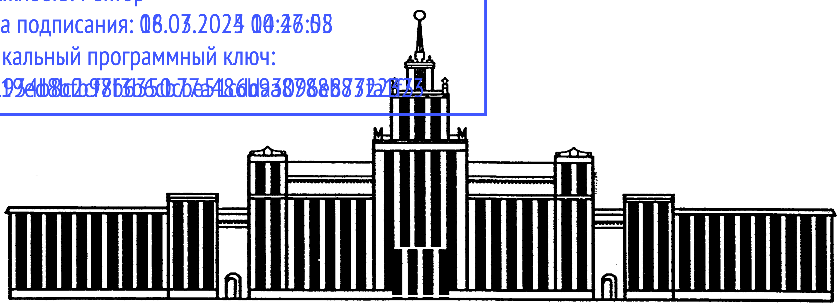
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ



ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

004.4 (07)
Д304

А.К. Демидов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Учебное пособие

Челябинск
2017

Министерство образования и науки Российской Федерации
Южно-Уральский государственный университет
Кафедра прикладной математики и программирования

004.4 (07)
Д304

А.К. Демидов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Учебное пособие

Челябинск
Издательский центр ЮУрГУ
2017

УДК 004.4 (075.8)
Д304

Одобрено
учебно-методической комиссией факультета математики,
механики и компьютерных технологий

Рецензенты:
и.о. директора департамента математики, механики и компьютерных наук Института естественных наук и математики УрФУ, к.ф.-м.н., доцент М.О. Асанов;
доцент кафедры вычислительной математики и информационных технологий ФГБОУ ВО «Челябинский государственный университет», к.п.н. М.Н. Алексеев

Демидов, А.К.
Д304 Объектно-ориентированное программирование на C++: учебное пособие / А.К. Демидов. – Челябинск: Издательский центр ЮУрГУ, 2017. – 157 с.

Пособие предназначено для студентов, обучающихся по направлениям 01.03.02 «Прикладная математика и информатика» и 01.03.04 «Прикладная математика», в качестве базового при изучении курса «Объектно-ориентированное программирование», а также как дополнительного при выполнении курсовой работы по курсу «Проектирование прикладного программного обеспечения».

В пособии рассматриваются возможности языка C++ для создания объектно-ориентированных программ, приводятся основные положения объектно-ориентированного подхода и паттерны проектирования. Изложение материала иллюстрируется примерами программ на языке C++. В пособие включены контрольные вопросы и упражнения для проверки освоения материала студентами.

УДК 004.4 (075.8)

ЧАСТЬ 1 ЯЗЫК С++

1 УЛУЧШЕНИЯ ЯЗЫКА С

Термины, знание которых необходимо для понимания главы:

- именованные константы в С;
- препроцессор;
- макроопределение;
- функция;
- заголовок функции;
- параметр;
- вызов функции;
- аргумент;
- передача по значению;
- указатель;
- динамическое выделение памяти;
- куча.

1.1 Замена для препроцессора

Квалификатор `const` позволяет создавать типизированные именованные константы, с ограниченной областью видимости в отличие от констант, определенных с помощью `#define`. Задать значение константе можно только при создании.

```
const int size=100;
int a[size];
const double pi=3.14159265358979323846;
```

Также этот квалификатор используется для указания, что параметр не меняется внутри функции.

```
char *strcpy(char *dst, const char *src);
```

Функция, определенная с помощью спецификатора `inline` становится встраиваемой, т.е. её код подставляется в точку вызова. При этом устраняются все недостатки, свойственные макроопределениям. Время работы программы уменьшается за счет операций сохранения/восстановления регистров и передачи управления, но размер программы увеличивается. Спецификатор `inline` носит рекомендательный характер, для больших функций компилятор генерирует обычный вызов. Тело встраиваемой функции определяется в заголовочном файле.

```
inline int max(int x, int y)
{ return x>y?x:y;
}
```

Макроопределение `#define max(x,y) ((x)>(y)?(x):(y))` работает для любых типов данных. Чтобы функция max работала так же, её нужно определить как шаблон:

```
template <typename T>
inline T max(T x, T y)
{ return x>y?x:y;
}
```

1.2 Ссылки

В языке С реализована только передача параметров по значению. Если потребуется изменение переменной внутри функции, то нужно передать функции адрес переменной, применив операцию &, а внутри функции выполнять операцию разыменования для параметра при всех обращениях к этой переменной. В С++ появилась замена для указателей – ссылка, которую можно рассматривать как константный указатель, который всегда разыменован.

Ссылка должна инициализироваться при создании некоторым объектом, адрес которого будет храниться в ссылке. Перенаправить ссылку на другой объект нельзя. Изменение ссылки приводит к изменению объекта.

```
int a=5;
int &b=a;
b=6; // a=6
```

Чаще всего ссылки используются в качестве параметров функций.

```
void swap(int &x, int &y)
{ int t;
  t=x;
  x=y;
  y=t;
}
int main()
{ int a=1, b=2;
  swap(a,b);
  // a=2, b=1
}
```

Функция может возвращать ссылку. Результат такой функции можно использовать в качестве левого операнда присваивания. Можно возвращать только ссылку на аргумент функции (или его часть), переданный по ссылке, или на статический объект.

```
// Ассоциативный массив
struct item { int key; double val; }  data[100];
int ndata;
double &getdata(int key)
```

```
{ for(int i=0;i<ndata;++i)
    if(data[i].key==key) // есть элемент с таким ключом
      return data[i].val;
  data[ndata].key=key; // добавить новый элемент
  return data[ndata++].val;
}
...
getdata(-5)=0.5;
getdata(4)=getdata(-5)+7; // 7.5
```

1.3 new и delete

Функции `malloc` и `calloc` не обеспечивают соответствия между размером выделенной памяти и типом данных.

```
double *p; // до модификации было int *p;
...
p=calloc(n,sizeof(int)); // забыл исправить тип,
                         // но компилятор не заметит этой ошибки
```

Результат `calloc` имеет тип `void *`, который в С может быть преобразован к указателю любого типа без явного вызова операции преобразования, но даже использование явного преобразования результата не гарантирует, что в аргументах не будет ошибки:

```
p=(double *)calloc(n,sizeof(int));
```

Кроме того, многие программисты не заботятся о проверке результата функции на 0, считая, что памяти много. Для устранения недостатков функций `malloc`, `calloc` и `free` в C++ были введены операции `new` и `delete`.

Операция `new` возвращает *типизированный* указатель, а после выделения памяти вызывается конструктор, который инициализирует память. Если память не может быть выделена, возникает исключительная ситуация `bad_alloc`, т.е. проверять возвращаемое значение на 0 нет необходимости. Допускаются следующие формы для операции `new`:

`new` **тип**
`new` **тип**`()`
`new` **тип**`[`**выражение**`]`
`new` **тип**`[`**выражение**`] ()`
`new` **тип**`(`**список_выражений**`)`

3-я и 4-я форма используется для создания массива объектов, остальные – для создания одиночных объектов. Различие между 1-й и 2-й, 3-й и 4-й формами проявляется только для стандартных типов данных и классов без конструкторов. В 1-м и 3-м случаях состояние объектов остается неопределенным, во 2-м и 4-м – инициализируется 0. Для классов с

конструкторами в первых четырех формах вызывается конструктор по умолчанию. 5-я форма предназначена для инициализации созданного объекта с помощью конструктора с параметрами, а для стандартных типов данных можно указать только одно выражение, значение которого используется для инициализации созданного объекта.

Операция `delete` освобождает память, предварительно вызвав деструктор для объектов. Существуют две формы операции `delete`:

`delete` **указатель**

`delete[]` **указатель**

1-я форма используется для уничтожения одиночного объекта, 2-я форма – массива объектов.

Примеры:

```
int *p;
p=new int(10);
...
delete p;
p=new int[10];
...
delete[] p;
```

Нельзя смешивать формы `new` и `delete`. Нельзя выделять память с помощью функций `malloc` и `calloc`, а освобождать с помощью `delete`, или выделять с помощью `new`, а освобождать с помощью `free` даже для стандартных типов данных, не требующих вызова конструкторов и деструкторов, так как эти операции могут сохранять дополнительную информацию о выделенной памяти или использовать другую реализацию кучи.

1.4 Функции и операции

Функциям, выполняющим одинаковые действия с данными различных типов, в C++ можно дать одинаковые имена. Компилятор сможет определить, какую функцию из набора вызвать, по типу аргументов.

```
void print(int x)
{ printf("%d",x); }
void print(char x)
{ printf("%c",x); }
int main()
{ print(10);
  print('A');
}
```

Чтобы на этапе компоновки программы можно было установить соответствие между функциями и их вызовами в имя функции на языке C++

включаются типы ее параметров. Для функций на языке С типы параметров не важны, поэтому перед объявлением функции из библиотек на языке С нужно написать `extern "C"`.

Обычно один и тот же заголовочный файл используется для программ на С и С++, поэтому в него добавляет следующие строки:

```
#ifdef __cplusplus
extern "C" {
#endif
// объявления функций на С
#ifdef __cplusplus
}
#endif
```

Вместо определения набора функций, отличающихся только количеством параметров, можно указать при объявлении функции в заголовочном файле значения по умолчанию для некоторых параметров. Эти параметры должны быть последними в списке. Если при вызове соответствующий аргумент пропущен, то должны быть пропущены и все аргументы за ним. В качестве значений по умолчанию можно использовать константы и глобальные переменные.

```
void f(int a, int n=100, double eps=1e-9);
...
f(1,200,1e-6);
f(1,200); // f(1,200,1e-9);
f(1); // f(1,100,1e-9);
```

В С++ можно переопределять операции для собственных типов данных и, также как для функций, нужный вариант операции будет вызван в зависимости от типов аргументов. Разработчиками языка эта возможность была использована, чтобы избавиться от проблемы несоответствия форматов и списка аргументов в функциях форматированного ввода-вывода. При изменении типа переменной в программе на С необходимо было отследить все появления этой переменной в списке аргументов в `printf/scanf` и изменить строки с форматами.

```
int a,b;
scanf("%d%d",&a,&b);
```

В С++ для ввода-вывода были переопределены операции << и >> (в С это операции поразрядного сдвига для целых чисел). Первым аргументом операции указывается поток ввода (например, cin – консольный ввод, console input) или поток вывода (например, cout – консольный вывод, console output), а вторым аргументом – вводимая переменная или выводимое значение. В качестве результата операция возвращает поток и поэтому

её можно применить последовательно для ввода или вывода нескольких значений.

```
int a,b;
cin>>a>>b;
cout<<"a="<<a<<" b="<<b<<"\n";
```

Обратите внимание на отсутствие форматов и операции &, так как в операции ввода происходит передача аргумента по ссылке.

Запомнить: направление уголков указывает направление передачи информации.

1.5 Пространства имен

Разные разработчики библиотек могут определить глобальные объекты с одинаковыми именами или функции с одинаковыми именами и параметрами. Чтобы избежать конфликта имен при использовании этих библиотек в одной программе, в C++ появилась возможность разделения глобального пространства имен. Каждый разработчик должен определить свое пространство имен:

```
namespace имя {
 объявления и определения
}
```

Описания, относящиеся к одному пространству имен, могут быть разделены на несколько модулей. Можно определять вложенные пространства имен.

Обращаться к объектам и функциям из того же пространства имен можно напрямую. При использовании этих объектов и функций вне этого пространства, нужно либо обращаться указывая их полное имя: **имя_пространства_имен::имя_функции(аргументы)**, либо сделать их доступными для прямого использования с помощью оператора

```
using имя_пространства_имен::имя_функции;
```

или

```
using namespace имя_пространства_имен;
```

По умолчанию все глобальные объекты и функции определяются в неименованном пространстве имен. Можно обращаться к глобальному объекту следующим образом:

```
int x=1;
int main()
{ int x=2;
  cout << x << "\n"; // 2
  cout << ::x << "\n"; // 1
}
```

Все объекты и функции стандартной библиотеки C++ STL определены в пространстве имен std. Поэтому необходимо в программе указывать оператор using:

```
#include <iostream>
using namespace std;
int main()
{ cout<<"Hello, world!\n";
}
```

Существуют также варианты заголовочных файлов для стандартной библиотеки C, в которых все функции объявлены в пространстве имен std. Например, <cstdio> содержит объявления из <stdio.h> (для получения имени C++ варианта из имени файла заголовочного файла C нужно убрать ".h" и добавить букву "c" вначале).

1.6 Операции преобразования

Операцию преобразования (**тип**) **выражение** можно записывать в форме **тип** (**выражение**), если имя типа состоит из одной лексемы. Правила преобразований в C++ совпадают с C, за исключением того, что преобразование void * к указателю любого типа требует явный вызов операции преобразования (см. пример ниже).

Любое преобразование является источником потенциальных ошибок, поэтому в C++ были введены четыре специализированных операции преобразования. При использовании этих операций компилятор сможет выполнить проверку на допустимость преобразования на этапе компиляции или при выполнении. Вызовы этих операций легче находить в тексте программы, чем вызовы в форме, унаследованной от C, что упрощает модификацию программы.

const_cast<**тип**>(**выражение**)

Служит для добавления или удаления модификатора const или volatile (антоним register). Так как добавление const не требует явного вызова операции преобразования, то вызывать эту операцию необходимо только для удаления const.

dynamic_cast<**тип**>(**выражение**)

Служит для преобразования указателя или ссылки на объект одного класса в указатель или ссылку соответственно на объект другого класса в пределах одной иерархии классов. Оба класса должны иметь хотя бы один виртуальный метод, например, деструктор. Если преобразование недопустимо, то в случае преобразования указателей операция возвращает нулевой указатель, а в случае ссылок – исключение bad_cast. Для повышающего преобразования (к указателю или ссылке на объект базового класса)

явного вызова операции преобразования не требуется, поэтому эта операция используется для понижающего (к указателю или ссылке на объект производного класса) или перекрёстного преобразования.

`static_cast<`**тип**`>(`**выражение**`)`

Служит для вызова преобразований для стандартных типов данных (например, вещественное в целое и обратно) и преобразований, определенных программистом. Также можно использовать для повышающих и понижающих преобразований в пределах одной иерархии классов *без проверки*, при этом не требуется наличие виртуальных методов. Так как в большинстве случаев такие преобразования выполняются даже без явного вызова, то можно использовать форму (тип)выражение или тип(выражение) в тех случаях, когда необходимо получить значение определенного типа, а форму `static_cast<тип>(выражение)` применять только для потенциально опасных понижающих преобразований.

`reinterpret_cast<`**тип**`>(`**выражение**`)`

Служит для преобразований не связанных между собой типов, например, указатель в целое и обратно, указатель `void *` к любому указателю. При переносе программы на компьютер с процессором другой архитектуры или разрядностью все операции преобразования этого вида должны тщательно проверяться.

```
// старая форма
int *p1=(int *)calloc(100,sizeof(int));
// новая рекомендуемая форма
int *p2=
  reinterpret_cast<int *>(calloc(100,sizeof(int)));
```

1.7 Другие изменения

Появился тип `bool` и резервированные слова `true` и `false`. Любое ненулевое значение преобразуется в `true`, а нулевое – в `false`.

Объекты можно объявлять в любой точке программы, в том числе в инициализирующей части цикла `for` и в условии `if` и `while` (эта возможность была добавлена потом и в стандарт языка С). Можно инициализировать объекты любым выражением, не только константным, в том числе и статические объекты.

Операции `typeid` в качестве аргумента можно указать любое выражение или тип (как операции `sizeof`). Операция возвращает информацию о типе выражения в виде объекта класса `type_info`. Если определить тип выражения невозможно, операция порождает исключение `bad_typeid`. Метод `name()` класса `type_info` возвращает строку `char *` с именем типа, а операции == и != позволяют сравнить два типа.

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:

- именованные константы в C++;
- неизменяемые параметры;
- встраиваемые функции;
- ссылка;
- передача по ссылке;
- возврат ссылки;
- операция new;
- операция delete;
- набор одноименных функций;
- функция из C;
- значение по умолчанию для параметра;
- потоки ввода-вывода;
- пространство имен;
- использование имен из пространства имен;
- неименованное пространство имен;
- 4 специализированные операции преобразования.

2. Определите следующие функции:

- создание матрицы (двумерного массива) заданного размера n x m;
- удаление матрицы (какие параметры, кроме указателя на матрицу, нужно передать этой функции?);
- ввод матрицы (функция должна вводить размеры матрицы, создавать матрицу соответствующих размеров, вводить значения элементов и затем возвращать указатель на матрицу и её размеры через параметры).

2 КЛАССЫ

Термины, знание которых необходимо для понимания главы:

- структуры данных;
- поле;
- обращение к полям структуры;
- динамическое выделение памяти.

2.1 Основные определения

Программа с точки зрения ООП является совокупностью взаимодействующих между собой объектов. *Объект* представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области.

Каждый объект обладает состоянием, поведением и идентичностью и является экземпляром некоторого класса. *Класс* – это множество объектов, имеющих схожую структуру и поведение.

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой описание реального объекта в виде данных и операций для работы с ними.

Данные-элементы класса называются *полями*, в них хранится текущее состояние объекта. Операции могут быть реализованы как *методы* (функции-элементы) или обычные функции. Операции могут быть следующих видов:

- *конструктор* служит для инициализации объекта;
- *деструктор* необходим для освобождения используемых объектом ресурсов;
- *селектор* позволяет считать состояние объекта, не изменяя его;
- *модификатор* используется для изменения состояния объекта;
- *итератор* позволяет получить доступ к частям объекта в некотором порядке.

2.2 Определение класса. Спецификаторы доступа

Синтаксис:

```
class имя_класса {
 элементы_класса
};
```

или

```
struct имя_класса {
 элементы_класса
};
```

С помощью спецификаторов доступа можно управлять видимостью элементов класса. Спецификатор `private` (закрытый) означает видимость элементов только для методов и друзей класса. Спецификатор `protected` (защищенный) означает видимость элементов только для методов и друзей класса и его наследников (наследники могут видеть эти элементы только в объектах собственного типа). Спецификатор `public` (открытый) означает видимость элементов из любого кода. Действие любого спецификатора распространяется до следующего спецификатора или до конца объявления класса. По умолчанию для элементов класса, объявленного с помощью `struct`, установлен доступ `public`, а для класса, объявленного с помощью `class`, – `private`.

Рекомендуется все поля в `class` делать закрытыми или защищенными элементами, а в `struct` – открытыми.

Пример объявления класса:

```
const int Stack_size=100; // размер стека
class Stack {
  int s[Stack_size]; // поля в закрытом разделе
  int t;
public:
  void clear(); // методы в открытом разделе
  bool isEmpty();
  bool isFull();
  void pop();
  int top();
  void push(int);
};
```

2.3 Определение и вызов методов. Указатель this

Объявление класса обычно помещается в заголовочном файле (.h), а сами методы определяются в файле реализации (.cpp). Определения простых методов можно поместить в объявление класса, в этом случае они будут считаться определенными со спецификатором `inline`. При определении методов вне интерфейса класса непосредственно перед именем метода необходимо указать имя класса и символы `::`

Пример объявления класса:

```
const int Stack_size=100; // размер стека
class Stack {
  int s[Stack_size];
  int t;
public:
  void clear() { t=-1; } // встраиваемые методы
  bool isEmpty() { return t==-1; }
  bool isFull() { return t==Stack_size-1; }
  void pop();
  int top();
  void push(int);
};
// классы для информирования об ошибках
class StackEmpty {};
class StackFull {};
```

Реализация методов класса:

```
void Stack::pop()
{ if(!isEmpty())
    --t;
```

```
}
int Stack::top()
{ if(isEmpty())
    throw StackEmpty(); // сообщаем об ошибке
  return s[t];
}
void Stack::push(int v)
{ if(isFull())
    throw StackFull(); // сообщаем об ошибке
  s[++t]=v;
}
```

При вызове метода сначала указывают имя объекта, затем после точки указывают имя метода и список аргументов в скобках. При использовании указателя на объект, вместо точки используют операцию ->

Пример вызова методов:

```
int main()
{ Stack s1;
  s1.clear();
  s1.push(10);
  cout<<s1.top()<<"\n";
  s1.pop();
...
  Stack *s2;
  s2=new Stack;
  s2->clear();
  s2->push(10);
  cout<<s2->top()<<"\n";
  s2->pop();
...
}
```

Адрес объекта, к которому примеряется метод, передается с помощью скрытого параметра и доступен внутри метода под именем `this`. С помощью указателя `this` можно также обратиться к полю при совпадении имени поля и имени параметра метода. Другой вариант решения этой проблемы – написать имя класса и `::` перед именем поля.

```
struct time {
  int h,m;
  void set(int h, int m)
  { this->h=h;
    time::m=m;
  }
};
```

2.4 Конструктор

Определение: Конструктор – это специальный метод, имя которого совпадает с именем класса. Конструктор не имеет возвращаемого значения, даже `void`. Конструктор не может быть объявлен как `const`, `static` или `virtual`. Класс может иметь несколько конструкторов с разными сигнатурами.

Синтаксис: **имя_класса** (**тип_параметра1**, **тип_параметра2**, ...) ;

Задача: Инициализировать *все* поля.

Назначение: Вызывается компилятором автоматически при создании объектов:

- для локальных объектов – при выполнении оператора, в котором они объявлены;
- для глобальных статических объектов – перед вызовом функции `main` (порядок вызовов *не определен*), для собственных статических значений функции – при первом выполнении этой функции;
- для объектов, создаваемых в динамической памяти – при выполнении операции `new`.

Замечания:

1. Перед выполнением тела конструктора, компилятором автоматически вызываются конструкторы для базовых классов, затем конструкторы для полей в порядке, указанном при объявлении класса.

2. Можно указать аргументы для вызова конструкторов базовых классов и полей, используя список инициализации, который записывается после : между заголовком и телом конструктора. В списке инициализации через запятую пишутся имена базовых классов или полей, в скобках указываются аргументы для конструкторов. Для базовых классов и полей, не указанных в списке инициализации вызываются конструкторы по умолчанию. Базовые классы и поля в списке инициализации рекомендуется перечислять в порядке, указанном при объявлении класса. Можно указать их в другом порядке, но ошибки использования неинициализированных полей станут менее заметными, так как вызовы конструкторов всё равно будут происходить в порядке, заданном при объявлении класса.

3. Конструктор не может быть вызван как метод, он вызывается только при создании объекта:

- Выражение **имя_класса**(**аргументы**) создает временный объект, уничтожаемый после выполнения оператора, в котором используется это выражение (скобки обязательны, даже если список аргументов пустой).

– Выражение `new` **имя_класса(аргументы)** создает объект в динамической памяти (если список аргументов пустой, скобки можно не писать).
– Оператор объявления **имя_класса имя_объекта(аргументы)**; создает новый объект в статической памяти или на стеке (если список аргументов пустой, скобки не пишутся).

Примеры:

```
class Stack {
  int size,t;
  int *s;
public:
  Stack(int);
  bool isEmpty() { return t==-1; }
  bool isFull() { return t==size-1; }
  void pop();
  int top();
  void push(int);
};
Stack::Stack(int size)
{
  Stack::size=size;
  s=new int [size];
  t=-1;
}
void fun(const Stack &);
int main()
{
  Stack s1(100); // создается объект на стеке
  s1.push(10);
  ...
  fun(Stack(100)); // создается временный объект
                   // и передается по ссылке в функцию
  Stack *s2;
  s2=new Stack(100); // объект создается в динамической памяти
  s2->push(10);
  ...
}
```

Аналогичный синтаксис можно использовать для создания объектов стандартных типов:

```
int a(100); // вместо int a=100;
int *b;
```

```
b=new int(100); // выделить память из кучи для int
                // и инициализировать её числом 100
```

Пример определения конструктора со списком инициализации:

```
Stack::Stack(int size): size(size),
      // здесь нет коллизии имен: поле size(параметр size)
  t(-1), s(new int [size])
{}
```

2.5 Деструктор

Определение: Деструктор – это специальный метод без параметров, имя которого состоит из символа ~ и имени класса. Деструктор не имеет возвращаемого значения, даже `void`. Деструктор не может быть объявлен как `const` или `static`, но может быть `virtual`.

Синтаксис: **~имя_класса** `();`

Задача: Освободить используемые объектом ресурсы, обычно это динамически выделяемая память, реже каналы ввода-вывода, связанные с файлами, устройствами и т. п.

Назначение: Вызывается компилятором автоматически при уничтожении объектов:

- для локальных объектов – при выходе из блока, в котором они объявлены;
- для статических объектов – после завершения функции `main`;
- для объектов, созданных в динамической памяти – при выполнении операции `delete`.

Замечания:

1. После выполнения тела деструктора, компилятором автоматически вызываются деструкторы для полей, а затем базовых классов в порядке, обратном указанному при объявлении класса.

2. Если деструктор явно не определен, компилятор сам создает деструктор с пустым телом: **~имя_класса** `(){}`

3. В случае использования классом динамической памяти необходимо определять деструктор самостоятельно.

4. Если класс будет использован как базовый в иерархии класс, рекомендуется сделать его деструктор виртуальным. Тогда при выполнении операции `delete` будет вызываться правильный деструктор, независимо от типа указателя.

5. Не рекомендуется вызывать деструктор явно, единственным исключением является случай повторной инициализации объекта (см. пример в 3.1)

Пример:

```
class Stack {
  int size,t;
  int *s;
public:
  ~Stack();
  ...
};
Stack::~Stack()
{ delete[] s;
}
```

2.6 Конструктор по умолчанию

Определение: Конструктор по умолчанию – это конструктор, который можно вызвать без аргументов. Конструктор может иметь параметры, но все они должны иметь значения по умолчанию.

Синтаксис:

имя_класса ( ) ;

имя_класса (**тип_параметра1** =**значение_по_умолчанию**,
тип_параметра2 =**значение_по_умолчанию**, ...) ;

Задача: Такая же, как у любого конструктора – инициализировать все поля.

Назначение: Без этого конструктора невозможно создать массив объектов.

Замечания:

1. Если в классе не определен **ни один** конструктор, компилятор создает сам конструктор по умолчанию. В этом автоматически созданном конструкторе вызываются конструкторы по умолчанию для всех базовых классов и полей.

2. Если у какого-то поля или базового класса нет конструктора по умолчанию, компилятор при попытке самостоятельно создать конструктор по умолчанию выдаст сообщение об ошибке.

Примеры:

```
class String {
  int len;
  char * str;
public:
  String(const char *s=""); // конструктор по умолчанию
  ...
};
```

```
String::String(const char *s):len(strlen(s)),
    str(new char[len+1])
{ strcpy(str,s);
}
int main()
{ String a; // создание строки, используя конструктор
            // по умолчанию
  String b[100]; // создание массива строк
  String c(); // это не определение объекта, а объявление
              // функции без параметров!!!
}
```

2.7 Конструктор копий

Определение: Конструктор копий – это конструктор, у которого один аргумент – ссылка на константный объект определяемого класса.

Синтаксис: **имя_класса** (`const` **имя_класса**`&`)`;`

Задача: Создание точной копии объекта, переданного в качестве аргумента.

