. Это и является вторым недостатком. И, в-третьих, при использовании такого механизма сигнализации об ошибках вся ответственность по их обработке ложится на программиста и могут возникнуть ситуации, когда серьезные ошибки будут оставлены без внимания.

Возврат кода ошибки через аргумент функции или использование глобальной переменной ошибки снимают, прежде всего, вторую проблему, однако

Библиотека БГУИР

по-прежнему остаются первая и третья. Кроме того, использование глобальных переменных не является особо позитивным фактором.

Использование оператора безусловного перехода в любых ситуациях является нежелательным, кроме того, оператор goto действует только в пределах функции. Пара функций setjmp/longjmp является довольно мощным средством, однако и этот метод имеет серьезнейший недостаток: он не обеспечивает вызов деструкторов локальных объектов при выходе из области видимости, что, естественно, влечет за собой утечки памяти.

И, наконец, макрос assert является скорее средством отладки, чем средством обработки нештатных событий, возникающих в процессе использования программы.

Таким образом, необходим некий другой способ обработки ошибок, который учитывал бы объектно-ориентированную философию. Таким способом и является механизм обработки исключительных ситуаций.

Основы обработки исключительных ситуаций

Обработка исключительных ситуаций лишена недостатков вышеназванных методов реагирования на ошибки. Этот механизм позволяет использовать для представления информации об ошибке объект любого типа. Поэтому можно, например, создать иерархию классов, которая будет предназначена для обработки аварийных событий. Это упростит, структурирует и сделает более понятной программу.

Рассмотрим пример обработки исключительных ситуаций. Функция div() возвращает частное от деления чисел, принимаемых в качестве аргументов. Если делитель равен нулю, то генерируется исключительная ситуация.

```
#include<iostream.h>
double div(double dividend, double divisor)
{  if(divisor==0)  throw 1;
   return dividend/divisor;
}
void main()
{ double result;
  try {
    result=div(77.,0.);
    cout<<"Answer is "<<result<<endl;
  }
  catch(int){
    cout<<"Division by zero"<<endl;
  }
}
```

Результат выполнения программы:

Division by zero

Библиотека БГУИР

В данном примере необходимо выделить три ключевых элемента. Во-первых, вызов функции div() заключен внутрь блока, который начинается с ключевого слова **try**. Этот блок указывает, что внутри него могут происходить исключительные ситуации. По этой причине код, заключенный внутри блока try, иногда называют охранным.

Далее за блоком try следует блок **catch**, называемый обычно обработчиком исключительной ситуации. Если возникает исключительная ситуация, выполнение программы переходит к этому catch-блоку. Хотя в этом примере имеется один-единственный обработчик, их в программах может присутствовать множество и они способны обрабатывать множество различных типов исключительных ситуаций.

Еще одним элементом процесса обработки исключительных ситуаций является оператор **throw** (в данном случае он находится внутри функции div()). Оператор throw сигнализирует об исключительном событии и генерирует объект исключительной ситуации, который перехватывается обработчиком catch. Этот процесс называется вызовом исключительной ситуации. В рассматриваемом примере исключительная ситуация имеет форму обычного целого числа, однако программы могут генерировать практически любой тип исключительной ситуации.

Если в инструкции   if(divisor==0) throw 1 значение 1 заменить на 1. , то при выполнении будет выдана ошибка об отсутствии соответствующего обработчика catch (так как возбуждается исключительная ситуация типа double).

Одним из главных достоинств использования механизма обработки исключительных ситуаций является обеспечение ***развертывания стека***. Развертывание стека – это процесс вызова деструкторов локальных объектов, когда исключительные ситуации выводят их из области видимости.

Сказанное рассмотрим на примере функции add() класса add_class, выполняющей сложение компонентов-данных объектов add_class и возвращающей суммарный объект. В случае, если сумма превышает максимальное значение для типа unsigned short, генерируется исключительная ситуация.

```
#include<limits.h>
#include<iostream.h>

class add_class
{ private:
    unsigned short num;
  public:
    add_class(unsigned short a)
     {  num=a;
        cout<<"Constructor "<<num<<endl;
     }
    ~add_class()  { cout<<"Destructor of add_class "<<num<<endl; }
    void show_num()  { cout<<" "<<num<<" ";     }
    void input_num(unsigned short a)  { num=a; }
```

Библиотека БГУИР

```
    unsigned short output_num(){return num;}
};
add_class add(add_class a,add_class b)
{ add_class sum(0);
  unsigned long s=(unsigned long)a.output_num()+
                  (unsigned long)b.output_num();
  if(s>USHRT_MAX) throw (int)1;
  sum.input_num((unsigned short) s);
  return sum;
}


void main()
{ add_class a(USHRT_MAX),b(1),s(0);
  try{
    s=add(a,b);
    cout<<"Result";
    s.show_num();
    cout<<endl;
  }
 catch(int){
    cout<<"Overflow error"<<endl;
 }
}
```

Результат выполнения программы:

```
Constructor 65535
Constructor 1
Constructor 0
Constructor 0
Destructor of add_class 0
Destructor of add_class 65535
Destructor of add_class 1
Overflow error
Destructor of add_class 0
Destructor of add_class 1
Destructor of add_class 65535
```

Сначала вызываются конструкторы объектов a, b и s, далее происходит передача параметров по значению в функцию (в этом случае происходит вызов конструктора копий, созданного по умолчанию, именно поэтому вызовов деструктора больше, чем конструктора), затем, используя конструктор, создается объект sum. После этого генерируется исключение и срабатывает механизм развертывания стека, то есть вызываются деструкторы локальных объектов sum, a и b. И, наконец, вызываются деструкторы s, b и a.

Библиотека БГУИР

Рассмотрим более подробно элементы try, catch и throw механизма обработки исключений.

Блок try. Синтаксис блока:

```
try{
охранный код
}
список обработчиков
```

Необходимо помнить, что после ключевого слова try всегда должен следовать составной оператор, т.е. после try всегда следует {…}. Блоки try не имеют однострочной формы, как, например, операторы if, while, for.

Еще один важный момент заключается в том, что после блока try должен следовать, по крайней мере, хотя бы один обработчик. Недопустимо нахождение между блоками try и catch какого-либо кода. Например:

```
int i;
try{
    throw исключение;
}
i=0;                // 'try' block starting on line ' номер ' has no catch handlers
catch(тип аргумент){
   блок обработки исключения
}
```

В блоке try можно размещать любой код, вызовы локальных функций, функции-компоненты объектов, и любой код любой степени вложенности может генерировать исключительные ситуации. Блоки try сами могут быть вложенными.

Обработчики исключительных ситуаций catch. Обработчики исключительных ситуаций являются важнейшей частью всего механизма обработки исключений, так как именно они определяют поведение программы после генерации и перехвата исключительной ситуации. Синтаксис блока catch имеет следующий вид:

```
catch(тип 1 <аргумент>)
{
тело обработчика
}
catch(тип 2 <аргумент>))
{
тело обработчика
}
.
.
.
catch(тип N <аргумент>))
{
тело обработчика
```

Библиотека БГУИР

}

Таким образом, так же как и в случае блока try, после ключевого слова catch должен следовать составной оператор, заключенный в фигурные скобки. В аргументах обработчика можно указать только тип исключительной ситуации, необязательно объявлять имя объекта, если этого не требуется.

У каждого блока try может быть множество обработчиков, каждый из которых должен иметь свой уникальный тип исключительной ситуации. Неправильной будет следующая запись:

```
typedef int type_int;
try{   . . .  }

catch(type_int error1){
        . . .
}
catch(int error2){
        . . .
}
```

Так, в этом случае type_int и int – это одно и то же.

Однако следующий пример верен.

```
class cls
{ public:
     int i;
};

try{
        . . .
}
catch(cls i1){
        . . .
}
catch(int i2){
}
```

В этом случае cls – это отдельный тип исключительной ситуации. Существует также абсолютный обработчик, который совместим с любым типом исключительной ситуации. Для написания такого обработчика надо вместо аргументов написать многоточие (эллипсис).

catch (…){
блок обработки исключения
}

Использование абсолютного обработчика исключительных ситуаций рассмотрим на примере программы, в которой происходит генерация исключительной ситуации типа char *, но обработчик такого типа отсутствует. В этом случае управление передается абсолютному обработчику.

Библиотека БГУИР

```
#include <iostream.h>
#include <conio.h>
void int_exception(int i)
{ if(i>100) throw 1;
}
void string_exception()
{ throw "Error";
}
void main()
{ try{
    int_exception(99);
    string_exception();
  }
  catch(int){
     cout<<"Обработчик для типа Int";
     getch();
  }
 catch(...){
     cout<<"Абсолютный обработчик ";
     getch();
 }
```

Результат выполнения программы:
Абсолютный обработчик

Так как абсолютный обработчик перехватывает исключительные ситуации всех типов, то он должен стоять в списке обработчиков последним. Нарушение этого правила вызовет ошибку при компиляции программы.

Для того чтобы эффективно использовать механизм обработки исключительных ситуаций, необходимо грамотно построить списки обработчиков, а для этого, в свою очередь, нужно четко знать следующие правила, по которым осуществляется поиск соответствующего обработчика:

– исключительная ситуация обрабатывается первым найденным обработчиком, т. е. если есть несколько обработчиков, способных обработать данный тип исключительной ситуации, то она будет обработана первым стоящим в списке обработчиком;

– абсолютный обработчик может обработать любую исключительную ситуацию;

– исключительная ситуация может быть обработана обработчиком соответствующего типа либо обработчиком ссылки на этот тип;

– исключительная ситуация может быть обработана обработчиком базового для нее класса. Например, если класс B является производным от класса A, то обработчик класса A может обработать исключительную ситуацию класса B;

– исключительная ситуация может быть обработана обработчиком, при-

Библиотека БГУИР

нимающим указатель, если тип исключительной ситуации может быть приведен к типу обработчика, путем использования стандартных правил преобразования типов указателей.

Если при возникновении исключительной ситуации подходящего обработчика нет среди обработчиков данного уровня вложенности блоков try, то обработчик ищется на следующем охватывающем уровне. Если обработчик не найден вплоть до самого верхнего уровня, то программа аварийно завершается.

Следствием из правил 3 и 4 является еще одно утверждение: исключительная ситуация может быть направлена обработчику, который может принимать ссылку на объект базового для данной исключительной ситуации класса. Это значит, что если, например, класс B – производный от класса A, то обработчик ссылки на объект класса A может обрабатывать исключительную ситуацию класса B (или ссылку на объект класса B).

Рассмотрим особенности выбора соответствующего обработчика на следующем примере. Пусть имеется класс C, являющийся производным от классов A и B; показано, какими обработчиками может быть перехвачена исключительная ситуация типа C и типа указателя на C.

```
#include<iostream.h>
class A{};
class B{};
class C : public A, public B {};
void f(int i)
{  if(i) throw C();      // возбуждение исключительной ситуации
                         // типа объект C
   else throw new C; //  возбуждение исключительной ситуации
                         // типа указатель на объект класса C
}
void main()
{ int i;
  try{
    cin>>i;
    f(i);
  }
  catch(A) {
    cout<<"A handler";
  }
  catch(B&) {
    cout<<"B& handler";
  }
  catch(C) {
    cout<<"C handler";
  }
```

Библиотека БГУИР

```
  catch(C*) {
    cout<<"C* handler";
  }
  catch(A*) {
    cout<<"A* handler";
  }
  catch(void*) {
    cout<<"void* handler";
  }
}
```

В данном примере исключительная ситуация класса C может быть направлена любому из обработчиков A, B& или C, поэтому выбирается обработчик, стоящий первым в списке. Аналогично для исключительной ситуации, имеющей тип указателя на объект класса C, выбирается первый подходящий обработчик A* или C*. Эта ситуация также может быть обработана обработчиками void*. Так как к типу void* может быть приведен любой указатель, то обработчик этого типа будет перехватывать любые исключительные ситуации типа указателя.

Генерация исключительных ситуаций throw. Исключительные ситуации передаются обработчикам с помощью ключевого слова throw. Как ранее отмечалось, обеспечивается вызов деструкторов локальных объектов при выходе из области видимости, то есть развертывание стека. Однако развертывание стека не обеспечивает уничтожение объектов, созданных динамически. Таким образом, перед генерацией исключительной ситуации необходимо явно освободить динамически выделенные блоки памяти.

Следует отметить также, что если исключительная ситуация генерируется по значению или по ссылке, то создается скрытая временная переменная, в которой хранится копия генерируемого объекта. Когда после throw указывается локальный объект, то к моменту вызова соответствующего обработчика этот объект будет уже вне области видимости и, естественно, прекратит существование. Обработчик же получит в качестве аргумента именно эту скрытую копию. Из этого следует, что если генерируется исключительная ситуация сложного класса, то возникает необходимость снабжения этого класса конструктором копий, который бы обеспечил корректное создание копии объекта.

Если же исключительная ситуация генерируется с использованием указателя, то копия объекта не создается. В этом случае могут возникнуть проблемы. Например, если генерируется указатель на локальный объект, к моменту вызова обработчика объект уже перестанет существовать и использование указателя в обработчике приведет к ошибкам.

Перенаправление исключительных ситуаций

Иногда возникает положение, при котором необходимо обработать исключительную ситуацию сначала на более низком уровне вложенности блока

Библиотека БГУИР

try, а затем передать ее на более высокий уровень для продолжения обработки. Для того чтобы сделать это, нужно использовать throw без аргументов. В этом случае исключительная ситуация будет перенаправлена к следующему подходящему обработчику (подходящий обработчик не ищется ниже в текущем списке – сразу осуществляется поиск на более высоком уровне). Приводимый ниже пример демонстрирует организацию такой передачи. Программа содержит вложенный блок try и соответствующий блок catch. Сначала происходит первичная обработка, затем исключительная ситуация перенаправляется на более высокий уровень для дальнейшей обработки.

```
#include<iostream.h>
void func(int i)
{  try{
       if(i) throw "Error";
   }
   catch(char *s) {
       cout<<s<<"-  выполняется первый обработчик"<<endl;
       throw;
   }
}

void main()
{  try{
      func(1);
   }
   catch(char *s) {
      cout<<s<<"- выполняется второй обработчик"<<endl;
   }
}
```

Результат выполнения программы:
Error – выполняется первый обработчик
Error – выполняется второй обработчик

Если ключевое слово trow используется вне блока catch, то автоматически будет вызвана функция terminate(), которая по умолчанию завершает программу.

Исключительная ситуация, генерируемая оператором new

Следует отметить, что некоторые компиляторы поддерживают генерацию исключений в случае ошибки выделения памяти посредством оператора new, в частности исключения типа bad_alloc. Оно может быть перехвачено и необходимым образом обработано. Ниже в программе рассмотрен пример генерации и обработки исключительной ситуаций bad_alloc. Искусственно вызывается ошибка выделения памяти и перехватывается исключительная ситуация.

Библиотека БГУИР

```
#include <new>
using std::bad_alloc;
#include <iostream.h>
void main()
{  double *p;
    try{
      while(1) p=new double[100];  // генерация ошибки выделения памяти
    }
   catch(bad_alloc exept) {                    // обработчик  xalloc
      cout<<"Возникло исключение"<<exept.what()<<endl;
   }
}
```

В случае если компилятором не генерируется исключение bad_alloc, то можно это исключение создать искусственно:

```
#include <new>
using std::bad_alloc;
#include <iostream.h>
void main()
{  double *p;
  bad_alloc exept;
    try{
        if(!(p=new double[100000000]))  // память не выделена   p==NULL
        throw exept;                    // генерация ошибки выделения памяти
    }
    catch(bad_alloc exept) {          // обработчик  bad_alloc
          cout<<"Возникло исключение   "<<exept.what()<<endl;
   }
}
```

Результатом работы программы будет сообщение:

Возникло исключение   bad_allocation

Оператор new появился в языке С++ еще до того, как был введен механизм обработки исключительных ситуаций, поэтому первоначально в случае ошибки выделения памяти этот оператор просто возвращал NULL. Если требуется, чтобы new работал именно так, надо вызвать функцию set_new_handler() с параметром 0. Кроме того, с помощью set_new_handler() можно задать функцию, которая будет вызываться в случае ошибки выделения памяти. Функция set_new_handler (в заголовочном файле new) принимает в качестве аргумента указатель на функцию для функции, которая не принимает никаких аргументов и возвращает void.

```
#include <new.h>
#include <iostream.h>

int error_alloc( size_t  ) // size_t unsigned integer  результат sizeof operator.
{   cout << "ошибка выделения памяти" <<endl;
```

```
   return 0;
}
void main( void )
{
  _set_new_handler( error_alloc );
  int *p = new int[10000000000];
}
```

Результатом работы функции является:

ошибка выделения памяти

В случае, если память не выделяется и не задается никакой функции-аргумента для set_new_handler, оператор new генерирует исключение bad_alloc.

