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

004.4(07)
Р159

Г.И. Радченко, Е.А. Захаров

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Конспект лекций

Челябинск
Издательский центр ЮУрГУ
2013

УДК 004.4(075.8)
Р159

Одобрено
учебно-методической комиссией
факультета вычислительной математики и информатики.

Конспект лекций подготовлен в соответствии с ФГОС ВПО 3-го поколения по образовательным направлениям 010300.62 «Фундаментальная информатика и информационные технологии» и 010400.62 «Прикладная математика и информатика».

Рецензенты:
доктор физ.-мат. наук, профессор В.И. Ухоботов,
кандидат технических наук А.В. Созыкин

Р159

Радченко, Г.И.

Объектно-ориентированное программирование / Г.И. Радченко, Е.А. Захаров. – Челябинск: Издательский центр ЮУрГУ, 2013. – 167 с.

В учебном пособии представлены основы применения объектно-ориентированного программирования. Рассмотрены основные концепции объектно-ориентированного программирования на примере языка программирования C++. Рассматриваются понятия класса и объекта, концепция наследования, шаблоны функций и классов, методы перегрузки операторов, методы обработки исключительных ситуаций. Приводится обзор основных порождающих, структурных и поведенческих паттернов проектирования. Рассматриваются особенности использования стандартной библиотеки шаблонов (STL) в C++.

Пособие предназначено для студентов бакалавриата по направлению 010300 «Фундаментальная информатика и информационные технологии» при изучении курса «Объектно-ориентированное програмирование», а также для обучения студентов бакалавриата по направлению 010400 «Прикладная математика и информатика».

УДК 004.4(075.8)

1. ВВЕДЕНИЕ

1.1 Сложность разработки программного обеспечения

Мы окружены сложными системами:

– персональный компьютер;

– любое дерево, цветок, животное;

– любая материя – от атома до звезд и галактик;

– общественные институты – корпорации и сообщества.

Большинство сложных систем обладает иерархической структурой. Но не все ПО – сложное. Существует класс приложений которые проектируются разрабатываются и используются одним и тем же человеком. Но они имеют ограниченную область применения. Вопросы сложности появляются при разработке корпоративного ПО, промышленного программирования.

Сложность ПО вызывается четырьмя основными причинами:

– сложностью реальной предметной области, из которой исходит заказ на разработку;

– трудностью управления проектированием;

– необходимостью обеспечить достаточную гибкость программы;

– сложность описания поведения больших дискретных систем.

1.2 Декомпозиция

Декомпозиция – один из способов борьбы со сложностью.

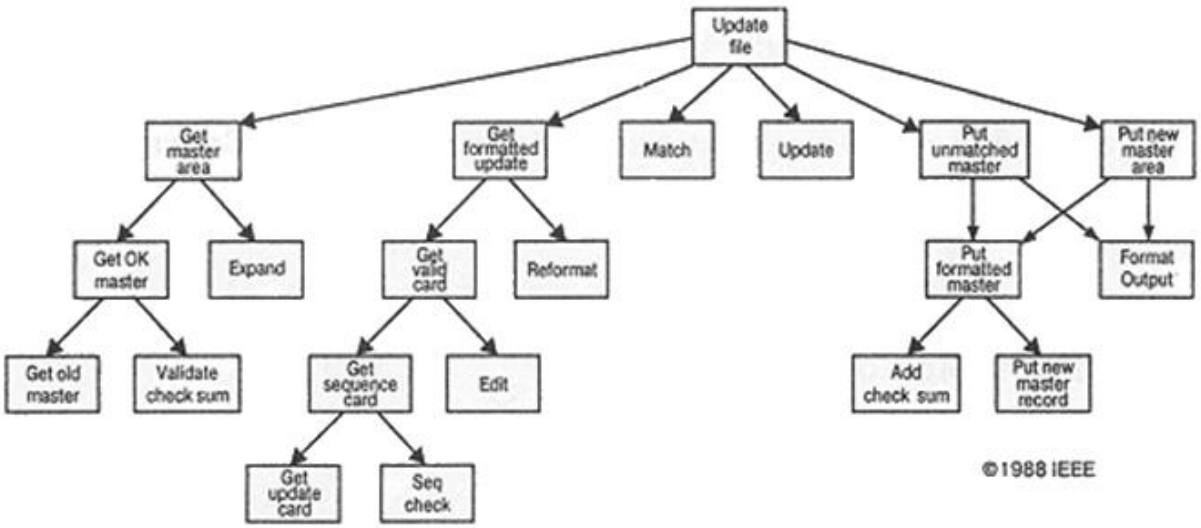

Рис. 1. Пример алгоритмической декомпозиции

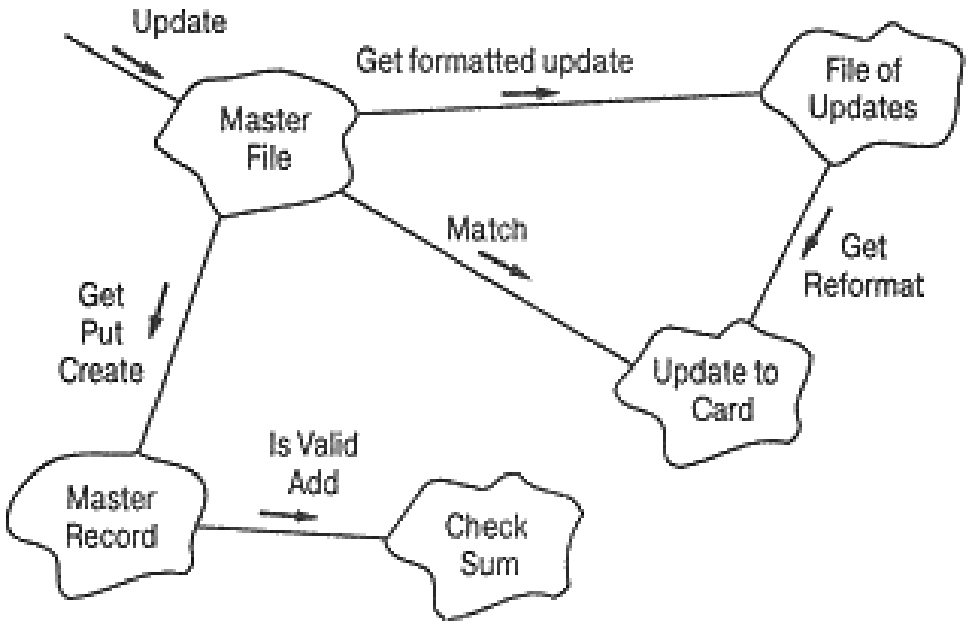

Рис. 2. Пример объектно-ориентированной декомпозиции

Необходимо разделять систему на независимые подсистемы, каждую из которых разрабатывать отдельно. Выделяют следующие методы декомпозиции:

- алгоритмическая декомпозиция (см. рис. 1);
- объектно-ориентированная декомпозиция (см. рис. 2).

1.3 Краткая история языков программирования

Выделяют следующие этапы развития языков программирования высокого уровня:

- Языки первого поколения (1954–1958)
  - FORTRAN 1 Математические формулы
  - ALGOL-58 Математические формулы
- Языки второго поколения (1959–1961)
  - FORTRAN II Подпрограммы
  - ALGOL-60 Блочная структура, типы данных
  - COBOL Описание данных, работа с файлами
  - LISP Обработка списков, указатели, сборка мусора
- Языки третьего поколения (1962–1970)
  - PL/I FORTRAN+ALGOL+COBOL
  - Pascal Простой наследник ALGOL-60
  - Simula Классы, абстракция данных
- Разрыв преемственности (1970–1980)

- C Эффективный высокоуровневый язык
- FORTRAN 77 Блочная структура, типы данных

– Бум ООП (1980–1990)
- Smalltalk 80 Чисто объектно-ориентированный язык
- C++ C + Simula
- Ada83 Строгая типизация; сильное влияние Pascal

– Появление инфраструктур (1990–…)
- Java Блочная структура, типы данных
- Python Объектно-ориентированный язык сценариев
- Visual C# Конкурент языка Java для среды Microsoft .NET

1-е поколение (рис. 3) преимущественно использовалось для научных и технических вычислений, математический словарь. Языки освобождали от сложностей ассемблера, что позволяло отступать от технических деталей реализации компьютеров.