Назначение: Конструктор вызывается компилятором при передаче аргументов в функцию по значению или возврате значения определяемого класса из функции. Можно также использовать явно при создании нового объекта, имеющего состояние одинаковое с другим объектом.

Замечания:

1. Если конструктор копий не определяется программистом, компилятор создает его сам. В этом автоматически созданном конструкторе вызываются конструкторы копий для всех базовых классов и полей.

2. В большинстве случаев нет необходимости определять конструктор копий самостоятельно.

3. Но если класс содержит указатели на динамически выделяемую память, то при создании копии произойдёт дублирование идентичности этих значений. В результате изменение одного объекта приведёт к изменению другого объекта. Поэтому в случае использования указателей необходимо определять конструктор копий самостоятельно.

4. Если создание копий не требуется, рекомендуется объявить конструктор копий в разделе private, не определяя его реализацию (в C++11 рекомендуется писать `=delete` после заголовка).

Примеры:

```
class Stack {
  int size,t;
  int *s;
public:
```

```
  Stack(const Stack &); // конструктор копий
  ...
};
Stack::Stack(const Stack &a):
  size(a.size),t(a.t),s(new int[size])
{
  for(int i=0; i<=t; ++i)
    s[i]=a.s[i];
}
Stack reverse(Stack x) // параметр x функции reverse
                       // передается по значению
{ Stack z(100);
  while(!x.isEmpty())
  { z.push(x.top());
    x.pop();
  }
  return z; // при возврате сначала вызывается конструктор копий,
            // чтобы создать копию z, а затем деструктор для z
}
int main()
{
  Stack a(100);
  ...
  Stack b(a); // b - это копия a
  ...
  a=reverse(b); // перед вызовом reverse будет вызван конструктор
// копий для создания копии b, также компилятор выделит память
// для возвращаемого значения, для инициализации этой памяти
// при завершении reverse будет вызван конструктор копий
}
```

2.8 Конструктор-преобразователь

Определение: Конструктор-преобразователь – это конструктор, который можно вызвать с одним аргументом. Конструктор может иметь более одного параметра, но все они должны иметь значения по умолчанию.

Синтаксис:

имя_класса (**другой_тип**) ;

имя_класса (const **другой_тип**&) ;

Задача: Инициализировать все поля, используя для установки начального состояния значение другого типа.

Назначение: Используется для преобразования в определяемый тип значения другого типа. Компилятор может вызывать его самостоятельно,

если аргумент функции или операции имеет тип **другой_тип**, а параметр – тип **имя_класса**.

Замечания:

1. Если конструктор с одним аргументом не должен использоваться неявно для преобразования, перед ним нужно написать `explicit` (явный).

Примеры:

```
class String {
public:
  String(const char *s=""); // конструктор-преобразователь
  ...
};
void fun(const String &s);
int main()
{
  String a("ABC"); // создание строки, вызывается конструктор
  char s[]="ABCDE";
  fun(s); // перед вызовом функции будет вызван
      // конструктор-преобразователь, временный объект будет
      // уничтожен после выполнения оператора
  a=s; // OK, будет выполнено неявное преобразование
       // с созданием временного объекта
  a=String(s); // явный вызов конструктора для создания
               // временного объекта
  a=(String)s; // явный вызов операции преобразования,
               // устаревшая форма
  a=static_cast<String>(s); // явный вызов операции
               // преобразования, новая форма
}

class Stack {
public:
  explicit Stack(int); // конструктор, требующий явного
          // вызова для преобразования
  ...
};
void fun(Stack x);
int main()
{
  Stack a(100);
  ...
```

```
  fun(100);  // ошибка, нет преобразователя int -> Stack
  fun(Stack(100));  // OK
  a=Stack(100);  // OK, явный вызов конструктора
  a=100;  // ошибка, нет преобразователя int -> Stack
  a=(Stack)100;  // OK, явный вызов операции
                 // преобразования, устаревшая форма
  a=static_cast<Stack>(100);  // OK, явный вызов операции
        // преобразования, новая форма
}
```

2.9 Специальные элементы класса

Поля, значения которых не меняются на протяжении всего времени существования объекта, рекомендуется описывать со спецификатором `const`. Инициализировать такие поля нужно в списке инициализации конструктора.

Методы, не изменяющие состояния объекта (селекторы), рекомендуется описывать как константные. Спецификатор `const` указывается в заголовке метода после списка параметров. Только константные методы можно применять к константным объектам. Напротив, к неконстантным объектам можно применять как обычные методы, так и константные. Можно определить одновременно константный и обычный методы с одинаковым списком параметров, тогда для константных объектов будет вызываться константный метод, для неконстантных – обычный.

```
class Vector {
  const int size;
  double *const data;
public:
  Vector(int size):
        size(size),data(new double[size]) {}
  ~Vector() { delete []data; }
  Vector(const Vector &);
  int length() const { return size; }
  double get(int i) const;
  double &get(int i);
};
Vector::Vector(const Vector &v):
  size(v.size),data(new double[size])
{
  for(int i=0; i<size; ++i)
    data[i]=v.data[i];
}
```

```
class BadIndex {};
double Vector::get(int i) const
{ if(i<0 || i>=size) throw BadIndex();
  return data[i];
}
double & Vector::get(int i)
{ if(i<0 || i>=size) throw BadIndex();
  return data[i];
}
```

Если некоторые поля нужно изменять в константных методах, то их нужно описать со спецификатором `mutable` (использовать эту возможность с аккуратностью).

Поля, описанные с помощью спецификатора `static`, являются общими для всех объектов класса. Фактически это глобальные переменные, видимостью которых можно управлять с помощью спецификаторов доступа. В файле реализации необходимо разместить поле в статической памяти и указать её начальное значение. Память, занимаемая статическим полем, не учитывается при определении размера объект с помощью операции `sizeof`.

Статические методы могут обращаться только к статическим полям и вызывать другие статические методы. Этим методам не передается указатель `this`. Вызывать эти методы можно либо обычным образом, либо через имя класса и `::`

```
// объявление
class One {
  // класс, допускающий создание только одного экземпляра
   int a;
   static One *o; // указатель на единственный экземпляр
   One(int x):a(x) {} // ограничиваем доступ к конструктору
   One(const One&)=delete; // запрещаем создание копий
   One& operator=(const One&)
          =delete; // и операцию присваивания
public:
   static One *instance(); // получение экземпляра класса
   ~One() { o=NULL; } // разрешаем уничтожение
   int get() { return a; } // прочие методы
};
// реализация
One * One::o=NULL;
One * One::instance()
{ if(o==NULL) //если нет экземпляра
```

```
    o=new One(10); // создать его
  return o;
}
// использование
int main()
{
  cout<<One::instance()->get()<<"\n";
  ...
  delete One::instance();
}
```

При использовании одновременно спецификаторов `const` и `static` получается аналог глобальной константы, видимостью которой можно управлять. Значение такой константы нужно непосредственно указывать при объявлении.

```
class Stack {
  static const int size=100;
  int s[size];
  int t;
public:
  void clear() { t=-1; }
  bool isEmpty() const { return t==-1; }
  bool isFull() const { return t==size-1; }
  void pop();
  int top() const;
  void push(int);
};
```

2.10 Друзья класса

Вызов метода похож на вызов обычной функции, за исключением того, что объект, к которому применяется метод, передается как скрытый аргумент. Получение длины строки можно реализовать как метод:

```
class String {
  int len;
  char * str;
public:
  int length() const { return len; }
  ...
};
int main()
{ String s;
  cout<<s.length()<<"\n";
}
```

Или можно использовать обычную функцию, которой строка передается в качестве параметра. Тогда эта функция для получения доступа к закрытым элементам класса должна быть объявлена внутри класса со спецификатором `friend`

```
class String {
  int len;
  char * str;
public:
  friend int length(const String &);
  ...
};
int length(const String & s)
{ return s.len; // необходим доступ к закрытому элементу
}
int main()
{ String s;
  ...
  cout<<length(s)<<"\n";
}
```

Как реализовать операцию – через метод или дружественную функцию – зависит только от предпочтений программиста, но чаще через обычные функции реализуются такие операции, которым необходимы два или более объекта определяемого типа.

Функция может быть дружественной сразу несколько классам, можно объявить дружественным метод другого класса. Если многие методы другого класса должны иметь доступ к закрытым элементам, то можно объявить дружественным весь класс:

`friend class` **имя_класса;**

Спецификаторы доступа не оказывают влияния на друзей класса. Друзей класса можно описывать в любом разделе класса.

2.11 Рекомендации по проектированию

Все поля и вспомогательные методы в `class` следует делать закрытыми или защищенными элементами. Для доступа к полям пользователям класса могут быть предоставлены тривиальные селекторы и модификаторы. Для запрета чтения или изменения некоторого поля соответствующий селектор или модификатор не определяется.

```
class A {
  int value;
public:
```

```
  // тривиальный селектор
  int getValue() const { return value; }
  // тривиальный модификатор
  void setValue(int newValue) { value=newValue; }
};
```

Друзьями класса следует делать только операции этого класса. Прочие функции и классы должны использовать открытый интерфейс. Более того, некоторые операции могут быть реализованы через обычные функции, не являющиеся друзьями класса, если открытого интерфейса достаточно для их реализации.

Методы и дружественные функции должны непосредственно обращаться к полям для получения их значений, а не через тривиальные селекторы. Более того, если какая-то операция класса, реализованная через обычную функцию, использует тривиальные селекторы для получения значений полей, необходимо сделать ее дружественной и обращаться к полям непосредственно.

Необходимо минимизировать интерфейс класса. В набор операций необходимо включать те операции, которые нельзя эффективно реализовать через другие методы и функции.

Методы, не изменяющие состояния объекта, должны быть объявлены как константные.

Методы, изменяющие состояние объекта, не должны возвращать информацию о состоянии объекта (текущем или предыдущем).

Взаимосвязанные поля (например, день и месяц в дате) должны устанавливаться вызовом одного метода.

Если параметр не изменяется в теле функции, а его размер больше или равен 16 байт или он содержит указатель на динамически выделенную память, то его необходимо передавать как ссылку на константный объект. Иначе можно передавать по значению.

Для классов с динамическим выделением памяти обязательно определяйте деструктор, конструктор копий и операцию присваивания или запретите их автоматическое создание компилятором.

Объявляйте деструкторы виртуальными в классах, являющихся базовыми в иерархии.

В объявлении класса определяйте только методы, состоящие из одного оператора присваивания или `return`.

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:

– объект;

– класс;
– метод;
– определение класса;
– спецификатор доступа;
– определение метода;
– вызов метода;
– указатель this;
– конструктор;
– список инициализации;
– деструктор;
– конструктор по умолчанию;
– конструктор копий;
– конструктор-преобразователь;
– константные поля;
– константные методы;
– статические поля;
– статические методы;
– константы класса;
– дружественная функция.

2. Спроектируйте интерфейс и реализуйте класс «Очередь целых чисел» с использованием циклического буфера, размер которого задается в конструкторе. Необходимые операции: проверка на наличие элементов в очереди и полное заполнение буфера, удаление первого элемента, получение первого элемента, добавление элемента, отладочная печать текущего состояния очереди. Напишите программу для тестирования реализованного класса.

3 ПЕРЕГРУЗКА ФУНКЦИЙ И ОПЕРАЦИЙ

Термины, знание которых необходимо для понимания главы:

– приоритет операции;
– ассоциативность;
– унарная операция;
– бинарная операция;
– постфиксная операция;
– префиксная операция;
– динамическое выделение памяти;
– присваивание;
– ссылка.

3.1 Правила связывания

Если требуется реализация одной и той же операции для данных различных типов, то можно определить нескольких функций с одним и тем же именем. Такие функции называют перегруженными. Перегруженные функции должны быть объявлены в одной области видимости (блоке, классе или пространстве имен) и иметь различные сигнатуры (список параметров функции в скобках). Сигнатуры должны отличаться количеством параметров или их типами. Возвращаемое значение в сигнатуру не входит.

Связывание – это процесс выбора экземпляра функции из набора в одной области видимости, который соответствует набору аргументов при ее вызове. Различают раннее (статическое, во время компиляции) и позднее (динамическое, во время выполнения) связывание. В C++ используется раннее связывание, но механизм виртуальных методов можно рассматривать как позднее связывание по первому, скрытому аргументу.

Выбор функции при связывании происходит на основе следующих приоритетов:

1) Точное соответствие.

2) Повышение точности: `char, short, wchar_t, enum, bool` → `int`; `float` → `double`.

3) Стандартные преобразования: любой числовой тип в любой числовой тип (близость типов во внимание не принимается); любой указатель в `void *`; константа 0 к любому числовому типу или указателю; указатель или ссылка на производный класс к указателю или ссылке на базовый класс.

4) Преобразования через конструкторы-преобразователи или операции преобразования, определенные программистом.

5) Соответствие `...` (специальное обозначение для любого количества параметров, см. printf).

Выбирается функция или метод с большим точным числом соответствий, при одинаковом количестве точных соответствий – рассматривается количество повышений точности, затем учитывается число стандартных преобразований и т.д. Если две или более функции из набора получают одинаковый приоритет, то компилятор выводит сообщение об ошибке.

```
void f(double);         // 1
void f(const char *); // 2
void f(...);            // 3
int main()
{
```

```
  int a[10];
  float b;
  f(b); // 1, повышение точности
  f('A'); // 1, стандартное преобразование
  f("A"); // 2, точное соответствие
  f(0); // ошибка, соответствует функциям 1 и 2
  f(1); // 1, стандартное преобразование
  f(a); // 3, соответствие ...
}
```

3.2 Правила перегрузки операций

1. Нельзя определить новый лексический символ для операции.

2. Нельзя изменить приоритет, арность (т.е. количество аргументов) и ассоциативность операций.

3. Нельзя перегрузить операцию для стандартных типов данных.

4. Нельзя перегружать операции `. :: .* ?: sizeof typeid`.

5. Операции `= [] () ->` можно перегружать только как методы.

6. Не рекомендуется перегружать операции `, || &&`.

7. Поведение перегруженных операций должно соответствовать поведению этих операций для стандартных типов данных: операция + не должна изменять своих аргументов, операция = должна возвращать ссылку на левый аргумент и т.п.

Формат перегрузки унарной операции @ как метода:

тип_результата `operator@()`

или

тип_результата `operator@() const`

Формат перегрузки унарной операции @ как функции:

тип_результата `operator@(`**имя_класса** `& )`

или

тип_результата `operator@(const` **имя_класса** `& )`

Исключение: В случае постфиксных `++` и `--` добавляется фиктивный параметр типа `int`.

как метод: **имя_класса** `operator@(int)`

как функция: **имя_класса** `operator@(`**имя_класса** `& , int)`

Формат перегрузки бинарной операции @ как метода:

тип_результата `operator@(`**тип_параметра2**`)`

или

тип_результата `operator@(`**тип_параметра2**`) const`

Левый аргумент операции передается как `*this` и может быть только

определяемого типа. Конструкторы-преобразователи к левому аргументу не применяются.

Формат перегрузки бинарной операции @ как функции:

тип_результата `operator@` (**тип_параметра1 , тип_параметра2**)

где по крайней мере один из параметров должен относиться к определяемому типу. Если левый аргумент может иметь тип, отличный от определяемого, необходимо определять операцию как функцию.

Замечания:

1. При перегрузке операции как функции, эту функцию нужно делать дружественной классу только в том случае, если для её эффективной реализации необходим доступ к закрытым элементам класса, иначе лучше определять обычную функцию (см. 2.11)

2. Если операция присваивания не определяется программистом, компилятор создает её сам. В этом автоматически созданном методе вызываются операции присваивания для всех базовых классов и полей.

3. В большинстве случаев нет необходимости определять операцию присваивания самостоятельно.

4. Но если класс содержит указатели на динамически выделяемую память, то при присваивании произойдет дублирование идентичности этих значений. В результате изменение одного объекта приведёт к изменению другого объекта. Поэтому в случае использования указателей необходимо определять операцию присваивания самостоятельно.

5. Если присваивание объектов не требуется, рекомендуется объявить операцию присваивания в разделе private, не определяя её реализацию (в C++11 рекомендуется писать `=delete` после заголовка).

Кроме присваивания компилятор сам определяет операцию взятия адреса.

3.3 Примеры перегрузки операций

Перегрузка операции присваивания:

а) для случая, когда перевыделения памяти не происходит

```
class BadSize {}; // класс для информирования об ошибке
class Vector {
  const int size;
  double *const data;
public:
  Vector &operator=(const Vector &);
};
Vector & Vector::operator=(const Vector &v)
{ if(size!=v.size)
```

```
    throw BadSize();
  for(int i=0;i<size;++i)
    data[i]=v.data[i];
  return *this;
}
```

б) для случая, когда память перевыделяется

```
class String {
  int len;
  char * str;
public:
  String &operator=(const String &);
};
String & String::operator=(const String &s)
{ String t(s); // создание копии
  std::swap(len,t.len); // обмен состояний объекта t и *this
  std::swap(str,t.str);
  return *this; // объект t со старым состоянием *this
                // уничтожается
}
```

Определение постфиксного ++ через префиксный:

```
class A {
  ...
public:
  A& operator++();
};
inline A operator++(A &x, int)
{ A t(x);
  ++x;
  return t;
}
```

Определение + через += :

```
class A {
  ...
public:
  A& operator+=(const A&);
};
inline A operator+(A x, const A &y) // первый параметр
                    // передается по значению
{ return x+=y;
}
```

Определение != через ==:

```
class A {
  ...
public:
  friend bool operator==(const A&, const A&);
};
inline bool operator!=(const A &x, const A &y)
{ return !(x==y);
}
```

Определение > >= <= == через <:

```
class A {
  ...
public:
  friend bool operator<(const A&, const A&);
};
inline bool operator>(const A &x, const A &y)
{ return y<x;
}
inline bool operator>=(const A &x, const A &y)
{ return !(x<y);
}
inline bool operator<=(const A &x, const A &y)
{ return !(y<x);
}
inline bool operator==(const A &x, const A &y)
{ return !(x<y || y<x);
}
```

Перегрузка операции индексирования:

```
class BadIndex {}; // класс для информирования об ошибке
class Vector {
  const int size;
  double *const data;
public:
  double &operator[](int); // для неконстантных объектов
  double operator[](int) const; // для константных объектов
};
double & Vector::operator[](int i)
{ if(i<0 || i>=size)
    throw BadIndex();
  return data[i];
}
```

```
double Vector::operator[](int i) const
{ if(i<0 || i>=size)
    throw BadIndex();
  return data[i];
}
int main()
{ Vector a(10); // вектор из 10 элементов
  a[4]=a[5]*2; // изменяем элемент вектора
}
```

*Перегрузка &, -> и *:*

```
class A;
class Ptr { // умный указатель
public:
  A *operator->();
  A &operator*();
  ...
};
class A {
public:
  Ptr operator&(); // создание умного указателя
  ...
};
```

Перегрузка операции () для создания функционалов:

```
struct str_less { // функционал для сравнения двух строк
  bool operator()(const char *a, const char *b)
  { return strcmp(a,b)<0;
  }
};
int main()
{ char *s[100];
  sort(s,s+100,str_less()); // использование функционала
}
```

Перегрузка операции () для индексации по двум или более ключам:

```
class Matrix {
public:
  Matrix(int n, int m);
  double &operator()(int i, int j);
  double operator()(int i, int j) const;
};
int main()
{ Matrix a(10,10); // матрица 10x10
```

```
  a(4,5)=1; // изменяем элемент матрицы
}
```

Перегрузка операций << и >> ввода-вывода:

```
class Point {
  double x,y;
public:
  friend istream &operator>>(istream &, Point&);
  friend ostream &operator<<(ostream &,
              const Point&);
};
istream &operator>>(istream &s, Point &p)
{ char c;
  return s>>c>>p.x>>c>>p.y>>c;
    // можно проверить, какие символы до, между и после чисел
}
ostream &operator<<(ostream &s, const Point &p);
{ return s<<"("<<p.x<<","<<p.y<<")"; // \n не выводится!
}
```

Формат ввода должен точно соответствовать формату вывода, не должно быть подсказок на ввод.

3.4 Операция преобразования

Операция преобразования определяется как метод с заголовком вида:

`operator` **другой_тип**`();`

и используется при необходимости для преобразования из определяемого типа в **другой_тип**.

Не следует определять одновременно операцию преобразования и обратный к ней конструктор-преобразователь.

```
class String {
  int len;
  char * str;
public:
  String(const char *s=""); // конструктор-преобразователь
  String& operator=(const String&);
            // операция присваивания
  operator const char *() const { return str; }
             // операция преобразования
  friend bool operator==(const String &,
                    const String &);
  ...
};
```

```
int main()
{
  int x;
  String fmt, txt("ABC");
  fmt="%d\n"; // Ok, неявное преобразование const char * -> String
  printf(fmt,x);
              // Ok, неявное преобразование String -> const char *
  if(fmt==txt) // Ok, преобразований не нужно
    ...
  if(fmt=="ABC") // Ошибка, какое преобразование выполнять?
      // можно сравнить два String, а можно два char *
    ...
}
```

В таких случаях не определяют операцию преобразования, вместо неё делают обычный метод:

```
class String {
  int len;
  char * str;
public:
  String(const char *s=""); // конструктор-преобразователь
  String& operator=(const String&);
            // операция присваивания
  const char * c_str() const { return str; }
          // метод вместо операции преобразования
  friend bool operator==(const String &,
              const String &);
  ...
};
int main()
{
  int x;
  String fmt, txt("ABC");
  fmt="%d\n"; // Ok, неявное преобразование const char * -> String
  printf(fmt.c_str(),x); // Ok, нужно явно вызывать метод
  if(fmt==txt) // Ok, преобразований не нужно
    ...
  if(fmt=="ABC")
      // Ok, неявное преобразование const char * -> String
    ...
}
```

3.5 Перегрузка new и delete

Реализовать выделение участков памяти одинакового размера из некоторого пула можно более эффективно, чем выделение участков памяти произвольного размера. Кроме того, при работе с участками памяти одинакового размера не возникает проблема фрагментации, когда свободная память делится на множество участков небольшого размера.

Чтобы повысить эффективность программ, использующих динамические структуры данных, можно создать собственные варианты операций `new` и `delete` для выделения памяти под объекты некоторых классов, например, элементов списка. Эти операции определяются как статические методы класса, отдельно для одиночных объектов и массивов:

```
static void *operator new(size_t, доп_параметры);
static void *operator new[](size_t, доп_параметры);
static void operator delete(void *, size_t);
static void operator delete[](void *, size_t);
```

Значения для дополнительных параметров можно указать при вызове `new` следующим образом:

```
new (доп_аргументы) тип (список_выражений);
```

Можно также перегрузить глобальную операцию `new`, указав дополнительные параметры. Глобальная операция `new` уже перегружена с дополнительным параметром типа `void *` (память не выделяется, а возвращается указанный адрес, используется для инициализации с помощью конструктора произвольного участка памяти, пример использования показан ниже) и `nothrow_t` (в случае ошибки не возникает исключительной ситуации `bad_alloc`, операция возвращает 0).

Пример повторной инициализации объекта с помощью перегруженной формы оператора `new` и явного вызова деструктора:

```
Vector a(100);
... // действия с вектором размером 100
a.~Vector();
new(&a) Vector(200);
... // действия с вектором размером 200
```

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:
   - перегруженные функции;
   - сигнатура;
   - связывание;

– количество соответствий;
– точное соответствие;
– повышение точности;
– стандартные преобразования;
– преобразования, определенные программистом;
– операция индексирования;
– операция преобразования.

2. Определите класс «День недели». В классе должны быть два конструктора-преобразователя: номер дня от 1 до 7 в день недели и строки из двух букв (ПН, ВТ, …, ВС) в день недели. Метод day() должен возвращать номер дня. В классе должны быть реализованы постфиксные и префиксные операции ++ и --, операции сравнения на равенство и неравенство, операции +=, -=, + и -, где 2-м аргументом является целое число — количество дней, операции ввода и вывода в виде сокращенного названия дня недели из двух букв. Напишите программу для тестирования реализованного класса.

4 ШАБЛОНЫ

Термины, знание которых необходимо для понимания главы:
– функция;
– определение функции;
– определение класса.

4.1 Шаблоны функций

С помощью шаблонов функций можно определить алгоритм, который может быть применен к данным различных типов. Компилятор генерирует код при первом вызове по заданному шаблону. Шаблонная функция определяется как обычная функция, но её определение должно быть в заголовочном файле и перед заголовком функции необходимо написать:

`template <`**параметры_шаблона**`>`

Параметрами шаблона чаще всего являются типы обрабатываемых функцией данных (перед именем такого параметра нужно написать `typename` или `class`), но допустимо также передавать константы и объекты, как и обычной функции. Для параметров шаблона можно указать значения по умолчанию.

Пример шаблона функции:

```
template <typename T>
inline T maximum(T a, T b)
{
  return  a < b ? b : a;
}
```

Если параметры шаблона относятся только к типам параметров функции, то её можно вызывать обычным образом, компилятор определит аргументы шаблона самостоятельно. В остальных случаях их нужно указать явно после имени функции в <>:

```
int a,b,c;
double d,e;
c=maximum(a,b); // оба аргумента функции имеют тип int,
                // значит аргумент шаблона - тип int
e=maximum<double>(d,a); // тип аргумента шаблона нельзя
// определить по типу аргументов функции, указываем явно тип double
```

Как и обычные функции, шаблоны функций можно перегружать. Из набора шаблонных функций вызывается функция с наиболее специализированными параметрами, точно соответствующая аргументам при вызове.

Если определены одновременно обычные и шаблонные функции с одинаковым именем и при вызове не указаны аргументы для шаблона в <>, то сначала компилятор выполняет поиск обычной функции с точным соответствием типов параметров, затем шаблона функции с точным соответствием типов параметров, а затем пытается добиться соответствия аргументов с параметрами обычной функции с помощью операций преобразования.

4.2 Шаблоны классов

Перед шаблоном класса необходимо написать:

`template <`**параметры_шаблона**`>`

Методы шаблонного класса должны быть определены в заголовочном файле либо в объявлении класса (тогда они являются встраиваемыми), либо как шаблонные функции.

Пример шаблона класса:

```
template <typename T=int, int size=100>
class Stack {
  T s[size];
  int t;
public:
  Stack():t(-1) {} // встраиваемые методы
  bool isEmpty() { return t==-1; }
  bool isFull() { return t==size-1; }
  void pop();
  const T & top();
  void push(const T &);
};
class StackEmpty {}; // классы для информирования об ошибках
```

```
class StackFull {};
template <typename T, int size>
void Stack<T,size>::pop()
{ if(!isEmpty())
    --t;
}
template <typename T, int size>
const T & Stack<T,size>::top()
{ if(isEmpty())
    throw StackEmpty();
  return s[t];
}
template <typename T, int size>
void Stack<T,size>::push(const T& v)
{ if(isFull())
    throw StackFull(); // сообщаем об ошибке
  s[++t]=v;
}
```

При создании объектов шаблонного класса после имени класса в <> указываем аргументы шаблона:

```
Stack<int,200> a; // стек целых чисел размером 200
Stack<double> b;  // стек вещественных чисел размером 100
Stack<> c;        // стек целых чисел размером 100
```

4.3 Специализация шаблонов

Можно специализировать (изменять реализацию) для определенных типов данных шаблон функции, отдельные методы шаблона класса или шаблон класса целиком.