Генерация исключений в конструкторах

Механизм обработки исключительных ситуаций очень удобен для обработки ошибок, возникающих в конструкторах. Так как конструктор не возвращает значения, то соответственно нельзя возвратить некий код ошибки и приходится искать альтернативу. В этом случае наилучшим решением является генерация и обработка исключительной ситуации. При генерации исключения внутри конструктора процесс создания объекта прекращается. Если к этому моменту были вызваны конструкторы базовых классов, то будет обеспечен и вызов соответствующих деструкторов. Рассмотрим на примере генерацию исключительной ситуации внутри конструктора. Пусть имеется класс B, производный от класса A и содержащий в качестве компоненты-данного объект класса local. В конструкторе класса B генерируется исключительная ситуация.

```
#include<iostream.h>
  class local
  {  public:
        local()   {  cout<<"Constructor of local"<<endl;   }
        ~local()  {  cout<<"Destructor of local"<<endl;    }
  };
  class A
  {  public:
       A()        {  cout<<"Constructor of A"<<endl;  }
      ~A()        {   cout<<"Destructor of A"<<endl;  }
  };
  class B : public A
  {  public:
       local ob;
       B(int i)
       {  cout<<"Constructor of B"<<endl;
          if(i ) throw 1;
       }
```

Библиотека БГУИР

```
        ~B()   {  cout<<"Destructor of B"<<endl; }
};
void main()
{   try {
        B ob(1);
    }
    catch(int) {
        cout<<"int exception handler";
    }
}
```

Результат выполнения программы:
Constructor of A
Constructor of local
Constructor of B
Destructor of local
Destructor of A
int exception handler

В программе при создании объекта производного класса B сначала вызываются конструкторы базового класса A, затем класса local, который является компонентом класса B. После этого вызывается конструктор класса B, в котором генерируется исключительная ситуация. Видно, что при этом для всех ранее созданных объектов вызваны деструкторы, а для объекта самого класса B деструктор не вызывается, так как конструирование этого объекта не было завершено.

Если в конструкторах выполнялось динамическое выделение памяти, то при генерации исключительной ситуации выделенная память автоматически освобождена не будет, об этом необходимо заботиться самостоятельно, иначе возникнут утечки памяти.

Задание собственной функции завершения

Если программа не может найти подходящий обработчик для сгенерированной исключительной ситуации, то будет вызвана процедура завершения terminate() (ее также называют обработчик завершения), по умолчанию выполнение программы будет остановлено и на экран будет выведено сообщение «Abnormal program termination». Однако можно установить собственный обработчик завершения, используя функцию set_terminate(), единственным аргументом которой является указатель на новую функцию завершения (*функция, принимающая и возвращающая void*), а возвращаемое значение – указатель на предыдущий обработчик. Ниже приведен пример установки собственного обработчика завершения и генерации исключительной ситуации, для которой не может быть найден обработчик.

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

Библиотека БГУИР

```
#include <exception>
#include <stdlib.h>
void  my_term()
{  cout<<"Собственная функция-обработчик";
   exit(1);
}
void main()
{   set_terminate(my_term);
    try {
      throw 1;        // генерация исключительной ситуации типа int
    }
    catch(char) {    // обработчик для типа   char
        cout<<"char handler";
    }
}
```

Результат выполнения программы:
Собственная функция-обработчик

Спецификации исключительных ситуаций

Иногда возникает необходимость заранее указать, какие исключения могут генерироваться в той или иной функции. Это можно сделать с помощью так называемой спецификации исключительных ситуаций. Это средство позволяет указать в объявлении функции типы исключительных ситуаций, которые могут в ней генерироваться. Синтаксис спецификации имеет вид:

объявление функции throw(тип1, тип2,…){тело функции}

Использование спецификации исключительных ситуаций не означает, что в функции не может быть сгенерирована исключительная ситуация некоторого не указанного в спецификации типа. Просто в этом случае программа по умолчанию завершится, так как подобные действия приведут к вызову неожиданного обработчика. Таким образом, когда функция генерирует исключительную ситуацию, не описанную в спецификации, выполняется неожиданный обработчик.

Задание собственного неожиданного обработчика

Так же как и обработчик terminate(), обработчик unexpected() позволяет перед завершением программы выполнить какие-то действия. Но в отличие от обработчика завершения неожиданный обработчик может сам генерировать исключительные ситуации. Таким образом, собственный неожиданный обработчик может сгенерировать исключительную ситуацию, на этот раз уже входящую в спецификацию. Установка собственного неожиданного обработчика выполняется с помощью функции set_unexpected().

Приведенная ниже программа демонстрирует применение спецификации

Библиотека БГУИР

исключений и перехват неожиданных исключительных ситуаций с помощью собственного обработчика.

```
#include <iostream.h>
#include <exception>
class first{};
class second : public first{};
class third : public first{};
class my_class{};

void my_unexpected()
{  cout<<"my_unexpected handler"<<endl;
   throw third();                  //  возбуждение исключения типа объект
}                                  // класса   third

void f(int i)  throw(first)        // указание спецификации исключения
{  if(i ) throw second();          //
   else throw my_unexpected();
}

void main()
{  set_unexpected(my_unexpected);
   try  {
      f(1);
   }
   catch(first)  {
         cout<<"first handler"<<endl;
   }
   catch(my_class) {
         cout<<"my_class handler"<<endl;
   }
   try{
      f(0);
   }
   catch(first)  {
      cout<<"first handler"<<endl;
   }
   catch(my_class) {
      cout<<"my_class handler"<<endl;
   }
}
```

Результат выполнения программы.

first handler
my_unexpected handler
first handler

Библиотека БГУИР

В данной программе вызов функции f() во втором блоке try приводит к тому, что генерируется исключительная ситуация, тип которой не указан в спецификации, поэтому вызывается установленный нами неожиданный обработчик, где происходит генерация исключения, которое успешно обрабатывается.

Иерархия исключений стандартной библиотеки

Вершиной иерархии является класс exceptioh (определенный в заголовочном файле <exception>). В этом классе содержится функция what(), переопределяемая в каждом производном классе для выдачи сообщения об ошибке.

Непосредственными производными классами от класса exception являются классы runtime_error и logic_error (определенные в заголовочном файле <stdexcept>), имеющие по несколько производных классов.

Производными от exception также являются исключения: bad_alloc, генерируемое оператором new, bad_cast, генерируемое dynamic_cast, и bad_typeid, генерируемое оператором typeid.

Класс logic_error и производные от него классы (invalid_argument, length_error, out_of_range) указывают на логические ошибки (передача неправильного аргумента функции, выход за пределы массива или строки).

Класс runtime_error и производные от него (overflow_error и underflow_error) указывают на математические ошибки переполнения сверху и снизу.

Стандартная библиотека шаблонов (STL)

Общее понятие о контейнере

Стандартная библиотека шаблонов (Standard Template Library, STL) входит в стандартную библиотеку языка C++. В неё включены реализации наиболее часто используемых контейнеров и алгоритмов, что избавляет программистов от рутинного переписывания их снова и снова. При разработке контейнеров и применяемых к ним алгоритмов (таких как удаление одинаковых элементов, сортировка, поиск и т. д.) часто приходится приносить в жертву либо универсальность, либо быстродействие. Однако разработчики STL поставили перед собой задачу: сделать библиотеку одновременно эффективной и универсальной. Для ее решения были использованы такие универсальные средства языка C++, как шаблоны и перегрузка операторов. В последующем изложении будем опираться на реализацию STL, поставляемую фирмой Microsoft вместе с компилятором Visual C++ 6.0. Тем не менее большая часть сказанного будет справедлива и для реализаций STL другими компиляторами.

Основными понятиями в STL являются понятия контейнера (container), алгоритма (algorithm) и итератора (iterator).

Контейнер – это хранилище объектов (как встроенных, так и определённых пользователем типов). Как правило, контейнеры реализуются в виде шаблонов классов. Простейшие виды контейнеров (статические и динамические массивы) встроены непосредственно в язык C++. Кроме того, стандартная биб-

Библиотека БГУИР

лиотека включает в себя реализации таких контейнеров, как вектор (vector), список (list), очередь (deque), ассоциативный массив (map), множество (set) и некоторых других.

Алгоритм – это функция для манипулирования объектами, содержащимися в контейнере. Типичные примеры алгоритмов – сортировка и поиск. В STL реализовано порядка 60 алгоритмов, которые можно применять к различным контейнерам, в том числе к массивам, встроенным в язык C++.

Итератор – это абстракция указателя, то есть объект, который может ссылаться на другие объекты, содержащиеся в контейнере. Основные функции итератора – обеспечение доступа к объекту, на который он ссылается (разыменование), и переход от одного элемента контейнера к другому (итерация, отсюда и название итератора). Для встроенных контейнеров в качестве итераторов используются обычные указатели. В случае с более сложными контейнерами итераторы реализуются в виде классов с набором перегруженных операторов.

Помимо отмеченных элементов в STL есть ряд ***вспомогательных понятий***; с некоторыми из них следует также познакомиться.

Аллокатор (allocator) – это объект, отвечающий за распределение памяти для элементов контейнера. С каждым стандартным контейнером связывается аллокатор (его тип передаётся как один из параметров шаблона). Если какому-то алгоритму требуется распределять память для элементов, он обязан делать это через аллокатор. В этом случае можно быть уверенным, что распределённые объекты будут уничтожены правильно.

В состав STL входит стандартный класс allocator (описан в файле xmemory). Именно его по умолчанию используют все контейнеры, реализованные в STL. Однако пользователь может реализовать собственный класс. Необходимость в этом возникает очень редко, но иногда это можно сделать из соображений эффективности или в отладочных целях.

Остановимся более подробно на рассмотрении введенных понятий.

Контейнеры. Каждый контейнер предоставляет строго определённый интерфейс, через который с ним будут взаимодействовать алгоритмы. Этот интерфейс обеспечивают соответствующие контейнеру итераторы. Важно подчеркнуть, что никакие дополнительные функции-члены для взаимодействия алгоритмов и контейнеров не используются. Это сделано потому, что стандартные алгоритмы должны работать, в том числе со встроенными контейнерами языка C++, у которых есть итераторы (указатели), но нет ничего, кроме них. Таким образом, при создании собственного контейнера реализация итератора – необходимый минимум.

Каждый контейнер реализует определённый тип итераторов. При этом выбирается наиболее функциональный тип итератора, который может быть эффективно реализован для данного контейнера. «Эффективно» означает, что скорость выполнения операций над итератором не должна зависеть от количества элементов в контейнере. Например, для вектора реализуется итератор с произвольным доступом, а для списка – двунаправленный. Поскольку скорость выполнения операции [] для списка линейно зависит от его длины, итератор с

Библиотека БГУИР

произвольным доступом для списка не реализуется.

Вне зависимости от фактической организации контейнера (вектор, список, дерево) хранящиеся в нём элементы можно рассматривать как последовательность. Итератор первого элемента в этой последовательности возвращает функция begin(), а итератор элемента, следующего за последним, – функция end(). Это очень важно, так как все алгоритмы в STL работают именно с последовательностями, заданными итераторами начала и конца.

Кроме обычных итераторов в STL существуют обратные итераторы (reverse iterator). Обратный итератор отличается тем, что просматривает последовательность элементов в контейнере в обратном порядке. Другими словами, операции + и – у него меняются местами. Это позволяет применять алгоритмы как к прямой, так и к обратной последовательности элементов. Например, с помощью функции find можно искать элементы как "с начала", так и "с конца" контейнера.

В STL контейнеры делятся на три основные группы (табл. 2): контейнеры последовательностей, ассоциативные контейнеры и адаптеры контейнеров. Первые две группы объединяются в контейнеры первого класса.

Таблица 2

Контейнерный класс STL	Описание
Контейнеры последовательностей	
vector	Динамический массив
deque	Двунаправленная очередь
List	Двунаправленный линейный список
Ассоциативные контейнеры	
Set	Ассоциативный контейнер с уникальными ключами
multiset	Ассоциативный контейнер, допускающий дублирование ключей
Map	Ассоциативный контейнер для наборов уникальных элементов
multimap	Ассоциативный контейнер для наборов с дублированием элементов
Адаптеры контейнеров	
Stack	Стандартный стек
queue	Стандартная очередь
priority_queue	Очередь с приоритетами

Каждый класс контейнера, реализованный в STL, описывает набор типов, связанных с контейнером. При написании собственных контейнеров следует придерживаться этой же практики. Вот список наиболее важных типов:

value_type – тип элемента;

size_type – тип для хранения числа элементов (обычно size_t);

iterator – итератор для элементов контейнера;

key_type – тип ключа (в ассоциативном контейнере).

Библиотека БГУИР

Помимо типов можно выделить набор функций, которые реализует почти каждый контейнер в STL (табл. 3). Они не требуются для взаимодействия с алгоритмами, но их реализация улучшает взаимозаменяемость контейнеров в программе. STL разработана с тем расчетом, чтобы контейнеры обеспечивали аналогичные функциональные возможности.

Таблица 3

Общие методы всех STL-контейнеров	Описание
1	2
default constructor	Конструктор по умолчанию. Обычно контейнер имеет несколько конструкторов
copy constructor	Копирующий конструктор
destructor	Деструктор
empty	Возвращает true, если в контейнере нет элементов, иначе false
max_size	Возвращает максимальное число элементов для контейнера
size	Возвращает число элементов в контейнере в текущее время
operator =	Присваивает один контейнер другому
operator <	Возвращает true, если первый контейнер меньше второго, иначе false
operator <=	Возвращает true, если первый контейнер не больше второго, иначе false
operator >	Возвращает true, если первый контейнер больше второго, иначе false
operator >=	Возвращает true, если первый контейнер не меньше второго, иначе false
operator ==	Возвращает true, если сравниваемые контейнеры равны, иначе false
operator !=	Возвращает true, если сравниваемые контейнеры не равны, иначе false
swap	Меняет местами элементы двух контейнеров
Функции, имеющиеся только в контейнерах первого класса	
begin	Две версии этой функции возвращают либо iterator, либо const_iterator, который ссылается на первый элемент контейнера
end	Две версии этой функции возвращают либо iterator, либо const_iterator, который ссылается на следующую позицию после конца контейнера
rbegin	Две версии этой функции возвращают либо reverse_iterator, либо reverse_const_iterator, который

Библиотека БГУИР

	ссылается на последний элемент контейнера
rend	Две версии этой функции возвращают либо reverse_iterator, либо reverse_const_iterator, который ссылается на позицию перед первым элементом контейнера
insert, erase,	Позволяют вставить или удалить элемент(ы) в середине последовательности

Окончание табл. 3

1	2
clear	Удаляет из контейнера все элементы
front, back	Возвращают ссылки на первый и последний элемент, хранящийся в контейнере
push_back, pop_back	Позволяют добавить или удалить последний элемент в последовательности
push_front, pop_front	Позволяют добавить или удалить первый элемент в последовательности

Итераторы обычно создаются как друзья классов, с которыми они работают, что позволяет выполнить прямой доступ к частным данным этих классов. С одним контейнером может быть связано несколько итераторов, каждый из которых поддерживает свою собственную «позиционную информацию» (табл. 4).

Таблица 4

Тип итератора	Доступ	Разыменование	Итерация	Сравнение
Итератор вывода (output iterator)	Только запись	*	++	
Итератор ввода (input iterator)	Только чтение	*, ->	++	==, !=
Прямой итератор (forward iterator)	Чтение и запись	*, ->	++	==, !=
Двунаправленный итератор (bidirectional iterator)	Чтение и запись	*, ->	++, --	==, !=
Итератор с произвольным доступом (random-access iterator)	Чтение и запись	*, ->, []	++, --, +, -, +=, -=	==, !=, <, <=, >, >=

Общее понятие об итераторе

Для структурированных итераций, например, при обработке массивов:

```
for(i=0;i<size;i++) sm+=a[i];
```

порядок обращения к элементу управляется индексом i, который изменяется явно. Можно зафиксировать получение следующего элемента в компоненте-функции.