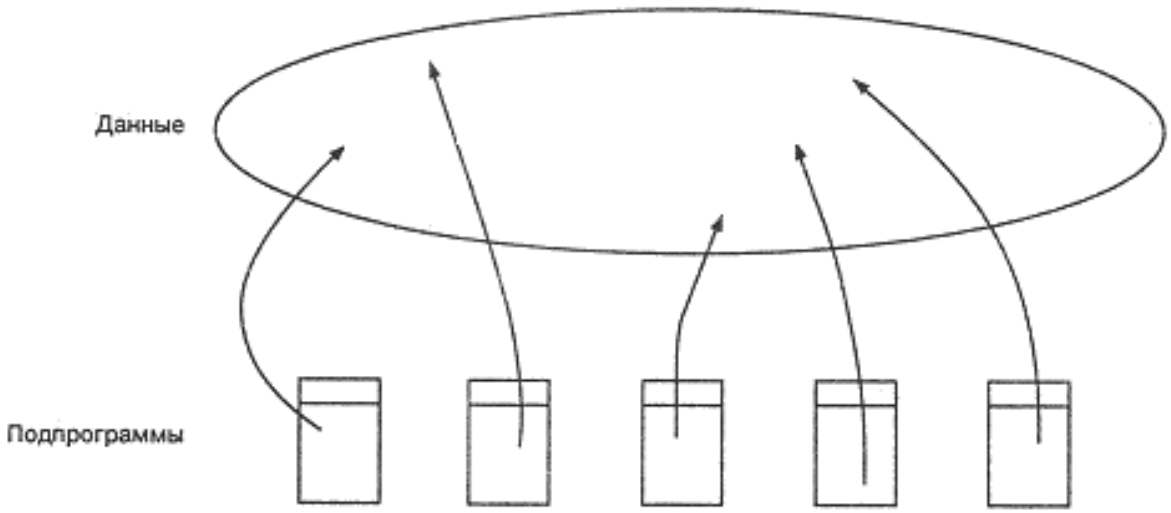

Рис. 3. Топология языков первого поколения

Программы, написанные на языках программирования первого поколения, имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм.

Языки *2-го поколения* (рис. 4) сделали акцент на алгоритмических абстракциях, что приблизило разработчиков к предметной области. Появилась процедурная абстракция, которая позволила описать абстрактные программные функции в виде подпрограмм.

На *3-е поколение* языков программирования (рис. 5) влияние оказало то, что стоимость аппаратного обеспечения резко упала, при этом, производительность экспоненциально росла. Языки поддерживали абстракцию данных и появилась возможность описывать свои собственные типы данных

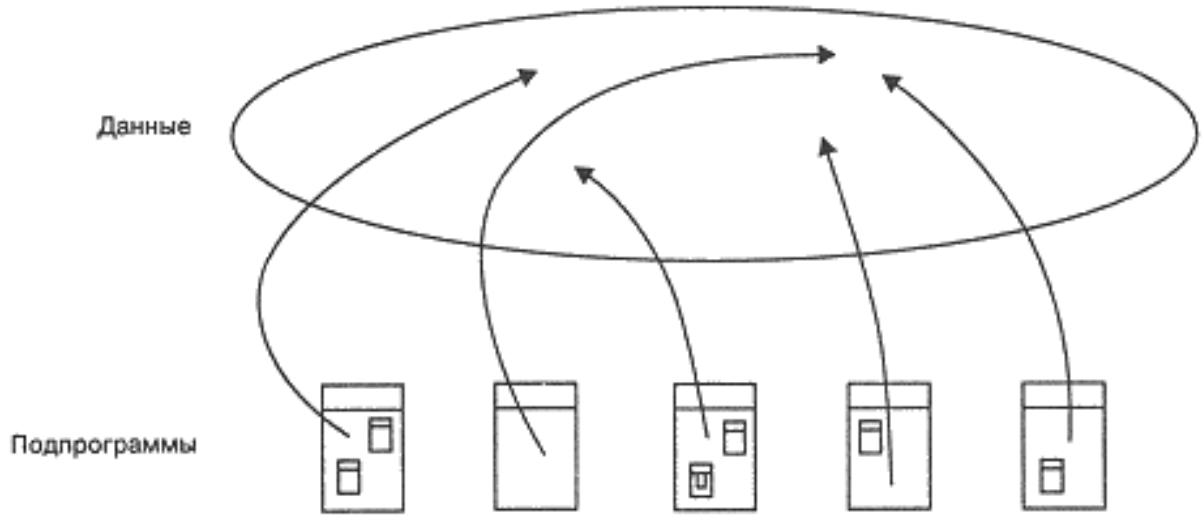

Рис. 4. Топология языков второго поколения

(структуры). В 1970-е годы было создано несколько тысяч языков программирования для решения конкретных задач, но практически все из них исчезли. Осталось только несколько известных сейчас языков, которые прошли проверку временем.

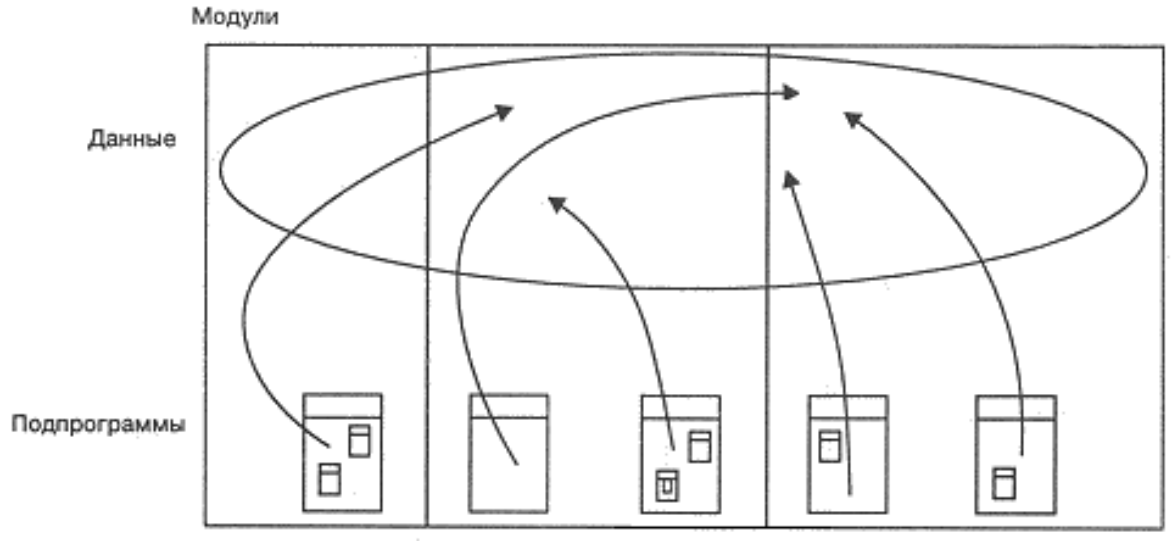

Рис. 5. Топология языков третьего поколения

В модули собирали подпрограммы, которые будут изменяться совместно, но их не рассматривали как новую технику абстракции.