При специализации функций и отдельных методов допускается только *полная специализация*. Перед определением специализации пишется `template <>`, а после имени функции или класса добавляется список аргументов основного шаблона в <>. Аналогичной функциональности можно добиться, если определить обычную функцию или перегрузить шаблон функции. Разница в том, что генерация кода для обычной функции будет происходить при компиляции её определения (т.е. всегда), а для шаблона и специализации – при первом использовании (инстанцировании). Разница между специализацией и перегрузкой шаблона чисто концептуальная.

Пример специализации шаблона функции:

```
template<>
inline const char * maximum<const char *>
       (const char *a, const char * b)
```

```
{ return strcmp(a,b)>0?a:b;
}
```

Специализация шаблона класса определяется как шаблон класса, но нельзя указывать значения по умолчанию для параметров шаблона, а после имени класса указывается список аргументов основного шаблона в <>, который позволяет компилятору установить связь между параметрами специализации и основного шаблона. При специализации шаблона класса можно полностью изменить весь интерфейс класса. Допустима частичная специализация шаблона класса.

Пример специализации шаблона класса:

```
template <int size> // частичная специализация
class Stack<char *,size> {
  char *s[size];
  int t;
public:
  Stack():t(-1) {} // встраиваемые методы
  bool isEmpty() { return t==-1; }
  bool isFull() { return t==size-1; }
  ~Stack() { while(!isEmpty()) pop(); }
  void pop();
  const char * top();
  void push(const char *);
};
template <int size>
void Stack<char *,size>::pop()
{ if(!isEmpty())
    delete[] s[t--];
}
template <int size>
const char * Stack<char *,size>::top()
{ if(isEmpty())
    throw StackEmpty();
  return s[t];
}
template <int size>
void Stack<char *,size>::push(const char *v)
{ if(isFull())
    throw StackFull(); // сообщаем об ошибке
  s[++t]=new char[strlen(v)+1];
  strcpy(s[t],v);
}
```

4.4 Инстанцирование шаблонов

Инстанцирование шаблона – это генерация кода функции или класса по шаблону для конкретных параметров. Различают неявное инстанцирование, которое происходит при вызове функции или создании объекта класса, и явное инстанцирование с помощью резервированного слова `template`. Инстанцирование можно делать только в точке программы, где доступна реализация шаблона функции или методов шаблонного класса.

Реализация шаблонов должна быть определена в заголовочном файле, чтобы компилятор смог выполнить инстанцирование шаблона для любых аргументов, но можно скрыть реализацию, используя явное инстанцирование. В заголовочном файле при этом остаётся только объявление шаблонов, а реализация помещается в отдельный файл. Для всех возможных вариантов применения шаблонов в этом модуле выполняется явное инстанцирование. Плюсы: компиляция выполняется быстрее, сохраняется know-how, минус: использование шаблонов с непредусмотренными аргументами приведёт к ошибке при компоновке программы.

Заголовочный файл my.h:

```
template <typename T> T sqr(T);
template <typename T> class X
{
  T x;
public:
  X(T x):x(x){}
  T get()const;
  void set(T);
};
```

Реализация модуля my.cpp:

```
#include "my.h"
template <typename T> T sqr(T x)
{ return x*x;
}
template <typename T> T X<T>::get()
{ return x;
}
template <typename T> void X<T>::set(T x)
{ this->x=x;
}
template double sqr(double);
        // явное инстанцирование шаблона функции
```

```
template int sqr(int);          // для двух типов аргументов
template class X<int>;          // и класса для типа int
```

Применение:

```
#include <iostream>
#include "my.h"
using namespace std;
int main()
{
  cout<<sqr(10)<<"\n";   // OK
  cout<<sqr(1.2)<<"\n";  // OK
  cout<<sqr(10L)<<"\n";  // Ошибка, нет sqr(long)

  X<int> x1(10);
  cout<<x1.get()<<"\n";  // OK
  X<double> x1(1.2);
  cout<<x1.get()<<"\n";  // Ошибка, нет X<double>::get()
  return 0;
}
```

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:
   - шаблон функции;
   - параметр шаблона;
   - шаблон класса;
   - создание объектов шаблонного класса;
   - специализация шаблона;
   - неявное инстанцирование;
   - явное инстанцирование.
2. Определите класс «Очередь» из упражнения 2 главы 2 как шаблонный.

5 НАСЛЕДОВАНИЕ

Термины, знание которых необходимо для понимания главы:

- класс;
- определение класса;
- спецификатор доступа;
- перегрузка функций;
- сигнатура.

5.1 Отношение наследования между классами

Механизм наследования позволяет создавать новые классы на основе существующих путем расширения или изменения их структуры и поведения. Наследование не следует путать с агрегацией (отношением часть-целое) и использованием:

- водитель → автомобиль: водитель сам никуда не может поехать, нет связей между их свойствами, но только водитель может выполнять действия над автомобилем, следовательно, это отношение использования;
- двигатель → автомобиль: двигатель самостоятельно никуда не может поехать (нет колес), характеристики двигателя являются частью свойств автомобиля (максимальная скорость, мощность, тип топлива, его расход и т.п.), следовательно, это отношение агрегации;
- грузовик → автомобиль: грузовик имеет такие же свойства и поведение, как автомобиль, добавляется новый элемент – кузов и возможность перевозить грузы, следовательно, это отношение наследования.

После имени *производного класса* можно написать символ `:` и указать список *базовых классов* через запятую. Перед именем базового класса можно указать спецификатор доступа для наследуемых элементов. По умолчанию для базовых классов в классах, объявленных с помощью `struct`, устанавливается спецификатор доступа `public`, а в классах, объявленных с помощью `class`, – `private`.

Можно изменить уровень доступа к отдельным наследуемым элементам класса (кроме закрытых элементов базового класса, доступа к которым из производного класса нет), используя `using`.

```
class A {
public:
  void f();
  void g();
};
class B : protected A {
public:
  using A::g; // g - public, f - protected
};
```

С помощью `using` можно в производном класса добавить метод к набору методов с тем же именем в базовом классе:

```
class A {
public:
  void f();
```

```
};
class B1 : public A {
public:
  void f(int);
};
class B2 : public A {
public:
  void f(int);
  using A::f;
};
int main()
{ B1 b1;
  b1.f(); // Ошибка, метод перекрыт
  B2 b2;
  b2.f(); // OK, метод перегружен
}
```

5.2 Виртуальные методы и абстрактные классы

Полиморфизм обеспечивает взаимозаменяемость объектов с одинаковым интерфейсом. В C++ полиморфизм возможен только для объектов, относящихся к одной иерархии, и реализуется с помощью виртуальных методов. Те методы, которые могут быть переопределены в производных классах, должны быть объявлены в базовом классе с помощью спецификатора `virtual`. При переопределении метод должен иметь то же имя и набор параметров. Спецификатор `virtual` при переопределении метода в производных классах можно не указывать. Допускается изменять тип возвращаемого результата – вместо ссылки или указателя на базовый класс вернуть ссылку или указатель на производный класс. Переопределить виртуальный метод можно в сколь угодно дальнем потомке и сколько угодно раз. Если метод в каком-то классе этой иерархии не переопределяется, то используется определение из базового класса этого класса.

Статический метод нельзя объявить виртуальным.

При вызове виртуального метода вызывается его реализация из реального класса объекта, хотя объект может быть представлен ссылкой или указателем на базовый класс. Если необходимо вызвать реализацию виртуального метода из базового класса, то перед именем метода нужно написать имя базового класса и `::`.

Если определить виртуальный метод в базовом классе нельзя, то после заголовка вместо тела метода нужно указать =0. Такой метод называется *чисто виртуальным*, а класс, содержащий хотя бы один такой метод, – *абстрактным*. Нельзя создавать объекты абстрактных классов, но можно ра-

ботать со ссылками или указателями на абстрактные классы. В производном классе все чисто виртуальные методы должны быть определены, чтобы класс перестал быть абстрактным.

```
class Shape {
public:
  virtual double square()=0; // чисто виртуальный метод
};
class Rectangle: public Shape
{ double w,h;
public:
  Rectangle(double w, double h):w(w),h(h){}
  double square() // переопределение виртуального метода
  { return w*h; }
};
class Circle: public Shape
{ double r;
public:
  Circle(double r):r(r){}
  double square() // переопределение виртуального метода
  { return r*r*acos(0.)*2; }
};
int main()
{ Shape *s=new Circle(10);
  cout<<(s->square())<<"\n"; // вызов виртуального метода
}
```

Чисто виртуальному методу в абстрактном классе можно дать определение, которое можно вызвать по имени класса.

```
class Figure {
  int color, x, y;
public:
  Figure(int color, int x, int y):
    color(color),x(x),y(y) {}
  virtual void draw()=0; // чисто виртуальный метод
};
void Figure::draw() // определение чисто виртуального метода
{ setcolor(color);
  moveto(x,y);
}
class Rectangle: public Figure
{ int w,h;
public:
```

```
  Rectangle(int color, int x, int y, int w, int h)
    :Figure(color,x,y),w(w),h(h){}
  void draw(); // переопределение виртуального метода
};
void Rectangle::draw()
{ Figure::draw(); // вызов реализации метода из базового класса
  linerel(w,0);
  linerel(0,h);
  linerel(-w,0);
  linerel(0,-h);
}
int main()
{ Figure *s=new Rectangle(RED,10,20,100,50);
  ...
  s->draw(); // вызов виртуального метода
}
```

5.3 Множественное наследование

При множественном наследовании класс наследует структуру и поведение от нескольких классов.

При множественном наследовании могут появиться следующие проблемы:

1.Наследование от общего предка. В этом случае используют виртуальные классы.

```
class A {};
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public C {};
```

При использовании виртуальных классов конструкторы выполняются в специальном порядке: сначала конструкторы всех виртуальных классов, а потом конструкторы базовых классов и полей в обычном порядке и тело конструктора. Для деструкторов порядок строго противоположный.

2. Методы с одинаковой сигнатурой в разных базовых классах. В этом случае используют явное указание имени класса перед именем метода при вызове.

```
class A {
public:
  void f();
};
class B {
public:
  void f();
```

```
};
class C: public A, public B {};
int main()
{ C c1;
  c1.A::f(); // метод из класса A
  c1.B::f(); // метод из класса B
}
```

3. Переопределение виртуальных методов с одинаковой сигнатурой в разных базовых классах. В этом случае вводят промежуточные классы.

```
class A {
public:
  virtual void f();
};
class B {
public:
  virtual void f();
};
class A1: public A {
public:
  virtual void fA() { A::f(); }
  void f() { fA(); }
};
class B1: public B {
public:
  virtual void fB() { B::f(); }
  void f() { fB(); }
};
class C: public A1, public B1
{
  void fA();
  void fB();
};
```

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:

- наследование;
- агрегация;
- использование;
- базовый класс;
- производный класс;
- полиморфизм;

– виртуальный метод;
– чисто виртуальный метод;
– абстрактный класс;
– множественное наследование;
– виртуальные классы.

2. Определите абстрактный класс Figure с методами для изменения видимости, координат и цвета линий фигуры и чисто виртуальными методами для рисования и получения координат прямоугольной области, которая содержит фигуру (для удаления фигуры с экрана рисуем черный прямоугольник по полученным координатам). Определите производный класс Page (прямоугольник с загнутым углом) с методом изменения размеров и класс производный от него FillPage, у которого есть метод изменения цвета внутренних областей фигуры. Напишите программу для тестирования реализованного класса.

6 ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Термины, знание которых необходимо для понимания главы:
– стек;
– аварийное завершение программы;
– деструктор;
– статическая память;
– заголовок функции.

6.1 Назначение. Стандартные исключения

В ходе выполнения функции может произойти *исключительная ситуация*, когда из-за обнаруженной ошибки продолжить нормальное выполнение алгоритма невозможно. Выполнение этого алгоритма должно завершиться, затем функция может:

– Выдать сообщение об ошибке и аварийно завершить выполнение программы. Недостатки: снижается надежность программы, введенная пользователем информация теряется, нет возможности предпринять альтернативные действия, например, сохранить информацию на другом диске.
– Вернуть специальное значение. Например, функция `malloc` при ошибке выделения памяти возвращает NULL. Недостатки: область значений функции может содержать все значения типа результата функции, проверка результата каждого вызова в сложном выражении невозможна, а в более простых приводит к существенному увеличению размера программы, поэтому часто проверка не выполняется.

– Установить состояние ошибки в специальной глобальной переменной. Например, математические функции записывают код ошибки в `errno`. Недостатки: проверка этой переменной после каждого оператора с вызовом функции приводит к существенному увеличению размера программы, поэтому часто проверка не выполняется.
– Вызвать специальную функцию обработки ошибки. Например, некоторые реализации математических функций при ошибке вызывают функцию `matherr`. Недостатки: при отсутствии исключений эта функция должна выполнить одно из трех рассмотренных ранее действий.

Механизм исключений позволяет передать управление и всю необходимую информацию в ту часть программы, которая сможет обработать данную ошибку, а затем продолжить нормальное выполнение программы. Размер программы при этом не увеличивается.

Все стандартные исключения являются производными от класса `exception`. К стандартным исключениям относятся:

– `bad_alloc`, генерируемое операцией `new`;
– `bad_cast`, генерируемое операцией `dynamic_cast`;
– `bad_typeid`, генерируемое операцией `typeid`;
– `bad_exception`, генерируемое для обработки исключения, непредусмотренном в заголовке функции;
– исключения, порождаемые классами и алгоритмами STL.

6.2 Порождение и перехват

Для порождения исключений используется оператор

`throw` **выражение;**

Рекомендуется указывать в качестве аргумента объекты специальных классов, а не стандартных типов (`int`) или классов общего назначения (`vector`). Классы исключений рекомендуется выстраивать в иерархии.

Операторы, которые могут генерировать исключения, записываются внутри блока `try`, после которого идет набор обработчиков исключений `catch`. У каждого обработчика указывается тип исключений, который он может перехватывать, и опционально имя, с помощью которого можно обращаться к созданному оператором `throw` объекту внутри обработчика.

Перехват исключения рекомендуется делать по ссылке, чтобы избежать *срезки* объекта при преобразовании к базовому классу.

```
try { // операторы
}
catch(bad_alloc &)
{ // обработка ошибки выделения памяти
```

```
}
catch(io_error & e)
{ e.stream.clear(); // обработка ошибки ввода-вывода
}
catch(exception &)
{ // обработка других стандартных исключений
}
```

Порядок обработки исключений следующий:

1. Создается копия аргумента `throw` в статической памяти.

2. Происходит поиск подходящего обработчика. В процессе поиска выполняется *раскрутка* стека, т.е. выход из функций, где нет подходящих обработчиков и вызов деструкторов для локальных объектов.

3. Если подходящий обработчик найден, управление передается ему, иначе вызывается функция `terminate`, которая по умолчанию аварийно завершает программу, но можно задать другое действие с помощью функции `set_terminate`

4. После выполнения обработчика копия аргумента `throw` в статической памяти уничтожается и управление передается на оператор, следующий за последним из обработчиков того набора, где был найден обработчик.

Если обработчик не смог полностью обработать исключение, то можно повторно сгенерировать исключение внутри обработчика с помощью оператора

```
throw;
```

Проверка на соответствие типа объекта и обработчика выполняется в порядке перечисления обработчиков, поэтому обработчики для исключений производных классов должны быть указаны до обработчиков базовых классов. Самым последним обработчиком в наборе может быть указан `catch(...)`, который перехватывает все исключения. Такой обработчик можно использовать для корректного освобождения ресурсов:

```
int g(int n)
{ int *a=new int[n];
  try
  { //...
  }
  catch(...)
  { delete[] a;
    throw;
  }
  delete[] a;
}
```

Более простым способом добиться такого же эффекта можно, используя умные указатели, автоматически освобождающие ресурсы при уничтожении.

```
{ smart_ptr<int> a=new int[n];
  ...
  // освобождение памяти при уничтожении указателя
}
```

6.3 Спецификация исключений в заголовке функции

Возврат исключения – это альтернативный вариант возврата результата из функции, область значения функции при этом расширяется новым типом. Тип результата указывается в заголовке функции, аналогично должна быть возможность указать тип альтернативного результата. Для этого в заголовке функции после списка параметров указывается резервированное слово `throw` со списком типов исключений в скобках. По умолчанию функция может вернуть исключение любого типа.

```
void f(); // исключение любого типа
void f()throw(); // никаких исключений
void f()throw(bad_alloc,io_error); // только исключения
                  // bad_alloc, io_error и производных от них классов
```

Если функция попытается вернуть непредусмотренное исключение, то вызывается функция `unexpected`, которая по умолчанию пытается создать исключение `bad_exception`. Можно задать другое действие с помощью функции `set_unexpected`. Если `bad_exception` нет в списке допустимых исключений, вызывается функция `terminate`.

Рекомендуется, чтобы деструктор не возвращал исключений, так как деструкторы вызываются при раскрутке стека и возникновение исключений в этот момент приведет к вызову `terminate` и аварийному завершению программы.

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:

- исключительная ситуация (исключение);
- порождение исключения;
- перехват исключения;
- обработчик;
- повторное порождение;
- раскрутка стека;
- возврат исключения;
- спецификация исключений в заголовке функции.

2. Определите класс «Массив целых чисел», размер которого задается в конструкторе. Класс должен обеспечивать операции индексации, копирования и присваивания. При неверном индексе или несоответствии размеров массивов при присваивании операции должны возвращать исключительные ситуации.

7 STL

Термины, знание которых необходимо для понимания главы:

- шаблон функции;
- шаблон класса;
- специализация шаблона;
- указатель;
- последовательность;
- деревья бинарного поиска (BST);
- поразрядные операции;
- форматированный ввод-вывод.

Standard **T**emplate **L**ibrary – это набор классов и алгоритмов, входящих в стандарт языка C++. Готовые структуры данных и алгоритмы из библиотеки позволяют ускорить написание программ.

7.1 Вспомогательные компоненты

В заголовочном файле <utility> определены шаблоны функций для `operator!=` из `operator==` и `operator>, <=, >=` из `operator<` (для использования шаблонов написать `using namespace std::rel_ops;`).

Также <utility> включает шаблон класса для разнородных пар значений `pair<T1,T2>`. Для этого типа определены операции == и <. Для создания пары используется функция `make_pair(5, 3.1415926)`.

Функциональные объекты – это объекты, для которых определён `operator()`. Они важны для эффективного использования библиотеки. В местах, где ожидается передача указателя на функцию алгоритмическому шаблону, интерфейс установлен на приём объекта с определённым `operator()`. Это не только заставляет алгоритмические шаблоны работать с указателями на функции, но также позволяет им работать с произвольными функциональными объектами. Использование функциональных объектов вместе с шаблонами функций увеличивает выразительную мощность библиотеки также, как делает результирующий код более эффективным. Например, если мы хотим поэлементно сложить два вектора `a` и `b`, содержащие `double`, и поместить результат в `a`, мы можем сделать это так:

```
transform(a.begin(), a.end(), b.begin(),
    a.begin(), plus<double>());
```

В заголовочном файле <functional> определены функционалы plus, minus, times, divides, modulus, negate, equal_to, not_equal_to, less, greater, less_equal, greater_equal, logical_or, logical_and, logical_not. Пользователь может производить свои функционалы, наследуя от `unary_function<TArg, TRez>` и `binary_function<TArg1, TArg2, Trez>` и определяя `operator()`. Можно превращать бинарные функционалы в унарные с помощью связывателей `bind1st` и `bind2nd`, заменяющих соответствующий аргумент на константу: `bind2nd(less<int>(),40)` (значение меньше 40). В C++11 вместо функционалов можно использовать более простой механизм: лямбда-выражения и анонимные функции.

7.2 Итераторы

У всех контейнерных классов определены следующие итераторы (аналог указателя), позволяющие получить доступ к отдельным элементам: **класс**`::iterator`, **класс**`::const_iterator`, **класс**`::reverse_iterator`, **класс**`::const_reverse_iterator` (const если не хотим изменять значения, `reverse` если от конца к началу)

Начальный итератор контейнера: `v.begin()` `v.rbegin()`

Конечный итератор: `v.end()` `v.rend()`

К итераторам применимы операции ++ и --, к некоторым $+*n*, $-*n*

```
vector<double> v(100);
for(vector<double>::iterator iv=v.begin();
      iv!=v.end();++iv)
  *iv=0.5;
```

Для сравнения:

```
double v[100];
for(double *iv=v;iv!=v+100;++iv)
  *iv=0.5;
```

7.3 Алгоритмы

Все шаблонные алгоритмы работают не только со структурами данных в библиотеке STL, но также и с встроенными структурами данных C++. Например, все алгоритмы работают с обычными указателями.

Обработка скалярных значений

`max(a,b)`	максимальное значение
`min(a,b)`	минимальное значение
`swap(a,b)`	обмен

Обработка последовательностей

`max_element` (first,last)	значение максимального элемента набора
`min_element` (first,last)	значение минимального элемента набора
`for_each` (first,last,fun)	применить к каждому элементу последовательности функциональный объект (функцию) fun
`find` (first,last,value)	поиск первого элемента, равного value
`find_if` (first,last,fun)	поиск первого элемента, для которого fun(x)==true
`count` (first,last,value)	подсчет количества элементов, равных value
`count_if` (first,last,fun)	подсчет количества элементов, для которых fun(x)==true
`search` (first1,last1, first2,last2)	поиск вхождения подпоследовательности, заданной итераторами (first2,last2)
`search_n` (first1,last1, n,value)	поиск подпоследовательности из n значений value
`copy` (first,last,where)	копирование последовательности в where
`transform` (first,last, where,fun)	преобразование элементов последовательности с помощью функционального объекта (функции) fun

`transform` (first1,last1, first2,where,fun)	преобразование элементов двух последовательностей, например, суммирование двух векторов a и b:
	`vector<int> a(n),b(n),c(n);`
	`transform(a.begin(),a.end(),`
	`b.begin(),b.end(),c.begin(),`
	`plus<int>())`

`fill` (first,last,value)	заполнение значением value
`fill_n` (where,n,value)	записать n значений value
`generate` (first,last,fun)	заполнение результатом функционального объекта (функции) fun()
`generate_n` (where,n,fun)	записать n результатов функционального объекта (функции) fun()
`random_shuffle` (first, last)	перемешивание
`remove` (first,last,value)	удаление элементов, равных value, возвращается итератор на конец новой последователь-

	ности, например, удаление всех 0: `a.erase(remove(a.begin(), a.end(),0),a.end())`
`remove_if` (first,last,fun)	удаление элементов, для которых fun(x)==true
`remove_copy` (first,last, where,value)	копирование элементов, не равных value
`remove_copy_if` (first, last,where,fun)	копирование элементов, для которых fun(x)==false
`replace` (first,last, oldvalue,newvalue)	замена элементов, равных oldvalue на newvalue
`replace_if` (first,last, fun,newvalue)	замена элементов, для которых fun(x)==true на newvalue
`replace_copy` (first,last, where,oldvalue,newvalue)	копирование с заменой
`replace_copy_if` (first, last,where,fun,newvalue)	копирование с заменой
`reverse` (first,last)	изменение порядка элементов на обратный
`reverse_copy` (first,last, where)	копирование в обратном порядке
`rotate` (first,middle,last)	циклический сдвиг, элемент *middle становится первым
`rotate_copy` (first, middle,last,where)	копирование с циклическим сдвигом
`partition` (first,last,fun)	перемещение элементов, для которых fun(x)==true, в начало последовательности
`stable_partition` (first, last,fun)	перемещение элементов, для которых fun(x)==true, в начало последовательности с сохранением порядка в исходной последовательности
`next_permutation` (first, last)	следующая перестановка, возвращается false, если следующей перестановки не существует
`prev_permutation` (first, last)	предыдущая перестановка
`accumulate` (first,last, init,fun=plus())	подсчет суммы элементов, можно вместо сложения указать другую бинарную функцию

Сортировка и обработка упорядоченных последовательностей, множеств (упорядоченных последовательностей без повторений)

`sort(first,last)`	сортировка последовательности
`stable_sort(first,last)`	сортировка с сохранением порядка элементов с равным ключом
`unique(first,last)`	удалить повторения элементов, возвращается итератор на конец новой последовательности
`unique_copy(first,last, where)`	копирование без повторений
`merge(first1,last1, first2,last2,where)`	слияние
`binary_search(first,last, value)`	проверка наличия значения value
`lower_bound(first,last,value)`	поиск первой позиции для вставки значения
`upper_bound(first,last,value)`	поиск последней позиции для вставки значения
`includes(first1,last1, first2,last2)`	проверка, что множество (first2,last2) является подмножеством множества (first1,last1)
`set_intersection(first1, last1,first2,last2,where)`	пересечение множеств
`set_difference(first1, last1,first2,last2,where)`	разность множеств
`set_symmetric_difference( first1, last1,first2,last2,where)`	симметрическая разность множеств
`set_union(first1,last1, first2,last2,where)`	объединение множеств

Работа со структурой данных куча (heap)

`make_heap(first,last)`	создать кучу из элементов последовательности
`push_heap(first,last)`	добавить значение *(last-1) к куче (first,last-1), после выполнения куча станет (first,last)
`pop_heap(first,last)`	удалить наибольшее значение из кучи (first,last) и поместить его в *(last-1), после выполнения куча станет (first,last-1)
`sort_heap(first,last)`	превратить кучу в упорядоченную последовательность

Для аргумента where можно использовать либо итератор (указатель) на начало контейнера достаточного размера, либо `back_inserter(s)`, который возвращает добавляющий итератор для контейнера s.

7.4 Класс vector

`vector` – вид последовательности, которая поддерживает итераторы произвольного доступа, он поддерживает операции вставки и удаления в конце с постоянным временем; вставка и удаление в середине занимают линейное время.

Создание:

```
vector<int> a; // 0 элементов
vector<double> b(10); //10 элементов
vector<double> c(10,0.0);//10 элементов, начальное значение 0
```

Доступ к элементам:

```
b[0]=c[4]; // без проверки
b.at(0)=c.at(4); // с проверкой выхода за границы
c.back()=b.front(); // последний и первый элементы
```

Текущий размер: `c.size()`

При необходимости можно изменять размер:

```
a.resize(100);
b.resize(200,2.0); // всем новым элементам присвоить 2.0
a.clear(); // опять 0 элементов
```

Добавлять новое значение в конец: `c.push_back(1.5);`

Удалять последнее значение: `c.pop_back();`

Вставка в середину:

```
c.insert(c.begin()+pos,1.5);
c.insert(c.begin()+pos,5,1.5); // 5 значений
c.insert(c.begin()+pos,b.begin(),b.end());// из вектора b
```

Удаление из середины:

```
c.erase(c.begin()+pos);
c.erase(c.begin()+pos,c.end());// все элементы с pos до конца
```

Для повышения эффективности операций изменения размера можно зарезервировать память: `c.reserve(100);`

Работают операции сравнения (в лексикографическом порядке) и присваивание.