Библиотека БГУИР

```
class vect
{  int *p;        // массив чисел
   int size;      // размерность массива
   int ind;       // текущий индекс
 public:
   vect();              // размерность массива const
   vect(int SIZE);   // размерность массива size
  ~vect();
  int ub(){return size-1;}
  int next()
  { if(ind==pv->size)
       return pv->p[(ind=0)++];
     else
       return pv->p[ind++];
  }
};
```

Это соответствует тому, что обращение к объекту ограничивается использованием одного индекса ind. Другая возможность состоит в том, чтобы создать множество индексов и передавать функции обращения к элементу один из них. Это ведет к существенному увеличению числа переменных. Более удобным представляется создание отдельного, связанного с vect класса (класса итераций), в функции которого входит обращение к элементам класса vect.

```
#include <iostream.h>
class vect
{  friend class vect_iterator;          // предварительное friend-объявление
   int *p;                              // базовый указатель (массив)
   int size;                            // размерность массива
 public:
   vect(int SIZE):size(SIZE)
   { p=new int[size];
     for(int i=0; i<size; *(p+i++)=i);
   }
   int ub(){return size-1;}             // возвращается размер массива
   void add()                          // изменение содержимого массива
   { for(int i=0; i<size; *(p+i++)+=1);}
   ~vect(){delete [] p;}
};
class vect_iterator
{  vect *pv;                           // указатель на класс vect
   int ind;                             // текущий индекс в массиве
 public:
   vect_iterator(vect &v): ind(0),pv(&v){}
   int &next();//возвращается текущее значение из массива (с индекса ind)
```

Библиотека БГУИР

```
};
int &vect_iterator::next()
{ if(ind==pv->size)
    return pv->p[(ind=0)++];
  else
    return pv->p[ind++];
}
void main()
{ vect v(5);
  vect_iterator v_i1(v),v_i2(v);              // создано 2 объекта-итератора
                                              // для прохода по объекту vect
  cout<<v_i1.next()<<' '<<v_i2.next()<<endl;
  v.add();                                    // модификация объекта v
  cout<<v_i1.next()<<' '<<v_i2.next()<<endl;
  for(int i=0;i<v.ub();i++)
  cout<<v_i1.next()<<endl;
}
```

Результат работы программы:

```
0  0
2  2
3
4
5
1
```

Полное отсоединение обращения от составного объекта позволяет объявлять столько объектов итераторов, сколько необходимо. При этом каждый из объектов итераторов может просматривать объект vect независимо от других.

Итератор представляет собой операцию, обеспечивающую последовательный доступ ко всем частям объекта. Итераторы имеют свойства, похожие на свойства указателей, и могут быть использованы для указания на элементы контейнеров первого класса. Итераторы реализуются для каждого типа контейнера. Также имеется целый ряд операций (*, ++ и другие) с итераторами, стандартными для контейнеров.

Если итератор **a** указывает на некоторый элемент, то **++a** указывает на следующий элемент, а ***a** ссылается на элемент, на который указывает **a**.

Объект типа **iterator** может использоваться для ссылки на элемент контейнера, который может быть модифицирован, а **const_iterator** для ссылки на немодифицируемый элемент контейнера.

Категории итераторов

Итераторы, как и контейнеры, реализуются в виде шаблонов классов. Итераторы обладают такими достоинствами, как, например, автоматическое от-

слеживание размера типа, на который указывает итератор, автоматизированные операции инкремента и декремента для перехода от элемента к элементу. Именно благодаря таким возможностям итераторы и являются фундаментом всей библиотеки.

Итераторы можно условно разделить на две категории: **основные** и **вспомогательные**.

Основные итераторы

Основные итераторы используются наиболее часто. Они взаимозаменяемы, однако при этом нужно соблюдать иерархию старшинства (рис.8).

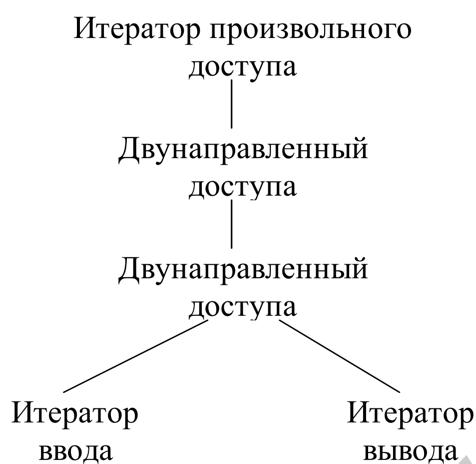

Рис. 8. Иерархия итераторов

Итераторы ввода. Итераторы ввода (input iterator) стоят в самом низу иерархии итераторов. Это наиболее простые из всех итераторов STL, и доступны они только для чтения. Итератор ввода может быть сравнен с другими итераторами на предмет равенства или неравенства, чтобы узнать, не указывают ли два разных итератора на один и тот же объект. Можно использовать оператор разыменовывания (*) для получения содержимого объекта, на который итератор указывает. Перемещаться от первого элемента, на который указывает итератор ввода, к следующему элементу можно с помощью оператора инкремента (++). Ниже приведен пример, демонстрирующий некоторые приемы работы с итератором ввода.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
main(void)
{  int init1[4];
   vector<int> v(4);
   istream_iterator<int> ii(cin);
   for(int j,i=0;i<4;i++)
       //  v.push_back(*ii++);     добавление в конец вектора
       // *(v.begin()+i)=*ii++;
```

Библиотека БГУИР

```
        //  v[i]=*ii++;
    copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
}
```

Итераторы ввода возвращает только шаблонный класс istream_iterator. Однако, несмотря на то что итераторы ввода возвращаются единственным классом, ссылки на него присутствуют повсеместно. Это связано с тем, что вместо итератора ввода может подставляться любой из основных итераторов, за исключением итератора вывода, назначение которого прямо противоположно итератору ввода.

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f)
{
   while (first != last) f(*first++);
   return f;
}
```

В примере первые два параметра – итераторы ввода на начало цепочки объектов и на первое значение, находящееся «за пределом» для этой цепочки. Тело алгоритма выполняет переход от объекта к объекту, вызывая для каждого значения, на которое указывает итератор ввода first, функцию. Указатель на нее передается в третьем параметре. Здесь задействованы все три перегруженных оператора, допустимые для итераторов ввода: сравнения (!=), инкремента (++) и разыменовывания (*). Ниже приводится пример использования алгоритма for_each для однонаправленных итераторов.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void Print(int n)
{ cout << n << " " ; }
void main()
{  const int SIZE = 5 ;
   typedef vector<int > IntVector ;  //  создание синонима для  vector<int>
   typedef IntVector::iterator IntVectorItr;//аналогично для IntVector::iterator
   IntVector Numbers(SIZE) ;                 //вектор, содержащий целые числа
   IntVectorItr start, end, it ;              // итераторы для IntVector
   int i ;
   for (i = 0; i < SIZE; i++)    // инициализация вектора
      Numbers[i] = i + 1 ;
   start = Numbers.begin() ;   // итератор на начало вектора
   end = Numbers.end() ;       // итератор на запредельный элемент вектора

   for_each(start, end, Print);
   cout << "\n\n" ;
```

Библиотека БГУИР

```
}
```

Чтобы включить в программу возможность использования потоков, добавляется включаемый файл iostream, а для описания прототипа алгоритма for_each в программу включается заголовочный файл algorithm (algorith для продуктов Borland). Обязательным при использовании STL является использование директивы:

using namespace std,

включающей пространство имен библиотеки STL.

Итераторы вывода. Если итератор ввода предназначен для чтения данных, то итератор вывода (output iterator) служит для ссылки на область памяти, куда выводятся данные. Итераторы вывода можно встретить повсюду, где происходит хоть какая-то обработка информации средствами STL. Для данного итератора определены операторы присвоения (=), разыменовывания (*) и инкремента (++). Однако следует помнить, что первые два оператора предполагают, что итератор вывода располагается в левой части выражений, то есть во время присвоения он должен быть целевым итератором, которому присваиваются значения. Разыменовывание нужно делать лишь для того, чтобы присвоить некое значение объекту, на который итератор ссылается. Итераторы ввода могут быть возвращены итераторами потоков вывода (ostream_iterator) и итераторами вставки inserter, front_inserter и back_inserter (рассмотрены ниже в разделе "Итераторы вставки"). Рассмотрим пример использования итераторов вывода:

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
main(void)
{ int init1[] = {1, 2, 3, 4, 5};
  int init2[] = {6, 7, 8, 9, 10};
  vector<int> v(10);
  merge(init1, init1 + 5, init2, init2 + 5, v.begin());
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
}
```

В примере помимо потоков и алгоритмов использован контейнер vector (представляющий одномерный массив, вектор). У него имеются специальные компоненты-функции begin() и end(). В приведенном нами примере создаются и инициализируются два массива – init1 и init2. Далее их значения соединяются вместе алгоритмом merge и записываются в вектор. А для проверки полученного результата мы пересылаем данные из вектора в поток вывода, для чего вызываем алгоритм копирования copy и специальный итератор потока вывода ostream_iterator. Он перешлет данные в поток cout, разделив каждое пересылаемое значение символом окончания строки. Для шаблонного класса ostream_iterator требуется указать тип выводимых значений. В нашем случае

Библиотека БГУИР

это int.

Если в примере описать еще один вектор vv:

```
vector<int> vv(10);
```

то в алгоритме copy вместо выходного итератора можно, например, использовать итератор вектора vv для копирования данных из одного вектора в другой:

```
copy(v.begin(), v.end(), vv.begin());
```

Рассмотрим еще один пример использования итераторов ввода-вывода.

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

#define N 2
void main()
{ cout<<"Введите  "<<N<<" числа"<<endl;
  std::istream_iterator<int>  in_obj(cin);
  int ms[N],i;
  for(i=0;i<N;i++)
  { ms[i]=*in_obj;
    ++in_obj;
  }
 ostream_iterator<int> out_obj(cout);
 for(i=0;i<N;i++)
 *out_obj=ms[i];
  cout<<endl;
}
```

В инструкциях:

```
std::istream_iterator<int>  in_obj(cin);
```

и

```
ostream_iterator<int> out_obj(cout);
```

создаются итераторы istream_iterator и ostream_iterator для ввода и вывода int значений из объектов cin и cout соответственно.

Использование операции * (разыменовывания) ms[i]=*in_obj приводит к получению значения из потока in_obj и занесению его в элемент массива, а в инструкции *out_obj=ms[i] к получению ссылки на объект out_obj, ассоциируемый с выходным потоком, и посылке значения элемента массива в поток. Перегруженная операция ++in_obj перемещает итератор in_obj к следующему элементу во входном потоке. Отметим, что для выходного потока операции разыменование и инкремент возвращают одно значение – ссылку на поток.

```
ostream_iterator<_U, _E, _Tr>& operator*()
        {return (*this); }
ostream_iterator<_U, _E, _Tr>& operator++()
        {return (*this); }
```

Однонаправленные итераторы. Если соединить итераторы ввода и вы-

Библиотека БГУИР

вода, то получится однонаправленный итератор (forward iterator), который может перемещаться по цепочке объектов в одном направлении, за что и получил такое название. Для такого перемещения в итераторе определена операция инкремента (++). Кроме этого, в однонаправленном итераторе есть операторы сравнения (== и !=), присвоения (=) и разыменовывания (*). Все эти операторы можно увидеть, если посмотреть, как реализован, например, алгоритм replace, заменяющий одно определенное значение на другое:

```
template <class ForwardIterator, class T>
void replace (ForwardIterator first, ForwardIterator last,
              const T& old_value, const T& new_value)
{  while (first != last)
   {  if (*first == old_value) *first = new_value;
      ++first;
   }
}
```

Чтобы убедиться в правильности работы всех операторов однонаправленных итераторов, составим программу, заменяющую в исходном массиве все единицы на нули и наоборот, т. е. произведем инверсию. С этой целью все нули заменяются на некоторое значение, например на двойку. Затем все единицы обнуляются, а все двойки становятся единицами:

```
#include <algorithm>
#include <iostream>
using namespace std;
main(void)
{
 replace(init, init + 5, 0, 2);
 replace(init, init + 5, 1, 0);
 replace(init, init + 5, 2, 1);
 copy(init, init + 5, ostream_iterator<int>(cout, "\n"));
}
```

Алгоритм replace, используя однонаправленные итераторы, читает значения, заменяет их и перемещается от одного к другому.

Двунаправленные итераторы. Двунаправленный итератор (bidirectional iterator) аналогичен однонаправленному итератору. В отличие от последнего двунаправленный итератор может перемещаться не только из начала в конец цепочки объектов, но и наоборот. Это становится возможным благодаря наличию оператора декремента (--). На двунаправленных итераторах базируются различные алгоритмы, выполняющие реверсивные операции, например reverse. Этот алгоритм меняет местами все объекты в цепочке, на которую ссылаются переданные ему итераторы. Следующий пример был бы невозможен без двунаправленных итераторов:

```
#include <algorithm>
#include <iostream>
```

Библиотека БГУИР

```
using namespace std;
main(void)
{
  int init[] = {1, 2, 3, 4, 5};
  reverse(init, init + 5);
  copy(init, init + 5, ostream_iterator<int>(cout, "\n"));
}
```

Итераторы двунаправленного доступа возвращаются несколькими контейнерами STL: list, set, multiset, map и multimap.

Итераторы произвольного доступа. Итераторы этой категории – наиболее универсальные из основных итераторов. Они не только реализуют все функции, свойственные итераторам более низкого уровня, но и обладают большими возможностями. Глядя на исходные тексты, в которых используются итераторы произвольного доступа, можно подумать, что имеешь дело с арифметикой указателей языка C++. Реализованы такие операции, как сокращенное сложение и вычитание (+= и -=), сравнение итераторов (<, >, <= и >=), операция обращения к заданному элементу массива ([]), а также и некоторые другие операции.

Как правило, все сложные алгоритмы, требующие расширенных вычислений, оперируют итераторами произвольного доступа. Ниже приводится пример, в котором мы используем практически все операции, допустимые для них. Исходный текст разбит на части, к каждой из которых приведены комментарии. Сначала нужно включить требуемые заголовочные файлы и определить константу пробела:

```
#include <algorithm>
#include <iostream>
#include <vector>
#define space " "
```

Затем следует включить использование STL:

```
using namespace std;
```

В функции main мы описываем массив числовых констант и вектор из пяти элементов:

```
int main(void)
{
  const int init[] = {1, 2, 3, 4, 5};
  vector<int> v(5);
```

Создаем переменную типа «итератор произвольного доступа». Для этого берем итератор и на его основе создаем другой, более удобный:

```
 typedef vector<int>::iterator vectItr;
 vectItr itr ;
```

Инициализируем вектор значениями из массива констант и присваиваем адрес его первого элемента итератору произвольного доступа:

```
 copy(init, init + 5, itr = v.begin());
```

Библиотека БГУИР

Далее, используя различные операции над итераторами, последовательно читаем элементы вектора, начиная с конца, и выводим их на экран:

```
cout << *( itr + 4 ) << endl;
cout << *( itr += 3 ) << endl;
cout << *( itr -= 1) << endl;
cout << *( itr = itr - 1) << endl;
cout << *( --itr ) << endl;
```

После этого итератор, претерпев несколько изменений, снова указывает на первый элемент вектора. А чтобы убедиться, что значения в векторе не были повреждены, и проверить оператор доступа ([]), выведем в цикле значения вектора на экран:

```
 for(int i = 0; i < (v.end() - v.begin()); i++)
    cout << itr[i] << space;
  cout << endl;
}
```

Операции с итераторами произвольного доступа реализуются таким образом, чтобы не чувствовалось разницы между использованием обычных указателей и итераторов.

Итераторы произвольного доступа возвращают такие контейнеры, как vector и deque.

Вспомогательные итераторы

Вспомогательные итераторы названы так потому, что они выполняют вспомогательные операции по отношению к основным.

Реверсивные итераторы. Некоторые классы-контейнеры спроектированы так, что по хранимым в них элементам данных можно перемещаться в заданном направлении. В одних контейнерах это направление от первого элемента к последнему, а в других – от элемента с самым большим значением к элементу, имеющему наименьшее значение. Однако существует специальный вид итераторов, называемых реверсивными. Такие итераторы работают «с точностью до наоборот», то есть если в контейнере итератор ссылается на первый элемент данных, то реверсивный итератор ссылается на последний. Получить реверсивный итератор для контейнера можно вызовом метода rbegin(), а реверсивное значение "за пределом" возвращается методом rend(). Следующий пример использует нормальный итератор для вывода значений от 1 до 5 и реверсивный итератор для вывода этих же значений, но в обратном порядке:

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
main(void)
{
  const int init[] = {1, 2, 3, 4, 5};
  vector<int> v(5);
```

Библиотека БГУИР

```
  copy(init, init + 5, v.begin());
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
  copy(v.rbegin(), v.rend(), ostream_iterator<int>(cout, " "));  }
```

Итераторы потоков. Важную роль в STL играют итераторы потоков, которые делятся на итераторы потоков ввода и вывода. Практически во всех рассмотренных примерах имеется итератор потока вывода для отображения данных на экране. Суть применения потоковых итераторов в том, что они превращают любой поток в итератор, используемый точно так же, как и прочие итераторы: перемещаясь по цепочке данных, он считывает значения объектов и присваивает им другие значения.