В 1980-е произошел бум развития объектно-ориентированного программирования (рис. 6). Языки данного времени лучше всего поддерживают объектно-ориентированную декомпозицию ПО. В 90-х появились инфраструктуры (J2EE, .NET), предоставляющие огромные объемы интегрированных сервисов.

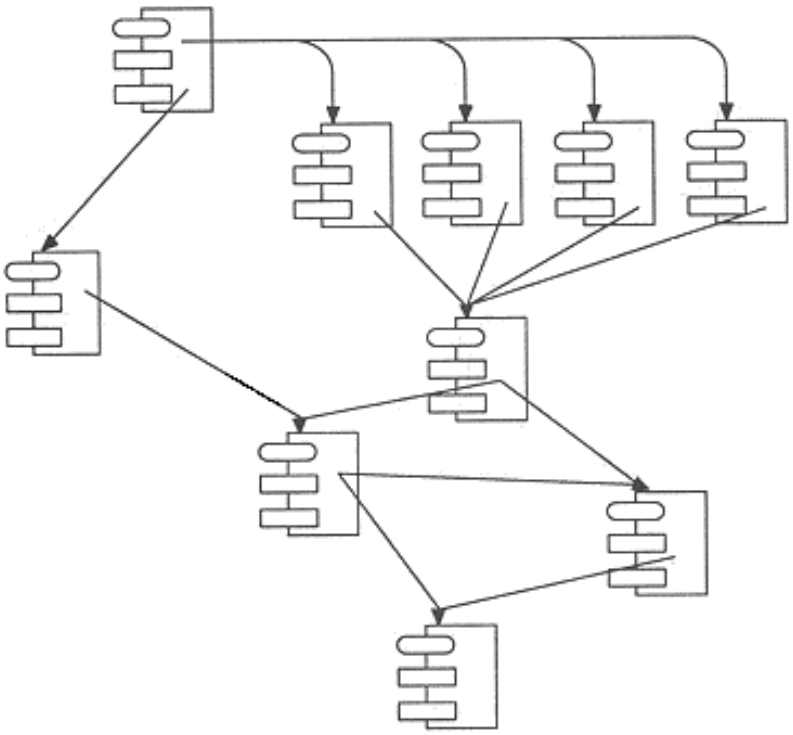

Рис. 6. Топология объектно-ориентированного программирования

Основным элементом конструкции служит модуль, составленный из логически связанных классов и объектов, а не подпрограмма.

1.4 Объектно-ориентированное программирование

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

Объект – это нечто, имеющее четко определенные границы. Однако, этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

Класс – это множество объектов, обладающих общей структурой, поведением и семантикой. Отдельный объект – это экземпляр класса. Класс представляет лишь абстракцию существенных свойств объекта.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств. Например: торговый автомат имеет свой-

ство: способность принимать монеты; этому свойству соответствует динамическое значение – количество принятых монет. Пример описания состояния объекта:

```
struct PersonnelRecord {
   char name[100];
   int socialSecurityNumber;
   char department[10];
   float salary;
};
```

Поведение объекта – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений. Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции `append()` и `pop()` для того, чтобы управлять объектом-очередью:

```
class Queue {
public:
   Queue();
   Queue(const Queue&);
   virtual ~Queue();
   virtual Queue& operator=(const Queue&);
   virtual int operator==(const Queue&) const;
   int operator!=(const Queue&) const;
   virtual void clear();
   virtual void append(const void*);
   virtual void remove(int at);
   virtual int length() const;
   virtual int isEmpty() const;
   ...
};
```

Индивидуальность объекта – это такое свойство объекта, которое отличает его от всех других объектов. В большинстве языков программирования при создании объект именуется, поэтому многие путают адресуемость и индивидуальность. Невозможность отличить имя объекта от самого объекта является источником множества ошибок в ООП.

2. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ЯЗЫК C++

Объектно-ориентированное программирование строится на трех основополагающих принципах: инкапсуляция, полиморфизм и наследование.

2.1 Инкапсуляция

Инкапсуляция – 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.

Пусть члену класса требуется защита от «несанкционированного доступа». Как разумно ограничить множество функций, которым такой член будет доступен? Очевидный ответ для языков, поддерживающих объектно-ориентированное программирование, таков: доступ имеют все операции, которые определены для этого объекта, иными словами, все функции-члены. Например:

```
class window
{
   // ...
protected:
   Rectangle inside;
   // ...
};

class dumb_terminal: public window
{
   // ...
public:
   void prompt();
   // ...
};
```

Здесь в базовом классе `window` член `inside` типа `Rectangle` описывается как защищенный (`protected`), но функции-члены производных классов, например, `dumb_terminal::prompt()`, могут обратиться к нему и выяснить, с какого вида окном они работают. Для всех других функций член `window::inside` недоступен.

В таком подходе сочетается высокая степень защищенности с гибкостью, необходимой для программ, которые создают классы и используют их.

Неочевидное следствие из этого: нельзя составить полный и окончательный список всех функций, которым будет доступен защищенный член, поскольку всегда можно добавить еще одну, определив ее как функцию-член в новом производном классе. Для метода абстракции данных такой подход часто бывает мало приемлемым. Если язык ориентируется на метод абстракции данных, то очевидное для него решение – это требование указывать в описании класса список всех функций, которым нужен доступ к члену. В C++ для этой цели используется описание частных (`private`) членов.

Важность инкапсуляции, т.е. заключения членов в защитную оболочку, резко возрастает с ростом размеров программы и увеличивающимся разбросом областей приложения.

2.2 Наследование

Наследование представляет собой способность производить новый класс из существующего базового класса. Производный класс – это новый класс, а базовый класс – существующий класс. Когда вы порождаете один класс из другого (базового класса), производный класс наследует элементы базового класса. Для порождения класса из базового начинайте определение производного класса ключевым словом `class`, за которым следует имя класса, двоеточие и имя базового класса, например `class dalmatian: dog`.

Когда вы порождаете класс из базового класса, производный класс может обращаться к общим элементам базового класса, как будто эти элементы определены внутри самого производного класса. Для доступа к частным данным базового класса производный класс должен использовать интерфейсные функции базового класса.

Внутри конструктора производного класса ваша программа должна вызвать конструктор базового класса, указывая двоеточие, имя конструктора базового класса и соответствующие параметры сразу же после заголовка конструктора производного класса.

Чтобы обеспечить производным классам прямой доступ к определенным элементам базового класса, в то же время защищая эти элементы от оставшейся части программы, C++ обеспечивает защищенные (`protected`) элементы класса. Производный класс может обращаться к защищенным элементам базового класса, как будто они являются общими. Однако для оставшейся части программы защищенные элементы эквивалентны частным.

Если в производном и базовом классе есть элементы с одинаковым именем, то внутри функций производного класса C++ будет использовать эле-

менты производного класса. Если функциям производного класса необходимо обратиться к элементу базового класса, вы должны использовать оператор глобального разрешения, например `base class:: member`.

2.3 Полиморфизм

Полиморфный объект представляет собой такой объект, который может изменять форму во время выполнения программы.

В объектно-ориентированных языках класс является абстрактным типом данных. Полиморфизм реализуется с помощью наследования классов и виртуальных функций. Класс-потомок наследует сигнатуры методов класса-родителя, а реализация, в результате переопределения метода, этих методов может быть другой, соответствующей специфике класса-потомка. Другие функции могут работать с объектом класса-родителя, но при этом вместо него во время исполнения будет подставляться один из классов-потомков. Это называется поздним связыванием.

Класс-потомок сам может быть родителем. Это позволяет строить сложные схемы наследования – древовидные или сетевидные.