Замечание: `vector<bool>` хранит биты упаковано, поэтому нельзя получить адрес элемента такого вектора.

7.5 Класс string

Создание:

```
string a;
```

```
string b("строка");
```
Длина строки: `b.length()` или `b.size()`

Сцепление: `a="текст"; a=a+b; a+=b; a+='!';`

Обращение к символам: `a[i]=b[j]; a.at(i)='A';`

Подстрока:

`a.substr(pos,n)` // длиной n символов, начиная с pos,

`a.substr(pos)` //до конца строки

Строка Си: `a.c_str()`

Вставка: `a.insert(pos,b);`

Удаление:

`a.erase(pos,n)` // n символов

`a.erase(pos)` // до конца строки

`a.clear()` // всей строки

Замена: `a.replace(pos,n,b);`

Поиск:

```
size_type i=a.find(b); // первого вхождения подстроки
i=a.find('A'); // или символа
i=a.rfind(b); // последнего вхождения
i=a.rfind('A');
```

Вторым аргументом функции можно указать начальную позицию поиска.

Ввод строки:

```
getline(cin,s);
getline(cin,s,'\n');
```

7.6 Ассоциативные контейнеры

`set` – это ассоциативный контейнер, который поддерживает уникальные ключи (не содержит ключи с одинаковыми значениями) и обеспечивает быстрый поиск ключей

```
set<int> s;
s.insert(5); // добавить значение
s.erase(5); // удалить значение
s.erase(s.begin()); // удалить первый
if(s.find(5)!=s.end()) // значение найдено
```

`multiset` допускает множественные копии того же самого значения ключа, для нахождения границ можно использовать `s.upper_bound(5)` и `s.lower_bound(5)`

`map` – ассоциативный контейнер, который поддерживает уникальные ключи и обеспечивает быстрый поиск значений другого типа `T`, связанных с ключами. Итератор в случае map является «указателем» на `pair<const` **ключ,значение**>

```
map<int,string> m;
m[5]="текст"; // доступ по ключу
m.insert(make_pair(5,string("текст")); // добавить значение
m.erase(5); // удалить значение
m.erase(m.begin()); // удалить первый
if(m.find(5)!=m.end()) // значение найдено
```

`multimap` допускает множественные копии того же самого значения ключа, для нахождения границ можно использовать `m.upper_bound(5)` и `m.lower_bound(5)`

7.7 Прочие контейнерные классы

Наиболее часто используемым классом является вектор бит:

```
bitset<100> b; // вектор из 100 битов
```

Допустимы все поразрядные операции `& | << >> ^ ~`

Быстрый подсчет единичных битов: `b.count()`

Проверка битов:

```
b.any() // хотя бы один бит равен 1
b.none() // все биты равны 0
b.test(i) // состояние i-го бита
```

Установка битов:

```
b.set() // все биты в 1
b.set(i) // i-й бит в 1
b.reset() // все биты в 0
b.reset(i) // i-й бит в 0,
b.set(i,val) // i-й бит в val
```

Операция индексации `b[i]` позволяет проверить и установить i-й бит.

В STL есть реализация всех базовых структур данных:

```
queue<double> p; // очередь
deque<int> d; // двухстороняя очередь
list<int> l; // список
stack<double> s; // стек
priority_queue<double> p; // очередь с приоритетами
valarray<double> v(100); // числовой массив,
        // определены операции + - * / и т. д.
```

Дополнительную информацию по классам и алгоритмам STL можно найти в книгах [4, 9].

7.8 Поточные классы и форматированный вывод

Поточные классы `istream` и `ostream` получаются инстанцированием классов-шаблонов для типа `char`:

```
typedef basic_ostream<char> ostream; // обычные символы
```

```
typedef basic_ostream<wchar_t> wostream;
                    // широкие символы (например, Unicode)
```

Для ввода из файла или вывода в файл используются поточные классы `ifstream`, `ofstream` и `fstream`. При создании потока в первом аргументе указывается строка с именем файла, для `fstream` нужно указать и второй аргумент – режим доступа. Деструктор автоматически закрывает файл.

Для неформатированного ввода и вывода удобно использовать объекты `cin` и `cout`, но при необходимости управлять форматированием проще воспользоваться `printf` и `scanf`.

Управление форматированием с помощью методов поточных классов

Управление форматированием

`setf(flag)`	установка негрупповых флагов форматирования
`setf(flag,group)`	установка флага из группы
`unsetf(flag)`	сброс негрупповых флагов
`fmtflags flags()`	текущие флаги
`width(w)`	установка минимальной ширины поля, после вывода значения ширина поля сбрасывается на 0
`int width()`	текущая ширина поля
`precision(n)`	установка максимального количества десятичных знаков, для вывода вещественных чисел по умолчанию – 6
`int precision()`	текущая точность
`fill(ch)`	установка символа заполнения, по умолчанию – пробел
`int fill()`	текущий символ заполнения

Проверка и установка состояния

`bool good()`	true, если ошибок не было
`bool eof()`	true, если при выполнении последнего ввода был обнаружен конец файла
`bool fail()`	true, если при выполнении последнего ввода была ошибка ввода-вывода
`clear()`	очистить состояние ошибки для продолжения ввода
`clear(ios::failbit)`	установить состояние ошибки

Посимвольный ввод-вывод

`put(ch)`	вывод символа

`write(buf,n)`	вывод n символов
`flush()`	записать информацию из буфера на диск
`get(ch)` `int get()`	ввод символа
`read(buf, n)`	ввод n символов
`int peek()`	подсмотреть следующий символ, для конца файла возвращается EOF
`putback(ch)`	вернуть символ в поток

Позиционирование в файле

`pos_type tellg()`	получить текущую позицию чтения
`seekg(pos)` `seekg(offs,dir)`	установить позицию чтения
`pos_type tellp()`	получить текущую позицию записи
`seekp(pos)` `seekp(offs,dir)`	установить позицию записи

Открытие и закрытие файлов (классы ifstream, ofstream, fstream)

`open(name)` `open(name, mode)`	открыть файл, где mode комбинируется из флагов in, out, app, ate, trunc, binary
`close()`	закрыть файл

Группы и флаги форматирования определены в классе `ios`

Группа `basefield` – система счисления для целых чисел

`dec`	десятичная
`oct`	восьмеричная
`hex`	шестнадцатеричная

Группа `floatfield` – формат представления для вещественных чисел

`fixed`	с фиксированной точкой
`scientific`	в экспоненциальной форме

Группа `adjustfield` – выравнивание в поле вывода

`right`	вправо
`left`	влево
`internal`	знак числа – слева, значение – справа

Без группы

`showbase`	выводить `0x` перед шестнадцатеричными и `0` перед восьмеричными числами
`showpoint`	всегда выводить `.` и дробную часть

`uppercase`	выводить шестнадцатеричные цифры и E в вещественных числах в верхнем регистре
`showpos`	выводить знак числа для положительных чисел
`boolalpha`	выводить значения bool как true и false
`skipws`	игнорирование пробелов при вводе символов с помощью операции >>

```
// вывод целого числа в шестнадцатиричном виде с ведущими нулями
int x=123;
cout.setf(ios::hex,ios::basefield);
cout.setf(ios::uppercase);
cout.width(8);
cout.fill('0');
cout<<x<<"\n";
cout.fill(' ');
// вывод вещественного числа в поле шириной 10 с 2 дес. знаками
double f=12.3;
cout.setf(ios::fixed,ios::floatfield);
cout.setf(ios::showpoint);
cout.width(10);
cout.precision(2);
cout<<f<<"\n";
```

Вывод с помощью `printf` (*для сравнения*)

```
// вывод целого числа в шестнадцатиричном виде с ведущими нулями
int x=123;
printf("%08X\n",x);
// вывод вещественного числа в поле шириной 10 с 2 дес. знаками
double f=12.3;
printf("%10.2lf\n",f);
```

Управление форматированием с помощью манипуляторов из <iomanip>

`setw(w)`	установка ширины поля вывода
`setprecision(p)`	установка количества десятичных знаков для вывода вещественных чисел
`setfill(ch)`	установка символа заполнения
`flush`	записать информацию из буфера на диск
`endl`	вывести переход на новую строку, а затем выполнить `flush`
`dec oct hex`	выбор системы счисления для целых чисел
`right left internal`	выравнивание в поле вывода

`fixed scientific`	формат представления для вещественных чисел

`showbase noshowbase`
`showpoint`
`noshowpoint`
`showpos noshowpos`
	установка и сброс соответствующих флагов форматирования

...

Полезный совет

Использование `endl` существенно замедляет работу программы, так как для каждой строки выполняется принудительный вывод буфера в файл (это полезно только для ведения лога действий программы). Поэтому вместо `endl` лучше использовать `'\n'` или `"\n"`.

```
// вывод целого числа в шестнадцатиричном виде с ведущими нулями
int x=123;
cout<<hex<<uppercase<<setw(8)<<setfill('0')
    <<x<<setfill(' ')<<"\n";
// вывод вещественного числа в поле шириной 10 с 2 дес. знаками
double f=12.3;
cout<<fixed<<showpoint<<setw(10)<<setprecision(2)
   <<f<<"\n";
```

Управление форматированием с помощью класса format из библиотеки boost

```
// вывод целого числа в шестнадцатиричном виде с ведущими нулями
int x=123;
cout<<format("%08X\n")%x;
// вывод вещественного числа в поле шириной 10 с 2 дес. знаками
double f=12.3;
cout<<format("%10.2f\n")%f;
```

Управление форматированием с помощью собственных манипуляторов

```
// Пример манипулятора для вещественных чисел
struct fmt_f {
  int w,p;
  fmt_f(int w=0, int p=6):w(w),p(p){}
};
ostream& operator<<(ostream& s, fmt_f f)
{ s.width(f.w);
  s.fill(' ');
  s.setf(ios::fixed,ios::floatfield);
  s.precision(f.p);
  if(f.p>0)
    s.setf(ios::showpoint);
```

```
  else
    s.unsetf(ios::showpoint);
  return s;
}
// вывод вещественного числа
double f=12.3;
cout<<fmt_f(10,2)<<f<<"\n";// printf("%10.2f\n",f);
cout<<fmt_f(10,0)<<f<<"\n";// printf("%10.0f\n",f);
cout<<fmt_f(10)<<f<<"\n";// printf("%10f\n",f);
cout<<fmt_f()<<f<<"\n";// printf("%f\n",f);
```

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:
   - функциональные объекты (функционалы);
   - связыватель;
   - контейнерный класс;
   - итератор;
   - ассоциативный контейнер;
   - манипуляторы.
2. Напишите программу, которая находит 10 самых часто встречающихся слов в заданном тесте. При реализации используйте ассоциативный массив map и алгоритм сортировки.

8 СТАНДАРТЫ C++11/14/17

Термины, знание которых необходимо для понимания главы:
- ссылки;
- контейнерный класс (контейнер);
- шаблон функции;
- указатель.

8.1 История развития C++

Язык C++ возник в начале 1980-х годов, когда сотрудник фирмы Bell Laboratories Бьёрн Страуструп придумал ряд усовершенствований к языку C для собственных нужд. До начала официальной стандартизации язык развивался в основном силами Страуструпа в ответ на запросы программистского сообщества. В 1998 году был принят международный стандарт языка C++, а в 2003 году в стандарте были сделаны исправления неточностей и внесены небольшие изменения. Например, была введена разница между формами операции `new T` и `new T()` (см. 1.3).

В 2005 году к STL были добавлены новые классы и шаблоны (регулярные выражения, хэш-таблицы, кортежи, массивы фиксированного разме-

ра) и функции для генерации случайных чисел. Эти расширения должны были войти в предыдущий стандарт языка, но разработчики решили сделать провести дополнительное обсуждение и согласование перед стандартизацией. Большая часть нововведений была взята из библиотеки boost.

В 2011 году в стандарт языка C++ были внесены изменения, упрощающие написание программ, добавлены параллелизм и лямбда-выражения, которые стали необходимостью в современных языках программирования.

В 2011 году был принят также новый стандарт языка C, в котором также добавлен параллелизм. Стоит отметить, что некоторые кардинальные изменения в языке C, которые появились еще в стандарте 1999 года, не нашли отражения в новом стандарте C++ (например, массивы переменного размера, указание имени поля или номера элемента массива при инициализации, комплексный тип). Напротив, в стандарт C99 добавлены возможности из C++ или их аналоги, позволяющие минимизировать изменения при копировании кода (например, объявление переменных в любом месте программы, макросы, генерирующие вызов нужной функции в зависимости от типа аргумента).

В 2014 году был разработан очередной стандарт C++, который немного увеличил удобство языка (см. 8.2 константы), добавил некоторые обобщения (см. 8.2 лямбда-выражения) к революционным изменениям, которые были сделаны в стандарте 2011 года.

В стандарт 2017 было решено добавить параллельные версии алгоритмов, работу с файловой системой, дополнительные атрибуты для управления предупреждениями при компиляции. К сожалению, в новый стандарт не попало предложение Б.Страуструпа и Г.Саттера об унификации вызовов методов и функций: можно писать как x.size(), так и size(x).

8.2 Изменения в языке

Вывод типа объекта компилятором при объявлении

```
auto x=a+b; // тип x определяется типом выражения a+b
decltype(c) y; // тип y совпадает с типом объекта c
map<string,int> m;
auto x=m.find("name"); // auto вместо map<string,int>::iterator
if(x!=m.end()) // элемент найден
  x->second=1;
```

В C++14 допускается указывать `auto` для типа возвращаемого значения функции, если определение функции дано перед её вызовом в компилируемом модуле, тогда тип определяется по результату, возвращаемому `return`.

Новый вид ссылки &&

Ссылка && используется для ссылок на временные значения, получающиеся в результате вычислений. Разницу между этими видами ссылок можно использовать для оптимизации вычислений.

```
class A {
  int size;
  double *data;
public:
  A(int);
  A(const A&); // конструктор копий
  A(A&&); // перемещающий конструктор
  A& operator=(const A&); // операция присваивания
  A& operator=(A&&); // перемещающая операция присваивания
  ~A();
};
A::A(const A& a):size(a.size),data(new double[size])
{ std::copy(a.data,a.data+size,data);
}
A::A(A&& a):size(a.size),data(a.data)
{ a.data=NULL;
}
A& A::operator=(const A& a)
{ A t(a);
  std::swap(size,t.size);
  std::swap(data,t.data);
  return *this;
}
A& A::operator=(A&& a)
{ std::swap(size,a.size);
  std::swap(data,a.data);
  return *this;
}
A::~A() { delete []data; }
...
A f() { A a(100); ... return a; }
A x;
x=f();
```

При отсутствии перемещающих конструктора и присваивания массив data локально созданного в функции f объекта будет дважды скопирован из одной области памяти в другую (a → временный объект → x), а при их наличии – это не потребуется делать ни разу.

Инициализация

Контейнерные классы теперь можно инициализировать, используя списки значений, как и обычные массивы. Для этого в C++ был добавлен шаблонный класс `std::initializer_list<T>`. Объекты этого класса задаются в программе с помощью перечисления значений в `{}`, а класс должен иметь конструктор вида:

```
class IntList {
public:
  IntList(std::initializer_list<int> a);
};
IntList list={1,2,3,4,5};
```

Все стандартные контейнерные классы теперь имеют такой конструктор:

```
vector<int> v={1,2,3};
map<string,int> m={{"Tom",5},{"John",10}};
```

Списки значений можно также использовать в качестве аргумента для функций:

```
void f(std::initializer_list<int> a);
...
f({3,5,7});
```

При создании объектов аргументы конструктора можно указывать в фигурных скобках `{}`. Если у класса есть конструктор с параметром `initializer_list`, то при использовании `{}` будет вызван именно он, в остальных случаях разницы между этими вариантами нет. Также форму с `{}` можно использовать для инициализации полей структур без конструкторов. Форма с `{}` может использоваться неявно при возврате результата из функции.

```
struct Vector { int x,y; };
Vector a{1,2};
Vector turn(Vector b)
{ return {-b.y,b.x};
}
a=turn({2,3});
```

Нестатическим полям можно указать значение по умолчанию, которое будет использовано, если в конструкторе это поле не будет инициализировано другим значением (ранее при объявлении можно было указывать значение только у `static const` полей).

```
class Rational {
  int p(0),q(1);
  Rational(){} // p=0, q=1
  Rational(int z):p(z){} // p=z, q=1
```

```
  Rational(int a, int b):p(a),q(b)
  { int c=gcd(a,b);
    p/=c; q/=c;
  }
};
```

Цикл по контейнеру (массиву)

```
vector<int> v;
for(auto &x: v) x*=2; // удвоить элементы вектора v
int m[4];
for(int &e: m) cin>>e; // ввод массива m
for(auto a:{2,3,5,7})
  cout<<a<<"\n"; // вывести числа из списка
```

for(*объявление переменной* : *контейнер*) *оператор;*

превращается в

```
{ auto &&r=контейнер;
  auto b=begin(r);
  auto e=end(r)
  for(; b!=e; ++b)
  { объявление переменной=*b;
    оператор;
  }
}
```

где begin(r) – перегруженная функция, которая возвращает начальный адрес массива или результат вызова метода r.begin() для контейнерных классов, а end(r) – конечный адрес массива или результат вызова метода r.end() для контейнерных классов.

Лямбда-выражения и анонимные функции

Лямбда-выражения и анонимные функции можно использовать везде, где требуется передавать в качестве аргумента функцию или функциональный объект.

Лямбда-выражения создаются следующим образом:

[*замыкание*](*параметры*)**{ return** *выражение*; **}**

Тип лямбда-выражения определяется по типу возвращаемого значения. В C++14 можно указать `auto` для параметров лямбда-выражения и компилятор сам выведет их тип.

Анонимные функции создаются следующим образом:

[*замыкание*](*параметры*) –> *тип_результата* { *операторы* }

```
vector<int> v;
// подсчитать количество положительных элементов вектора v
```

```
int n=count_if(v.begin(),v.end(),
          [](int x){ return x>0; });
int a[20];
// удвоить элементы a и поместить в v (C++14)
transform(begin(a),end(a),begin(v),
     [](auto x){ return x*2; });
```

В замыкании нужно указать список имен объектов, которые используются для вычислений, но не передаются как параметры. Перед именем объекта можно указать &, если объект нужно передавать по ссылке, иначе в замыкании будет копия значения объекта в момент определения лямбда-выражения. Можно не перечислять самостоятельно объекты поименно, а указать `[=]`, если все объекты должны храниться по значению, или `[&]` для передачи по ссылке. Важно: лямбда-выражение, использующее в замыкании ссылки на локальные объекты блока, становится ошибочным после завершения блока, так как объекты будут уничтожены. При использовании в замыкании копий значений, лямбда-выражение останется корректным. В C++14 можно поместить в замыкание любое значение и дать ему имя.

```
int i=0;
// заполнить вектор квадратами натуральных чисел
generate(v.begin(),v.end(),
    [&i]()->int { ++i; return i*i; });
// C++14
auto one = [value=1](){return value;};
cout<<one(); // будет выведено 1
```

Альтернативный синтаксис можно использовать при определении обычных функций. Такой синтаксис особенно полезен для шаблонов функций, когда тип результата определить затруднительно.

```
auto f(int x) -> int;
template <typename T1, typename T2>
auto add(const T1 &a, const T2 &b) -> decltype(a+b)
{ return a+b;
}
```

Константы и проверки во время компиляции

Спецификатор `const` можно использовать не только для определения констант, но и для указания, что значение объекта не должно меняться. Теперь для всех значений, вычисляемых во время компиляции, можно использовать новый спецификатор `constexpr`. С его помощью можно даже определять функции для обработки констант, которые будут вычислены во время компиляции.

Появилась специальная константа nullptr, которая которая неявно может преобразована в указатель любого типа, но не в число. Ранее для определения нулевого указателя NULL использовалась константа 0, что могло при связывании привести к неверному выбору функции.

Для сложных строк таких как регулярные выражения, включающих много символов \, можно использовать новый вид строковой константы R"(...)":

```
// Регулярное выражение для поиска \/
std::regex re(R"(\\\/)"); // вместо "\\\\\/"
```

Программисты теперь могут определять свои виды констант следующим образом:

```
long operator "" b(const char *s) // двоичное число
{ long r=0;
  for(;*s;++s) r=(r<<1)+(*s!='0');
  return r;
}
constexpr complex<double> operator "" i
          (long double d) // мнимая константа
{ return {0,d};
}
constexpr double operator "" _deg
        (long double d) // градусы
{ return d*PI/180.0;
}
std::string operator "" s(const char* p,
    size_t n) // строковая константа
{ return string(p,n);
}
void f(string);
void f(char *);

int x=1011b;
complex<double> c=2+3i;
f("Text"s); // вызов f(string);
double r=sin(30_deg); // найти синус от 30 градусов
```

В C++14 для записи двоичных чисел используется префикс 0b (0b1010), разряды в длинных числах можно разделять апострофами (12'345'671). Также уже определены некоторые из указанных выше суффиксов (""s и 2i) и суффиксы для промежутков времени (2h, 15min, 9s, 11ms, 50us, 2ns).

С помощью enum class можно перечислить константы, которые являются типобезопасными и требуют явного преобразования в int и обратно. Также для перечислений можно указать базовый целый тип (по умолчанию int).

```
void f(int);
enum Switch: char {on, off};
enum class Color { green, red, blue, white };
Switch s1=off;
Color c1=Color::red;
f(s1); // OK
f(c1); // Ошибка
f(int(c1)); // OK
```

Не все условия применимости программы можно проверить с помощью препроцессора (директивы #if...#endif и #error), поэтому была добавлена возможность выполнять необходимые проверки компилятору:

```
static_assert(sizeof(int)!=sizeof(void *),
  "Cannot convert pointer to int.");
```

Классы и шаблоны

Компилятор производит генерацию кода по шаблону, когда происходит явное или неявное инстанцирование. Это может существенно увеличить время компиляции и компоновки, особенно в тех случаях, когда шаблон инстанцируется с одинаковыми параметрами во многих модулях. С помощью extern теперь можно указать компилятору не инстанцировать шаблон в данной единице трансляции.

```
extern template vector<int>;
```

Естественно, в одном из модулей должно быть сделано явное или неявное инстанцирование, чтобы при компоновке программы все ссылки были разрешены.

```
template vector<int>; // явное инстанцирование
```

Новый стандарт позволяет вызывать одни конструкторы класса из других в списках инициализации, чтобы не дублировать код. Также для упрощения списка инициализации для нестатических полей можно указать значение по умолчанию (см. выше).

explicit можно указать не только у конструкторов-преобразователей, но и у операций преобразования.

При перегрузке виртуальных методов теперь можно указать спецификатор override. Это позволит компилятору найти ошибку в случае неверной сигнатуры виртуального метода в производном классе.

Новый спецификатор `final` используется, чтобы запретить дальнейшую перегрузку виртуального метода или наследование от класса.

```
struct A {
  virtual void f(int);
  virtual void g(double);
};
struct B: A {
  void f(int) override final; // OK
  void g(int) override; // Ошибка
};
struct C final: B {
  void f(int) override; // Ошибка
};
struct D final: A {...};
struct E: D { ... }; // Ошибка
```

Теперь можно явно запретить автоматическое создание конструкторов и операций для класса или, наоборот, указать, что используется определение по умолчанию. Ранее для запрета приходилось помещать заголовок метода в private и не определять его (что иногда приводило к более сложной для локализации ошибке компоновки), а определение по умолчанию при смене видимости реализовать явно.

```
class A {
  ...
public:
  A(const A&)=delete; // запрет копирования
private:
  A& operator=(const A&)=default;
  // использовать определение присваивания по умолчанию,
  // но только в методах класса
};
```

8.3 Изменения STL

Добавлены новые классы и алгоритмы.

Кортежи

Класс `tuple` (кортеж), аналог `pair` для неограниченного числа значений. Для доступа к элементам кортежа используется функция `std::get<i>` (нумерация с 0). Для извлечения всех значений из кортежа используется функция `tie` (для ненужных полей указывается `ignore`). Для создания кортежа используется функции `make_tuple` и `forward_as_tuple` (кортеж из ссылок на временные значения). Функция `tuple_cat` позволяет соединить несколько кортежей в один.

```
tuple<int,char,double> t1(10,'x',1.0),t2;
get<0>(t2)=get<0>(t1);
t2=make_tuple(1,'b',2.0);
int x;
double y;
tie(x,ignore,y)=t2;
print(forward_as_tuple(string("Name"),10));
...
void print(tuple<std::string&&,int&&> arg)
{ cout<<get<0>(arg)<<" "<<get<1>(arg)<<"\n";
}
```

Хэш-таблицы

Классы `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap` позволяют создать ассоциативные массивы и множества с временем работы меньше чем $\mathcal{O}(\log(N))$ за счет использования большего количества памяти и хороших хэш-функций. Можно применять аналогичные методы `insert`, `find`, `erase`, но при обходе контейнера с помощью итератора порядок обработки элементов является неопределенным. Для нестандартных типов в аргументах шаблона указывается класс-функционал, метод `()` которого возвращает хэш для ключа.

Распараллеливание программ

Классы и функции, необходимые для распараллеливания программ, описаны в заголовочном файле `<thread>`. Кроме того, для распараллеливания вычислений, не требующих синхронизации и блокировок, можно использовать набор атомарных операций и типов, описанный в `<atomic>`

```
mutex m;
void input(string msg, int &n)
{ lock_guard<mutex> lm(m);
    // блокировка выполнения функции в других потоках
  cout<<msg;
  cin>>n;
}
void f(string t)
{ int n; // функция, которая будет выполняться параллельно
  input(t+" thread:",n);
  for(int i=0;i<n;++i)
    this_thread::sleep_for(
         chrono::milliseconds(100)); // задержка 100мс
  cout<<t<<" thread finished\n";
}
```

```
int main()
{ thread t1(f, "First"s);// запуск потоков управления
  thread t2(f, "Second"s);
  t1.join(); // ожидание завершения
  t2.join();
}
```

Для обработки данных большого объема в C++17 добавлены параллельные версии для многих стандартных алгоритмов. В качестве первого аргумента при вызове указывается режим выполнения par (возможна разделение данных на части и их параллельная обработка) или par_vec (возможна векторизация, т.е. одновременная обработка групп данных), по умолчанию применяется режим sequential (без распараллеливания).

Умные указатели

Вместо `auto_ptr` рекомендуется использовать умные указатели:

- `unique_ptr` – уникальный, при уничтожении указателя удаляется и объект, при присваивании и копировании указателей передается владение объектом, а старый указатель обнуляется.
- `shared_ptr` – с подсчетом ссылок на объект, когда количество указателей на этот объект (или какую-то его часть) становится равно нулю, объект удаляется.
- `weak_ptr` – вспомогательный, не учитывается при подсчете ссылок, но с помощью метода позволяет `expired` обнаружить, что объект был удален.