Итератор потока ввода – это удобный программный интерфейс, обеспечивающий доступ к любому потоку, из которого требуется считать данные. Конструктор итератора имеет единственный параметр – поток ввода. А поскольку итератор потока ввода представляет собой шаблон, то ему передается тип вводимых данных. Вообще-то должно передаваться четыре типа, но последние три имеют значения по умолчанию. Каждый раз, когда требуется ввести очередной элемент информации, используйте оператор ++ точно так же, как с основными итераторами. Считанные данные можно узнать, если применить разыменовывание (*).

Итератор потока вывода весьма схож с итератором потока ввода, но у его конструктора имеется дополнительный параметр, которым указывают строку-разделитель, добавляемую в поток после каждого выведенного элемента. Ниже приведен пример программы, читающей из стандартного потока cin числа, вводимые пользователем и дублирующие их на экране, завершая сообщение строкой – «last entered value». Работа программы заканчивается при вводе числа 999:

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
main(void)
{ istream_iterator<int> is(cin);
  ostream_iterator<int> os(cout, " – введенное значение \n");
  int input;
  while((input = *is) != 999)
  { *os++ = input;
     is++ ;
  }
}
```

Потоковые итераторы имеют одно существенное ограничение: в них нельзя возвратиться к предыдущему элементу. Единственный способ сделать это – заново создать итератор потока.

Итераторы вставки. Появление итераторов вставки (insert iterator) было продиктовано необходимостью. Без них просто невозможно добавить значения

Библиотека БГУИР

к цепочке объектов. Так, если в массив чисел, на которые ссылается итератор вывода, вы попытаетесь добавить пару новых значений, то итератор вывода попросту запишет новые значения на место старых и они будут потеряны. Любой итератор вставки: front_inserter, back_inserter или inserter – выполнит ту же операцию вполне корректно. Первый из них добавляет объекты в начало цепочки, второй – в конец. Третий итератор вставляет объекты в заданное место цепочки. При всем удобстве итераторы вставки имеют довольно жесткие ограничения. Так, front_inserter и back_inserter не могут работать с наборами (set) и картами (map), а front_inserter к тому же не умеет добавлять данные в начало векторов (vector). Зато итератор вставки inserter добавляет объекты в любой контейнер. Одной интересной особенностью обладает front_inserter: данные, которые он вставляет в цепочку объектов, должны передаваться ему в обратном порядке.

Рассмотрим пример программы, в которой создается список (list) с двумя значениями, равными нулю. После этого в начало и конец списка добавляются значения 1, 2, 3. Третья последовательность 1-1-1 вставляется в середину списка между нулями. Итак, после описания заголовочных файлов мы создаем массивы, необходимые для работы, и контейнер типа «список» из двух элементов:

```
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;
main(void)
{
 int init[]  = {0, 0};
 int init1[] = {3, 2, 1};
 int init2[] = {1, 2, 3};
 int init3[] = {1, 1, 1};
 list<int> l(2);
```

Затем список инициализируется нулями из массива init и его значения отображаются на экране:

```
 copy(init, init + 2, l.begin());
 copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
 cout << " – before front_inserter" << endl;
```

Итератором вставки в начало списка в обратном порядке добавляются значения массива init1, и производится повторный показ данных из списка на экране:

```
 copy(init1, init1 + 3, front_inserter(l));
 copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
 cout << " – before back_inserter" << endl;
```

Теперь итератор вставки в конец добавит элементы массива init2 в «хвост» списка:

```
 copy(init2, init2 + 3, back_inserter(l));
 copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
```

Библиотека БГУИР

```
cout << " – before inserter" << endl;
```

Сложнее всего обстоит дело с итератором insertes. Для него, кроме ссылки на сам контейнер, нужен итератор, указывающий на тот объект в контейнере, за которым будет произведена вставка элементов массива init3. С этой целью мы создаем переменную типа «итератор», инициализируя ее итератором, указывающим на начало списка:

```
list<int>::iterator& itr = l.begin();
```

Теперь специальной операцией advance делаем приращение переменной итератора так, чтобы она указывала на четвертый объект в цепочке данных списка:

```
advance(itr, 4);
```

Остается добавить данные в цепочку посредством insertes и отобразить содержимое «списка» на дисплее:

```
copy(init3, init3 + 3, inserter(l, itr));
copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
cout << " – the end!" << endl;
}
```

Константный итератор. Последний итератор, который мы рассмотрим, – константный (constant iterator). Он образуется путем модификации основного итератора. Константный итератор не допускает изменения данных, на которые он ссылается. Можно считать константный итератор указателем на константу. Чтобы получить константный итератор, можно воспользоваться типом const_iterator, предопределенным в различных контейнерах. К примеру, так можно описать переменную типа константный итератор на список:

```
list<int>::const_iterator c_itr;
```

Операции с итераторами

Существуют две важных операции для манипуляции ими. С одной из них – advance – мы познакомились в последнем примере. Это просто удобная форма инкрементирования итератора itr на определенное число n:

```
void advance (InputIterator& itr, Distance& n);
```

Вторая операция измеряет расстояние между итераторами first и second, возвращая полученное число через ссылку n:

```
void distance(InputIterator& first, InputIterator& second, Distance& n);
```

Контейнерные классы

Контейнеры последовательностей

Стандартная библиотека C++ предоставляет три контейнера последовательностей – **vector**, **list** и **deque**. Первые два представляют собой классы, организованные по типу массивов, а последний реализует связанный список. Класс vector является одним из наиболее популярных контейнерных классов в STL, динамически изменяющим свои размеры.

Использование контейнера **vector** наиболее эффективно при добавлении элементов в конец контейнера. Для приложений, выполняющих частые вставки

Библиотека БГУИР

и удаления в конец и начало контейнера, более предпочтительным является **deque**. Если требуется выполнять вставки и удаление элементов в любое место контейнера, то обычно используется **list**.

Кроме перечисленных выше компонент-функций, общих для всех STL-контейнеров, контейнеры последовательностей имеют несколько особых: **front** и **back** – для возврата ссылки на первый и последний элемент в контейнере, **push_back** и **pop_back** – для вставки и удаления последнего элемента контейнера.

Контейнер последовательностей vector

Аналогично обычным массивам в C и C++ класс **vector** обеспечивает непрерывную область памяти для размещения данных. Следовательно, возможен доступ к любому элементу контейнера посредством индексации. В случае, если при добавлении новых элементов в контейнер объема выделенной памяти недостаточно, vector выделяет память большего размера. Выполняется копирование информации в выделенную область памяти и освобождение старой области памяти.

Контейнер vector поддерживает итераторы произвольного доступа, то есть все операции итераторов (табл. 4) могут быть применены к итератору контейнера vector. Все алгоритмы STL могут работать с контейнером vector.

В приводимой ниже программе демонстрируется использование некоторых компонент-функций класса vector.

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <vector>
template<class T>
void PrintVector(const std::vector<T> &vect);
main()
{ std::vector<int>  v,vv;
  PrintVector(v);
  v.push_back(2);
  v.push_back(5);
  v.push_back(7);
  v.push_back(1);
  v.push_back(9);
  v[4]=3;                // изменить значение 5-го элемента на 3
  v.at(3)=6;             // изменить значение 3-го элемента на 6
  try{ v.at(5)=0;        //
  }
  catch(std::out_of_range e){  //доступ к элементу вне массива (вектора)
    cout<<"\nИсключение : "<<e.what();
```

Библиотека БГУИР

```
 }
 PrintVector(v);
 v.erase(v.begin() + 2);     // удаление 3-го элемента (начальный индекс 0)
 PrintVector(v);
 v.insert(v.begin() + 3,7);  // добавление 7 после 3-го элемента вектора
 PrintVector(v);
 vv.push_back(6);
 v.swap(vv);              // замена массивов v и vv
 PrintVector(v);
 PrintVector(vv);
 vv.erase(vv.begin()+1,vv.end()-2); // удаление со 2-го по n-2 элементов
 PrintVector(vv);
 vv.clear();              // чистка всего вектора
 PrintVector(vv);
 return 0;
}
template<class T>
void PrintVector(const std::vector<T> &vect)
{ std::vector<T>::const_iterator pt;
 if (vect.empty())
   { cout << endl << "Vector is empty." << endl;
     return;
   }
 cout<<"ВЕКТОР   :"
          <<" Размер ="<<vect.size()
          <<"   вместимость ="<<vect.capacity()<<endl;
 cout<<"содержимое  :";
 for(pt=vect.begin();pt!=vect.end();pt++)
 cout<<*pt<<' ';
 cout<<endl;
}
```

Инструкция

```
std::vector<int>  v,vv;
```

определяет два экземпляра класса v и vv для хранения int чисел. При этом создаются два пустых контейнера с нулевым числом элементов и нулевой вместимостью (числом элементов, которые могут быть сохранены в контейнере). Компонента-функция push_back(), имеющаяся во всех контейнерах последовательностей, используется для добавления элемента в контейнер v. Инструкции

```
v[4]=3;
v.at(3)=6;
```

демонстрируют два способа индексирования контейнера vector (используются и для контейнера deque). Первая инструкция реализуется перегрузкой операции [], возвращающей либо ссылку на значение в требуемой позиции, либо константную ссылку на это значение, в зависимости от того, является ли контейнер

константным. Следующая функция at() выполняет то же, осуществляя дополнительно проверку на выход индекса из диапазона. Функции size() и capacity() возвращают число элементов вектора (диапазон) и его вместимость. Вместимость удваивается каждый раз в том случае, когда при попытке разместить очередной элемент в контейнере все выделенное пространство памяти занято.

При добавлении первого элемента размер стал равен 1 вместимость 1, второго – размер 2 вместимость 2, третьего – 3 и 4 соответственно, четвертого – 4 и 8 и т.д. В примере приведена внешняя функция просмотра вектора и его размера и вместимости PrintVector(). Для прохода по вектору используется итератор pt:

```
std::vector<T>::const_iterator pt;
```

Этот итератор (const_iterator) допускает считывание, но не модификацию элементов контейнера vector. При проходе по вектору используются функции begin() и end(), доступные для всех контейнеров первого класса.

Кроме перечисленных выше в программе используются функции класса vector: clear(), empty() и swap().

Контейнер vector не должен быть пустым, иначе результаты функций **front** и **back** будут не определены.

Контейнер последовательностей list

Контейнерный класс **list** реализуется как двусвязный список, что позволяет ему поддерживать двунаправленные итераторы для прохождения контейнера как в прямом, так и в обратном направлениях. Для контейнера list может быть использован любой алгоритм, использующий однонаправленные и двунаправленные итераторы. Контейнер последовательностей list эффективно реализует операции вставки и удаления в любое место контейнера.

Шаблон класса list предоставляет еще восемь функций-членов: **splice**, **push_front**, **pop_fron**t, **remove**, **unique**, **merge**, **reverse** и **sort**. Многие функции класса vector поддерживаются также и в классе list. Для работы с объектом (его компонентами) необходимо включать заголовочный файл <list>.

```
#include <iostream>
using std::cout;
using std::endl;
#include <list>
template<class T>
void PrintList(const std::list<T> &lst);
main()
{ std::list<int>  ls,ll;
  std::list<int>::iterator itr;
  int ms[]={2,5,3};
  int mm[]={2,15,23,1};
  ls.push_back(2);
  ls.push_front(5);
```

Библиотека БГУИР

```
 ls.push_front(7);
 ls.push_back(9);
 PrintList(ls);
 ll.insert(ll.begin(),ms,ms+sizeof(ms)/sizeof(int));// добавление
 PrintList(ll);
 ll.push_front(6);
 ls.splice(ls.end(),ll);
 PrintList(ls);
 PrintList(ll);
 ls.sort();              // сортировка ls
 PrintList(ls);
 ll.insert(ll.begin(),mm,mm+sizeof(mm)/sizeof(int));
 PrintList(ll);
 ll.sort();              // сортировка ll
 ls.merge(ll);           // перенос элементов ll в ls
 PrintList(ls);
 ls.pop_front();         // удаление первого элемента списка
 ls.pop_back();          // удаление последнего элемента списка
 ls.reverse();           // реверсивный переворот списка  ls
 PrintList(ls);
 ls.unique();            // удаление из списка ls одинаковых эл-тов
 PrintList(ls);
 ls.swap(ll);            // обмен содержимым списков ls и ll
 PrintList(ls);
 PrintList(ll);
 ls.push_front(2);
 ls.assign(ll.begin(),ll.end());// замена ls содержимым ll
 PrintList(ls);
 PrintList(ll);
 ls.remove(2);           // удаление из ls всех  2
 PrintList(ls);
 itr=ls.begin();
 itr++;
 ls.erase(itr,ls.end());// удаление из ls элементов с itr до ls.end
 PrintList(ls);
 return 0;
}
template<class T>
void PrintList(const std::list<T> &lst)
{ std::list<T>::const_iterator pt;
 if (lst.empty())
   { cout << endl << "List is empty." << endl;
     return;
   }
```
Библиотека БГУИР

```
 cout<<"Двусвязный список   = " <<" Размер     = "<<lst.size()<<endl;
 cout<<" Содержимое           = ";
 for(pt=lst.begin();pt!=lst.end();pt++)
 cout<<*pt<<' ';
 cout<<endl;
}
```

Рассмотрим некоторые конструкции, использованные в программе.

Используя функцию push_back (общую для всех контейнеров последовательностей) вставки чисел в конец ls и push_front (определенную для классов list и deque) добавления значений в начало ls, создаем последовательность целых чисел. В инструкции

```
ll.insert(ll.begin(),ms,ms+sizeof(ms)/sizeof(int));
```

используется функция insert класса list, вставляющая в начало последовательности ll элементы массива ms.

Далее в строке

```
ls.splice(ls.end(),ll);
```

используется компонента-функция splice класса list, выполняющая удаление элементов списка ll с одновременным переносом их в список ls до позиции итератора ls.end() – первый аргумент функции. В классе list имеются две другие версии этой функции:

```
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first, iterator last);
```

Функция с тремя аргументами позволяет удалить один элемент из контейнера, определенного в качестве второго элемента, начиная с позиции, определенной третьим аргументом. В функции с четырьмя элементами последние два параметра определяют диапазон удаляемых из контейнера (второй параметр) значений. Как и первая функция, два других удаляемых значения помещают в позицию, отмеченную итератором (первый аргумент).

Выполнение инструкции

```
ls.sort();
```

вызывает функцию sort() класса list, производящую упорядочивание элементов list по возрастанию.

После сортировки списков ls и ll выполняется функция

```
ls.merge(ll);
```

удаляющая все объекты контейнера ll и вставки их в отсортированном виде в контейнер ls (оба списка должны быть отсортированы в одном порядке).

Далее следуют функции pop_front – удаления элемента из начала последовательности и доступная во всех контейнерах последовательности функция pop_back – удаление из конца последовательности.

В строке

```
ls.unique();
```

используется функция класса list, выполняющая удаление элементов-дубликатов. При этом список должен быть отсортирован.

Использование далее функции

Библиотека БГУИР

```
ls.swap(ll);
```
доступной во всех контейнерах, приводит к обмену содержимым контейнеров ls и ll.