Абстрактные (или чисто виртуальные) методы не имеют реализации вообще. Они специально предназначены для наследования. Их реализация должна быть определена в классах-потомках.

Класс может наследовать функциональность от нескольких классов. Это называется множественным наследованием. Множественное наследование создаёт проблему (когда класс наследуется от нескольких классов-посредников, которые в свою очередь наследуются от одного класса (так называемая «Проблема ромба»): если метод общего предка был переопределён в посредниках, неизвестно, какую реализацию метода должен наследовать общий потомок. Решается эта проблема через виртуальное наследование.

2.4 История появления C++

В 1980 году Бьерн Страуструп в AT&T Bell Labs стал разрабатывать расширение языка C под условным названием C++. Стиль ведения разработки вполне соответствовал духу, в котором создавался и сам язык C, – в него вводились те или иные возможности с целью сделать более удобной работу конкретных людей и групп. Первый коммерческий транслятор нового языка, получившего название C++ появился в 1983 году. Он представлял собой препроцессор, транслировавший программу в код на C. Однако факти-

ческим рождением языка можно считать выход в 1985 году книги Страуструпа. Именно с этого момента C++ начинает набирать всемирную популярность.

Главное нововведение C++ – механизм классов, дающий возможность определять и использовать новые типы данных. Программист описывает внутреннее представление объекта класса и набор функций-методов для доступа к этому представлению. Одной из заветных целей при создании C++ было стремление увеличить процент повторного использования уже написанного кода. Концепция классов предлагала для этого механизм наследования. Наследование позволяет создавать новые (производные) классы с расширенным представлением и модифицированными методами, не затрагивая при этом скомпилированный код исходных (базовых) классов. Вместе с тем наследование обеспечивает один из механизмов реализации полиморфизма базовой концепции объектно-ориентированного программирования, согласно которой, для выполнения однотипной обработки разных типов данных может использоваться один и тот же код. Собственно, полиморфизм тоже один из методов обеспечения повторного использования кода.

Введение классов не исчерпывает всех новаций языка C++. В нем реализованы полноценный механизм структурной обработки исключений, отсутствие которого в C значительно затрудняло написание надежных программ, механизм шаблонов – изощренный механизм макрогенерации, глубоко встроенный в язык, открывающий еще один путь к повторной используемости кода, и многое другое.

Таким образом, генеральная линия развития языка была направлена на расширение его возможностей путем введения новых высокоуровневых конструкций при сохранении сколь возможно полной совместимости с ANSI C. Конечно, борьба за повышение уровня языка шла и на втором фронте: те же классы позволяют при грамотном подходе упрятывать низкоуровневые операции, так что программист фактически перестает непосредственно работать с памятью и системно-зависимыми сущностями.

На данный момент существуют следующие стандарты языка C++:

- ANSI C++ / ISO-C++ – 1996 год,
- ISO/IEC 14882:1998 – 1998 год,
- ISO/IEC 14882:2003 – 2003 год,
- C++/CLI – 2005 год,
- TR1 – 2005 год,
- C++11 – 2011 год.

2.5 Нововведения и отличия C++ от C

В C++ появились классы и объекты. Технически, класс C++ – это тип структуры в C, а объект – переменная такого типа. Разница только в том, что в C++ есть еще модификаторы доступа и полями могут быть не только данные, но и функции (функции-методы).

В C++ появились две новые операции: `new` и `delete`. В первую очередь это – сокращения для распространенных вызовов функций `malloc` и `free`:

При вызове `new` автоматически вызывается конструктор, а при вызове `delete` – деструктор. Так что нововведение можно описать формулой: `new` = `malloc` + конструктор, `delete` = `free` + деструктор.

В C++ появились функции, которые вызываются автоматически после создания переменной структуры (конструкторы) и перед ее уничтожением (деструкторы). Во всех остальных отношениях это – обычные функции, на которые наложен ряд ограничений. Некоторые из этих ограничений ничем не оправданы и мешают: например, конструктор нельзя вызвать напрямую (деструктор, к счастью, можно). Нельзя вернуть из конструктора или деструктора значение. Что особенно неприятно для конструктора. А деструктору нельзя задать параметры.

При программировании на C часто бывает так, что имеется несколько вариантов одной и той же структуры, для которых есть аналогичные функции. Например, есть структура, описывающая точку (`Point`) и структура, описывающая окружность (`Circle`). Для них обоих часто приходится выполнять операцию рисования (`Point`). Так что, если у нас есть блок данных, где перемешаны точки, окружности и прочие графические примитивы, то перед нами стоит задача быстро вызвать для каждого из них свою функцию рисования.

Обычное решение – построить таблицу соответствия «вариант структуры – функция». Затем берется очередной примитив, определяется его тип, и по таблице вызывается нужная функция. В C++ всем этим занимается компилятор: достаточно обозначить функцию-метод как `virtual`, и для всех одноименных функций будет создана таблица и поле типа, за которыми следить будет также компилятор. При попытке вызвать функцию с таким именем, будет вызвана одна из серии одноименных функций в зависимости от типа структуры.

Исключение по своей сути – это последовательность `goto` и `return`. Основано на C-технологии `setjmp`/`longjmp`. `try` и `catch` – это `setjmp` с проверкой. `throw` – это `longjmp`. Когда вызывается `throw`, то проверяется: если

он окажется внутри блока `try`, то выполняется `goto` на парный блок `catch`. Если нет, то делается `return` и ищется `catch` на уровень выше и так далее.

2.5.1 Аргументы по-умолчанию

Аргумент по-умолчанию – это то, о чем мечтали программисты С: чтобы иногда не надо было при вызове задавать некоторые параметры, которые в этом случае должны иметь некоторое «обычное» значение:

```
void f(int x, int y=5, int z=10);
void g(int x=5, int y); /* Неправильно! По умолчанию задаются только
                                                    последние аргументы
*/
f(1); // будет вызвано f(1, 5, 10)
f(1, 2); // будет вызвано f(1, 2, 10)
f(1, 2, 3); // будет вызвано f(1, 2, 3)
```

Желание программистов С контролировать типы параметров в `define`-ах породило в C++ `inline`-функции. Такая функция – это обычный `define` с параметрами, но без использования символов «\» и с проверкой типов.

Желание узаконить в параметрах `define` имя типа породило `template`. Главный плюс `template` – то, что `#define` с одинаковыми параметрами породит два одинаковых куска кода. А `template` в компиляторе скорее всего будет с оптимизирован: одинаковые куски кода будут соединены в один. Имеется больший контроль типов по сравнению с `#define`.

2.5.2 Ссылки и указатели

В C++ ссылка – это простой ссылочный тип, менее мощный, но более безопасный, чем указатель, унаследованый от языка Си. Если мы объявили переменную, то для хранения значений выделяется память. Чтобы изменить или прочитать значение переменной (то есть значение находящейся в этой области памяти), мы обращаемся по имени этой переменной. В языке Си имя сущности (переменной, типа, функции и т.д.) – это идентификатор. С точки зрения программиста, объявляя ссылку (или же указывая, что она будет возвращаемым значением или аргументом функции), мы задаём альтернативный идентификатор для уже созданного объекта. В языке Си ссылок нет. С точки зрения реализации, ссылка – это, по сути, указатель, который жестко привязан к области памяти, на которую он указывает, и который автоматически разыменовывается, когда мы обращаемся по имени ссылки. Например:

```
int a;       // переменная типа int, размещеннная по адресу 0xbfd86d6c
int &ra = a; // альтернативное имя для переменной по адресу 0xbfd86d6c
```

```
cout << &a << "\n" << &ra << "\n";
```

Приведенному выше коду будет соответствовать следующий вывод:

```
0xbfd86d6c
0xbfd86d6c
```

То есть оба имени `a` и `ra` привязаны к одному и тому же адресу.