У всех типов указателей перегружены операции `->` и `*`(разыменования), а с помощью метода `get` можно получить обычный указатель.

```
struct data {int x; };
{ unique_ptr<data> p1(new data);
  p1->x=1;
  // автоматическое уничтожение объекта
}
{ shared_ptr<data> p1(new data),p2;
  weak_ptr<data> p3;
  p2=p1;
  p3=p1;
  p1=nullptr;
  p2=nullptr;
  // автоматическое уничтожение объекта
  if(!p3.expired()) p3->x=1;
}
```

Регулярные выражения

Для создания регулярных выражений используется класс `regex`. В первом аргументе конструктора указывается строка, во втором аргументе – тип регулярного выражения (POSIX, awk, grep, ECMA-262), режим оптимизации, чувствительность к регистру и т.п. Для обработки строк используются следующие функции:

- `regex_match` – проверка соответствия строки регулярному выражению, результат сопоставления может быть помещен в объект класса `smatch` или `cmatch` в зависимости от типа первого аргумента (`string` или `char *`), 0-й уровень соответствует всему выражению, *i*-й – подвыражению в *i*-х скобках;
- `regex_search` – поиск первой подстроки, соответствующей регулярному выражению, аналогично можно получить результат сопоставления; для обработки всех подстрок для `string` используются итераторы sregex_iterator и sregex_token_iterator (если необходима обработка подвыражений определенного уровня), а для `char *` – cregex_iterator и cregex_token_iterator, итератор указывает на результат сопоставления;
- `regex_replace` – замена подстрок, соответствующим регулярному выражению.

```
regex price("\\$([0-9]+)(\\.([0-9]+))?)");
cmatch m;
if(regex_match("$0.99", price))
  cout<<"OK\n";
if(regex_match("$3.25", m, price))
  cout<<m.str(2)<<" dollars "<<m.str(4)<<" cents\n";
    // 3 dollars 25 cents
if(regex_search("Total sum $100", m, price))
  cout<<"sum="<<m[1].str()<<"\n"; // sum=100
string s("Prices are from $75 to $80");
for(auto t=sregex_iterator(s.begin(),s.end(),price);
         t!=sregex_iterator();++t)
  cout<<t->str()<<"\n"; // $75 $80
for(auto
t=sregex_token_iterator(s.begin(),s.end(),price,1);
         t!=sregex_token_iterator();++t)
  cout<<t->str()<<"\n"; // 75 80
```

Генерация случайных чисел

Для генерации случайных чисел используются следующие классы:

- `minstd_rand0` (`default_random_engine`) реализует линейный конгруэнтный метод генерации 32-битных псевдослучайных чисел;
- `mt19937` и `mt19937_64` реализуют более качественный метод генерации 32- и 64-битных псевдослучайных чисел с помощью вихря Мерсенна;
- `ranlux24` и `ranlux48` реализуют метод Фибоначчи генерации 24- и 48-битных псевдослучайных чисел;
- `random_device` использует системный генератор случайных чисел.

Используя генераторы, можно получать числа с различными законами распределения: равномерное, Пуассона, Бернулли, нормальное и заданное образцом.

```
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> digit(0, 9);
for(int n=0; n<10; ++n)
  cout << digit(gen); // 10 случайных цифр
```

Новые алгоритмы

`iota` (first,last,value)	заполнение увеличивающимися на 1 значениями, начиная с value
`minmax_element` (first, last)	значения максимального и максимального элемента набора
`is_sorted` (first,last)	проверка, что последовательность упорядочена в неубывающем порядке
`move` (first,last,where)	перемещение последовательности в where
`all_of` (first,last,fun)	проверка, что все элементы последовательности удовлетворяют предикату fun
`any_of` (first,last,fun)	проверка, что существует элемент последовательности, удовлетворяющий предикату fun
`none_of` (first,last,fun)	проверка, что не существует элемента последовательности, удовлетворяющего предикату fun
`copy_if` (first,last, where,fun)	копирование элементов, для которых fun(x)==true

Работа с файловой системой

В C++17 добавлен новый класс для работы с путями к файлам `path`, у которого определены методы `filename()` (имя файла), `extension()` (расширение), `stem()` (имя без расширения), `parent_path()` (путь к

папке, содержащей файл). Также есть методы для замены имени и расширения файла, а операции / и /= позволяют добавить имя папки в путь, используя разделитель, заданный в ОС. Для получения и изменения текущего пути используется функция current_path.

Для обхода папок используются итераторы directory_iterator и recursive_directory_iterator, которые указывают на объект типа directory_entry с методами path() и status().

```
#include <iostream>
#include <filesystem>
namespace fs = std::filesystem;
int main()
{ path p=fs::current_path();
  std::cout << "Current path is " << p << '\n';
  for(auto& f: fs::directory_iterator(p))
     std::cout << f.path().filename() << '\n';
}
```

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:
   - вывод типа объекта;
   - ссылки на временные значения;
   - инициализация списком значений;
   - цикл по коллекции;
   - лямбда-выражение;
   - анонимная функция;
   - константы, вычисляемые во время компиляции;
   - нулевой указатель;
   - типобезопасные перечисляемые константы
   - спецификаторы для виртуальных методов;
   - запрет определения по умолчанию;
   - кортеж;
   - хэш-таблица;
   - распараллеливание;
   - умные указатели;
   - регулярные выражения;
   - генераторы случайных чисел.
2. Напишите функцию для создания аббревиатуры из первых букв слов в заданной строке с помощью regex.
3. Напишите программу, которая распараллеливает вычисление квадрата матрицы 1000x1000. Сравните время её работы с обычной версией.

ЧАСТЬ 2 ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

9 СЛОЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ООП

Термины, знание которых необходимо для понимания главы:

- предметная область;
- программное обеспечение;
- процесс разработки;
- дискретная система;
- компонент.

В этой главе цитируются основные идеи главы 1 книги [6].

9.1 Сложность программного обеспечения

Сложность вызывается четырьмя основными причинами:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Сложность реального мира. Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. Знакомство с первыми версиями системы позволяет пользователям лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время процесс разработки повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

Трудности управления процессом разработки. Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Поэтому такой объем работ потребует привлечения команды разработчиков. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга.

Гибкость программного обеспечения. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми

необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она заставляет разработчика создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Кроме того, часто приходится выполнять настройку программы под индивидуальные требования конкретного пользователя и системное окружение. Поэтому программные разработки остаются очень трудоемким делом.

Проблема описания поведения больших дискретных систем. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных. С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. Всеобъемлющее тестирование таких программ провести невозможно. При неблагоприятных условиях небольшое внешнее событие может привести к критической ошибке системы.

Чем сложнее система, тем легче ее полностью развалить.

9.2 Пять признаков сложной системы

1. Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня.

Многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже "увидеть" такие системы и их части. Скорее всего, мы можем понять лишь те системы, которые имеют иерархическую структуру. Архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов.

2. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя. Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого.

3. Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять "высокочастотные"

взаимодействия внутри компонентов от "низкочастотной" динамики взаимодействия между компонентами.

Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

4. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.

Иными словами, разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты, такие как клетки, или более крупные структуры, типа сосудистых систем, имеющиеся и у растений, и у животных.

5. Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная "с нуля", никогда не заработает. Следует начинать с работающей простой системы.

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы. Более того, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

9.3 Разработка сложной системы

Роль декомпозиции

Как отмечает Дейкстра, "Способ управления сложными системами был известен еще в древности – divide et impera (разделяй и властвуй)". При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае мы не превысим пропускной способности человеческого мозга: для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях (отнюдь не о всех). Именно сложность системы вынуждает делить пространство состояний системы.

Для разделения можно использовать либо алгоритмическую, либо объектно-ориентированную декомпозицию. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Однако мы не можем сконструировать сложную систему одновременно двумя способами, тем более, что эти способы по сути ортогональны. Мы должны начать разделение системы либо по алгоритмам, либо по объектам, а затем, используя полученную

структуру, попытаться рассмотреть проблему с другой точки зрения. В первом случае мы получаем сложные подпрограммы, обрабатывающие переменные, массивы, простые структуры данных, во втором случае – сложные структуры, включающие не только данные, но и небольшие подпрограммы для их обработки. Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложности программных систем.

Роль абстракции

В экспериментах Миллера было установлено, что обычно человек может одновременно воспринять лишь 7 ± 2 единицы информации. Это число, по-видимому, не зависит от содержания информации. Как замечает сам Миллер: "Размер нашей памяти накладывает жесткие ограничения на количество информации, которое мы можем воспринять, обработать и запомнить. Организуя поступление входной информации одновременно по нескольким различным каналам и в виде последовательности отдельных порций, мы можем прорвать… этот информационный затор". В современной терминологии это называют разбиением или выделением абстракций. Люди развили чрезвычайно эффективную технологию преодоления сложности. Мы абстрагируемся от нее. Будучи не в состоянии полностью воссоздать сложный объект, мы просто игнорируем не слишком важные детали и, таким образом, имеем дело с обобщенной, идеализированной моделью объекта. И хотя мы по-прежнему вынуждены охватывать одновременно значительное количество информации, но благодаря абстракции мы пользуемся единицами информации существенно большего семантического объема. Это особенно верно, когда мы рассматриваем мир с объектно-ориентированной точки зрения, поскольку объекты как абстракции реального мира представляют собой отдельные насыщенные связные информационные единицы.

Роль иерархии

Другим способом, расширяющим информационные единицы, является организация внутри системы иерархий классов и объектов. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которое осуществляется с помощью механизмов взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы. Зачем, например, изучать фотосинтез каждой клетки отдельного листа растения, когда достаточно изучить одну такую клетку, поскольку мы ожидаем, что все остальные ведут себя подобным же образом. И хотя мы рассматриваем каждый объект определенного типа как отдельный, можно предположить, что его поведение будет похоже на поведение других объектов того же типа. Классифицируя

объекты по группам родственных абстракций (например, типы клеток растений в противовес клеткам животных), мы четко разделяем общие и уникальные свойства разных объектов, что помогает нам затем справляться со свойственной им сложностью.

9.4 Эволюция объектной модели

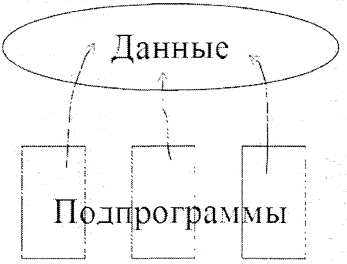
Объектно-ориентированный подход связан со следующими событиями:

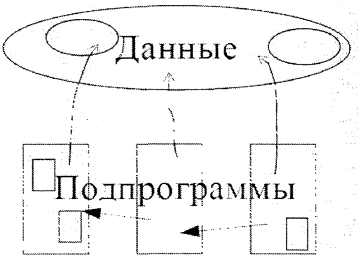
- развитие методологии программирования, включая принципы модульности и скрытия данных;
- развитие языков программирования;
- прогресс в области архитектуры ЭВМ и операционных систем;
- развитие теории баз данных;
- исследования в области искусственного интеллекта;
- достижения философии и теории познания.

Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра. Позднее Парнас ввел идею скрытия информации, а в 70-х годах ряд исследователей разработали механизмы абстрактных типов данных. Хоар дополнил эти подходы теорией типов и подклассов.

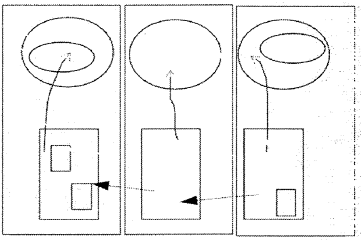
Развитие языков программирования:

Первое поколение (1954-1958)	FORTRAN I ALGOL-58	Математические формулы
Второе поколение (1959-1961)	FORTRAN II	Подпрограммы, раздельная компиляция
	ALGOL-60	Блочная структура, типы данных
	COBOL	Описание данных, работа с файлами
	LISP	Обработка списков, сборка мусора

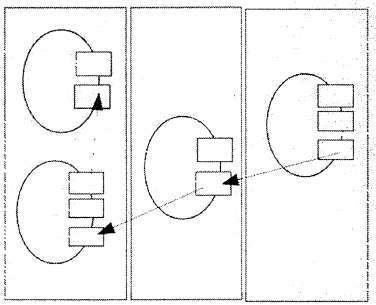




Третье поколение (1962-1970)	PL/I Pascal Simula	Объединение возможностей Строгая типизация, модули Классы, абстрактные данные
Четвертое (потерянное) поколение (1970-1980)	Множество языков Smalltalk Ada C++	Проблемно-ориентированные ООП Параллелизм Шаблоны



Модули



Модули

В 70-80-х годах делались попытки отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программной абстракции и низким уровнем ЭВМ. По мнению сторонников этих подходов, тогда были созданы более качественные средства, обеспечивающие: лучшее выявление ошибок, большую эффективность реализации программ, сокращение набора инструкций, упрощение компиляции, снижение объема требуемой памяти. Были разработаны компьютеры Burroughs 5000, SWARD, Intel 432, IBM System/38 (AS/400). Для объектно-ориентированной архитектуры потребовались объектно-ориентированные операционные системы.

Развивавшиеся достаточно независимо технологии построения баз данных также оказали влияние на объектный подход, в первую очередь благодаря так называемой модели "сущность-отношение" (ER, entity-relationship), в которой моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.

Разработчики способов представления данных в области искусственного интеллекта также внесли свой вклад в понимание объектно-ориентированных абстракций. В 1975 г. Мински выдвинул теорию фреймов для представления реальных объектов в системах распознавания образов и

естественных языков. Фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах.

Объектный подход известен еще издавна. Грекам принадлежит идея о том, что мир можно рассматривать в терминах как объектов, так и событий. А в XVII веке Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир. В XX веке эту тему развивала Рэнд в своей философии объективистской эпистемологии. Позднее Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различно мыслящих агентов. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека.

9.5 Объектно-ориентированный подход

Объектно-ориентированное программирование (OOP) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований.

Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.

- Объекты относятся к соответствующим типам (классам).
- Типы (классы) могут наследовать атрибуты супертипов (суперклассов).

Объектно-ориентированное проектирование (OOD) – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором – алгоритмами.

Объектно-ориентированный анализ (OOA) – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

На результатах OOA формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии ООР.

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:
   - сложная система;
   - декомпозиция;
   - абстракция;
   - иерархия.
2. Укажите проявления признаков сложной системы для следующих систем:
   - дерево;
   - компьютер;
   - компания Apple;
   - физика (как наука).

10 КОНЦЕПЦИИ ООП

Термины, знание которых необходимо для понимания главы:
- объект;
- класс;
- операция;
- исключительная ситуация;
- сложная система;
- заголовочный файл;
- препроцессор;
- область видимости.

В этой главе цитируются основные идеи главы 2 книги [6].

Каждый стиль программирования имеет свою концептуальную базу и требует своего способа восприятия решаемой задачи. Для объектно-ориентированного стиля концептуальная база – это объектная модель. Она имеет четыре главных элемента:
- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Эти элементы являются главными в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три дополнительных элемента:
- типизация;

– параллелизм;
– сохраняемость.

Называя их дополнительными, мы имеем в виду, что они полезны в объектной модели, но не обязательны.

10.1 Абстрагирование

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Такое разделение основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения и ничего больше. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу ОО проектирования.

Существует целый спектр абстракций, перечисляемых далее от наиболее полезных к наименее полезным:

Абстракция сущности	Объект представляет собой полезную модель некой сущности в предметной области
Абстракция поведения	Объект состоит из обобщенного множества операций
Абстракция виртуальной машины	Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
Произвольная абстракция	Объект включает в себя набор операций, не имеющих друг с другом ничего общего

Поведение объекта можно охарактеризовать услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. В *контрактной модели* программирования внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет ответственность объекта – то поведение, за которое он отвечает.

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значе-

ния. Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется *протоколом*. Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.

Центральной идеей абстракции является понятие *инварианта*. Инвариант – это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать предусловия (инварианты, предполагаемые операцией) и постусловия (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и объект-сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия обычно порождается исключительная ситуация.

10.2 Инкапсуляция

Инкапсуляция (ограничение доступа) – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части. Абстракция помогает создавать программы, а инкапсуляция упрощает их модификацию. Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его методов.

Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Практически это означает наличие двух частей в классе: интерфейса и реализации. Интерфейс отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя реализация описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Скрытие информации – понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. "Инкапсуляция защищает от ошибок, но не от жульничества."

10.3 Модульность

Модульность – это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

Разделение программы на части (модули) до некоторой степени позволяет уменьшить ее сложность. Внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом. Классы и объекты составляют логическую структуру системы, они помещаются в модули, образующие физическую структуру системы. Выявление классов и объектов в проекте и организация модульной структуры – независимые действия, выполняемые итеративно. Модульность дополняет инкапсуляцию и абстрагирование.

В разных языках программирования модульность поддерживается поразному. В Smalltalk модулей нет, а в Delphi модуль – это специальная языковая конструкция.

```
unit mymodule;
interface // интерфейс
  function len(a,b:extended):extended;
implementation // реализация
  uses math; // использование модуля math
  function len(a,b:extended):extended;
  begin
    len:=sqrt(a*a+b*b);
  end;
end.
```

В языке C++ модулями являются раздельно компилируемые файлы. Интерфейсная часть модуля помещается в заголовочный файл с расширением `.h` (может быть общим для нескольких модулей), а реализация помещается в файл с расширением `.cpp`. Заголовочный файл подключается к файлу с реализацией и в модули-клиенты с помощью директивы препроцессора `#include`. Такой подход строится исключительно на соглашении и не является строгим требованием самого языка. Пример программы, состоящей из двух модулей.

Файл module1.h:

```
struct point {
  double x,y;
  double len();
```

```
};
extern point p1;
```
Файл module1.cpp:
```
#include "module1.h"
#include <math.h>
double point::len()
{ return sqrt(x*x+y*y);
}
point p1;
```
Файл module2.cpp:
```
#include "module1.h"
...
int main()
{ point a=p1;
  cout<<a.len()<<"\n";
  ...
}
```

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений этот процесс можно стандартизовать, но для новых задач задача может быть очень трудной.

Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми только те элементы, которые необходимо видеть другим модулям. Конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны.

Если изменяется реализация, то заново компилируется только данный модуль, и программа перекомпонуется. Перекомпиляция интерфейсной части модуля, напротив, более трудоемка, так как приходится перекомпилировать все модули, связанные с данным, для очень больших программ могут потребоваться многие часы на перекомпиляцию. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была возможно более узкой. Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информа-

цию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях.

При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные – за реализацию. Документирование проекта также делается по модульному принципу – модуль служит единицей описания и администрирования.

10.4 Иерархия

Иерархия – это упорядочение абстракций, расположение их по уровням.

Абстракция – вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is a") и структура объектов (иерархия "part of").

Основным видом иерархии "is a" является наследование – такое отношение между классами, когда один класс заимствует структурную и функциональную часть одного или нескольких других классов (соответственно, одиночное и множественное наследование). Часто подкласс дополняет или изменяет элементы вышестоящего класса. В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля".

Наследование порождает иерархию "обобщение-специализация", в которой подкласс представляет собой специализированный частный случай своего суперкласса. Множественным наследованием часто злоупотребляют, используя его вместо агрегации. "Лакмусовая бумажка" наследования – обратная проверка; так, если В не есть А, то В не стоит производить от А. Множественное наследование усложняет реализацию языков программирования, поэтому многие языки ее не поддерживают (Delphi) или поддерживают частично (Java).

В иерархии "part of" класс находится на более высоком уровне абстракции, чем любой из использовавшихся при его реализации. Агрегация есть во всех языках, использующих структуры, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции. При уничтожении объекта главного класса должны быть уничтожены все объекты, являющиеся его частями, т.е. объект является владельцем своих составляющих.

10.5 Типизация

Типизация – это способ защититься от использования объектов одного класса вместо другого или, по крайней мере, управлять таким использованием.

Конкретный язык программирования может иметь сильный или слабый механизм типизации, и даже не иметь вообще никакого, оставаясь объектно-ориентированным. В сильно типизированных языках нарушение согласования типов может быть обнаружено во время трансляции программы. С другой стороны, в Smalltalk типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение об ошибке. Нарушение согласования типов может не обнаружиться во время трансляции и обычно проявляется как ошибка исполнения.

Языки, в которых типизация отсутствует, обладают большей гибкостью (например, можно создавать наборы из разнородных объектов), но даже в таких языках программисты обычно знают, какие объекты ожидаются в качестве аргументов и какие будут возвращаться. На практике при разработке сложных систем надежность языков со строгой типизацией с лихвой компенсирует некоторую потерю в гибкости по сравнению с нетипизированными языками.

Строго типизированные языки имеют следующие преимущества:

- Все выражения будут согласованы по типу.
- Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.
- В большинстве систем процесс редактирование-компиляция-отладка утомителен, и раннее обнаружение ошибок просто незаменимо.
- Объявление типов улучшает документирование программ.
- Многие компиляторы генерируют более эффективный объектный код, если им явно известны типы.

По времени связывания имен объектов с их типами выделяют также статическое связывание (раннее связывание, во время компиляции) и динамическое связывание (позднее связывание, в момент выполнения программы). Концепции типизации и связывания являются независимыми, поэтому в языке программирования может использоваться разная стратегия связывания и типизации.

Одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять. Это особенность называется *полиморфизмом*. Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание.

Полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют одинаковые протоколы. Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.

Наследование без полиморфизма возможно, но не очень полезно. Это видно на примере Ada, где можно объявлять производные типы, но из-за мономорфизма языка операции жестко задаются на стадии компиляции.

При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В C++ программист имеет возможность выбирать между ранним и поздним связыванием имени с операцией. Если функция виртуальная, связывание будет поздним и, следовательно, функция полиморфна. Если нет, то связывание происходит при компиляции и ничего изменить потом нельзя.

10.6 Параллелизм

Параллелизм позволяет различным объектам действовать одновременно. Параллелизм – это способность объектов находиться либо в активном, либо в пассивном состоянии.

Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (поток управления) – это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: одни существуют недолго, а другие живут в течении всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

Кроме этого "аппаратного" различия, будем различать "тяжелые" процессы, для которых выделяется отдельное защищенное адресное пространство, и "легкие" процессы, которые сосуществуют в одном адресном пространстве. "Тяжелые" процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь "легких" процессов осуществляется гораздо проще, часто они используют одни и те же данные.

Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах.

В то время, как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется активным. Для систем, построенных на основе OOD, мир может быть представлен, как совокупность взаимодействующих объектов, часть из которых является активной.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщения третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет растерзан на части несколькими независимыми процессами.

Существует три подхода к параллелизму.

Во-первых, параллелизм – это внутреннее свойство некоторых языков программирования, таких как Ada, Smalltalk, Java и т.д. Во всех этих языках можно создавать активные объекты, код которых постоянно выполняется параллельно с другими активными объектами.

Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность "легкого" параллелизма. Ее реализация, естественно, зависит от платформы, хотя интерфейс достаточно хорошо переносим. При этом подходе механизмы параллельного выполнения не встра-

иваются в язык (и, значит, не влияют на системы без параллельности), но в то же время практически воспринимаются как встроенные.

Наконец, в-третьих, можно создать иллюзию многозадачности с помощью прерываний. Например, аппаратный таймер может периодически прерывать приложение и при изменении состояния внешних устройств соответствующие им объекты обращались бы, если нужно, к своим функциям вызова.

10.7 Сохраняемость

Сохраняемость (устойчивость) – способность объекта существовать во времени, переживая породивший его процесс, и/или в пространстве, перемещаясь из своего первоначального адресного пространства.

Любой программный объект существует в памяти и живет во времени. Спектр сохраняемости объектов охватывает:

- Промежуточные результаты вычисления выражений.
- Локальные переменные в вызове процедур.
- Собственные (статические) переменные функции, глобальные переменные и динамически создаваемые данные.
- Данные, сохраняющиеся между сеансами выполнения программы.
- Данные, сохраняемые при переходе на новую версию программы.
- Данные, которые вообще переживают программу.

Традиционно, первыми тремя уровнями занимаются языки программирования, а последними – базы данных. Языки программирования, как правило, не поддерживают сохраняемость в полном объеме (исключение – язык Smalltalk). Введение сохраняемости, как нормальной составной части объектного подхода приводит к объектно-ориентированным базам данных (OODB). На практике подобные базы данных строятся на основе проверенных временем моделей – последовательных, индексированных, иерархических, сетевых или реляционных, но программист может ввести абстракцию объектно-ориентированного интерфейса, через который запросы к базе данных и другие операции выполняются в терминах объектов, время жизни которых превосходит время жизни отдельной программы. При использовании одноуровневой памяти как в System/38 разработка OODB существенно упрощается.

Сохраняемость – это не только проблема сохранения данных. В OODB имеет смысл сохранять и классы, так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно, если класс объекта вдруг потребовалось изменить.

В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве, так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти.

Контрольные вопросы и упражнения

1. Укажите определение и примеры применения следующих изученных терминов:
   - абстракция;
   - абстрагирование;
   - контрактная модель;
   - протокол;
   - инвариант;
   - предусловие;
   - постусловие;
   - инкапсуляция (ограничение доступа);
   - интерфейс;
   - реализация;
   - модульность;
   - модуль;
   - иерархия;
   - наследование;
   - типизация;
   - статическое связывание;
   - динамическое связывание;
   - полиморфизм;
   - мономорфизм;
   - параллелизм;
   - сохраняемость (устойчивость).
2. Укажите в какой мере и в какой форме реализуются концепции ООП в языке C++.

11 ОБЪЕКТЫ

Термины, знание которых необходимо для понимания главы:
- предметная область;
- утечка памяти;
- куча;
- стек;
- статическая память.

В этой главе цитируются основные идеи подглав 3.1 и 3.2 книги [6].

11.1 Определение объекта

С точки зрения восприятия человеком объектом может быть:

- осязаемый и (или) видимый предмет (мяч);
- нечто, воспринимаемое мышлением (алгоритм);
- нечто, на что направлена мысль или действие (время).

С точки зрения ОО подхода *объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области.*

Объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве. Объект может быть определен как нечто, имеющее четко очерченные границы. Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение. Два тела, например, сфера и куб, имеют как правило нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами. Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей.

Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.

11.2 Состояние

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер), поэтому в данном определении использован термин "обычно динамическими".

Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта, но в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера). Состояние ОО системы в целом инкапсулировано в объекты.

11.3 Поведение

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Поведение объекта определяется выполняемыми над ним операциями и его состоянием, причем некоторые операции имеют побочное действие: они изменяют состояние. Состояние объекта представляет суммарный результат его поведения.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Операция – это услуга, которую класс может предоставить своим клиентам. Выделяют следующие виды операций:

- модификатор – операция, которая изменяет состояние объекта;
- селектор – операция, считывающая состояние объекта, но не меняющая состояния;
- итератор – операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности;
- конструктор – операция создания объекта и/или его инициализации;
- деструктор – операция, освобождающая состояние объекта и/или разрушающая сам объект.