В строке программы
```
ls.assign(ll.begin(),ll.end());
```
использована функция для замены содержимого объекта ls содержимым объекта ll в диапазоне, определяемом двумя аргументами итераторами.

Строка
```
ls.remove(2);
```
содержит вызов функции remove, удаляющей из ls все копии значения 2.

И, наконец, в строке
```
ls.erase(itr,ls.end());
```
вызывается функция класса list, удаляющая из ls элементы с itr до ls.end.

Контейнер последовательностей deque

Класс **deque** объединяет многие возможности классов **vector** и **list**. Этот класс представляет собой двунаправленную очередь. В классе deque реализован механизм эффективного индексного доступа (подобно тому, как и в классе vector).

Класс deque реализован для эффективных операций вставки в начало и конец. Класс deque обеспечивает поддержку итераторов прямого доступа, что дает возможность использовать при работе с ним любых алгоритмов STL.

Класс deque имеет базовые функции, аналогичные классу vector, при этом в класс deque добавлены компоненты-функции push_front и pop_front.

```
#include <iostream>
using std::cout;
using std::endl;
#include <deque>
#include <algorithm>
template<class T>
void PrintDeque(const std::deque<T> &dq);

#define SIZE 6
main()
{ std::deque<float>  d,dd(3,1.5);
  PrintDeque(dd);
  d.push_back(2);
  d.push_front(5);
  d.push_back(7);
  d.push_front(9);
  d[4]=3;              // изменить значение 5-го элемента на 3
  d.at(3)=6;
  try{ d.at(5)=0;
  }
```

Библиотека БГУИР

```
  catch(std::out_of_range e){
    cout<<"\nИсключение : "<<e.what();
  }
  PrintDeque(d);
  d.erase(d.begin() + 2);   // удаление 3-го элемента (начальный индекс 0)
  PrintDeque(d);
  d.insert(d.begin() + 3,7);// добавление 7 после 3-го элемента вектора
  PrintDeque(d);
  d.insert(d.end()-1,2,1.6);
  PrintDeque(d);
  d.insert(d.end(),dd.begin(),dd.end());
  PrintDeque(d);
  std::sort(d.begin(),d.end());
  PrintDeque(d);
  d.swap(dd);              // замена массивов  v  и vv
  PrintDeque(d);
  PrintDeque(dd);
  dd.erase(dd.begin(),dd.end()); //удаление всех элементов dd
  PrintDeque(dd);
  d.clear();              // чистка всего вектора
  PrintDeque(d);
  return 0;
}
template<class T>
void PrintDeque(const std::deque<T> &dq)
{ std::deque<T>::const_iterator pt;
  if (dq.empty())
  { cout << endl << "Deque is empty." << endl;
    return;
  }
  cout<<"ОЧЕРЕДЬ   :"
       <<" Размер ="<<dq.size()<<endl;
  cout<<"содержимое  :";
  for(pt=dq.begin();pt!=dq.end();pt++)
  cout<<*pt<<' ';
  cout<<endl;
}
```

В приведенном примере использованы все ранее рассмотренные функции классов vector и list, также являющиеся компонентами класса deque. В программе были использованы все три версии функции insert:

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
void insert(iterator it, const_iterator first, const_iterator last);
```

Первая версия функции предназначена для вставки после элемента, на ко-

Библиотека БГУИР

торый указывает итератор, значения, соответствующего второму параметру. Вторая версия вставляет n значений, равных третьему параметру. Наконец, третья версия вставляет значения из интервала от второго аргумента (итератора) до третьего.

Ассоциативные контейнеры

Ассоциативные контейнеры предназначены для обеспечения прямого доступа посредством использования ключей. В STL имеется четыре ассоциативных контейнерных класса: **multiset**, **set**, **multimap** и **map**. Во всех контейнерах ключи отсортированы. Классы **multiset** и **set** манипулируют множествами значений, одновременно являющихся ключами. При этом **multiset** допускает одинаковые ключи, а **set** нет. Классы **multimap** и **map** манипулируют множествами значений, ассоциируемых с ключами. При этом **multimap** допускает хранение одинаковых ключей с ассоциированными значениями, а **map** нет.

Ассоциативный контейнер multiset

Ассоциативный контейнер **multiset** обеспечивает быстрое сохранение и выборку ключей. Упорядочение элементов контейнера определяется компараторным объектом-функцией less<тип>, при этом отсортированные ключи должны поддерживать сравнение с помощью operator<, иначе (для пользовательских типов) необходимо перегружать операцию сравнения.

Класс multiset поддерживает двунаправленные итераторы (но не итераторы произвольного доступа).

```
#include <iostream>
using std::cout;
using std::endl;
#include <set>
#include <algorithm>
typedef std::multiset<int,std::less<int> > intMSET;
#define size 10
main()
{ int mas[]={2,4,1,6,19,17,1,7,17,14};
  intMSET mset;
  std::ostream_iterator<int> out(cout," ");
  intMSET::const_iterator res;
  cout<<"элемент 8 содержится в multiset " << mset.count(8)<<" раз\n";
  mset.insert(8);      //
  mset.insert(8);
  cout<<"содержимое multiset :";
  std::copy(mset.begin(),mset.end(),out);
  res=mset.find(6);
  cout<<'\n' ;
  if(res!=mset.end())
         cout<<"найдено значение  6\n";
```

```
    else cout<<"не найдено значение  6\n";
    mset.insert(mas,mas+sizeof(mas)/sizeof(int));      //
    cout<<"содержимое multiset :" ;
    std::copy(mset.begin(),mset.end(),out);
    cout<<"\nнижняя  граница 17  " << *(mset.lower_bound(17));
    cout<<"\nверхняя граница 17  " << *(mset.upper_bound(17));
    std::pair<intMSET::const_iterator, intMSET::const_iterator> pr;
    pr=mset.equal_range(17);
    cout<<"\nнижняя  граница 17  " << *(pr.first);
    cout<<"\nверхняя граница 17  " << *(pr.second);
    return 0;
   }
```

В приведенной программе использованы следующие компоненты контейнерного класса multiset:

mset.count(8) – функция, доступная во всех ассоциативных контейнерах, возвращает число вхождений значения 8 в multiset. Затем в программе использованы две из трех версий функции insert:

```
mset.insert(8);
mset.insert(mas,mas+sizeof(mas)/sizeof(int));
```

первая из двух функций insert вставляет значение 8 во множество, а вторая – числа из интервала.

Далее используются функции lower_bound(17) и upper_bound(17) (доступные во всех ассоциативных контейнерах) для определения позиции первого вхождения числа 17 во множество и позиции элемента после последнего вхождения. Обе функции возвращают iterator или const_iterator соответствующих позиций, или итератор, возвращаемый функцией end.

В строке

```
std::pair<intMSET::const_iterator, intMSET::const_iterator> pr;
```

создается объект класса pair. Объекты класса pair используются для связывания пар значений. Объект pr используется для сохранения в нем значения pair, возвращаемого функцией equal_range и содержащего результаты lower_bound() и upper_bound(). Тип pair содержит две открытые компоненты с именами first и second. Для доступа к lower_bound() и upper_bound используются pr.first и pr.second.

Ассоциативный контейнер set

Контейнерный класс **set** используется для обеспечения быстрого сохранения и доступа к уникальным ключам. При попытке поместить в контейнер set дубликата ключа это действие игнорируется без идентификации ошибки. Контейнер set поддерживает двунаправленные итераторы. Работа с контейнером set может быть продемонстрирована на предыдущем примере, если в нем строку

```
typedef std::multiset<int,std::less<int> > intMSET;
```

заменить на строку

```
typedef set<int,std::less<int> > intMSET;
```

Библиотека БГУИР

что приведет далее к созданию и работе с объектами класса set.

Ассоциативный контейнер multimap

Ассоциативный контейнер **multimap** эффективен для быстрого сохранения и нахождения ключей и ассоциированных с ними значений. Многие методы, используемые в контейнерах set и multiset, применимы к контейнерам map и multimap.

Элементами multimap и map являются объекты **pair** – пары ключей и соответствующих им значений. Порядок сортировки ключей в контейнере определяется компараторным объектом-функцией **less<тип>**. В контейнере multimap допускается дублирование ключей. Это означает, что несколько значений могут быть ассоциированы с одним ключом (отношение ”*один ко многим*”). Например, ученик изучает много предметов, один человек может иметь несколько банковских счетов и т.д.

Контейнер multimap поддерживает двунаправленные итераторы. Для работы с контейнерным классом multimap необходимо подключить заголовочный файл <map>. Рассмотрим пример использования контейнера multimap.

```
#include <iostream>
using std::cout;
using std::endl;
#include <map>
#include <algorithm>
typedef float T1;
typedef float T2;
typedef std::multimap<T1,T2,std::less<T1> > MUL_MAP;
T1 key;
main()
{ MUL_MAP mmap,mm;
  MUL_MAP::const_iterator itr;
  MUL_MAP::value_type pr;        // элемент типа pair
  key=3.1;
  cout<<"пар с ключом "<<key<<" в multimap " << mmap.count(key)<<
        " раз"<<endl;
  mmap.insert(MUL_MAP::value_type(key,1.1));
  mmap.insert(MUL_MAP::value_type(key,2.2));
  cout<<"пар с ключом "<<key<<" в multimap " << mmap.count(key)<<
        " раз"<<endl;
  mmap.insert(MUL_MAP::value_type(5,12));
  mmap.insert(MUL_MAP::value_type(1,20.12));
  mmap.insert(MUL_MAP::value_type(12,20.12));
  mmap.erase(5);
  cout<<"SIZE= "<<mmap.size()<<"  MAX_SIZE= "<<mmap.max_size()<<
        endl;
  for(itr=mmap.begin();itr!=mmap.end();itr++)
```

Библиотека БГУИР

```
            cout<<itr->first<<'\t'<<itr->second<<'\n';
    cout<<"нижняя граница "<<key<<'\t'<<(mmap.lower_bound(key)->first);
    cout<<"верхняя граница "<<key<<'\t'<<(mmap.upper_bound(key)->first);
    itr=mmap.find(key);
    cout<<'\n' ;
    if(itr!=mmap.end())
           cout<<"найден ключ "<<itr->first<<'\t'<<itr->second<<'\n';
    else cout<<"не найден ключ  "<<key<<'\n';
    mmap.clear();
    return 0;
}
```

В строках:

```
typedef float T1;
typedef float T2;
typedef std::multimap<T1,T2,std::less<T1> > MUL_MAP;
```

используя typedef, типам float и double назначаются псевдонимы T1 и T2 и экземпляру шаблонного класса multimap псевдоним MUL_MAP.

Компонента-функция mmap.count(key) возвращает число пар ключа key, содержащихся в multimap. Далее следуют функции insert для ввода пар ключей и соответствующих значений в контейнер.

```
mmap.insert(MUL_MAP::value_type(5,12));
```

В этой инструкции используется функция value_type(5,12), создающая объект pair, в котором first – это ключ (5) типа T1, а second – значение (12) типа T2.

В цикле for выводится содержимое контейнерного класса multimap (ключи и значения). Для доступа к компонентам pair элементов контейнера используется const_iterator itr. При этом ключи выводятся в порядке возрастания.

```
for(itr=mmap.begin();itr!=mmap.end();itr++)
cout<<itr->first<<'\t'<<itr->second<<'\n';
```

Для вывода нижней и верхней границ ключа key в контейнере используются

```
cout<<"нижняя граница "<<key<<'\t'<<(mmap.lower_bound(key)->first);
cout<<"верхняя граница "<<key<<'\t'<<(mmap.upper_bound(key)->first);
```

функции lower_bound() и upper_bound(), возвращающие итератор соответствующего элемента pair.

Ассоциативный контейнер map

Контейнерный класс **map** используется для обеспечения быстрого сохранения и доступа к уникальным ключам и соответствующих значений. При этом между ними устанавливается взаимно однозначное соответствие. Попытка поместить в контейнер map дубликат ключа игнорируется без идентификации ошибки. Контейнер map поддерживает двунаправленные итераторы. Работа с контейнером map может быть продемонстрирована на предыдущем примере, если в нем строку

```
typedef std::multimap<T1,T2,std::less<T1> > MUL_MAP;
```

Библиотека БГУИР

заменить на строку

typedef std:map<T1,T2,std::less<T1> > MUL_MAP;

что приведет далее к созданию и работе с объектами класса map.

Адаптеры контейнеров

В состав STL входят три адаптера контейнеров – **stack**, **queue** и **priority_queue**. Адаптеры не предоставляют реализации фундаментальной структуры данных и не поддерживают работу с итераторами. Это отличает их от контейнеров первого класса. Преимущество класса адаптеров состоит в возможности выбирать требуемую базовую структуру данных. Все три класса адаптеров содержат компоненты-функции **push** и **pop**, реализуемые посредством вызова соответствующих функций базового класса.

Адаптеры stack

Класс **stack** обеспечивает возможность вставки и удаления данных в базовой структуре с одной стороны. Адаптер **stack** может быть реализован с любым из контейнеров последовательностей: **vector**, **list** и **deque** (по умолчанию реализуется с контейнером deque). Для класса stack определены следующие операции (реализуемые через соответствующие функции базового контейнера): **push** – помещение элемента на вершину стека, **pop** – удаление элемента с вершины стека, **top** – получение ссылки на вершину стека, **empty** – проверки на пустоту стека и **size** – получение числа элементов стека.

```
#include <iostream>
using std::cout;
using std::endl;
#include <stack>
#include <vector>
#include <list>
typedef char T;
template<class E>
void popElement(E &e)
{ while(!e.empty())          // пока стек не пустой
  { cout<<e.top()<<' ';      // получение значения элемента на вершине стека
    e.pop();                 // удаление элемента с вершины стека
  }
}
main()
{ std::stack <T> deque_st;                    // стек на основе deque
  std::stack <T, std::vector<T> > vect_st;    // стек на основе vector
  std::stack <T, std::list<T> > list_st;      // стек на основе list
  char c='a';
  for(int i=0;i<5;i++)             // занесение в стеки
  { deque_st.push(c++);
    vect_st.push(c++);
    list_st.push(c++);
```

Библиотека БГУИР

```
  }
  cout << "\nСтек deque  :";
  popElement(deque_st);
  cout << "\nСтек vector :";
  popElement(vect_st);
  cout << "\nСтек list   :";
  popElement(list_st);
  cout<<endl;
  return 0;
}
```

Результат работы программы:
Стек deque : m j g d a
Стек vector :   n k h e b
Стек   list   :   o l i f c

В первых трех строках функции main создаются три стека для хранения символов, использующие в качестве базовых структур контейнеры deque (по умолчанию), vector и list соответственно. Далее в программе использована функция push для помещения элементов на вершину соответствующего стека.

Реализованная в программе template функция выводит на экран удаляемый с вершины стека элемент. Для этого использованы функции top – нахождения (но не удаления) элемента на вершине стека и pop – удаления его с вершины стека.

Адаптеры queue

Класс **queue** предназначен для вставки элементов в конец базовой структуры данных и удаления элементов из ее начала. Адаптер **queue** реализуется с контейнерами **list** и **deque** (по умолчанию).

Наряду с общими для всех классов адаптеров операциями **push**, **pop**, **empty** и **size** в классе queue имеются операции **front** – получения ссылки на первый элемент очереди, **back** – ссылки на последний элемент очереди.