Ссылки нельзя объявлять без привязки к переменной (то есть не инициализировав при объявлении). После объявления ссылки её невозможно привязать к другой переменной.

Важно отличать ссылки от оператора взятия адреса `&` (address of). Оператор взятия адреса используется для уже созданного объекта с целью получить его адрес (то есть адрес области памяти, где хранятся значения), а ссылка это только задание альтернативного имени объекта (с точки зрения программиста, а не реализации). Например:

```
int a = 3;        // переменная типа int размещена по адресу 0xbfd86d6c

int *p = &a; /* указатель типа int* с именем "p" по адресу 0xbf971c4c,
значение этого указателя - адрес объекта с именем "a" - 0xbfd86d6c (это
значение можно будет менять): */

p = &b;
```

Отличие указателя от ссылки в том, что получить само значение переменной, на которую указывает указатель, можно только выполнив операцию разыменовывания `*` (символ «`*`» в объявлении является объявлением указателя, а при применении к уже созданной переменной является оператором разыменовывания). Например:

```
int a = 3;
int *p = &a; // объявили, создали, и инициализировали объект

cout << *p << '\n'; /* здесь к уже созданному объекту с именем "p"
применяется оператор "*", который означает "считать значение из "p",
которое является адресом и далее считать данные по этому адресу" */
```

Итого есть два оператора: `*` и `&`. Первый по данному адресу, который хранится в переменной типа `int*`, возвращает собственно данные, расположенные по этому адресу. Второй по данной переменной узнаёт её адрес в памяти.

2.6 Ключевое слово static

В C/C++ имеется пять применений ключевого слова `static`.

```
static int i = 0;
static void foo() {}
```

Глобальная переменная, объявленная с ключевым словом `static`, будет иметь `internal linkage`, т.е. она будет объявлена глобальной только в рамках одной единицы трансляции (чаще всего, такой единицей трансляции является файл). Таким образом использование `static` помогает избавиться от ошибок линковки из-за того что в разных объектах трансляции были объявлены глобальные переменные с одинаковыми именами.

```
void foo() {
   static int i = 0;
}
```

Заметим, что наличие глобальных переменных скорее всего свидетельствует об ошибках проектирования. В крайнем случае следует использовать синглтоны (см. главу 12 «Паттерны проектирования»).

Глобальная функция, объявленная с ключевым словом `static`, будет также иметь `internal linkage`. Наличие глобальных функций об ошибках проектирования не свидетельствует, как правило.

```
void foo() {
    static int i = 0;
}
```

Локальная переменная, объявленная с ключевым словом `static`, будет иметь локальную область видимости и время жизни – от инициализации до завершения программы. Таким образом, состояние статической переменной сохраняется между вызовами функции. Инициализация локальной статической переменной будет происходить в тот момент когда выполнение программы пройдёт через строчку с объявлением переменной. Если конструктор локальной статической переменной выбросит исключение (которое будет обработано), то при следующем прохождении этой строчки будет также выполнена попытка инициализации переменной. Если инициализация статической локальной переменной прошла успешно, инициализации более происходить не будет. По-умолчанию, статические переменные POD-типов инициализируются нулями.

```
class MyClass {
      static void foo();
      static int i;
};

int MyClass::i = 0;
void MyClass::foo() { }
```

Атрибут класса, объявленный с ключевым словом `static`, будет иметь глобальную область видимости (через класс, разумеется) и время жизни – от инициализации до завершения программы. Инициализация статических атрибутов происходит так же как и глобальных переменных: в глобальной области видимости объявляется тип переменной, затем её имя (с указанием класса в котором она содержится), и, опционально, инициализатором, например: `int MyClass::variable_ = 5;` По-умолчанию, статические переменные-члены также будут инициализированы нулями.

Метод класса, объявленный с ключевым словом `static`, будет иметь глобальную область видимости. В отличие от других функций-членов, статический метод не будет получать указатель `T * this` на текущий объект (см. п. 3.6) и соответственно не может быть объявлен со спецификаторами `const` или `virtual`, по этой же причине статические методы не имеют прямого доступа к нестатическим полям класса.

Каждый нестатический метод, помимо явно объявленных параметров, получает еще один скрытый параметр: константный указатель на объект, для которого он вызван. В C++ это указатель обозначается зарезервированным словом `this`. Когда имя параметра метода совпадает с именем поля класса, доступ к полю выполняется через этот указатель (например, `this->num`).

2.7 Ключевое слово const

Есть две точки зрения на использование `const`:

`const` – это плохо. От него больше хлопот, чем пользы, ошибки какие-то странные вылезать начинают, лучше им не пользоваться.

`const` – это хорошо. `const` не дает менять объекты, которые не должны меняться, таким образом оберегает от ошибок, его надо использовать везде где только можно.

В английской литературе можно часто встретить термины const correctness и const correct code, для кода, который корректно использует `const`. `const` имеет немного разный смысл в зависимости от того где находится.

2.7.1 Объявление переменных

Самый простой случай, обычная переменная. Переменная объявляется, тут же инициализируется, менять ее значение больше нельзя:

```
const int p=4;
p=5; //ошибка
```

Про использование const с указателями есть известный C++ паззл, который любят давать на собеседованиях при приеме на работу. Чем отличаются:

```
int *const p1
int const* p2
const int* p3
```

Правило тут такое: провести мысленно вертикальную черту по звездочке. То, что находится справа относится к переменной. То, что слева – к типу, на который она указывает. Вот например:

```
int *const p1
```

Справа находится p1, и это p1 константа. Тип, на который p1 указывает, это int. Значит получился константный указатель на int. Его можно инициализировать лишь однажды и больше менять нельзя. Нужно так:

```
int q=1;
int *const p1 = &q; //инициализация в момент объявления
*p1 = 5;            //само число можно менять
```

Вот так компилятор не пропустит, потому что идет попытка присвоения константе:

```
int q=1;
int *const p1;
p1 = &q;        //ошибка
```

Следующие объявления – это по разному записанное одно и то же объявление. Указатель на целое, которое нельзя менять.

```
int const* p2
const int* p3
```

Обычно в реальных программах используется вариант объявления const int, а int const используется, чтобы запутать на собеседовании.

```
int q=1;
const int *p;
p = &q; //на что указывает p можно менять
*p = 5; //ошибка, число менять уже нельзя
```

const можно использовать со ссылками, чтобы через ссылку нельзя было поменять значение переменной.

```
int p = 4;
const int& x = p; //нельзя через x поменять значение p
x = 5; //ошибка
```

Константная ссылка (например, `int& const x`) – это нонсенс. Она по определению константная. Компилятор скорее всего выдаст предупреждение, что он проигнорировал `const`.

2.7.2 Передача параметров в функцию

`const` удобен, если нужно передать параметры в функцию, но при этом надо обязательно знать, что переданный параметр не будет изменен:

```
void f1(const std::string& s);
void f2(const std::string* sptr);
void f3(std::string s);
```

В первой и второй функции попытки изменить строку будут пойманы на этапе компиляции. В третьем случае в функции будет происходить работа с локальной копией строки, исходная строка не пострадает.

Приведение `Foo**` к `const Foo**` приводит к ошибке потому что такое приведение может позволить менять объекты, которые константны.

```
class Foo {
   ...
public:
   void modify(); //вносит какие-либо изменения
};

int main() {
   const Foo x;
   Foo* p;
   const Foo** q = &p; // q теперь указывает на p; и это ошибка
   *q = &x; // p теперь указывает на x
   p->modify(); // попытка изменить const Foo!!
}
```

Самый простой способ это исправить – это поменять `const Foo**` на `const Foo* const*`.