В чисто объектно-ориентированных языках, таких как Smalltalk, операции могут быть только методами, в Delphi, C++ допускается описывать операции как независимые от объектов подпрограммы. Таким образом, можно утверждать, что все методы – операции, но не все операции – методы: некоторые из них представляют собой свободные подпрограммы.

Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение. Это наводит на мысль

представить объект в качестве небольшого вычислительного устройства, например, конечного автомата. Объекты могут быть активными и пассивными. Активный объект имеет свой поток управления, а пассивный – нет. Активный объект в общем случае автономен, то есть он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы – источники управляющих воздействий. Если система имеет несколько потоков управления, то и активных объектов может быть несколько.

11.4 Идентичность

Идентичность (уникальность) – это такое свойство объекта, которое отличает его от всех других объектов.

В большинстве языков программирования для различения объектов используют имя, тем самым путая адресуемость и идентичность. В базах данных различают объекты по набору ключевых полей, тем самым смешивая идентичность и значение данных.

Объект может использоваться в программе под несколькими синонимичными именами. Такая ситуация называется структурной зависимостью и порождает в объектно-ориентированном программировании много проблем. Трудность распознания побочных эффектов при действиях с объектами, имеющими имена-синонимы, часто приводит к "утечкам памяти", неправильному доступу к памяти, и, хуже того, непрогнозируемому изменению состояния.

Объекты в функцию можно передавать по ссылке и по значению. По ссылке передается адрес объекта, что дает возможность изменять переданный объект, реализовать полиморфное поведение и повысить эффективность при работе со сложными объектами. При передаче по значению создается объект-копия, который имеет такое же состояние. Копирование может быть "поверхностным" (копируется только сам объект, а состояние является разделяемым) и "глубоким" (копируется объект и состояние рекурсивно). В C++ конструктор копий по умолчанию копирует объект поэлементно, что приводит к созданию синонимов на составные части (разделению состояния), когда объект содержит ссылки или указатели на другие объекты.

Аналогично, присваивание может быть сделать "глубоким" и "поверхностным".

С вопросом присваивания тесно связан вопрос равенства. Равенство можно понимать двумя способами. Во-первых, два имени могут обозначать один и тот же объект. Во-вторых, это может быть равенство состоя-

ний у двух разных объектов. В C++ нет предопределенного операции равенства, поэтому нужно самостоятельно определять операции равенства и неравенства.

Операции присваивания и равенства можно определить как виртуальные, чтобы подклассы могли переопределять их поведение.

11.5 Время жизни объектов

Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием – возвращение отведенного участка памяти системе.

Объекты могут создаваться явно или неявно. Есть два способа создать объекты явно. Во-первых, это можно сделать при объявлении – тогда объект размещается в стеке или в статической памяти. Во-вторых, можно разместить объект, то есть выделить ему память из "кучи". В C++ в любом случае при этом вызывается конструктор, который инициализирует объект. Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке временную копию объекта.

Создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики конструктора копий и операции присваивания в C++ разрешает явное управление тем, когда части объекта создаются и уничтожаются.

Уничтожение объектов также может выполняться явно и неявно. В Smalltalk и Java при потере последней ссылки на объект его забирает сборщик мусора. В языках без сборки мусора, типа C++, объекты, созданные в стеке, уничтожаются неявно при выходе из блока, в котором они были определены, но объекты, созданные в "куче" оператором new, продолжают существовать и занимать место в памяти: их необходимо явно уничтожать оператором delete. Если объект "забыть", не уничтожить, это вызовет утечку памяти. Если же объект попробуют уничтожить повторно (например, через другой указатель), последствием будет сообщение о нарушении памяти или полный крах системы.

При явном или неявном уничтожении объекта в C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить, что делать с другими ресурсами, например, с открытыми файлами.

В некоторых системах объекты могут быть долгоживущими; под этим понимается, что их время жизни может выходить за время жизни породивших их программ. Все объекты, которым мы хотим обеспечить долгую жизнь, должны наследовать от специальных сохраняемых классов.

11.6 Отношения между объектами

Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система.

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов: связь и агрегация.

Связь – это физическое или концептуальное соединение между объектами. Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь – это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому. Только вдоль связи один объект может послать сообщение другому. Связь между объектами и передача сообщений обычно односторонняя и инициализируется клиентом, но данные передаются в обоих направлениях.

Участвуя в связи, объект может выполнять одну из следующих трех функций:

- Воздействие. Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов; в определенном смысле это соответствует понятию активный объект.
- Исполнение. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта.
- Посредничество. Такой объект может выступать как в активной, так и в пассивной роли; как правило, объект-посредник создается для выполнения операций в интересах какого-либо активного объекта или посредника.

Чтобы клиент мог послать сообщение серверу, надо, чтобы сервер был видим для клиента. В принципе есть следующие четыре способа обеспечить видимость:

- Сервер глобален по отношению к клиенту.
- Сервер (или указатель на него) передан клиенту в качестве параметра операции.
- Сервер является частью клиента.
- Сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Какой именно из этих способов выбрать – зависит от тактики проектирования.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями – это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое по своей природе.

Выбирая одно из двух – связь или агрегацию – надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Иногда, наоборот, предпочтительнее связи, поскольку они слабее и менее ограничительны. Принимая решение, надо взвесить все.

Объект, являющийся частью другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

Контрольные вопросы и упражнения

1. Укажите определение следующих изученных терминов:
- объект;
- состояние объекта;
- поведение объекта;
- операция;
- модификатор;
- селектор;
- итератор;
- конструктор;
- деструктор;
- активный объект;
- пассивный объект;
- идентичность (уникальность);
- структурная зависимость;
- поверхностное копирование;
- глубокое копирование;
- равенство объектов;
- время жизни объектов;
- долгоживущие объекты;
- отношение связи;
- агрегация.

2. Реализуйте поверхностную и глубокую операцию сравнения на равенство для класса Stack на языке C++.

12 КЛАССЫ

Термины, знание которых необходимо для понимания главы:

- объект;
- спецификаторы доступа;
- иерархия;
- структурная зависимость;
- ссылка;
- указатель;
- шаблон;
- статические элементы класса.

В этой главе цитируется текст подглав 3.3, 3.4 и 3.6 книги [6].

12.1 Определение класса. Отношения между классами

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте.

Класс – это некое множество объектов, имеющих общую структуру и общее поведение.

Любой конкретный объект является экземпляром какого-то класса. Даже если класс имеет только одного представителя, этот объект не является классом, хотя в некоторых случаях класс можно рассматривать как объект.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, на достаточно высоком уровне абстракции графический интерфейс пользователя, база данных или система учета как целое, это явные объекты, но не классы. Лучше считать их некими совокупностями сотрудничающих классов.

По своей природе, класс – это генеральный контракт между абстракцией и всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.

Идея контрактного программирования приводит к разграничению внешнего облика, то есть интерфейса, и внутреннего устройства класса, реализации. Главное в интерфейсе – объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других

классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Напротив, реализация класса никому, кроме него самого, не интересна. По большей части реализация состоит в определении операций, объявленных в интерфейсе класса.

Интерфейс класса обычно делится на три части:

- открытую (public) – видимую всем клиентам;
- защищенную (protected) – видимую самому классу, его подклассам и друзьям;
- закрытую (private) – видимую только самому классу и его друзьям.

Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента. Структура объекта определяется в интерфейсной части класса, а не в его реализации, чтобы компилятор знал, сколько памяти необходимо выделить под объект. Если бы эта информация содержалась в реализации класса, нам пришлось бы написать реализацию полностью до определения его клиентов. То есть, весь смысл отделения интерфейса от реализации был бы потерян.

Классы, как и объекты, не существуют изолированно, они взаимодействуют разными способами. Между классами возникают иерархические отношения ("обобщение/специализация" и "целое/часть") и семантические, смысловые отношения, ассоциации.

Большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

- ассоциация;
- наследование;
- агрегация;
- использование;
- инстанцирование (параметризация);
- метакласс.

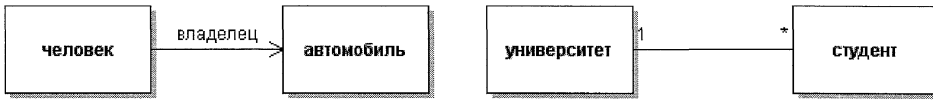
12.2 Ассоциация

Из шести перечисленных видов отношений наиболее общим и неопределенным является ассоциация. Обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

Ассоциация – смысловая связь. По умолчанию, она не имеет направления (если не оговорено противное) и не объясняет, как классы общаются друг с другом (мы можем только отметить семантическую зависимость, указав, какие роли классы играют друг для друга). Однако именно это требуется на ранней стадии анализа. Мы фиксируем участников, их роли и

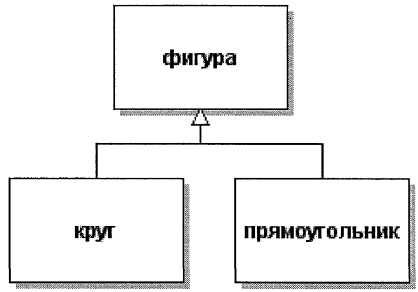
мощность отношения. На практике важно различать три случая мощности ассоциации:

- один-к-одному
- один-ко-многим
- многие-ко-многим



12.3 Наследование

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Обычно подклассы повторяют структуры их суперклассов. Поведение суперклассов также наследуется. В большинстве языков допускается не только наследование методов суперкласса, но также добавление новых и переопределение существующих методов. В Smalltalk любой метод суперкласса можно переопределить в подклассе. В C++ степень контроля за этим несколько выше. Функция, объявленная виртуальной, может быть в подклассе переопределена, а остальные – нет.



Самый общий класс в иерархии классов называется базовым. В многих языках языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов (в Smalltalk – object, в Delphi – TObject), в C++ хорошо сделанная структура классов – это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем.

У класса обычно бывает два вида клиентов:

– экземпляры;
– подклассы.

Часто полезно иметь для них разные интерфейсы. В частности, мы хотим показать только внешне видимое поведение для клиентов-экземпля-

ров, но нам нужно открыть служебные функции и представления клиентам-подклассам. Этим объясняется наличие открытой, защищенной и закрытой частей описания класса в языке C++: разработчик может четко разделить, какие элементы класса доступны для экземпляров, а какие для подклассов.

Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему классу некоторые секреты. На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его суперклассы в их внутренних деталях.

Наследование можно рассматривать, как способ управления повторным использованием программ, то есть, как простое решение разработчика о заимствовании полезного кода. В этом случае механика наследования должна быть гибкой и легко перестраиваемой. Другая точка зрения: наследование отражает принципиальную родственность абстракций, которую невозможно отменить. В Smalltalk эти два аспекта неразделимы. C++ более гибок. В частности, при определении класса его суперкласс можно объявить public. В этом случае подкласс считается также и подтипом, то есть обязуется выполнять все обязательства суперкласса, в частности обеспечивая совместимое с суперклассом подмножество интерфейса и обладая неразличимым с точки зрения клиентов суперкласса поведением. Но если при определении класса объявить его суперкласс как private, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его подтипом. Это означает, что открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и следовательно они будут недоступны подклассам более низкого уровня. Кроме того, тот факт, что подкласс не будет подтипом, означает, что класс и суперкласс обладают несовместимыми (вообще говоря) интерфейсами с точки зрения клиента.

Одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами. Это ограничивает возможность повторного использования предопределенных классов и заставляет дублировать уже имеющиеся коды.

Необходимость множественного наследования в ООР остается предметом горячих споров. Множественное наследование можно сравнить с парашютом: как правило, он не нужен, но, когда вдруг он понадобится, будет жаль, если его не окажется под рукой.

Проектирование структур классов со множественным наследованием – трудная задача, решаемая путем последовательных приближений. Есть две специфические для множественного наследования проблемы – как разре-

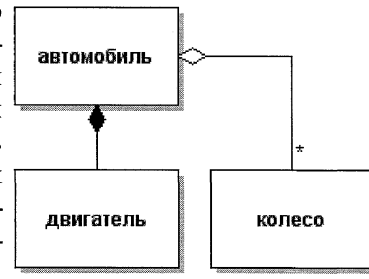
шить конфликты имен между суперклассами и что делать с повторным наследованием.

Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Борются с этим конфликтом тремя способами. Во-первых, можно считать конфликт имен ошибкой и отвергать его при компиляции (Smalltalk). Во-вторых, можно считать, что одинаковые имена означают одинаковый атрибут (так делает CLOS). В третьих, для устранения конфликта разрешается добавить к именам префиксы, указывающие имена классов, откуда они пришли (C++).

Проблема повторного наследования решается тремя способами. Во-первых, можно его запретить, отслеживая при компиляции (Smalltalk), во-вторых, можно явно развести две копии унаследованного элемента, добавляя к именам префиксы в виде имени класса-источника (C++), в-третьих, можно рассматривать множественные ссылки на один и тот же класс, как обозначающие один и тот же класс (виртуальные базовые классы C++)

При множественном наследовании часто используется прием создания примесей (mixin). Идея примесей происходит из языка Flavors: можно комбинировать (смешивать) небольшие классы, чтобы строить классы с более сложным поведением. Примесь синтаксически ничем не отличается от класса, но назначение их разное. Примесь не предназначена для порождения самостоятельно используемых экземпляров – она смешивается с другими классами, выражая какую-то одну какую-то особенность, которую можно привить другим классам через наследование. Эта особенность обычно ортогональна собственному поведению наследующего ее класса.

12.4 Агрегация



Различают физическое включение (по значению) и включение по ссылке (указателю). При использовании ссылок объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо. Чтобы избежать структурной зависимости через ссылки важно придерживаться какой-то договоренности относительно создания и уничтожения объектов, ссылки на которые могут содержаться в разных местах. Нужно, чтобы это делал кто-то один.

Агрегация является направленной, как и всякое отношение "целое/часть". Физическое вхождение одного в другое нельзя "зациклить",

а вот указатели – можно (каждый из двух объектов может содержать указатель на другой).

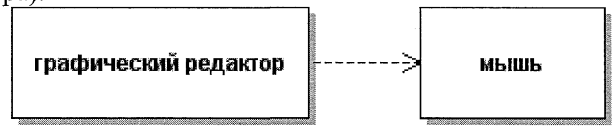
Агрегация не требует обязательного физического включения, ни по значению, ни по ссылке. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть совершенно различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть очень косвенной. "Лакмусовая бумажка" для выявления агрегации такова: если (и только если) налицо отношение "целое/часть" между объектами, их классы должны находиться в отношении агрегации друг с другом.

Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Решая, с чем вы имеете дело – с наследованием или агрегацией – будьте осторожны. Если вы не уверены, что налицо отношение общего и частного (is a), вместо наследования лучше применить агрегацию или что-нибудь ещё.

Для физического включения часто используют термин композиция (объединение). Некоторые авторы разницу между видами агрегации проводят по времени жизни объектов, то есть физическое включение означает, что время жизни составных частей совпадает с временем жизни составного объекта, так как многие современные языки программирования (C#, Java) не поддерживают включение по значению.

12.5 Использование

Отношение использования между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера).



На самом деле, один класс может использовать другой по-разному. В типичном случае отношение использования проявляет себя, если в реализации какой-либо операции происходит объявление локального объекта используемого класса.

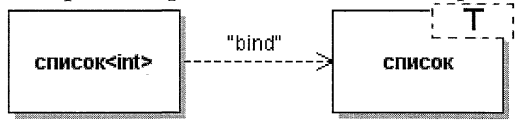
Строгое отношение использования иногда несколько ограничительно, поскольку клиент имеет доступ только к открытой части интерфейса сервера. Иногда по тактическим соображениям мы должны нарушить инкап-

суляцию, для чего, собственно, и служат "дружеские" отношения классов в C++.

12.6 Инстанцирование (конкретизация)

Существует четыре основных способа создавать параметризованные классы. Во-первых, мы можем использовать макроопределения. Например, так было сделано в раннем C++, но этот подход годился только для небольших проектов. Во-вторых, можно положиться на позднее связывание и наследование, как это делается в Smalltalk. При таком подходе мы можем строить только неоднородные контейнерные классы, так как в языке нет средства ввести нужный класс элементов контейнера; каждый элемент в контейнере трактуется как экземпляр некоторого удаленного базового класса. Третий способ реализован в языке Delphi, которые имеют и сильные типы, и наследование, но не поддерживают никакой разновидности параметризованных классов. В этом случае приходится создавать обобщенные контейнеры, как в Smalltalk, но использовать явную проверку типа объекта, прежде чем помещать его в контейнер. Наконец, есть собственно параметризованные классы, впервые появившиеся в CLU.

Параметризованный класс представляет собой что-то вроде шаблона для построения других классов; шаблон может быть параметризован другими классами, объектами или операциями. Параметризованный класс должен быть инстанцирован перед созданием экземпляров.



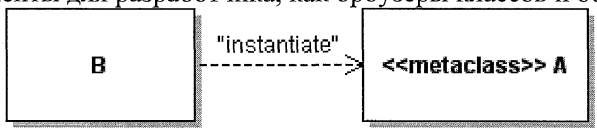
Хотя наследование является более мощным механизмом, чем обобщенные классы и через него можно получить большинство преимуществ обобщенных классов, лучше, когда языки поддерживают и то, и другое.

При проектировании обобщенные классы позволяют выразить некоторые свойства протоколов классов. Класс экспортирует операции, которые можно выполнять над его экземплярами. Наоборот, параметризующий аргумент класса служит для импорта классов и значений, предоставляющих некоторый протокол. C++ проверяет их взаимное соответствие при компиляции, когда фактически и происходит инстанцирование. Например, мы могли бы определить упорядоченную очередь объектов, отсортированных по некоторому критерию. Этот параметризованный класс должен иметь аргумент (класс Item), и требовать от этого аргумента определенное поведение (наличие операции <). При инстанцировании в качестве класса Item годится любой класс, который имеет соответствующий протокол. Таким

образом, поведение классов в семействе, происходящем от одного параметризованного класса, может изменяться в весьма широких пределах.

12.7 Метакласс

Любой объект является экземпляром какого-либо класса. Что будет, если мы попробуем и с самими классами обращаться как с объектами? Для этого нам надо ответить на вопрос, что же такое класс класса? Ответ – это метакласс. Иными словами, метакласс – это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Главное назначение – возможность экспериментировать с другими объектно-ориентированными парадигмами и создавать такие инструменты для разработчика, как броузеры классов и объектов.



Метаклассы есть в Smalltalk и Delphi, но не в C++. Первичное назначение метакласса – поддержка переменных класса (которые являются общими для всех экземпляров этого класса), операций инициализации переменных класса и создания единичного экземпляра метакласса. Хотя в C++ метаклассов нет, семантика его конструкторов и деструкторов служит целям, аналогичным тем, что вызвали к жизни метаклассы. C++ имеет средства поддержки и переменных класса, и операций метакласса (элементы данных или методы класса описанные как статические).

12.8 Выявление классов и выбор операций

Опыт показывает, что процесс выделения классов и объектов является последовательным, итеративным. Очень важно, следовательно, с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев: зацепление, связность, достаточность, полнота, примитивность.

Зацепление – это степень глубины связей между отдельными модулями, классами и объектами. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения зацепления между отдельными модулями. Существует определенное противоречие между явлениями зацепления и наследования. С одной стороны, желательно избегать сильного зацепления классов; с другой стороны, механизм наследования,

тесно связывающий подклассы с суперклассами, помогает выгодно использовать сходство абстракций.

Связность – это степень взаимодействия между элементами отдельного модуля, класса или объекта, характеристика его насыщенности. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели.

Под достаточностью подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Для примера рассмотрим класс set (множество). Операция удаления элемента из множества в этом классе, очевидно, необходима, но будет ошибкой не включить в этот класс и операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию.

Под полнотой подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты применения абстракции. Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, добавляя в интерфейс такие операции, которые можно реализовать на более низком уровне.

Из этого вытекает требование примитивности. *Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции.* Так, в примере с классом set операция добавления к множеству элемента примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Конечно, эффективность тоже вещь субъективная. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Операция, которая может быть описана в терминах существующих примитивных операций, но ценой значительно больших вычислительных затрат, также является кандидатом на включение в разряд примитивных (например, операция добавления к множеству другого множества).

Описание интерфейса класса или модуля – трудная работа. Обычно первое приближение делается, исходя из структурного смысла класса, а затем, когда появляются клиенты класса, интерфейс уточняется, модифицируется и дополняется. В частности может возникнуть потребность в создании новых классов или в изменении взаимодействия существующих.

В пределах каждого класса принято иметь только примитивные операции, отражающие отдельные аспекты поведения. Такие методы называются точными. Принято также отделять методы, не связанные между собой. Это облегчает образование подклассов с переопределением поведения. Решение о количестве методов может быть обусловлено двумя причинами: описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода; расщепление метода усложняет интерфейс, но делает каждый из методов проще. Обычно операции объявляются как методы класса, к объектам которого относятся данные действия, но многие языки допускают описание операций в виде свободных подпрограмм.

В объектно-ориентированном проектировании принято рассматривать методы класса как единое целое, поскольку все они взаимодействуют друг с другом для реализации протокола абстракции. Таким образом, определив поведение, нужно решить, в каком из классов это поведение реализуется. Критериями для принятия решения служат следующие вопросы:

Повторная используемость: Будет ли это поведение полезно более чем в одном контексте?

Сложность: Насколько трудно реализовать такое поведение?

Применимость: Насколько данное поведение характерно для класса, в который мы хотим включить поведение?

Знание реализации: Надо ли для реализации данного поведения знать секреты класса?

При ответе на эти вопросы нужно использовать принципы проектирования. Большинство принципов ориентированы на уменьшение сложности, создание как можно более простого кода. Принцип KISS (keep it simple stupid, keep it short and simple, делай проще) требует избегать усложнений кода или функциональности, если в этом нет необходимости. Принцип DRY (don't repeat youself, не повторяйся) или DIE (duplication is evil, повторение — зло) требует избегать повторения кода («copy-paste»). Принцип YAGNI (you ain't gonna need it, вам это не понадобится) требует реализовывать только поставленные задачи, не отвлекаясь на задачи, которые *возможно* потребуется решать в будущем. Принцип «Less is more» (меньше — лучше) объявляет простоту реализации и простоту интерфейса более важными, чем любые другие свойства системы. Также на поддержание простоты направлены правила «не заставляйте меня думать», «пиши код для сопровождающего», «принцип наименьшего удивления», «избегай преждевременной оптимизации», «повторное использование кода – это хорошо», «делай самую простейшую вещь, которая скорее всего заработает». Пять принципов ООП получили отдельную аббревиатуру SOLID:

- Single responsibility principle (принцип единственности ответственности) — один класс отвечает за один функционал.
- Open/closed principle (принцип открытости/закрытости) — сущности должны быть открыты к расширению, но закрыты — к изменению, т. е. новая функциональность должна приводить к добавлению новых подклассов, методов, а не к изменению существующего кода.
- Liskov substitution principle (принцип подстановки Барбары Лисков) — сущность, использующая объект, который реализует определенный интерфейс, должна иметь возможность использовать другой объект с тем же интерфейсом, не зная о факте подмены.
- Interface segregation principle (принцип разделения интерфейса) — клиентский код не должен зависеть от методов, которые не использует. Лучше иметь несколько интерфейсов для разных клиентов с небольшим числом методов, чем один — с большим.
- Dependency inversion principle (принцип инверсии зависимостей) — модули высшего порядка не должны зависеть от модулей низшего порядка, и те, и другие должны зависеть от абстракций; детали должны зависеть от абстракций, но не наоборот.

Контрольные вопросы и упражнения

1. Укажите определение следующих изученных терминов:
- класс;
- контрактная модель;
- интерфейс;
- реализация;
- отношение ассоциации;
- наследование;
- агрегация;
- отношение использования;
- инстанцирование (конкретизация);
- параметризованный класс;
- метакласс;
- зацепление;
- связность;
- достаточность;
- полнота;
- примитивность.

2. Укажите примеры реализации всех отношений между классами на языке C++.

ЧАСТЬ 3 ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

В подглаве 2.11 были рассмотрены вопросы проектирования нижнего уровня, а в подглаве 12.8 — общие принципы проектирования объектно-ориентированной системы, в последующих главах будут рассмотрены типовые задачи, возникающие при проектировании, и способы их решения.

Назначение, структура и результаты паттернов проектирования цитируются по книге [7]. Для каждого паттерна приводится пример реализации на C++.

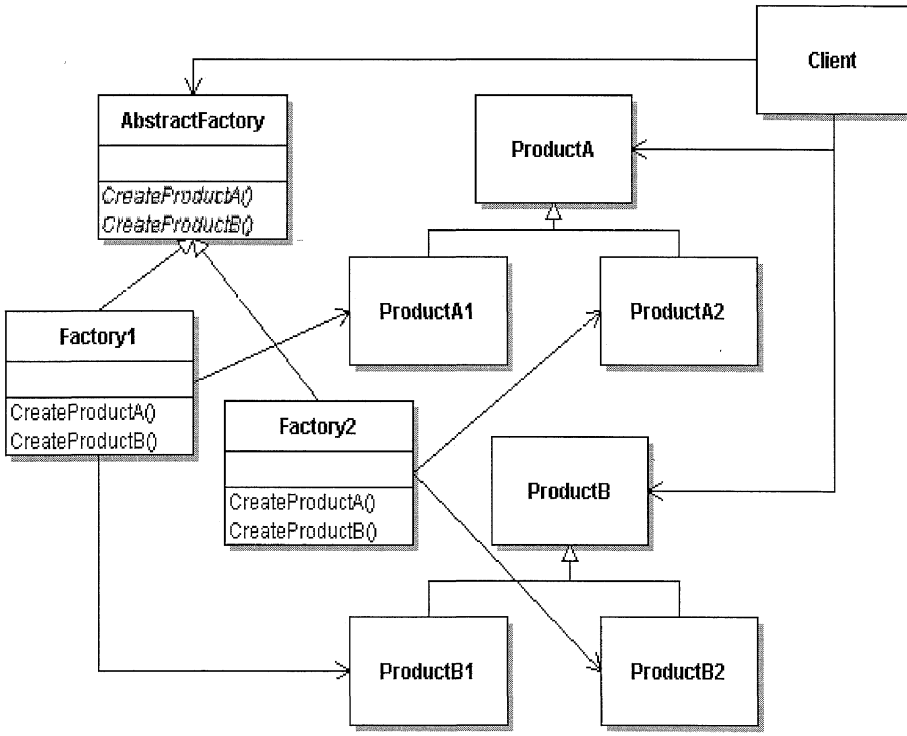
13 ПОРОЖДАЮЩИЕ ПАТТЕРНЫ

13.1 Абстрактная фабрика (Abstract Factory)

Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Структура



Результаты

1. Имена классов изготавливаемых объектов известны только конкретной фабрике, в коде клиента они не упоминаются. Клиент манипулируют объектами через их абстрактные интерфейсы.