```
#include <iostream>
using std::cout;
using std::endl;
#include <queue>
// #include <list>                          для базового контейнера list
typedef int T;
main()
{ std::queue <T> och;
 // std::queue <T, std::list<T> > lst;      очередь на основе контейнера list
  och.push(1);                         // занесение в очередь
  och.push(2);
  och.push(3);
  int k=och.size();                    //  количество элементов в очереди
  cout << "Очередь  : ";
```

```
  if(och.empty()) cout<<" пустая"; // проверка на пустоту очереди
  else
  { for(int i=0;i<k;i++)
    { cout<<och.front()<<' ';     // вывод значения первого элемента очереди
      och.pop();                  // удаление из очереди первого элемента
    }
  }
  cout<<endl;
  return 0;
}
```

Результат работы программы:
Очередь : 1 2 3

Инструкция
std::queue <T> och;
создает очередь для хранения в ней значений типа T. Очередь создается на основе контейнера deque по умолчанию. Функция front считывает значение первого элемента очереди, не удаляя его из нее. Для этого используется функция pop.

Адаптеры priority_queue

Класс **priority_queue** используется для вставки элементов в отсортированном порядке в базовую структуру данных и удаления элементов из ее начала. Адаптер **priority_queue** реализуется с контейнерами **vector** (по умолчанию) и **deque**.

Элементы в очередь с приоритетом заносятся в соответствии со своим значением. Это означает, что элемент с максимальным значением помещается в начало очереди и будет первым из нее удален, а с минимальным – в конец очереди. Это достигается с помощью метода, называемого *сортировкой кучи*. Сравнение элементов выполняется функцией-объектом less<Тип> или другой компораторной функцией.

Как и предыдущие адаптеры, priority_queue использует операции push, pop, empty, size, а также операцию top – получения ссылки на элемент с наивысшим приоритетом.

Пассивные и активные итераторы

Можно определять итерацию как часть объекта или создавать отдельные объекты, ответственные за итеративный опрос других структур. К достоинствам второго подхода относятся:

– наличие выделенного итератора классов, позволяющего одновременно проводить несколько просмотров одного и того же объекта;

– наличие итерационного механизма в самом классе, что несколько нарушает его инкапсуляцию, выделение итератора в качестве отдельного механизма поведения способствует достижению большей ясности в описании класса.

Библиотека БГУИР

Для каждой структуры определены две формы итераций. **Активный итератор** требует каждый раз от клиента явного обращения к себе для перехода к следующему элементу. **Пассивный итератор** применяет функцию, предоставляемую клиентом, и, таким образом, требует меньшего вмешательства клиента. С целью обеспечения безопасности типов для каждой структуры создаются свои итераторы.

```
template<class Item>
class ActivIterator
{ public:
    ActivIterator (const Queue<Item>&);
    ~ActivIterator();
    void reset();
    int next();
    int isDone() const;
    const Item* currentItem() const;
  protected:
    const Queue<Item>& queue;
    int index;
};
```

Каждому итератору ставится в соответствие определенный объект.

Используя функцию currentItem(), клиент может получить доступ к текущему элементу. Если итерация завершена или последовательность структур пуста, то возвращается нулевое значение. Переход к следующему элементу последовательности выполняется функцией next(), возвращающей 0, если итерация завершена. Функция isDone() возвращает информацию о состоянии процесса (0 – если итерация завершена или структура пуста). Функция reset() позволяет осуществлять неограниченное число итерационных проходов по объекту. Например, при наличии объявления:

```
BoundedQueue<NetworkEvent> eventQueue;
```

Фрагмент кода, использующий активный итератор для прохода по элементам очереди, будет выглядеть так:

```
QueueActiveIterator<NetworkEvent> iter(eventQueue);
while(!iter.isDone())
{  iter.currentItem()->dispatch();
   iter.next();
}
```

Конструктор класса QueueActiveIterator сначала устанавливает связь между итератором и конкретной очередью. Затем вызывается защищенная функция cardinality, которая определяет количество элементов в очереди. Конструктор может иметь вид:

```
template<class Item>
QueueActiveIterator<Item> ::QueueActiveIterator(const Queue<Item>& q)
:queue(q), index(q,cardinality()? 0 : -1;) {}
```

Библиотека БГУИР

Класс QueueActiveIterator имеет доступ к защищенной функции cardinality класса Queue, являясь дружественным ему.

Операция итератора isDone проверяет принадлежность текущего индекса допустимому диапазону, определяемому количеством элементов очереди.

```
template<class Item>
int QueueActiveIterator<Item> ::isDone() const
{ return((index<0 || (index>=queue.cardinality())); }
```

Функция currentItem возвращает указатель на элемент, на котором остановился итератор. Реализация итератора в виде индекса объекта в очереди дает возможность в процессе итерации добавлять и удалять элементы из очереди.

```
template<class Item>
const Item* QueueActiveIterator<Item> ::currentItem() const
{ return isDone()? 0 : &queue.itemAt(index); }
```

При выполнении операции итератор вновь вызывает защищенную функцию itemAt. Операция next имеет вид:

```
template<class Item>
int QueueActiveIterator<Item> ::next()
{ index++;
  return !isDone();
}
```

Пассивный итератор, который также называют *аппликатором*, характеризуется тем, что он применяет определенную функцию к каждому элементу структуры. Для класса Queue пассивный итератор можно определить следующим образом:

```
template <class Item>
class QueuePassiveIterator
{public:
    QueuePassiveIterator(const Queue<Item>&);
    ~QueuePassiveIterator();
    int apply(int (*) (const Item&));
 protected:
    const Queue<Item>& queue
};
```

Пассивный итератор действует на все элементы структуры за (логически) одну операцию. Таким образом, функция apply() последовательно производит одну и ту же операцию над каждым элементом структуры, пока передаваемая итератору функция не возвратит нулевое значение или пока не будет достигнут конец структуры. В первом случае функция apply() сама возвратит нулевое значение в знак того, что итерация не была завершена.

Алгоритмы

Более ранние библиотеки контейнеров для реализации алгоритмов обычно использовали наследование и полиморфизм, что влекло за собой потерю

Библиотека БГУИР

производительности, связанную с вызовом виртуальных функций. Алгоритмы были встроены в контейнерные классы. В STL алгоритмы отделены от контейнеров, что упрощает расширение их числа. Доступ к элементам контейнеров в STL осуществляется посредством итераторов. STL-алгоритмы используют итераторы в качестве аргументов (наряду с этим STL-алгоритмы также могут работать с любыми массивами языка С на основе указателей).

Каждый алгоритм использует итераторы определённого типа. Например, алгоритм простого поиска (find) просматривает элементы подряд, пока нужный не будет найден. Для такой процедуры вполне достаточно итератора ввода. С другой стороны, алгоритм более быстрого двоичного поиска (binary_search) должен иметь возможность переходить к любому элементу последовательности и поэтому требует итератора с произвольным доступом. Вполне естественно, что вместо менее функционального итератора можно передать алгоритму более функциональный, но не наоборот.

Все стандартные алгоритмы описаны в заголовочном файле algorithm, в пространстве имён std.

Ниже при описании прототипов некоторых алгоритмов будем использовать следующие обозначения итераторов:

OutIt – iterator вывода;
InIt – iterator ввода;
FwdIt – однонаправленный iterator;
BidIt – двунаправленный iterator;
RanIt – iterator прямого доступа.

Алгоритмы сортировки sort, partial_sort, sort_heap

Для сортировки данных в STL имеются различные алгоритмы. Рассмотрим некоторые из них.

Алгоритм **sort** является алгоритмом обычной сортировки, единственным ограничением которого является то, что он используется для контейнеров, поддерживающих итераторы произвольного доступа. Прототипы функции sort приведены ниже.

```
template<class RanIt>
   void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
   void sort(RanIt first, RanIt last, Pred pr);
```

Действие первого из них основано на использовании перегруженной операции operator<() для сортировки данных по возрастанию. Второй вариант заменяет операцию сравнения функцией сравнения pr(x,y), тем самым позволяя сортировать данные в порядке, определяемом пользователем.

```
#include <vector>
   #include <iostream>
   #include <algorithm>
   using namespace std;
   void Print(int x)
```

Библиотека БГУИР

```
{ cout << x <<' '; }
int main()
{ vector<int> v(4);
  v[0] = 3;
  v[1] = 1;
  v[2] = 5;
  v[3] = 2;
  sort(v.begin(), v.end() );
  for_each(v.begin(), v.end(), Print);
  return 0;
}
```

Результатом работы программы будет массив
1 2 3 5

Использованный в программе алгоритм **for_each** полезен тогда, когда надо произвести перебор всех элементов контейнера и выполнить некоторое действие для каждого из них.

Для использования алгоритма sort с тремя параметрами требуется в качестве третьего аргумента использовать указатель на функцию или функциональный объект. Например, для сортировки в обратном порядке требуется включить заголовок<functional>:

```
sort(v.begin(), v.end(), greater<int>() );
```

Алгоритм **partil_sort** предназначен для сортировки только части массива.

Алгоритм **sort_heap** предназначен для сортировки накопителя.

Алгоритмы поиска find, find_if, find_end, binary_search

В STL имеется несколько алгоритмов, выполняющих поиск в контейнере. Приводимая ниже программа демонстрирует возможности этих алгоритмов.

```
#include <vector>
#include <iostream>
#include <algorithm>
#define T int
using namespace std;
bool fun(T i){return i%2==0;}
int main()
{ T m1[]={5,3,4,7,3,12};
   std::vector<T> v1(m1,m1+sizeof(m1)/sizeof(T));
   std::ostream_iterator<T> out(cout," ");
   std::vector<T>::iterator itr;
   cout<<"вектор v1 : ";
   std::copy(v1.begin(),v1.end(),out);
   itr=std::find(v1.begin(),v1.end(),5);
   cout<<"\nзначение 5 ";
   if(itr!=v1.end()) cout<<"найдено в позиции "<<itr-v1.begin()<<endl;
```

Библиотека БГУИР

```
    else cout<<"не найдено\n";
    itr=std::find_if(v1.begin(),v1.end(),fun);
    if(itr!=v1.end()) cout<<"первый четный элемент вектора v1["<<
                             itr-v1.begin()<<"]= "<<*itr<<endl;
    else cout<<"четные элементы в векторе v1 отсутствуют\n";
   // std::sort(v1.begin(),v1.end());                    // необходимо выполнить
    if(std::binary_search(v1.begin(),v1.end(),3))       // сортировку вектора
         cout<<"число 3 найдено в векторе v1\n"; // для   binary_search
    else cout<<"число 3 не найдено в векторе v1\n";
    return 0;
}
```

В приведенной программе использован алгоритм find, выполняющий поиск в векторе v1 значения 5.

```
itr=std::find(v1.begin(),v1.end(),5);
```

Далее в программе использована функция find_if нахождения первого значения вектора v, для которого унарная предикатная функция fun возвращает true:

```
itr=std::find_if(v1.begin(),v1.end(),fun);
```

Каждый из алгоритмов find и find_if возвращает итератор ввода на найденный элемент либо (если элемент не найден) итератор, равный v.end(). Аргументы find и find_if должны быть, по крайней мере, итераторами ввода.

В строке:

```
if(std::binary_search(v1.begin(),v1.end(),3))
```

для поиска значения 3 в векторе v1 использована функция binary_search. При этом последовательность элементов вектора в анализируемом диапазоне должна быть отсортирована в возрастающем порядке. Функция возвращает значение bool. В STD имеется вторая версия алгоритма binary_search, имеющая четвертый параметр, – бинарная предикатная функция, возвращающая true, если два сравниваемых элемента упорядочены.

Алгоритмы fill, fill_n, generate и generate_n

Алгоритмы данной группы предназначены для заполнения определенным значением некоторого диапазона элементов контейнера. При этом алгоритмы generate и generate_n используют порождающую функцию. Порождающая функция не принимает никаких аргументов и возвращает значение, заносимое в элемент контейнера.

Прототип функций fill, fill_n имеет вид:

```
template<class FwdIt, class T>
   void fill(FwdIt first, FwdIt last, const T& x);
template<class OutIt, class Size, class T>
   void fill_n(OutIt first, Size n, const T& x);
```

Пример программы, использующей алгоритм generate, приведен ниже.

```
#include <iostream>
#include <vector>
```

Библиотека БГУИР

```
#include <algorithm>
using namespace std;

// функция нахождения чисел Фибоначчи
int Fibonacci(void)
{  static int r;
   static int f1 = 0;
   static int f2 = 1;
   r = f1 + f2 ;
   f1 = f2 ;
   f2 = r ;
   return f1 ;
}

void main()
{   const int v_size = 8 ;
    typedef vector<int > vect;
    typedef vect::iterator vectIt ;
    vect numb(v_size);       // вектор, содержащий числа
    vectIt start, end, it ;
    start = numb.begin() ;   // позиция первого элемента
    end = numb.end() ;       // позиция последнего эл-та
    // заполнение [first, last +1) числами Фибоначчи
    // используя функцию Fibonacci()
    generate(start, end, Fibonacci) ;
    cout << "numb { " ;   // вывод содержимого
    for(it = start; it != end; it++)
        cout << *it << " " ;
    cout << " }\n" << endl ;
}
```

Алгоритмы equal, mismatch и lexicographical_compare

Алгоритмы данной группы используются для выполнения сравнения на равенство последовательностей значений.

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{ int m1[]={2,3,5,7,12};
  int m2[]={2,3,55,7,12};
  std::vector<int> v1(m1,m1+sizeof(m1)/sizeof(int)),
                   v2(m2,m2+sizeof(m2)/sizeof(int)),
                   v3(m1,m1+sizeof(m1)/sizeof(int));
  bool res=equal(v1.begin(), v1.end(),v2.begin());
```

Библиотека БГУИР

```
    cout<<"\nВектор v1 "<<(res?"":" не ")<<" равен вектору v2";
    res=equal(v1.begin(), v1.end(),v3.begin());
    cout<<"\nВектор v1 "<<(res?"":" не ")<<" равен вектору v3";
    std::pair<std::vector<int>::iterator,
                        std::vector<int>::iterator> pr;
    pr=std::mismatch(v1.begin(), v1.end(),v2.begin());
    cout<<"\nv1  и  v2 имеют различие в позиции "
            <<(pr.first-v1.begin())<<" где v1= "<<*pr.first
            <<" а v2= "<<*pr.second<<'\n';
    char s1[]="abbbw", s2[]="hkc";
    res=std::lexicographical_compare(s1,s1+sizeof(s1)/sizeof(char),
                                   s2,s2+sizeof(s2)/sizeof(char));
    cout<<s1<<(res?" меньше ":" не меньше ")<<s2<<'\n';
    return 0;
  }
```

В строке

```
bool res=equal(v1.begin(), v1.end(),v2.begin());
```

для сравнения двух последовательностей чисел на равенство используется алгоритм **equal**, получающий в качестве аргументов три итератора (по крайней мере для чтения). Если последовательности неравной длины или их элементы не совпадают, то equal возвращает false (используя функцию operator==).

Имеется также версия equal, принимающая четвертым параметром предикатную функцию, получающую два сравниваемых элемента и возвращающую значение типа bool. Это может быть полезно для последовательностей объектов или указателей на сравниваемые значения.

Алгоритм **mismatch** возвращает пару итераторов для двух сравниваемых последовательностей (v1 и v2), указывающих позиции, где элементы различаются:

```
std::pair<std::vector<int>::iterator,
std::vector<int>::iterator> pr;
pr=std::mismatch(v1.begin(), v1.end(),v2.begin());
```

Для определения позиции, в которой векторы различаются, требуется выполнить pr.first-v1.begin(). Согласно арифметике указателей это соответствует числу элементов от начала вектора v1 до элемента, отличного от соответствующего элемента вектора v2.

Алгоритм **lexicographical_compare** использует четыре итератора (по крайней мере для чтения) для сравнения, обычно строк. Если элемент первой последовательности (итерируемый первыми двумя итераторами) меньше элемента второй (два последних итератора), то функция возвращает true, иначе false.

Математические алгоритмы

Приводимая ниже программа демонстрирует использование нескольких математических алгоритмов: random_shuffle, count, count_if, min_element,

Библиотека БГУИР

max_element, accumulate и transform.