2.7.3 const данные классов

Значения `const` данных класса задаются один раз и навсегда в конструкторе.

```
class CFoo
{
   const int num;
public:
   CFoo(int anum);
};

CFoo::CFoo(int anum) : num(anum)
{
   ...
}
```

Интересный момент со `static const` данными класса. Вообще для данных целого типа (`enum`, `int`, `char`) их значения можно задавать прямо в объявлении класса. Следующий код правильный с точки зрения стандарта:

```
class CFoo
{
public:
   static const int num = 50;
};
```

Но в Visual C++ 6.0 такое задание значения не работает, это один из багов Visual C++ 6.0. Тут задавать значение `static const` переменной следует отдельно. Вместо того, чтобы запоминать, когда можно при объявлении писать инициализацию, когда нельзя, лучше сразу написать так:

```
class CFoo
{
public:
   static const int num;
};
const int CFoo::num = 20;
```

2.7.4 const функции классов

Функция класса, объявленная `const`, трактует `this` как указатель на константу. Вообще тип `this` в методе класса `X` будет `X*`. Но если метод класса объявлен как `const`, то тип `this` будет `const X*`. В таких методах не может быть ничего присвоено переменным класса, которые не объявлены как `static` или как `mutable`. Также `const`-функции не могут возвращать не `const` ссылки и указатели на данные класса и не могут вызывать не `const` функции класса. `const`-функции иногда называют инспекторами (inspector), а остальные мутаторами (mutator).

```
class CFoo
{
public:
   int inspect() const; // Эта функция обещает не менять *this
   int mutate();        // Эта функция может менять *this
};
```

В классе могут присутствовать две функции отличающиеся только `const`:

```
class CFoo
{
   ...
public:
   int func () const;
   int func ();
};
```

Не всякая функция может быть объявлена константной. Конструкторы и деструкторы не могут быть объявлены как `const`. Также не бывает `static const` функций.

```
class CFoo
{
   int i;
public:
   static int func () const; //ошибка
};
```

Официально такого понятия как константный класс (`const class`) не существует. Но часто под этим понимается объявление вида `const CFoo p;`. Экземпляр класса `CFoo`, объявленный таким образом, обещает сохранить физическое состояние класса, не менять его. Как следствие, он не может вызвать не `const` функции класса `CFoo`. Все данные, не объявленные как `const`, начинают трактоваться как `const`. Например:

```
int становится int const
int * становится int * const
const int * становится int const *const
```

Единственный способ инициализировать константные поля, поля-ссылки и вызвать конструкторы родительских классов с определёнными параметрами – список инициализации. Список инициализации отделяется от прототипа конструктора двоеточием и состоит из инициализаторов разделённых запятыми. Например он может выглядеть так:

```
struct A {
   A(int){ }
};
struct B: A {
   const int c_;
   unsigned d_;
   unsigned& r_;
   B(): A(5), c_(4), r_(d_)
   {
      d_ = 5;
   }
};
```

Отметим, что всегда первыми будут вызваны конструкторы родительских классов, а затем уже произойдёт инициализация членов класса, в порядке их объявления в классе. Т.е. порядок полей в списке инициализации на порядок инициализации влияния иметь не будет.

2.7.5 Несколько фактов о const

Ключевое слово `const` перед объявлением массива или указателя относится к элементам массива, а не самой переменной-массиву. Т.е. `const int* p;` указывает на элемент типа `const int` и его значение нельзя изменять. При этом ничто не запрещает изменять значение самого указателя, если хотите это запретить – напишите `const` после «звёздочки».

Любой тип `T` приводим к типу `const T`, массивы из элементов таких типов также приводимы, так что не стоит волноваться из-за того что у вас указатель на строку типа `char*`, а функция принимает в себя аргумент типа `const char*`. Вообще слова `const` и `static` в объявлениях функций следует расставлять строго до тех пор пока программа не прекратит компилироваться. Я ещё ни разу не видел, чтобы компилирующаяся программа переставала правильно работать от расстановки `const` и `static` (на нормальных компиляторах). *Винт закручивается следующим образом: до срыва, затем пол-оборота назад.*

Ключевое слово `const` перед структурой или классом, по сути, добавляет ключевое слово `const` ко всем его полям. Исключение составляют поля-ссылки, поля объявленные с ключевым словом `mutable` и статические поля. Т.е., для примера выше, следующий код будет успешно скомпилирован (и, по идее, изменит значение поля `d_`):

```
const B b;
b.r_ = 7;
```

Константные методы отличаются от неконстантных лишь тем что указатель `this` имеет тип не `T* const`, а `const T* const` со всеми вытекающими отсюда последствиями. Неконстантные методы не могут быть вызваны у объектов являющимися константными.

3. КЛАССЫ В ООП

3.1 Что такое класс?

В окончательном виде любая программа представляет собой набор инструкций процессора. Все, что написано на любом языке программирования – более удобная, упрощенная запись этого набора инструкций, облегчающая написание, отладку и последующую модификацию программы. Чем выше уровень языка, тем в более простой форме записываются одни и те же действия.

С ростом объема программы становится невозможным удерживать в памяти все детали, и становится необходимым структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс называется повышением степени абстракции программы.

Для языка высокого уровня первым шагом к повышению абстракции является использование функций, позволяющее после написания и отладки функции отвлечься от деталей ее реализации, поскольку для вызова функции требуется знать только ее интерфейс. Если глобальные переменные не используются, интерфейс полностью определяется заголовком функции.

Следующий шаг – описание собственных типов данных, позволяющих структурировать и группировать информацию, представляя ее в более естественном виде. Например, все разнородные сведения, относящиеся к одному виду товара на складе, можно представить с помощью одной структуры.

Для работы с собственными типами данных требуются специальные функции. Естественно сгруппировать их с описанием этих типов данных в одном месте программы, а также по возможности отделить от ее остальных частей. При этом для использования этих типов и функций не требуется полного знания того, как именно они написаны – необходимы только описания интерфейсов. Объединение в модули описаний типов данных и функций, предназначенных для работы с ними, со скрытием от пользователя модуля несущественных деталей является дальнейшим развитием структуризации программы.

Все три описанных выше метода повышения абстракции преследуют цель упростить структуру программы, то есть представить ее в виде меньшего количества более крупных блоков и минимизировать связи между ними. Это позволяет управлять большим объемом информации и, следовательно, успешно отлаживать более сложные программы.

Введение понятия класса является естественным развитием идей модульности. В классе структуры данных и функции их обработки объединяются. Класс используется только через его интерфейс – детали реализации для пользователя класса не существенны.

Идея классов отражает строение объектов реального мира – ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением. Программы в основном предназначены для моделирования предметов, процессов и явлений реального мира, поэтому удобно иметь в языке программирования адекватный инструмент для представления моделей.

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы (напомню, что тип задает внутреннее представление данных в памяти компьютера, множество значений, которое могут принимать величины этого типа, а также операции и функции, применяемые к этим величинам).

Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом. Интерфейсом класса являются заголовки его открытых методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

Идея классов является основой объектно-ориентированного программирования (ООП). Основные принципы ООП были разработаны еще в языках Simula-67 и Smalltalk, но в то время не получили широкого применения из-за трудностей освоения и низкой эффективности реализации. В C++ эти концепции реализованы эффективно и непротиворечиво, что и явилось основой успешного распространения этого языка и внедрения подобных средств в другие языки программирования.

Идеи ООП не очень просты для практического использования (их неграмотное применение приносит гораздо больше вреда, чем пользы), а освоение существующих стандартных библиотек требует времени и высокого уровня первоначальной подготовки.