2. Класс конкретной фабрики появляется в приложении только один раз при её создании. Это облегчает замену используемой приложением конкретной фабрики и семейства используемых продуктов.

3. Паттерн гарантирует сочетаемость создаваемых продуктов семейства.

Реализация

```
// абстрактные продукты
class ProductA {};
class ProductB {};
// Абстрактная фабрика - содержит методы
// для создания абстрактных объектов-продуктов
class AbstractFactory {
public:
  virtual ProductA *CreateProductA()=0;
  virtual ProductB *CreateProductB()=0;
};
// Конкретные продукты 1-го семейства
class ProductA1:public ProductA {};
class ProductB1:public ProductB {};
// Конкретная фабрика для создания продуктов 1-го семейства
class Factory1: public AbstractFactory {
public:
  ProductA1 *CreateProductA()
  { return new ProductA1(); }
  ProductB1 *CreateProductB()
  { return new ProductB1(); }
};
// Конкретные продукты 2-го семейства
class ProductA2:public ProductA {};
class ProductB2:public ProductB {};
// Конкретная фабрика для создания продуктов 1-го семейства
class Factory2: public AbstractFactory {
public:
  ProductA2 *CreateProductA()
  { return new ProductA2(); }
  ProductB2 *CreateProductB()
  { return new ProductB2(); }
```

```
};
// Создание фабрики
AbstractFactory *factory=new Factory1();
...
// Создание объектов в клиенте
ProductA *a=factory->CreateProductA();
```

13.2 Одиночка (Singleton)

Назначение

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Результаты

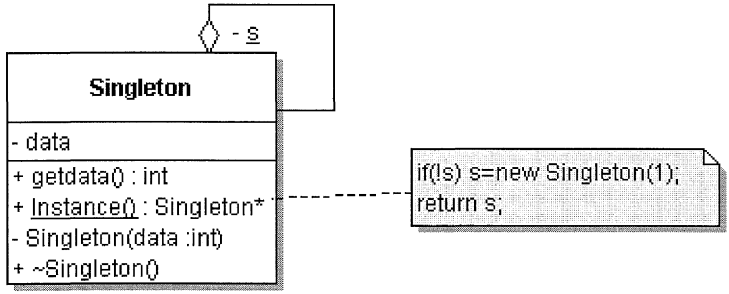
1. Класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему.

2. Паттерн одиночка позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры.

3. От класса Singleton можно порождать подклассы, а во время выполнения создавать экземпляр класса, заданного в конфигурации.

4. Паттерн позволяет использовать и более одного экземпляра класса Singleton. Для этого нужно изменить операцию доступа к экземпляру класса.

Структура



Реализация

```
// Интерфейс класса
class Singleton {
  int data; // какие-то данные
  static Singleton *s;
// Создавать объекты могут только методы этого класса
  Singleton(int d):data(d){}
// Запрещение создания копий и присваивания
  Singleton(const Singleton &)=delete;
```

```
  Singleton& operator=(const Singleton &)=delete;
public:
  // Метод для доступа к единственному экземпляру
  static Singleton *Instance();
  ~Singleton() { s=NULL; }
  // метод для доступа к данным
  int getdata() const { return data; }
};
// Реализация класса
Singleton *Singleton::s=NULL;
Singleton *Singleton::Instance()
{ if(!s)
    s=new Singleton(1);
  return s;
}
// Обращение к объекту в клиенте
int d=Singleton::Instance()->getdata();
```

Вариант реализации паттерна без возможности удаления клиентом экземпляра.

```
// Интерфейс класса
class Singleton {
  int data; // какие-то данные
// Создавать объекты могут только методы этого класса
  Singleton(int d):data(d){}
// Запрещение создания копий и присваивания
  Singleton(const Singleton &)=delete;
  Singleton& operator=(const Singleton &)=delete;
// Запрещение уничтожения объекта
  ~Singleton() {}
public:
  // Метод для доступа к единственному экземпляру
  static Singleton *Instance();
  // метод для доступа к данным
  int getdata() const { return data; }
};
// Реализация класса
Singleton *Singleton::Instance()
{ // Собственные объекты функции создаются только
  // один раз при первом вызове этой функции
  static Singleton s(1);
  return &s;
}
```

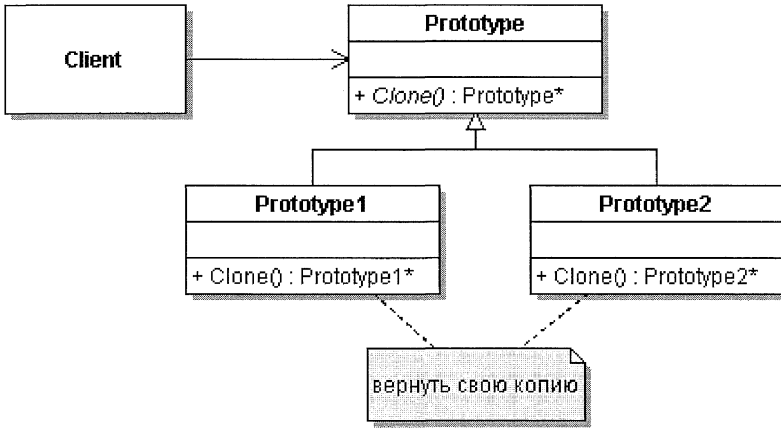
```
// Обращение к объекту в клиенте
int d=Singleton::Instance()->getdata();
```

13.3 Прототип (Prototype)

Назначение

Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

Структура



Результаты

1. Паттерн прототип позволяет добавлять и удалять продукты во время выполнения.

2. Можно определять новые виды объектов, инстанцируя уже существующие классы, изменяя значений полей и регистрируя экземпляры как прототипы клиентских объектов. В результате число необходимых системе классов может быть уменьшено.

3. Многие приложения строят объекты из более мелких составляющих. В этом случае можно определять новые виды объектов путем изменения структуры.

4. Можно избавиться от иерархии класса Creator в паттерне фабричный метод, если не запрашивать фабричный метод создать новый объект, а клонировать прототип.

5. Возможно динамическое конфигурирование приложения классами, если исполняющая среда будет автоматически создавать экземпляр каждого класса и регистрировать экземпляр в диспетчере прототипов. Затем приложение может запросить у диспетчера прототипов экземпляры вновь загруженных классов, которые изначально не были связаны с программой.

6. Каждый подкласс иерархии Prototype и их составляющие должны реализовывать операцию Clone, но ее добавление может оказаться затруднительным, если используются уже существующие классы.

Реализация

```
// Прототип объявляет интерфейс для клонирования самого себя
class Prototype {
public:
  virtual Prototype *Clone()=0;
};
// Конкретные прототипы реализуют операцию клонирования себя
class Prototype1 : public Prototype {
  Prototype1(const Prototype1 &);
public:
  Prototype1 *Clone(){ return new Prototype1(*this);}
};
class Prototype2 : public Prototype {
  Prototype2(const Prototype2 &);
public:
  Prototype2 *Clone(){ return new Prototype2(*this);}
};
// Образец
Prototype *proto;
// Создание объектов в клиенте
Prototype *p=proto->Clone();
```

13.4 Строитель (Builder)

Назначение

Отделяет конструирование сложного объекта от его представления, так, что в результате одного и того же процесса конструирования могут получаться разные представления.

Результаты

1. В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

2. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя.

3. Паттерн изолирует код, реализующий конструирование и представление. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, так как они отсутствуют в интерфейсе строителя. Разные распорядители могут строить разные варианты продукта из одних и тех же частей.

Реализация

```
// Строитель задает абстрактный интерфейс
// для создания частей объекта Product
class Builder {
public:
  virtual void BuildPartA()=0;
  virtual void BuildPartB()=0;
};
// Распорядитель конструирует объект, пользуясь интерфейсом Builder
class Director {
  Builder *builder;
public:
  Director(Builder *b):builder(b){}
  void BuildProduct() {
    builder->BuildPartA();
    builder->BuildPartB();
    ...
  }
};
// Продукт представляет собой сложный конструируемый объект
class Product {...};
// Конкретный строитель предоставляет интерфейс
// для доступа к продукту
class ConcreteBuilder : public Builder {
  Product *product;
public:
  ConcreteBuilder():product(new Product()) {}
  void BuildPartA();
  void BuildPartB();
  Product *GetProduct() { return product; }
};
// Клиент создает конкретного строителя и распорядителя
// для создания продукта
ConcreteBuilder builder;
Director director(&builder);
director.BuildProduct();
Product *product=builder.GetResult();
```

13.5 Фабричный метод (Factory Method)

Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

Результаты

1. Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми классами конкретных продуктов.

2. Паттерн может использоваться для соединения параллельных иерархий, например, основных и вспомогательных продуктов типа итераторов. С помощью фабричного метода локализуется знание о том, какие классы должны работать совместно.

Реализация

```
// Продукт определяет интерфейс объектов,
// создаваемых фабричным методом
class Product {};
// Создатель объявляет фабричный метод, возвращающий объект
// типа Product  и может вызывать этот метод для создания объектов
class Creator {
public:
  virtual Product *CreateProduct()=0;
  void Operation() {
    Product *product=CreateProduct();
    ...
  }
};
// Конкретный продукт
class Product1 : public Product {};
// Конкретный создатель замещает фабричный метод
class Creator1 : public Creator {
public:
  Product1 *CreateProduct()
   { return new Product1(); }
};
```

Контрольные вопросы

1. Укажите назначение следующих изученных паттернов:
- абстрактная фабрика;
- одиночка;
- прототип;

– строитель;
– фабричный метод.

2. В чем сходство паттернов абстрактная фабрика и фабричный метод?

3. Почему паттерн прототип иногда называют виртуальный конструктор?

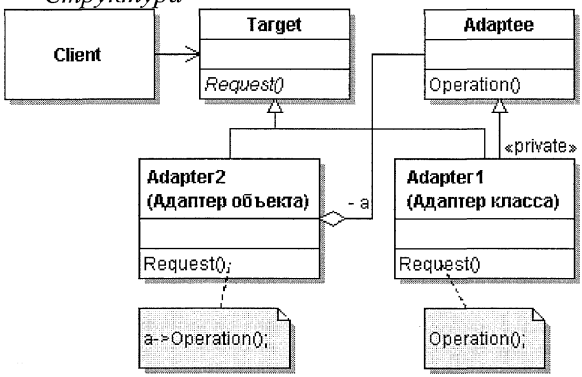
14 СТРУКТУРИРУЮЩИЕ ПАТТЕРНЫ

14.1 Адаптер (Adapter)

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Структура



Результаты

1. Используя различные сменные адаптеры можно использовать один и тот же класс в разных системах, независимо от требуемого интерфейсом.

2. Адаптер объектов уже не обладает интерфейсом Adaptee, так что его нельзя использовать там, где Adaptee был применим. В тех случаях, когда клиенты должны видеть объект по-разному, применяются двусторонние адаптеры.

Адаптер класса:

1. Можно легко заместить некоторые виртуальные методы адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee.

2. Адаптер класса вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

3. Паттерн не применим, если требуется адаптирование и класса Adaptee и его подклассов.

Адаптер объектов:

1. Позволяет адаптеру работать и с самим классом Adaptee и его подклассами.

2. Для замещения виртуальных методов класса Adaptee требуется породить от Adaptee подкласс и инициализировать указатель адресом объекта этого подкласса.

Реализация

```
// Адаптируемый класс
class Adaptee {
public:
  void Operation();
};
// Требуемый интерфейс
class Target {
public:
  virtual void Request()=0;
};
// Адаптер класса
class Adapter1: public Target, private Adaptee {
public:
  void Request() { Operation(); }
};
// Адаптер объектов
class Adapter2: public Target {
  Adaptee *a;
public:
  void Request() { a->Operation(); }
};
```

14.2 Декоратор (Decorator)

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Результаты

1. Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае простого наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. Кроме того, применение нескольких декораторов к одному

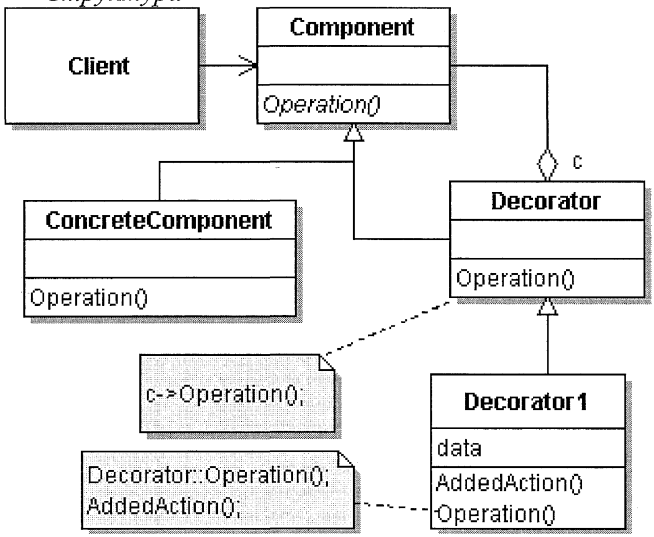
компоненту позволяет произвольным образом сочетать обязанности или добавить одно и то же свойство дважды.

2. Декоратор позволяет добавлять новые обязанности по мере необходимости и избежать перегруженных функциями классов ни верхних уровнях иерархии. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались.

3. Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному.

4. При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

Структура



Реализация

```
// Компонент определяет интерфейс для объектов
class Component {
public:
  virtual void Operation()=0;
};
// Конкретный компонент определяет класс объектов,
// на который возлагаются дополнительные обязанности
```

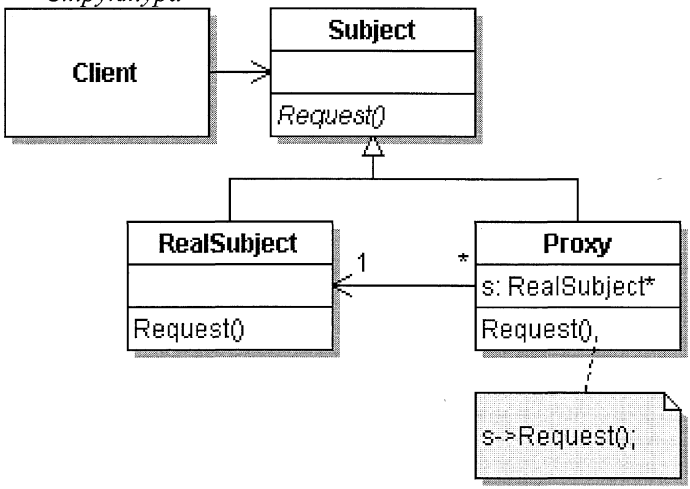
```
class ConcreteComponent: public Component {
public:
  void Operation();
};
// Декоратор хранит указатель на объект Component
// и определяет интерфейс, соответствующий интерфейсу Component
class Decorator: public Component {
protected:
  Component *c;
public:
  void Operation() { c->Operation(); }
};
// Конкретный декоратор добавляет дополнительные
// обязанности компоненту
class Decorator1: public Decorator {
  int data;
public:
  void AddedAction();
  void Operation()
  { Decorator::Operation(); AddedAction(); }
};
```

14.3 Заместитель (Proxy)

Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

Структура



Результаты

1. Удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве.

2. Виртуальный заместитель может выполнять оптимизацию, например, создание объекта по требованию.

3. Защищающий заместитель и "умная" ссылка позволяют решать дополнительные задачи при доступе к объекту, например, контроль прав доступа или автоматическое уничтожение объекта при уменьшении ссылок на объект до 0.

Реализация

```
// Так как Subject определяет общий для RealSubject и Proxy интерфейс,
// класс Proxy можно использовать везде, где ожидается RealSubject
class Subject {
public:
  virtual void Request()=0;
};
// Реальный субъект
class RealSubject : public Subject {
public:
  void Request();
};
// Заместитель
class Proxy: public Subject {
  RealSubject *s;
public:
  void Request() { s->Request(); }
};
```

Умные указатели за счет перегрузки операций обеспечивают интерфейс как у обычного указателя и автоматическое уничтожение созданных объектов.

```
// Умный указатель
class SmartPtr {
  Data *ptr;
public:
  SmartPtr():ptr(0) {}
  SmartPtr(const SmartPtr& p);
  SmartPtr(Data *data);
  SmartPtr& operator=(const SmartPtr&p);
  ~SmartPtr();
  Data *operator->(){ return ptr; }
  Data &operator*() { return *ptr; }
```

```
  friend bool operator==(const SmartPtr &a,
      const SmartPtr &b) { return a.ptr==b.ptr; }
};
// Данные
class Data {
  int count; // счетчик указателей
  int data;  // данные
  friend class SmartPtr;
public:
  Data():count(0),data(1){}
  SmartPtr operator&() { return SmartPtr(this); }
  int getData() const { return data; }
};
SmartPtr::SmartPtr(const SmartPtr& p)
{ if((ptr=p.ptr)!=NULL)
    ++(ptr->count);
}
SmartPtr::SmartPtr(Data *data)
{ ptr=data;
  ++(ptr->count);
}
SmartPtr& SmartPtr::operator=(const SmartPtr&p)
{ SmartPtr t(p);
  std::swap(t.ptr,ptr);
  return *this;
}
SmartPtr::~SmartPtr()
{ if(ptr && --(ptr->count)==0)
    delete ptr;
}
bool operator!=(const SmartPtr &a, const SmartPtr &b)
{ return !(a==b);
}
// Использование
SmartPtr p=new Data;
cout<<p->getData()<<"\n";
```

14.4 Компоновщик (Composite)

Назначение

Компонует объекты в древовидные структуры для представления иерархии часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Результаты

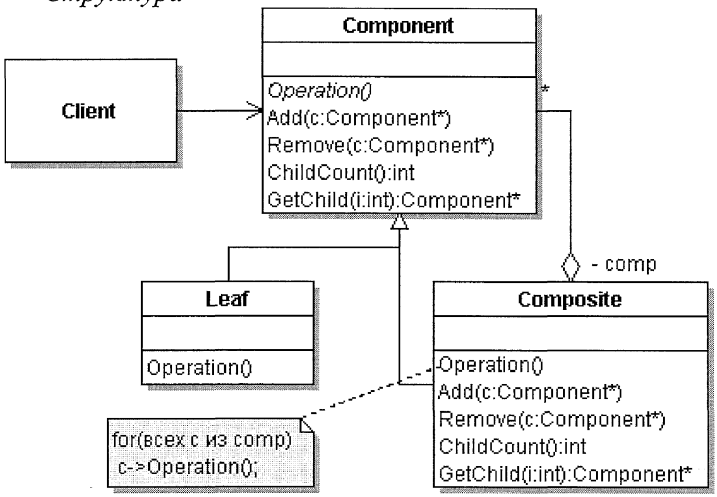
1. Паттерн компоновщик позволяет создавать иерархии классов, состоящие из примитивных и составных объектов. Из примитивных объектов составляются более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее.

2. Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с простым или составным объектом. Использование паттерна упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают.

3. Паттерн компоновщик облегчает добавление новых видов компонентов. Существующие классы и клиенты будут автоматически работать с новыми подклассами.

4. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Проверку этих ограничений нужно проводить во время выполнения, так как паттерн не позволяет проверять ограничения при компиляции.

Структура



Реализация

```
// Компонент определяет интерфейс для всех объектов в иерархии,
// объявляет интерфейс для доступа к потомкам и определяет
// реализацию операций по умолчанию, общую для всех классов
class Component {
public:
```

```
  virtual void Operation()=0;
  virtual void Add(Component *) { throw Error(); }
  virtual void Remove(Component *) {}
  virtual int ChildCount() { return 0; }
  virtual Component *GetChild(int i) { throw Error();
}
};
// Простой компонент
class Leaf : public Component {
public:
  void Operation();
};
// Составной компонент
class Composite : public Component {
  vector<Component *> comp;
public:
  void Operation()
  { for(int i=0;i<comp.size();++i)
      comp[i]->Operation();
    ...
  }
  void Add(Component *c) { comp.push_back(c); }
  void Remove(Component *c)
  { comp.erase(find(comp.begin(),comp.end(),c)); }
  int ChildCount() { return comp.size(); }
  Component *GetChild(int i) { return comp[i]; }
};
```

14.5 Мост (Bridge)

Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

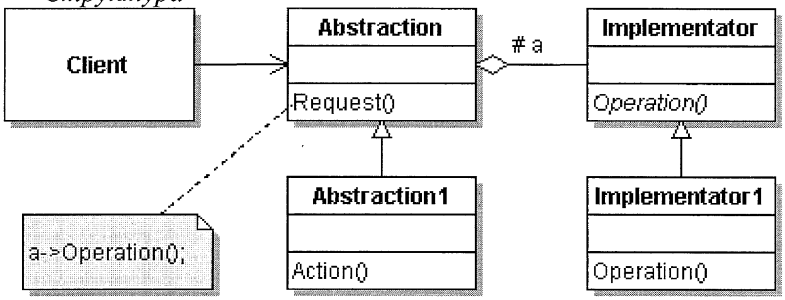
Результаты

1. Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно конфигурировать во время выполнения. Объект может даже динамически изменять свою реализацию.

2. Можно расширять независимо иерархии классов Abstraction и Implementor.

3. Можно скрыть детали реализации от клиентов, используя указатель типа `void *`.

Структура



Реализация

```
// Инструмент определяет интерфейс,
// содержащий примитивные операции
class Implementator {
public:
  virtual void Operation()=0;
};
// Конкретный инструмент определяет реализацию
// примитивных операций
class Implementator1 : public Implementator {
public:
  void Operation();
};
// Абстракция определяет интерфейс и реализацию
// по умолчанию для высокоуровневых операций
class Abstraction {
protected:
  Implementator *a;
public:
  virtual void Request() { a->Operation(); }
};
// Уточненная абстракция расширяет интерфейс
class Abstraction1 : public Abstraction {
public:
  void Action();
};
```

14.6 Приспособленец (Flyweight)

Назначение

Использует разделение для эффективной поддержки множества мелких объектов.

Результаты

При использовании приспособленцев не исключены затраты на передачу, поиск или вычисление внутреннего состояния. Однако такие расходы с лихвой компенсируются экономией памяти за счет разделения объектов-приспособленцев, которая получается из-за:

1) уменьшения общего числа экземпляров;

2) сокращения объема памяти, необходимого для хранения внутреннего состояния;

3) вычисления, а не хранения внешнего состояния.

Чем выше степень разделения приспособленцев, тем существеннее экономия.

Реализация

```
// Приспособленец, объявляет интерфейс, с помощью которого можно
// получать внешнее состояние или как-то воздействовать на него
class Flyweight {
public:
  virtual char geta() const=0; // символ
  virtual int getb() const=0; // размер
};
// Фабрика приспособленцев создает объекты-приспособленцы
// и управляет ими
class FlyweightFactory {
  vector<char> a; // уникальное значение для каждого объекта
  map<int, int> b; // значение одинаково
                   // для нескольких последовательных объектов
  map<int, Flyweight *> fw;
  char geta(int n) { return a[n]; }
  int getb(int n)
  { return (--b.upper_bound(n))->second; }
  void release(int n) { fw.erase(n); }
  friend class ConcreteFlyweight;
  friend class UnsharedFlyweight;
public:
  Flyweight *getFlyweight(int n);
};
// Конкретный приспособленец, хранит состояние внутри
class ConcreteFlyweight {
  char a;
  int b;
  FlyweightFactory *f;
  int n;
```

```
  ConcreteFlyweight(FlyweightFactory *f,
    int n):f(f),n(n) { a=f->geta(n); b=f->getb(n); }
  friend class FlyweightFactory;
public:
  ~ConcreteFlyweight() { f->release(n); }
  char geta() const { return a; }
  int getb() const { return b; }
};
// Неразделяемый конкретный приспособленец
class UnsharedFlyweight {
  FlyweightFactory *f;
  int n;
  UnsharedFlyweight(FlyweightFactory *f,
     int n):f(f),n(n) {}
  friend class FlyweightFactory;
public:
  ~UnsharedFlyweight() { f->release(n); }
  char geta() const { return f->geta(n); }
  int getb() const { return f->getb(n); }
};
Flyweight *FlyweightFactory::getFlyweight(int n)
{ auto f=fw.find(n);
  if(f!=fw.end())
    return f->second;
  return fw[n]=new ConcreteFlyweight(this,n);
}
```

Этот вариант можно использовать, когда не будет удаления и добавления элементов в набор. В случае частых добавлений и удалений с целью повышения эффективности для хранения состояний элементов набора вместо vector и map нужно использовать специальные структуры данных (дерево интервалов, √-декомпозицию).

14.7 Фасад (Facade)

Назначение

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

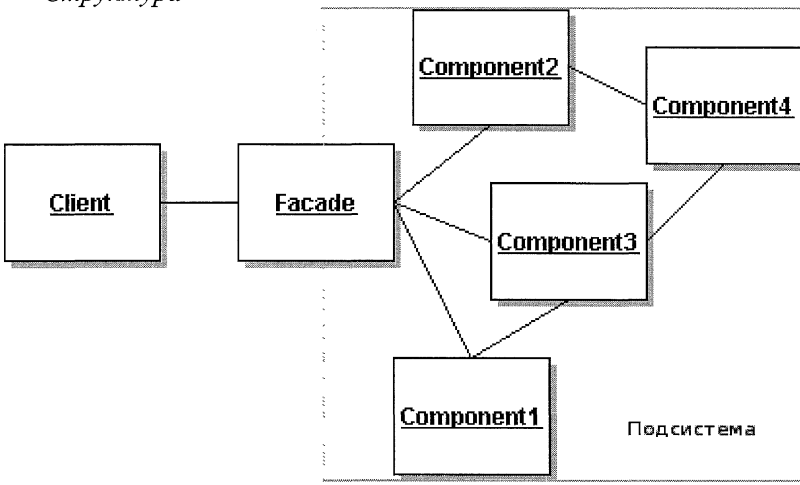
Результаты

1. Паттерн фасад изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой.

2. Зачастую компоненты подсистемы сильно связаны. Ослабление связей между подсистемой и ее клиентами позволяет видоизменять компоненты подсистемы, не затрагивая при этом клиентов. Также сокращение числа зависимостей за счет фасадов может уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. Фасад может упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные.

3. Фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

Структура



Реализация

```
// Компоненты подсистемы реализуют функциональность подсистемы,
// выполняют работу, порученную объектом Facade
class Component1 {
public:
  void Operation1();
} component1;
class Component2 {
public:
  void Operation2();
} component2;
// Фасад делегирует запросы клиентов подходящим
// объектам внутри подсистемы
```

```
class Facade {
public:
  void Operation1() { component1.Operation1(); }
  void Operation2() { component2.Operation2(); }
} facade;
// Использование
facade.Operation1();
facade.Operation2();
```

Контрольные вопросы

1. Укажите назначение следующих изученных паттернов:
- адаптер класса;
- адаптер объекта;
- декоратор;
- заместитель;
- компоновщик;
- мост;
- приспособленец;
- фасад.

2. Какие классы STL реализуют паттерн заместитель?