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>
using namespace std;
// возвращает целое число в диапазоне 0 – (n – 1)
int Rand(int n)
{ return rand() % n ; }

void main()
{ const int v_size = 8 ;
  typedef vector<int > vect;
  typedef vect::iterator vectIt ;
  vect Numbers(v_size) ;
  vectIt start, end, it ;
                        // инициализация вектора
  Numbers[0] = 4 ;   Numbers[1] = 10;
  Numbers[2] = 70 ;  Numbers[3] = 4 ;
  Numbers[4] = 10;   Numbers[5] = 4 ;
  Numbers[6] = 96 ;  Numbers[7] = 100;
  start = Numbers.begin() ;  // location of first
  end = Numbers.end() ;      // one past the location
  cout << "До выполнения random_shuffle:" << endl ;
  cout << "Numbers { " ;
  for(it = start; it != end; it++)
  cout << *it << " " ;
  cout << " }" << endl ;
  random_shuffle(start, end,pointer_to_unary_function<int, int>(Rand));
  cout << "После выполнения random_shuffle:" << endl ;
  cout << "Numbers { " ;
  for(it = start; it != end; it++)
    cout << *it << " " ;
  cout << " }" << endl ;
  cout<<"число 4 встречается"<<count(start, end,4)<<" раз"<<endl;
}
```

Результат работы программы:
До выполнения random_shuffle
Numbers {4 10 70 4 10 4 96 100}
После выполнения random_shuffle
Numbers {10 4 4 70 96 100 4 10}
число 4 встречается 3 раза

Библиотека БГУИР

Кратко охарактеризуем данные алгоритмы:

random_shuffle – имеется две версии функции для расположения в произвольном порядке чисел в диапазоне, определяемом аргументами-итераторами;

count, count_if – используются для подсчета числа элементов с заданным значением в диапазоне;

min_element, max_element – нахождение min- и max- элемента в диапазоне;

accumulate – суммирование чисел в диапазоне;

transform – применение общей функции к каждому элементу вектора.

Алгоритмы работы с множествами

Манипуляция множествами отсортированных значений выполняется алгоритмами: includes, set_difference, set_intersection, set_symmetric_difference и set_union. Приводимая ниже программа показывает пример использования алгоритма includes, проверяющего, входят ли элементы последовательности Deque в вектор Vector.

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include <deque>
using namespace std;
void main()
{  const int VECTOR_SIZE = 5 ;
   vector<string > Vector(VECTOR_SIZE) ;
   vector<string >::iterator start1, end1, it1;
   deque<string > Deque ;
   deque<string >::iterator start2, end2, it2 ;
    Vector[0] = "Коля";         // инициализация вектора
    Vector[1] = "Аня";
    Vector[2] = "Сергей";
    Vector[3] = "Саша";
    Vector[4] = "Вася";
    start1 = Vector.begin() ;    // итератор на начало Vector
    end1 = Vector.end() ;        // итератор на конец Vector
    Deque.push_back("Сергей") ;  // инициализация последовательности
    Deque.push_back("Аня") ;
    Deque.push_back("Саша") ;
    start2 = Deque.begin() ;     // итератор на начало Deque
    end2 = Deque.end() ;         // итератор на конец Deque
    sort(start1, end1) ;         //сортировка Vector и Deque
    sort(start2, end2) ;
                      // вывод содержимого  Vector и Deque
```

Библиотека БГУИР

```
    cout << "Vector { " ;
    for(it1 = start1; it1 != end1; it1++)
        cout << *it1 << ", " ;
    cout << " }\n" << endl ;
    cout << "Deque { " ;
    for(it2 = start2; it2 != end2; it2++)
        cout << *it2 << ", " ;
    cout << " }\n" << endl ;
    if(includes(start1, end1, start2, end2) )
        cout << "Vector включает Deque" << endl ;
    else
        cout << "Vector не включает Deque" << endl ;
}
```

Несколько слов о других алгоритмах:

set_difference – определяются элементы из первого множества, не входящие во второе;

set_intersection – противоположный предыдущему алгоритму;

set_symmetric_difference – выделяются элементы, которые содержатся только в одном из двух множеств;

set_union – объединение двух множеств в одно.

Алгоритмы swap, iter_swap и swap_ranges

Данная группа алгоритмов предназначена для выполнения перестановки элементов контейнера. Ниже приведены прототипы данных алгоритмов:

```
template<class T, class A>
    void swap(const vector <T, A>& lhs,const vector <T, A>& rhs);
```

Аргументами алгоритма swap являются ссылки на элементы для замены

```
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
```

Аргументами алгоритма являются два прямых итератора на элементы.

```
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
```

алгоритм swap_ranges используется для перестановки элементов от first до last с элементами начиная с x.

Все указанные замены производятся в одном и том же контейнере.

Алгоритмы copy, copy_backward, merge, unique и reverse

Дадим краткую характеристику алгоритмам копирования:

copy_backward – используется для копирования элементов из одного вектора в другой;

merge – используется для объединения двух отсортированных последовательностей в третью;

unique – исключает дублирование элементов в последовательности;

reverse – используется для перебора элементов в обратном порядке.

Библиотека БГУИР

Примеры реализации контейнерных классов

Связанные списки

Связанные списки – это способ размещения данных, при котором одни данные ссылаются на другие. Списки представляют собой пример контейнерных классов. В общем, список напоминает массив с тем отличием, что его размеры не фиксированы: он может расти и уменьшаться по мере работы с ним. Очередной элемент списка размещается в памяти динамически и связывается с остальной частью списка посредством указателей. В случае если некоторый элемент списка более не нужен, то освобождается память, отведенная под этот элемент. Частным случаем списков являются стеки, очереди и кольца.

Реализация односвязного списка

Односвязный список предполагает организацию хранения данных в памяти, при которой перемещение может быть выполнено только в одном направлении (от начала списка в конец). Расположение в памяти элементов односвязного списка можно изобразить следующим образом (рис. 9).

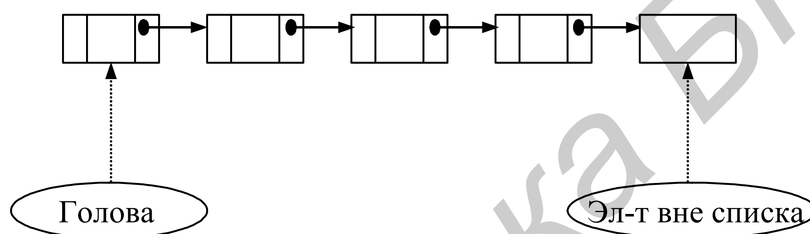

Рис. 9. Односвязный список

В настоящем пособии реализация односвязного списка не приводится (предлагается выполнить самостоятельно, основываясь на информации о реализации двусвязного списка, приводимого ниже).

Реализация двусвязного списка

Двусвязный список – это организация хранения данных в памяти, позволяющая выполнять перемещение в обоих направлениях (от начала списка в конец и наоборот). Расположение в памяти двусвязного списка можно изобразить следующим образом (рис. 10).

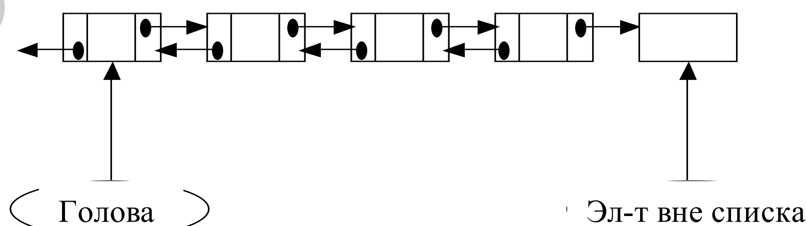

Рис. 10. Двусвязный список

Библиотека БГУИР

В приведенном ниже примере для доступа к элементам списка используется разработанный класс, реализующий простые итераторы.

```
#include <iostream>
#include <cassert>
using namespace std;
template <class T>
class D_List
{private:
  class D_Node
  { public:
      D_Node *next;   // указатель на следующий узел
      D_Node *prev;   // указатель на предыдущий узел
      T val;          // поле данного

      D_Node(T node_val) : val(node_val) { } // конструктор
      D_Node() {}                 // конструктор
      ~D_Node(){}                 // деструктор

     // для вывода элементов, тип которых определен пользователем,
     // неодходимо перегрузить операцию <<      operator<<( ).
    void print_val( ) { cout << val << " "; }
  };
public:
  class iterator
  {private:
     friend class D_List<T>;
     D_Node * the_node;
   public:
     iterator( ) : the_node(0) { }
     iterator(D_Node * dn) : the_node(dn) { }
      // Copy constructor
     iterator(const iterator & it) : the_node(it.the_node) { }

     iterator& operator=(const iterator& it)
           { the_node = it.the_node;
      return *this;
     }

    bool operator==(const iterator& it) const
          { return (the_node == it.the_node); }

   bool operator!=(const iterator& it) const
   { return !(it == *this); }
```

Библиотека БГУИР

```
  iterator& operator++( )
  { if ( the_node == 0 )
      throw "incremented an empty iterator";
    if ( the_node->next == 0 )
      throw "tried to increment too far past the end";

    the_node = the_node->next;
    return *this;
  }

  iterator& operator--( )
  { if ( the_node == 0 )
      throw "decremented an empty iterator";
    if ( the_node->prev == 0 )
      throw "tried to decrement past the beginning";

    the_node = the_node->prev;
    return *this;
  }

  T& operator*( ) const
  { if ( the_node == 0 )
     throw "tried to dereference an empty iterator";
    return the_node->val;
  }
};

private:
  D_Node *head;    // указатель на начало списка
  D_Node *tail;     // указатель на элемент вне списка
  D_List & operator=(const D_List &);
  D_List(const D_List &);

  iterator head_iterator;
  iterator tail_iterator;
public:
  D_List( )
  {  head = tail = new D_Node;
     tail->next = 0;
     tail->prev = 0;
     head_iterator = iterator(head);
     tail_iterator = iterator(tail);
  }
```

Библиотека БГУИР

```
    // конструктор  (создание списка, содержащего один элемент)
D_List(T node_val)
{ head = tail = new D_Node;
  tail->next = 0;
  tail->prev = 0;
  head_iterator = iterator(head);
  tail_iterator = iterator(tail);
  add_front(node_val);
}
    //  деструктор
~D_List( )
{  D_Node *node_to_delete = head;
   for (D_Node *sn = head; sn != tail;)
  { sn = sn->next;
    delete node_to_delete;
    node_to_delete = sn;
  }
  delete node_to_delete;
}
bool is_empty( ) {return head == tail;}
iterator front( ) { return head_iterator; }
iterator rear( )  { return tail_iterator; }
void add_front(T node_val)
{ D_Node *node_to_add = new D_Node(node_val);
  node_to_add->next = head;
  node_to_add->prev = 0;
  head->prev = node_to_add;
  head = node_to_add;
  head_iterator = iterator(head);
}
    // добавление нового элемента в начало списка
void add_rear(T node_val)
{  if ( is_empty( ) )    // список не пустой
    add_front(node_val);
  else
   // не выполняется для пустого списка, т.к. tail->prev =NULL
   // и, следовательно, tail->prev->next бессмысленно
  { D_Node *node_to_add = new D_Node(node_val);
   node_to_add->next = tail;
   node_to_add->prev = tail->prev;
   tail->prev->next = node_to_add;
```
Библиотека БГУИР

```
      tail->prev = node_to_add;
      tail_iterator = iterator(tail);
    }
  }
  bool remove_it(iterator & key_i)
  {  for ( D_Node *dn = head; dn != tail; dn = dn->next )
      if ( dn == key_i.the_node )    // найден узел для удаления
     { dn->prev->next = dn->next;
       dn->next->prev = dn->prev;
       delete dn;                        // удаление узла
       key_i.the_node = 0;
       return true;
    }
    return false;
  }
  //  поиск итератора по значению узла
  iterator find(T node_val) const
 {   for ( D_Node *dn = head; dn != tail; dn = dn->next )
      if ( dn->val == node_val ) return iterator(dn);
      return tail_iterator;
  }
int size( ) const
  {  int count = 0;
    for ( D_Node *dn = head; dn != tail; dn = dn->next ) ++count;
    return count;
  }
  //  Вывод содержимого списка
  void print( ) const
  {    for ( D_Node *dn = head; dn != tail; dn = dn->next )
      dn->print_val( );
      cout << endl;
  }
 };
```

В файле d_list.cpp содержится main-функция, демонстрирующая некоторые примеры работы со списком.

```
#include "dlist_2.h"
typedef int tip;
D_List<tip> the_list;      // создается пустой список
int main()
{ int ret  = 0;
```

Библиотека БГУИР

```
  D_List<tip>::iterator list_iter;
// занесение значений   0 1 2 3 4  в список
  for (int j = 0; j < 5; ++j)
    the_list.add_front(j);
    // вывод содержимого списка используя компоненту-функцию
    // класса D_List
  the_list.print( );
// повторный вывод значения содержимого списка
// используя итератор
  for ( list_iter = the_list.front( ) ;
       list_iter != the_list.rear( ) ;
       ++list_iter )
    cout << *list_iter << " ";
  cout << endl;
// вывод содержимого списка в обратном порядке
  for ( list_iter = the_list.rear( ) ; list_iter != the_list.front( ) ; )
  {   --list_iter;   // декремент итератора
      cout << *list_iter << " ";
  }
  cout << endl;
  the_list.remove_it(the_list.find(3));
  the_list.print( );
  cout<<the_list.size( )<<endl;
  return 0;
}
```

Результат работы программы:

```
3  2  1  0
3  2  1  0
0  1  2  3  4
2  1  0
4
```

Итератор реализован как открытый вложенный класс D_List::iterator. Так как класс открытый, может быть в main создан его объект. Класс iterator объявляет дружественным классу D_List, чтобы функция remuv_it класа D_List имела возможность обращаться к private-члену the_node класса iterator.

В дополнение к стандартным указателям на голову и хвост списка (адрес за пределами списка) в программе (в d_list.h) объявлены итераторы head_iterator и tail_iterator, также ссылающиеся на голову и хвост списка.

Использование итератора позволяет скрыть единственный элемент данных класса D_List:

D_Node * the_node.

В классе iterator выполнена перегрузка некоторых операций, позволяю-

Библиотека БГУИР

щих манипулировать узлом в строго определенных правилах (табл. 5).

Таблица 5

Функция	Описание
operator=()	Присваивает the_node значение the_node правой части
operator==()	Возвращает true, если оба итератора ссылаются на один узел
operator!=()	Отрицание операции ==
operator++()	Перемещает итератор на следующий узел
operator--()	Перемещает итератор на предыдущий узел
operator*()	Возвращает значение node_val узла D_node

Для поддержки использования итераторов в качестве аргументов или возвращаемых значений определен конструктор копирования.

В программе функция find(T) возвращает не bool, а iterator, который может быть передан другим функциям, принимающим параметры-итераторы, для доступа к данным или прохода по списку.

Использование итераторов позволяет пользователю манипулировать списком, при этом детали его реализации остаются надежно скрытыми.

Реализация двоичного дерева

В отличие от списков двоичные деревья представляют собой более сложные структуры данных.

Дерево – непустое конечное множество элементов, один из которых называется ***корнем***, а остальные делятся на несколько непересекающихся подмножеств, каждое из которых также является деревом. Одним из разновидностей деревьев являются **бинарные деревья**. Бинарное дерево имеет один корень и два непересекающихся подмножества, каждое из которых само является бинарным деревом. Эти подмножества называются **левым** и **правым поддеревьями** исходного дерева. На рис. 11 приведены графические изображения бинарных деревьев. Если один или более узлов дерева имеют более двух ссылок, то такое дерево не является бинарным.

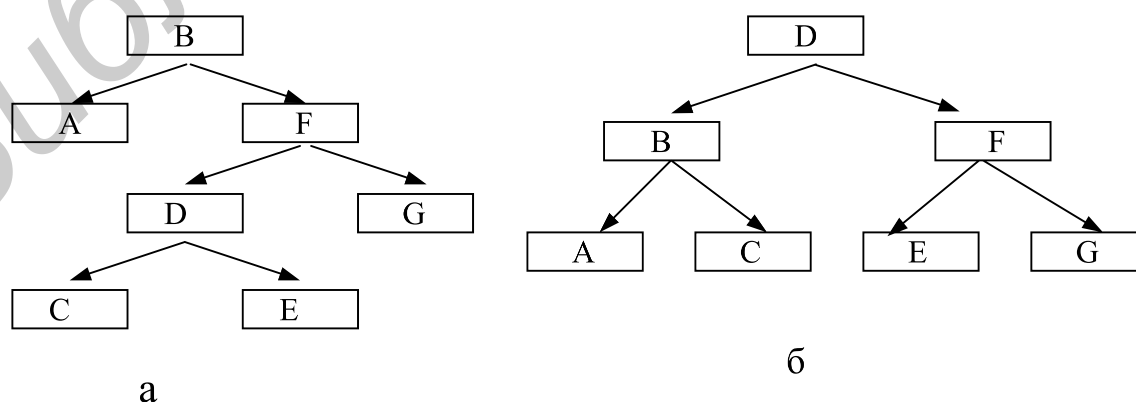

Рис. 11. Структура бинарного дерева

Библиотека БГУИР

Бинарные деревья с успехом могут быть использованы, например, при сравнении и организации хранения очередной порции входной информации с информацией, введенной ранее, и при этом в каждой точке сравнения может быть принято одно из двух возможных решений. Так как информация вводится в произвольном порядке, то нет возможности предварительно упорядочить ее и применить бинарный поиск. При использовании линейного поиска время пропорционально квадрату количества анализируемых слов. Каким же образом, не затрачивая большое количество времени, организовать эффективное хранение и обработку входной информации. Один из способов постоянно поддерживать упорядоченность имеющейся информации – это перестановка ее при каждом новом вводе информации, что требует существенных временных затрат. Построение бинарного дерева осуществляется на основе *лексикографического упорядочивания* входной информации.