Конкретные переменные типа данных «класс» называются экземплярами класса, или объектами. Объекты взаимодействуют между собой, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реали-

зуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП легко реализуется так называемая «событийно-управляемая модель», когда данные активны и управляют вызовом того или иного фрагмента программного кода.

Примером реализации событийно-управляемой модели может служить любая программа, управляемая с помощью меню. После запуска такая программа пассивно ожидает действий пользователя и должна уметь правильно отреагировать на любое из них. Событийная модель является противоположностью традиционной (директивной), когда код управляет данными: программа после старта предлагает пользователю выполнить некоторые действия (ввести данные, выбрать режим) в соответствии с жестко заданным алгоритмом.

Класс – это описание определяемого типа. Любой тип данных представляет собой множество значений и набор действий, которые разрешается выполнять с этими значениями. Например, сами по себе числа не представляют интереса – нужно иметь возможность ими оперировать: складывать, вычитать, вычислять квадратный корень и т. д. В C++ множество значений нового типа определяется задаваемой в классе структурой данных, а действия с объектами нового типа реализуются в виде функций и перегруженных операций C++.

Данные класса называются полями (по аналогии с полями структуры), а функции класса – методами. Поля и методы называются элементами класса. Описание класса в первом приближении выглядит так:

```
class <имя> {
[ private: ]
      <описание скрытых элементов>
  public:
      <описание доступных элементов>
};// Описание заканчивается точкой с запятой
```

3.2 Спецификаторы public, private, protected

Спецификаторы доступа `private` и `public` управляют видимостью элементов класса. Элементы, описанные после служебного слова `private`, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса описывается после спецификатора `public`. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций `private` и `public`, порядок их следования значения не имеет.

Поля класса:

– могут быть простыми переменными любого типа, указателями, массивами и ссылками (т.е. могут иметь практически любой тип, кроме типа этого же класса, но могут быть указателями или ссылками на этот класс);

– могут быть константами (описаны с модификатором `const`), при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;

– могут быть описаны с модификатором `static`, но не как `auto`, `extern` и `register`.

Инициализация полей при описании не допускается.

3.3 Глобальные и локальные классы

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, внутри функции или внутри другого класса). Обычно классы определяются глобально.

Локальные классы имеют некоторые особенности:

– локальный класс не может иметь статических элементов;

– внутри локального класса можно использовать из охватывающей его области типы, статические (`static`) и внешние (`extern`) переменные, внешние функции и элементы перечислений;

– запрещается использовать автоматические переменные из охватывающей класс области;

– методы локальных классов могут быть только встроенными (`inline`);

– если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

3.4 Примеры классов

В качестве примера создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество щупалец, силу или наличие гранатомета) и поведение. Естественно, пример будет схематичен, поскольку приводится лишь для демонстрации синтаксиса.

```
class monster {
  int health, ammo;
public:
  monster(int he = 100, int am  = 10) { health = he; ammo = am;}
  void draw(int x, int y, int scale, int position);
  int get_health(){return health;}
  int get_ammo(){return ammo;}
};
```

В этом классе два скрытых поля – `health` и `ammo`, получить значения которых извне можно с помощью методов `get_health()` и `get_ammo()`. Доступ к полям с помощью методов в данном случае кажется искусственным усложнением, но надо учитывать, что полями реальных классов могут быть сложные динамические структуры, и получение значений их элементов не так тривиально. Кроме того, очень важной является возможность вносить в эти структуры изменения, не затрагивая интерфейс класса.

Методы класса имеют неограниченный непосредственный доступ к его полям. Внутри метода можно объявлять объекты, указатели и ссылки как своего, так и других классов.

В приведенном классе содержится три определения методов и одно объявление (метод `draw`). Если тело метода определено внутри класса, он является встроенным (`inline`). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости:

```
void monster::draw(int x, int y, int scale, int position)
  { /* тело метода */}
```

Встроенные методы можно определить и вне класса с помощью директивы `inline` (как и для обычных функций, она носит рекомендательный характер):

```
inline int monster::get_ammo()
  { return ammo; }
```

Методы можно перегружать (это одно из проявлений полиморфизма), а также объявлять либо константными, либо статическими (но не одновременно).

В каждом классе есть метод, имя которого совпадает с именем класса. Он называется конструктором и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных. Подробнее конструкторы описываются далее в разделе «Конструкторы».

Типы данных `struct` и `union` являются специальными видами класса.

Конкретные переменные типа данных «класс» называются экземплярами класса, или объектами. Время жизни и видимость объектов зависит от вида и места описания и подчиняется общим правилам C++:

```
monster Vasia;          // Объект класса monster с параметрами по
умолчанию
```

```
monster Super(200, 300);// Объект с явной инициализацией
monster stado[100];     // Массив объектов с параметрами по умолчанию
/* Динамический объект (второй параметр задается по умолчанию) */
monster *beavis = new monster (10);
monster &butthead = Vasia;// Ссылка на объект
```

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор (деструкторы описаны далее).

Доступ к открытым (`public`) элементам объекта аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция `->` при обращении через указатель:

```
объект.поле
указатель -> поле
(*указатель).поле
объект.метод( параметры )
указатель -> метод( параметры )
(*указатель).метод( параметры )
```

Обращение к открытому полю и вызов метода для массива объектов:

```
имя_массива[ индекс ].поле
имя_массива[ индекс ].метод( параметры )
```

Например:

```
int n = Vasia.get_ammo();
stado[5].draw;
cout << beavis->get_health();
```

Получить или изменить значения `private` элементов можно только через обращение к соответствующим методам. Можно создать константный объект, значения полей которого изменять запрещается. К нему должны применяться только константные методы:

```
class monster {
   ...
   int get_health() const { return health; }
};
const monster Dead (0,0); // Константный объект
cout << Dead.get_health();
```

Константный метод:

- объявляется с ключевым словом `const` после списка параметров;
- не может изменять значения полей класса;
- может вызывать только константные методы;

– может вызываться для любых (не только константных) объектов.

Рекомендуется описывать как константные те методы, которые предназначены для получения значений полей.

3.5 Inline функции

Встроенная функция – это функция, код которой прямо вставляется в том месте, где она вызвана. Как и макросы, определенные через `#define`, встроенные функции улучшают производительность за счет стоимости вызова и (особенно!) за счет возможности дополнительной оптимизации («процедурная интеграция»).

В обычном C вы можете получить «инкапсулированные структуры», помещая в них указатель на `void`, и заставляя его указывать на настоящие данные, тип которых неизвестен пользователям структуры. Таким образом, пользователи не знают, как интерпретировать эти данные, а функции доступа преобразуют указатель на `void` к нужному скрытому типу. Так достигается некоторый уровень инкапсуляции.

К сожалению, этот метод идет вразрез с безопасностью типов, а также требует вызова функции для доступа к любым полям структуры (если вы позволили бы прямой доступ, то его мог бы получить кто угодно, поскольку будет известно, как интерпретировать данные, на которые указывает `void*`. Такое поведение со стороны пользователя приведет к сложностям при последующем изменении структуры подлежащих данных).

Стоимость вызова функции невелика, но дает некоторую прибавку. Классы C++ позволяют встраивание функций, что дает вам безопасность инкапсуляции вместе со скоростью прямого доступа. Более того, типы параметры встраиваемых функций проверяются компилятором, что является преимуществом по сравнению с `#define` макросами.

В отличие от `#define` макросов, встроенные (`inline`) функции не подвержены известным ошибкам двойного вычисления, поскольку каждый аргумент встроенной функции вычисляется только один раз. Другими словами, вызов встроенной функции – это то же самое что и вызов обычной функции, только быстрее.