3. Рассмотрите реализации адаптера объекта и моста, декоратора и заместителя. Почему при похожей реализации они имеют разное назначение?

15 ПАТТЕРНЫ ПОВЕДЕНИЯ

15.1 Интерпретатор (Interpreter)

Назначение

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

Результаты

1. Грамматику легко изменять и расширять.

2. Реализации классов, описывающих узлы абстрактного синтаксического дерева, достаточно тривиальна, их может автоматически создавать генератор синтаксических анализаторов.

3. Сложные грамматики трудно сопровождать, так как определяется по меньшей мере один класс для каждого правила грамматики.

4. Паттерн интерпретатор позволяет легко изменить способ вычисления выражений. При частом добавлении новых способов интерпретации выражений можно использовать паттерн посетитель.

Реализация

```
// Контекст
class Context {
public:
  int curr(); // текущий символ
  void next(); // переход к следующему символу
  int getpos(); // запомнить позицию
  void setpos(int); // восстановить позицию
};
// Исключительная ситуация для ошибки разбора
class SyntaxError {};
// Абстрактное выражение
class Expression {
public:
  virtual void Interpret(Context&)=0;
};
// Терминальное выражение - один символ
class Terminal : public Expression {
  int ch;
public:
  Terminal(int ch):ch(ch) {}
  void Interpret(Context &c)
  { if(c.curr()!=ch) throw SintaxError();
    c.next();
  }
};
// Нетерминальное выражение для правила грамматики R::=R1 R2 ... Rn
class Nonterminal : public Expression {
  vector <Expression *> rule;
public:
  void Add(Expression *e) { rule.push_back(e); }
  void Interpret(Context &c)
  { for(size_t i=0; i<rule.size(); ++i)
      rule[i]->Interpret(c);
  }
};
// Нетерминальное выражение для правила R::=R1 | R2 | ... | Rn
class Select : public Expression {
  vector <Expression *> rule;
public:
  void Add(Expression *e) { rule.push_back(e); }
  void Interpret(Context &c)
```

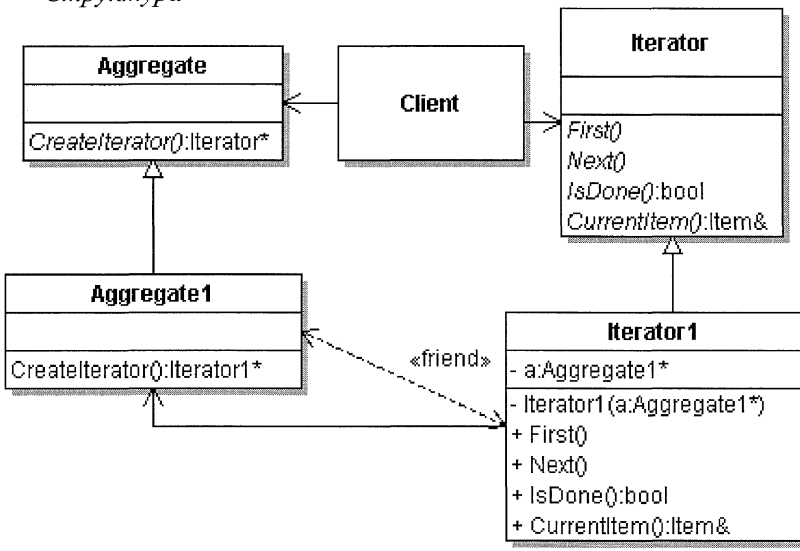
```
  { int p=c.getpos();
    for(size_t i=0; i<rule.size(); ++i)
    { try {
        rule[i]->Interpret(c);
        return;
      }
      catch(SyntaxError &e)
      { c.setpos(p); }
    }
    throw SyntaxError();
  }
};
```

15.2 Итератор (Iterator)

Назначение

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

Структура



Результаты

1. Сложные составные объекты можно обходить по-разному. Для изменения алгоритма обхода нужно определить подкласс класса Iterator и заменить один экземпляр итератора другим.

2. Итераторы упрощают интерфейс класса Aggregate.

3. Одновременно для данного агрегата может быть активно несколько обходов.

4. Модификация составного объекта в то время, как совершается его обход, может оказаться опасной. Для создания устойчивого к модификациям итератора необходимо изменять состояние всех созданных итераторов при вставке или удалении.

5. Если итерацией управляет клиент, то итератор называется внешним, в противном случае – внутренним. В случае внутреннего итератора клиент передает составному объекту некоторую операцию, а объект сам применяет эту операцию к каждому элементу. Внешние итераторы обладают большей гибкостью, чем внутренние. Например, сравнить две коллекции на равенство с помощью внешнего итератора очень легко, а с помощью внутреннего – практически невозможно.

Реализация

```
// Составной объект
class Aggregate {
public:
  virtual Iterator *CreateIterator()=0;
};
// Элементы составного объекта
class Item;
// Конкретный объект реализует интерфейс создания итератора
class Aggregate1 : public Aggregate {
  friend class Iterator1;
  Item *items;
  int size;
public:
  Iterator1 *CreateIterator()
  { return new Iterator1(this); }
};
// Итератор определяет интерфейс для доступа и обхода элементов
class Iterator {
public:
  virtual void First()=0;
  virtual void Next()=0;
  virtual bool IsDone()=0;
  virtual Item &CurrentItem()=0;
};
// Конкретный итератор реализует интерфейс класса Iterator
class Iterator1 {
  friend class Aggregate1;
```

```
  Aggregate1 *a;
  int i;
  Iterator1(Aggregate1 *);
public:
  void First() { i=0; }
  void Next() { ++i; }
  bool IsDone() { return i>=a->size; }
  Item &CurrentItem() { return a->items[i]; }
};
// Использование
Aggregate *a;
Iterator *it=a->CreateIterator();
for(it->First();!it->IsDone();it->Next())
  ... действия с it->CurrentItem() ...
```

В STL для работы с итераторами используется перегрузка операций: для получения текущего элемента – операция разъадресации (*), для перехода на следующий элемент – операция инкремента (++), для проверки завершения – сравнение с со специальным итератором контейнера `end()`.

Внутренний итератор (см. также паттерн посетитель)

```
// Элементы составного объекта
typedef int Item;
// Внутренний итератор
class Iterator {
public:
  virtual void Operation(Item&)=0;
};
// Составной объект
class Aggregate {
public:
  virtual void ApplyIterator(Iterator &)=0;
};
// Конкретный объект реализует интерфейс создания итератора
class Aggregate1 : public Aggregate {
  Item *items;
  int size;
public:
  void ApplyIterator(Iterator &)
  { for(int i=0;i<size;++i)
      Iterator.Operation(items[i]);
  }
};
// Использование
```

```
class SumIterator : public Iterator {
  int sum;
public:
  SumIterator():sum(0) {}
  void Operation(Item &it) { sum+=it; }
  int getSum() const { return sum; }
} sumIt;
...
Aggregate *a;
a->ApplyIterator(sumIt);
cout<<sumIt->getSum();
```

15.3 Команда (Command)

Назначение

Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

Результаты

1. Команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить.

2. Команды – это самые настоящие объекты. Допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами.

3. Из простых команд можно собирать составные, например, макрокоманды, при реализации которых можно использовать паттерн компоновщик.

4. Добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.

Реализация

```
// Команда объявляет интерфейс для выполнения операции
class Command {
public:
  virtual void Execute()=0;
};
// Получатель располагает информацией о способах выполнения
// операций, например, получателем может быть Окно,
// а операцией - закрытие окна
class Receiver {
public:
  void Action();
```

```
};
// Конкретная команда определяет связь между
// объектом-получателем и действием
class Command1 : public Command {
  Receiver *r;
public:
  Command1(Receiver *r):r(r) {}
  void Execute() { r->Action(); }
};
// Инициатор обращается к команде для выполнения запроса
// Например, инициатором может Кнопка, при нажатии
// на которую выполнится заданная команда
class Invoker {
  Command *cmd;
public:
  void AddCommand(Command *c) { cmd=c; }
  void Action() { cmd->execute(); }
};
// Использование
Invoker inv;
Receiver rec;
inv.AddCommand(new Command1(&rec));
...
inv.Action();
```

15.4 Наблюдатель (Observer)

Назначение

Определяет зависимость типа один ко многим между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Результаты

1. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму. Это дает возможность добавлять новые виды наблюдателей без модификации субъекта или других наблюдателей.

2. Для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюда-

телей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его.

3. Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Простая операция над субъектом может вызвать каскад обновлений наблюдателей и зависящих от них объектов. Если протокол обновления не содержит никаких сведений о том, что именно изменилось в субъекте, то наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

Реализация

```
// Наблюдатель определяет интерфейс уведомления
class Observer {
public:
  virtual void Update() {}
};
// Субъект предоставляет интерфейс для работы с наблюдателями
class Subject {
  vector <Observer *> obs;
public:
  void Attach(Observer *o)
  { obs.push_back(o);
  }
  void Detach(Observer *o)
  { obs.erase(find(obs.begin(),obs.end(),o));
  }
  void Notify()
  { for(size_t i=0; i<obs.size(); ++i)
      o->Update();
  }
};
// Конкретный субъект хранит состояние,
// представляющее интерес для наблюдателей
class Subject1 : public Subject {
  int state;
public:
  int GetState() const { return state; }
  void SetState(int s)
  { if(s!=state)
    { state=s;
      Notify();
```

```
    }
  }
};
// Конкретный наблюдатель хранит данные, которые должны
// быть согласованы с данными субъекта
class Observer1 : public Observer {
  Subject1 *s;
  int state;
public:
  Observer(Subject1 *s):s(s) { s->Attach(this); }
  ~Observer() { s->Detach(this); }
  void Update() { state=s->GetState(); }
};
```

15.5 Посетитель (Visitor)

Назначение

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

Результаты

1. С помощью посетителей легко добавлять операции, зависящие от компонентов сложных объектов. Для определения новой операции над структурой объектов достаточно просто ввести нового посетителя.

2. Родственное поведение не разносится по всем классам, присутствующим в структуре объектов, оно локализовано в посетителе. Не связанные друг с другом функции распределяются по отдельным подклассам класса Visitor. Это способствует упрощению как классов, определяющих элементы, так и алгоритмов, инкапсулированных в посетителях. Все относящиеся к алгоритму структуры данных можно скрыть в посетителе.

3. Паттерн посетитель усложняет добавление новых подклассов класса Element. Каждый новый конкретный элемент требует объявления новой абстрактной операции в классе Visitor, которую нужно реализовать в каждом из существующих его подклассов. Поэтому при решении вопроса о том, стоит ли использовать паттерн посетитель, нужно прежде всего посмотреть, что будет изменяться чаще: алгоритм, применяемый к объектам структуры, или классы объектов, составляющих эту структуру. Паттерн посетитель следует применять в случае, если иерархия классов Element стабильна, но постоянно расширяется набор операций или модифицируются алгоритмы.

4. Итератор может посещать объекты структуры по мере ее обхода, вызывая операции объектов. Но итератор не способен работать со структура-

ми, состоящими из объектов разных типов. У посетителя таких ограничений нет. Ему разрешено посещать объекты, не имеющие общего родительского класса.

5. Посетители могут аккумулировать информацию о состоянии при посещении объектов структуры. Если не использовать этот паттерн, состояние придется передавать в виде дополнительных аргументов операций, выполняющих обход, или хранить в глобальных переменных.

6. Применение посетителей подразумевает, что у элементов достаточно развитый интерфейс для того, чтобы посетители могли справиться со своей работой. Поэтому при использовании данного паттерна приходится предоставлять открытые операции для доступа к внутреннему состоянию элементов, что нарушает инкапсуляцию.

Реализация

```
class Visitor;
// Элемент определяет операцию Accept для приема посетителя
class Element {
public:
  virtual void Accept(Visitor &)=0;
};
// Конкретные элементы реализуют операцию Accept
class ElementA : public Element {
public:
  void Accept(Visitor &v) { v.Visit(*this); }
  void OperationA();
};
class ElementB : public Element {
public:
  void Accept(Visitor &v) { v.Visit(*this); }
  void OperationB();
};
// Посетитель объявляет операцию Visit для каждого подкласса Element
class Visitor {
public:
  virtual void Visit(ElementA &) {}
  virtual void Visit(ElementB &) {}
};
// Конкретный посетитель реализует нее операции,
// объявленные в классе Visitor, может содержать поля для
// накопления результатов в процессе обхода структуры
class Visitor1 : public Visitor {
public:
```

```
  void Visit(ElementA &e) {... e->OperationA(); ... }
  void Visit(ElementB &e) {... e->OperationB(); ... }
};
// Сложный объект предоставляет посетителю интерфейс
// для посещения своих элементов
class Object {
  Element *els;
  int n;
public:
  void Accept(Visitor &v)
  { for(int i=0;i<n;++i)
      els[i]->Accept(v);
  }
};
```

15.6 Посредник (Mediator)

Назначение

Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

Результаты

1. Посредник локализует поведение, которое в противном случае пришлось бы распределять между несколькими объектами. Для изменения поведения нужно породить подклассы только от класса посредника Mediator, классы коллег Colleague можно использовать повторно без каких бы то ни было изменений.

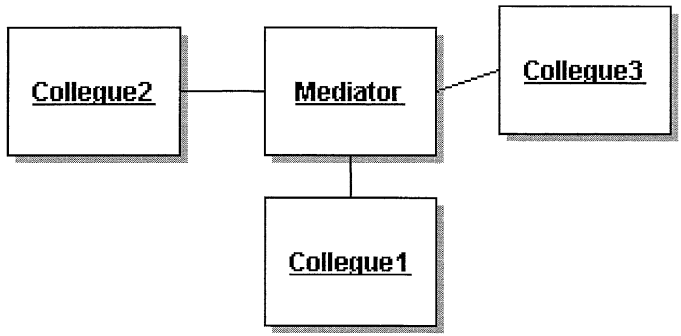
2. Посредник обеспечивает слабую связанность коллег. Можно изменять классы Colleague и Mediator независимо друг от друга.

3. Посредник заменяет способ взаимодействия «все со всеми» способом «один со всеми», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения.

4. Выделение механизма посредничества в отдельную концепцию и инкапсуляция ее в одном объекте позволяет сосредоточиться именно на взаимодействии объектов, а не на их индивидуальном поведении. Это дает возможность прояснить имеющиеся в системе взаимодействия.

5. Паттерн посредник переносит сложность взаимодействия в класс-посредник. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник становится монолитом, который трудно сопровождать.

Структура



Реализация

```
// Посредник реализует кооперативное поведение,
// координируя действия коллег
class Mediator {
public:
  void Operation1();
  void Operation2();
};
// Коллеги знают о своем посреднике и обмениваются
// информацией только через него
class Colleague {
protected:
  Mediator *m;
public:
  Colleague(Mediator *m):m(m) {}
};
class Colleague1 : public Colleague {
public:
  void Action1() { m->Operation1(); }
};
class Colleague2 : public Colleague {
public:
  void Action2() { m->Operation2(); }
};
```

15.7 Состояние (State)

Назначение

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Результаты

1. Паттерн состояние помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку зависящий от состояния код целиком находится в одном из подклассов класса State, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов. Вместо этого можно было бы использовать данные-члены для определения внутренних состояний, тогда операции объекта Context проверяли бы эти данные. Но в таком случае похожие условные операторы или операторы ветвления были бы разбросаны по всему коду класса Context. При этом добавление нового состояния потребовало бы изменения нескольких операций, что затруднило бы сопровождение. Паттерн состояние позволяет решить эту проблему, но одновременно порождает другую, поскольку поведение для различных состояний оказывается распределенным между несколькими подклассами State. Это увеличивает число классов. Конечно, один класс компактнее, но если состояний много, то такое распределение эффективнее, так как в противном случае пришлось бы иметь дело с громоздкими условными операторами. Наличие громоздких условных операторе в нежелательно, равно как и наличие длинных процедур. Они слишком монолитны, вот почему модификация и расширение кода становится проблемой. Паттерн состояние предлагает более удачный способ структурирования зависящего от состояния кода. Логика, описывающая переходы между состояниями, больше не заключена в монолитные операторы if или switch, а распределена между подклассами state. При инкапсуляции каждого перехода и действия в класс состояние становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение.

2. Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными. Кроме того, объекты State могут защитить контекст Context от рассогласования внутренних переменных, поскольку переходы с точки зрения контекста – это атомарные действия. Для осуществления перехода надо изменить значение только одной переменной (объектной переменной State в классе Context), а не нескольких.

3. Если в объекте состояния State отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект State. Когда состояния разделяются таким образом, они являются, по сути дела,

приспособленцами (см. описание паттерна приспособленец), у которых нет внутреннего состояния, а есть только поведение.

Реализация

```
// Состояние определяет интерфейс для инкапсуляции
// поведения, ассоциированного с конкретным состоянием
class State {
public:
  virtual void Action() {}
};
// Каждый подкласс реализует поведение,
// ассоциированное с некоторым состоянием
class State1 : public State {
public:
  void Action();
};
class State2 : public State {
public:
  void Action();
};
// Контекст определяет интерфейс,
// представляющий интерес для клиентов
class Context {
  State *state;
public:
  void Request() { state->Action(); }
};
```

15.8 Стратегия (Strategy)

Назначение

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Результаты

1. Иерархия классов Strategy определяет семейство алгоритмов или поведений, которые можно повторно использовать в разных контекстах. Наследование позволяет вычленить общую для всех алгоритмов функциональность.

2. Наследование поддерживает многообразие алгоритмов или поведений. Можно напрямую породить от Context подклассы с различными поведениями. Но при этом поведение жестко «зашивается» в класс Context. Вот почему реализации алгоритма и контекста смешиваются, что затруд-

няет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся. В результате вы получите множество родственных классов, отличающихся только алгоритмом или поведением. Инкапсуляции алгоритма в отдельный класс Strategy позволяют изменять его независимо от контекста.

3. Благодаря паттерну стратегия удается отказаться от условных операторов при выборе нужного поведения. Когда различные поведения помещаются в один класс, трудно выбрать нужное без применения условных операторов. Инкапсуляция же каждого поведения в отдельный класс Strategy решает эту проблему. Если код содержит много условных операторов, то часто это признак того, что нужно применить паттерн стратегия.

4. Стратегии могут предлагать различные реализации одного и того же поведения. Клиент вправе выбирать подходящую стратегию в зависимости от своих требований к быстродействию и памяти.

5. Потенциальный недостаток этого паттерна в том, что для выбора подходящей стратегии клиент должен понимать, чем отличаются разные стратегии. Поэтому наверняка придется раскрыть клиенту некоторые особенности реализации. Отсюда следует, что паттерн стратегия стоит применять лишь тогда, когда различия в поведении имеют значение для клиента.

6. Интерфейс класса Strategy разделяется всеми его подклассами – неважно, сложна или тривиальна их реализация. Поэтому вполне вероятно, что некоторые стратегии не будут пользоваться всей передаваемой им информацией, особенно простые. Это означает, что в отдельных случаях контекст создаст и проинициализирует параметры, которые никому не нужны. Если возникнет проблема, то между классами Strategy и Context придется установить более тесную связь.

7. Применение стратегий увеличивает число объектов в приложении. Иногда эти издержки можно сократить, если реализовать стратегии в виде объектов без состояния, которые могут разделяться несколькими контекстами. Остаточное состояние хранится в самом контексте и передается при каждом обращении к объекту-стратегии. Разделяемые стратегии не должны сохранять состояние между вызовами. В описании паттерна приспособленец этот подход обсуждается более подробно.

Реализация

```
class Context;
// Стратегия объявляет общий для всех поддерживаемых
// алгоритмов интерфейс
class Strategy {
public:
  virtual void Interface(Context *)=0;
```

```
};
// Конкретная стратегия реализует алгоритм
class Strategy1 : public Strategy {
public:
  void Interface(Context *);
};
// Контекст хранит все необходимые алгоритму данные
// и указатель на объект класса Strategy
class Context {
  Strategy *st;
public:
  Context(Strategy *st): st(st) {}
  void Operation() { st->Interface(this); }
};
```

15.9 Хранитель (Memento)

Назначение

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нём объект.

Результаты

1. Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн экранирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции.

2. При других вариантах дизайна, направленного на сохранение границ инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине. При перекладывании заботы о запрошенном состоянии на клиентов упрощается структура хозяина, а клиентам дается возможность не информировать хозяина о том, что они закончили работу.

3. С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой объем информации для занесения в память хранителя или если клиенты создают и возвращают хранителей достаточно часто. Если плата за инкапсуляцию и восстановление состояния хозяина велика, то этот паттерн не всегда подходит.

4. В некоторых языках сложно гарантировать, что только хозяин имеет доступ к состоянию хранителя.

Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

Реализация

```
// Хранитель сохраняет внутреннее состояние объекта Originator
// и запрещает доступ всем другим объектам, кроме хозяина
class Memento {
  int state;
  Memento(int s):state(s) {}
  friend class Originator;
public:
};
// Хозяин создает хранитель и использует его для восстановления
// внутреннего состояния
class Originator {
  int state;
public:
  Memento *CreateMemento()
  { return new Memento(state); }
  void SetMemento(Memento *m) { state=m->state; }
};
// Смотритель отвечает за сохранение хранителя
class Caretaker {
  Memento *m;
  Originator *orig;
public:
  void Action()
  { m=orig->CreateMemento();
    ...
  }
  void Undo()
  { orig->SetMemento(m);
  }
};
```

15.10 Цепочка обязанностей (Chain Of Responsibility)

Назначение

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Результаты

1. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неиз-

Результаты

1. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту – о структуре цепочки. Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике.

2. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков.

3. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

Реализация

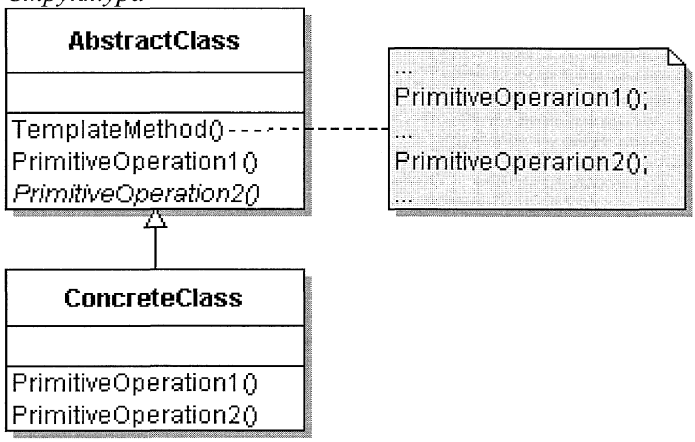
```
// Обработчик определяет интерфейс для обработки запросов
class Handler {
protected:
  Handler *next;
public:
  virtual void HandleRequest()=0;
};
class Handler1 : public Handler {
public:
  void HandleRequest()
  { // если конкретный обработчик способен обработать запрос,
    // то так и делает
     ...
     // иначе направляет его своему преемнику
     next->HandleRequest();
  }
};
// Использование
Handler *first;
first->HandleRequest();
```

15.11 Шаблонный метод (Template Method)

Назначение

Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шага алгоритма, не изменяя его структуру в целом.

Структура



Результаты

Шаблонные методы – один из фундаментальных приемов повторного использования кода. Они особенно важны в библиотеках классов, поскольку предоставляют возможность вынести общее поведение в библиотечные классы. Шаблонные методы приводят к инвертированной структуре кода, которую иногда называют принципом Голливуда, подразумевая часто употребляемую в этой киноимперии фразу «Не звоните нам, мы сами позвоним». В данном случае это означает, что родительский класс вызывает операции подкласса, а не наоборот.

Шаблонные методы вызывают операции следующих видов:

1) конкретные операции из производных классов;
2) конкретные операции из базового класса, то есть операции, полезные всем подклассам;
3) примитивные операции, то есть абстрактные операции;
4) фабричные методы (см. паттерн фабричный метод);
5) операции-зацепки (hook operations), реализующие поведение по умолчанию, которое может быть расширено в подклассах. Часто такая операция по умолчанию не делает ничего.

Важно, чтобы в шаблонном методе четко различались операции-зацепки, которые *можно* замещать, и абстрактные операции, которые *нужно* замещать. Чтобы повторно использовать абстрактный класс с максимальной

эффективностью, авторы подклассов должны понимать, какие операции предназначены для замещения. Подкласс может расширить поведение некоторой операции, заместив ее и явно вызвав эту операцию из родительского класса.

Реализация

```
// Абстрактный класс определяет абстрактные примитивные
// операции, замещаемые в конкретных подклассах
// для реализации шагов алгоритма и реализует
// шаблонный метод, определяющий скелет алгоритма
class AbstractClass {
public:
  void TemplateMethod()
  { ...
    PrimitiveOperarion1();
    ...
    PrimitiveOperarion2();
    ...
  }
  virtual void PrimitiveOperation1() {}
  virtual void PrimitiveOperation2()=0;
};
// Конкретный класс реализует примитивные операции
class ConcreteClass: public AbstractClass {
public:
  void PrimitiveOperation1();
  void PrimitiveOperation2();
};
```

Контрольные вопросы

1. Укажите назначение следующих изученных паттернов:
   - интерпретатор;
   - итератор;
   - команда;
   - наблюдатель;
   - посетитель;
   - посредник;
   - состояние;
   - стратегия;
   - хранитель;
   - цепочка обязанностей;
   - шаблонный метод.
2. Как реализован паттерн итератор в STL?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Дьюхерст С. Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ / С. Дьюхерст. — М.: ДМК Пресс, 2006. — 264 с.

2. Мейерс С. Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов / С. Майерс. — М.: ДМК, 2000. — 235 с.

3. Мейерс С. Эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / С. Майерс. — М.: ДМК, 2006. — 296 с.

4. Мейерс С. Эффективное использование STL. Библиотека программиста / С. Майерс. — СПб.: Питер, 2002. — 224 с.

5. Мейерс С. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14 / С. Майерс. — М.: Вильямс, 2016. — 304 с.

6. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч и др. — М.: Вильямс, 2010. — 720 с.

7. Приемы объектно-ориентированного проектирования: Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. — СПб. и др.: Питер, 2016. — 366 с.

8. Саттер Г. Решение сложных задач на C++ / Г. Саттер. — М.: Вильямс, 2002. — 400 с.

9. Страуструп, Б. Язык программирования C++: Специальное издание / Б. Страуструп. — М. : Бином-Пресс, 2008. — 1098 с.

ОГЛАВЛЕНИЕ

ЧАСТЬ 1 ЯЗЫК С++

Учебное издание

Демидов Андрей Константинович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ НА C++

Учебное пособие

Техн. редактор *А.В. Миних*

Издательский центр Южно-Уральского государственного университета

Подписано в печать 02.03.2017. Формат 60×84 1/16. Печать цифровая.
Усл. печ. л. 9,30. Тираж 30 экз. Заказ 47/612.

Отпечатано с оригинал-макета заказчика
в типографии Издательского центра ЮУрГУ.
454080, г. Челябинск, проспект Ленина, 76.