Лексикографическое упорядочивание информации в бинарном дереве заключается в следующем. Считывается первая информация и помещается в узел, который становится корнем бинарного дерева с пустыми левым и правым поддеревьями. Затем каждая вводимая порция информации сравнивается с информацией, содержащейся в корне. Если значения совпадают, то, например, наращиваем число повторов и переходим к новому вводу информации. Если же введенная информация меньше значения в корне, то процесс повторяется для левого поддерева, если больше – для правого. Так продолжается до тех пор, пока не встретится дубликат или пока не будет достигнуто пустое поддерево. В этом случае число помещается в новый узел данного места дерева.

Приводимый далее пример шаблонного класса B_tree демонстрирует простое дерево поиска. Как и для программы очереди, код программы организован в виде двух файлов, где файл Bt.h является непосредственно шаблонным классом дерева, а Bt.cpp демонстрирует работу функций шаблонного класса. Вначале приведен текст файла Bt.h.

```
#include <iostream>
#include <cassert>
using namespace std;
////////////////////////////////////////////////////////////
// реализация шаблона класса B_tree
//  Тип T должен поддерживать следующие операции:
//   operator=( );
//   operator<<( );
//   operator==( );
//   operator!=( );
//   operator<( );
//
////////////////////////////////////////////////////////////
template <class T>
class B_tree
{
```

Библиотека БГУИР

```
private:
  struct T_node
  { friend class B_tree;
    T val;                    // данные узла
    T_node *left;             // указатель на левый узел
    T_node *right;            // указатель на правый узел
    int  count;               // число повторов val при вводе
    T_node();                                    // конструктор
    T_node( const T node_val ) : val(node_val) { } // конструктор
    ~T_node(){}                                  // деструктор

   // печать данных в виде дерева "на боку" с корнем слева
   // "Обратный" рекурсивный обход (т.е. справа налево)
   // Листья показаны как "@".
   void print ( const int level = 0 ) const
   { // Инициализация указателем this (а не корнем)
     // это позволяет пройти по любому поддереву
     const T_node *tn = this;
     if(tn) tn->right->print(level+1); // сдвиг вправо до листа
     for (int n=0; n<level;++n)
       cout << "   ";
     if(tn)
           cout << tn->val << '(' << tn->count << ')' << endl;
     else
            cout << "@" << endl;
     if(tn) tn->left->print(level+1); // сдвиг на левый узел
   }
  };
private:
  T_node *root;
  T_node *zero_node;

  //  Запретить копирование и присваивание
  B_tree(const B_tree &);
  B_tree & operator=( const B_tree & );

  // Создать корневой узел и проинициализировать его
  void new_root( const T root_val )
  { root = new T_node(root_val);
    root->left  = 0;
    root->right = 0;
    root->count = 1;
  }
```
Библиотека БГУИР

```
// Find_node(T find_value) возвращает ссылку на
// указатель для упрощения реализации   remove(T).
T_node * & find_node( T find_value )
{ T_node *tn = root;
   while ((tn != 0) && (tn->val != find_value))
   { if(find_value < tn->val)
         tn = tn->left;            // движение налево
      else
         tn = tn->right;           // движение направо
   }
  if (!tn) return zero_node;
  else return tn;
}

// Присоединяет новое значение ins_val к соответствующему листу,
// если значения нет в дереве, и увеличивает count для каждого
// пройденного узла
T_node * insert_node( const T ins_val, T_node * tn = 0 )
{ if(!root)
  { new_root(ins_val);
    return root;
  }
  if(!tn) tn = root;

  if((tn ) && (tn->val != ins_val))
 {  if(ins_val < tn->val)
    { if(tn->left)           // просмотр левого поддерева
          insert_node(ins_val,tn->left);
       else
       { attach_node(tn,tn->left,ins_val); // вставка узла
          return tn->left;
       }
    }
  else
   { if(tn->right)           // просмотр правого поддерева
      insert_node(ins_val,tn->right);
    else
    { attach_node(tn,tn->right,ins_val); // вставка узла
       return tn->right;
    }
   }
}
else
 if(tn->val==ins_val) add_count(tn,1);
```

Библиотека БГУИР

```
  assert(tn); // Оценивает выражение и, когда результат ЛОЖЕН, печатает
              // диагностическое сообщение и прерывает программу
   return 0;
 }
// Создание нового листа и его инициализация
  void attach_node( T_node * new_parent,
              T_node * & new_node, T insert_value )
  { new_node = new T_node( insert_value );
    new_node->left  = 0;
    new_node->right = 0;
    new_node->count = 1;
  }
// Увеличение числа повторов содержимого узла
  void add_count( T_node * tn, int incr )
  { tn->count += incr; }

  // Удаление всех узлов дерева в обходе с отложенной
  // выборкой. Используется в   ~B_tree().
  void cleanup (T_node *tn)
  { if(!tn) return;
    if(tn->left)
          { cleanup(tn->left);
      tn->left = 0;
   }
   if(tn->right != 0 )
   { cleanup(tn->right);
      tn->right = 0;
   }
   delete tn;
  }

  // рекурсивно печатает значения в поддереве с корнем  tn
  //  (обход дерева с предварительной выборкой)
  void print_pre(const T_node * tn) const
  { if(!tn) return;
    cout << tn->val << "  ";
    if(tn->left)
    print_pre(tn->left);
    if(tn->right)
    print_pre( tn->right );
  }

  // рекурсивно печатает значения в поддереве с корнем  tn
  //  (обход дерева )
```

Библиотека БГУИР

```
  void print_in(const T_node * tn)  const
  { if(!tn) return;
    if(tn->left)
      print_in(tn->left);
    cout << tn->val << "  ";
    if(tn->right)
      print_in(tn->right);
  }

  // рекурсивно печатает значения в поддереве с корнем  tn
  //  (обход дерева с отложенной выборкой)
  void print_post(const T_node * tn) const
  { if(!tn) return;
    if(tn->left)
      print_post(tn->left);
    if(tn->right)
      print_post(tn->right);
   cout << tn->val << "  ";
  }
public:
  B_tree() : zero_node(0) {root = 0;}

  B_tree(const T root_val) : zero_node(0)
  { new_root(root_val); }

  ~B_tree()
  { cleanup(root); }

  // Добавляет к дереву через функцию insert_node() значение, если его
  // там еще нет. Возвращает true, если элемент добавлен, иначе false
  bool add(const T insert_value)
  { T_node *ret = insert_node(insert_value);
    if(ret) return true;
    else    return false;
  }

  // Find(T) возвращает true, если find_value найдено
  // в дереве, иначе false
  bool find(T find_value)
  { Tree_node *tn = find_node(find_value);
    if(tn) return true;
    else   return false;
  }
```

Библиотека БГУИР

```
  // Print() производит обратную порядковую выборку
  // и печатает дерево "на боку"

  void print() const
  { cout << "\n=--------------------------\n"<< endl;
   // Это вызов    Binary_tree::Tree_node::print( ),
   // а не рекурсивный вызов  Binary_tree::print( ).
    root->print( );
  }

  // последовательная печать дерева при обходе
  // с предварительной, порядковой и отложенной выборкой.
  // Вызываются рекурсивные private-функции, принимающие
  // параметр Tree_node *

  void print_pre_order( ) const
  { print_pre(root);
    cout << endl;
  }

  void print_in_order( ) const
  { print_in(root);
    cout << endl;
  }

  void print_post_order( ) const
  { print_post(root);
    cout << endl;
  }
};
```

Далее приведено содержимое файла Bt.cpp.

```
#include "bin_tree.h"
B_tree<int> my_bt( 7 ); // создание корня бинарного дерева
// Заполнение дерева целыми значениями
void populate( )
{ my_bt.add( 5 );
  my_bt.add( 5 );
  my_bt.add( 9 );
  my_bt.add( 6 );
  my_bt.add( 5 );
  my_bt.add( 9 );
  my_bt.add( 4 );
  my_bt.add( 11 );
  my_bt.add( 8 );
```

Библиотека БГУИР

```
  my_bt.add( 19 );
  my_bt.add( 2 );
  my_bt.add( 10 );
  my_bt.add( 19 );
}
int main( )
{ populate( );
  my_bt.print ( );
  cout << endl;
  cout << "Pre-order:  " ;
  my_bt.print_pre_order ( );
  cout << "Post-order: " ;
  my_bt.print_post_order ( );
  cout << "In-order:   " ;
  my_bt.print_in_order ( );
  return 0;
}
```

Результат работы программы:

```
                    @
                 19(2)
                    @
             11(1)
                    @
                 10(1)
                    @
         9(2)
                @
             8(1)
                @
 7(1)
                @
             6(2)
                @
         5(1)
                @
             4(1)
                    @
                  2(1)
                    @

Pre-order  :    7   5   4   2   6   9   8   11   10   19
Post-order :    2   4   6   5   8   10   19   11   9   7
In-order   :    2   4   5   6   7   8   9   10   11   19
```

B_tree содержит вложенный закрытый класс T_node (структуру) для представления внутреннего описания элемента (узла) бинарного дерева. Структура его скрыта от пользователя. Поле val содержит значение, хранимое в узле, left и right – указатели на левый и правый потомки данного узла.

В классе B_tree два открытых конструктора. Конструктор по умолчанию создает пустое бинарное дерево B_tree, а конструктор B_tree(const T) дерево с

Библиотека БГУИР

одним узлом. Деструктор ~B_tree() удаляет каждый T_node, производя обход бинарного дерева с отложенной выборкой. Отложенная выборка гарантирует, что никакой узел не будет удален, пока не удалены его потомки.

Для объектов класса B_tree с целью упрощения не определены (а только объявлены закрытыми) конструктор копирования и операция присваивания. Это позволяет компилятору сообщить об ошибке при попытке выполнить эти операции. Если эти операции потребуется использовать, то их необходимо сделать открытыми и определить.

Функция find_node() возвращает ссылку на указатель T_node. Для того чтобы функция могла сослаться на нулевой указатель, вводится поле zero_node.

Рекурсивные функции print_pre_order(), print_in_order() и print_post_order() печатают элементы дерева в линейной последовательности в соответствии со стандартными схемами обхода двоичного дерева. Функция print_pre_order() печатает значение узла ***до*** того, как будет напечатан любой из его потомков. Функция print_post_order(), наоборот, печатает значение узла только после того, как будут напечатаны все его потомки. И, наконец, функция print_in_order() выводит на печать элементы бинарного дерева в порядке возрастания.

Литература

1. Дейтел Х., Дейтел П. Как программировать на С++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2001. – 1152 с.: ил.

2. Шилд Г. Программирование на Borland C++ для профессионалов. –Мн.: ООО «Попурри» , 1998. – 800 с.

3. Скляров В.А. Язык С++ и объектно-ориентированное программирование. Мн.: Выш. шк., 1997. – 478 с.: ил.

4. Ирэ П. Объектно-ориентированное программирование с использованием С++: Пер. с англ. – Киев: НИПФ «ДиаСофт Лтд», 1995. – 480 с.

Библиотека БГУИР

Содержание

Введение 4
Объектно-ориентированное программирование 4
Абстрактные типы данных 4
Базовые принципы объектно-ориентированного программирования 5
Основные достоинства языка С++ 7
Особенности языка С++ 7
Ключевые слова 7
Константы и переменные 7
Операции 8
Типы данных 8
Передача аргументов функции по умолчанию 8
Простейший ввод и вывод 9
Объект cout 9
Манипуляторы hex и oct 9
Объект cin 12
Операторы для динамического выделения и освобождения памяти (new и delete) 12
Базовые конструкции объектно-ориентированных 15
программ 15
Объекты 15
Понятие класса 16
Конструктор explicit 28
Встроенные функции (спецификатор inline) 30
Организация внешнего доступа к локальным компонентам класса

Библиотека БГУИР

(спецификатор friend) 31
Вложенные классы 35
Static-члены (данные) класса 38
Указатель this 40
Компоненты-функции static и const 43
Ссылки 45
Параметры ссылки 47
Независимые ссылки 48
Наследование (производные классы) 48
Конструкторы и деструкторы.. 53
Виртуальные функции 57
Абстрактные классы 63
Множественное наследование... 69
Виртуальное наследование 72
Перегрузка функций... 75
Перегрузка операторов... 76
Перегрузка бинарного оператора... 77
Перегрузка унарного оператора 80
Дружественная функция operator 82
Особенности перегрузки операции присваивания 83
Перегрузка оператора [] 85
Перегрузка оператора () 87
Перегрузка оператора -> 89
Перегрузка операторов new и delete 90
Преобразование типа... 94
Явные преобразования типов 94
Преобразования типов, определенных в программе........................ 95
Шаблоны... 96
Параметризированные классы... 97
Передача в шаблон класса дополнительных параметров 99
Шаблоны функций.. 100
Совместное использование шаблонов и наследования 101
Некоторые примеры использования шаблона класса 103
Задание свойств класса 105
Пространства имен 108
Ключевое слово using как директива.. 109
Ключевое слово using как объявление 110
Псевдоним пространства имен.. 110
Организация ввода-вывода 111
Состояние потока.. 114
Строковые потоки... 116
Организация работы с файлами 117
Организация файла последовательного доступа............................ 120
Создание файла произвольного доступа 123
Библиотека БГУИР

Основные функции классов ios, istream, ostream 126
Исключения в C++ 128
Основы обработки исключительных ситуаций 129
Перенаправление исключительных ситуаций 136
Исключительная ситуация, генерируемая оператором new 137
Генерация исключений в конструкторах 139
Задание собственной функции завершения 140
Спецификации исключительных ситуаций 141
Задание собственного неожиданного обработчика 141
Иерархия исключений стандартной библиотеки 143
Стандартная библиотека шаблонов (STL) 143
Общее понятие о контейнере 143
Общее понятие об итераторе 147
Категории итераторов 149
Основные итераторы 150
Вспомогательные итераторы 156
Операции с итераторами 159
Контейнерные классы 159
Контейнеры последовательностей 159
Контейнер последовательностей vector 160
Контейнер последовательностей list 162
Контейнер последовательностей deque 165
Ассоциативные контейнеры 167
Ассоциативный контейнер multiset 167
Ассоциативный контейнер set 168
Ассоциативный контейнер multimap 169
Ассоциативный контейнер map 170
Адаптеры контейнеров 171
Адаптеры stack 171
Адаптеры queue 172
Адаптеры priority_queue 173
Пассивные и активные итераторы 173
Алгоритмы 175
Алгоритмы сортировки sort, partial_sort, sort_heap 176
Алгоритмы поиска find, find_if, find_end, binary_search 177
Алгоритмы fill, fill_n, generate и generate_n 178
Алгоритмы equal, mismatch и lexicographical_compare 179
Математические алгоритмы 180
Алгоритмы работы с множествами 182
Алгоритмы swap, iter_swap и swap_ranges 183
Алгоритмы copy, copy_backward, merge, unique и reverse 183
Примеры реализации контейнерных классов 184
Связанные списки 184
Реализация односвязного списка 184
Библиотека БГУИР

Реализация двусвязного списка 184
Реализация двоичного дерева.. 190
Литература.. 198

Библиотека БГУИР

Св. план 2003, резерв

Учебное издание

Луцик Юрий Александрович,
Ковальчук Анна Михайловна,
Лукьянова Ирина Викторовна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C++

Учебное пособие
по курсу «Объектно-ориентированное программирование»
для студентов специальности «Вычислительные машины,
системы и сети» всех форм обучения

Редактор Т.А. Лейко
Корректор Е.Н. Батурчик

Подписано в печать 12.12.2003.	Формат 60x84 1/16.	Бумага офсетная.
Печать ризографическая.	Гарнитура «Таймс».	Усл. печ. л. 11,97.
Уч.- изд. л. 12,0.	Тираж 200 экз.	Заказ 414.

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Лицензия ЛП № 156 от 30.12. 2002.
Лицензия ЛВ № 509 от 03.08. 2001.
220013, Минск, П. Бровки, 6

Библиотека БГУИР