Также, в отличие от макросов, типы аргументов встроенных функций проверяются, и выполняются все необходимые преобразования.

3.6 Указатель this

Каждый объект содержит свой экземпляр полей класса. Методы места в классе не занимают и не дублируются для каждого объекта. Единственный

экземпляр метода используется всеми объектами совместно, поэтому каждый нестатический метод класса должен «знать», для какого объекта он вызван. Для этого, при вызове каждого нестатического метода класса, ему неявно передается указатель на объект, вызвавший его `T * const this`.

Выражение `*this` представляет собой разыменование указателя и имеет тип определяемого класса. Обычно это выражение возвращается в качестве результата, если метод возвращает ссылку на свой класс (`return *this;`).

Для иллюстрации использования указателя `this` добавим в приведенный выше класс `monster` новый метод, возвращающий ссылку на наиболее здорового (поле `health`) из двух монстров, один из которых вызывает метод, а другой передается ему в качестве параметра (метод нужно поместить в секцию `public` описания класса):

```
monster & the_best(monster &M)
{
   if( health > M.get_health())
      return *this;
   return M;
}
...
monster Vasia(50), Super(200);
// Новый объект Best инициализируется значениями полей Super
monster Best = Vasia.the_best(Super);
```

4. КОНСТРУКТОРЫ КЛАССОВ

4.1 Конструкторы и их свойства

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов.

1. Конструктор не возвращает значения, даже типа `void`. Нельзя получить указатель на конструктор.

2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).

3. Конструктор, который можно вызвать без параметров, называется конструктором по умолчанию.

4. Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

5. Если программист не указал ни одного конструктора, компилятор создает его автоматически (кроме случая, когда класс содержит константы и ссылки, поскольку их необходимо инициализировать). Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы базовых классов.

6. Конструкторы не наследуются.

7. Конструктор не может быть константным, статическим и виртуальным (нельзя использовать модификаторы `const`, `virtual` и `static`).

8. Конструкторы глобальных объектов вызываются до вызова функции `main`. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

При объявлении объектов вызывается один из конструкторов. При отсутствии инициализирующего выражения в объявлении объекта вызывается конструктор по умолчанию, при инициализации другим объектом того же типа – конструктор копирования (см. далее), при инициализации полей – один из явно определенных конструкторов инициализации (т.е. конструкторов, которым передаются параметры для инициализации полей объекта).

4.2 Конструкторы по умолчанию

Конструкторы часто вызываются неявно для создания временных объектов. Обычно это происходит в следующих случаях:

– при инициализации;

– при выполнении операции присваивания;

– для задания значений параметров по умолчанию;

– при создании и инициализации массива;

– при создании динамических объектов;

– при передаче параметров в функцию и возврате результатов по значению.

Примеры конструкторов:

```
monster Super(200, 300), Vasia(50);
monster X = monster(1000);
```

В последнем операторе создается объект `X`, которому присваивается безымянный объект со значением параметра `health = 1000` (значения остальных параметров устанавливаются по умолчанию).

При создании динамического массива вызывается конструктор без аргументов.

В качестве примера класса с несколькими конструкторами усовершенствуем описанный ранее класс `monster`, добавив в него поля, задающие цвет (`skin`) и имя (`name`):

```
enum color {red, green, blue};      //    Возможные значения цвета
class monster
{
  int health, ammo;
  color skin;
  char *name;
public:
      monster(int he = 100, int am  = 10);
      monster(color sk);
      monster(char * nam);
...
};
//--------------------------------
monster::monster(int he, int am)
    { health = he; ammo = am; skin = red; name = 0;}
//--------------------------------
monster::monster(color sk)
{
   switch (sk)
      {
         case red: health = 100; ammo = 10; skin = red; name = 0; break;
         case green: health = 100;ammo = 20;skin = green;name = 0;break;
         case blue: health = 100; ammo = 40; skin = blue;name = 0;break;
      }
}
//--------------------------------
monster::monster(char * nam)
{
   /* К длине строки добавляется 1 для хранения нуль-символа */
   name = new char [strlen(nam) + 1];
```

```
   strcpy(name, nam);
   health = 100; ammo = 10; skin = red;
}
//--------------------------------
monster * m = new monster ("Ork");
monster Green (green);
```

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров. Объекты класса `monster` теперь можно инициализировать различными способами, требуемый конструктор будет вызван в соответствии со списком инициализации. При задании нескольких конструкторов следует соблюдать те же правила, что и при написании перегруженных функций – у компилятора должна быть возможность распознать нужный вариант.

Существует еще один способ инициализации полей в конструкторе (кроме уже описанного присваивания полям значений параметров) – с помощью списка инициализаторов, расположенным после двоеточия между заголовком и телом конструктора:

```
monster::monster(int he, int am):
      health (he), ammo (am), skin (red), name (0){}
```

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации полей-констант, полей-ссылок и полей-объектов. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

4.3 Конструктор копирования

Конструктор копирования – это специальный вид конструктора, получающий в качестве единственного параметра указатель на объект этого же класса:

```
T::T(const T&) { ... /* Тело конструктора */ }
```

Здесь `T` – имя класса. Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

- при описании нового объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции.

Если программист не указал ни одного конструктора копирования, компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки,

это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Запишем конструктор копирования для класса `monster`. Поскольку в нем есть поле `name`, содержащее указатель на строку символов, конструктор копирования должен выделять память под новую строку и копировать в нее исходную:

```
monster::monster(const monster &M)
{
  if (M.name)
  {
    name = new char [strlen(M.name) + 1];
    strcpy(name, M.name);
  }
  else name = 0;
  health = M.health; ammo = M.ammo; skin = M.skin;
}
...
monster Vasia (blue);
monster Super = Vasia;  // Работает конструктор копирования
monster *m = new monster ("Ork");
monster Green = *m;           // Работает конструктор копирования
```

4.4 Статические элементы класса

С помощью модификатора `static` можно описать статические поля и методы класса.

4.4.1 Статические поля

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора:

```
class A
{
public:
  static int count;
}
...
A::count = 0;
```

Статические поля доступны как через имя класса, так и через имя объекта:

```
/* будет выведено одно и то же */
A *a, b; * cout << A::count << a->count << b.count;
```

На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как `private`, нельзя инициализировать с помощью операции доступа к области действия, как описано выше. Им можно присвоить значения только с помощью статических методов, как описано ниже.

Память, занимаемая статическим полем, не учитывается при определении размера объекта операцией `sizeof`. Статические поля нельзя инициализировать в конструкторе, так как они создаются до создания любого объекта.

Классическое применение статических полей – подсчет объектов. Для этого в классе объявляется целочисленное поле, которое увеличивается в конструкторе и уменьшается в деструкторе.

4.4.2 Статические методы

Статические методы могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, поскольку им не передается скрытый указатель `this`. Обращение к статическим методам производится так же, как к статическим полям – либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Статические методы не могут быть константными (`const`) и виртуальными (`virtual`).

4.5 Дружественные функции и классы

Иногда желательно иметь непосредственный доступ извне к скрытым полям класса, то есть расширить интерфейс класса. Для этого служат дружественные функции и дружественные классы.

4.5.1 Дружественная функция

Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом `friend`. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель `this` ей не передается. Одна функция может «дружить» сразу с несколькими классами.

Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие

спецификаторов доступа, место размещения ее объявления в классе безразлично.

В качестве примера ниже приведено описание двух функций, дружественных классу `monster`. Функция `kill` является методом класса `hero`, а функция `steal_ammo` не принадлежит ни одному классу. Обеим функциям в качестве параметра передается ссылка на объект класса `monster`.

```
class monster;    // Предварительное объявление класса
class hero
{
...
   void kill(monster &);
};
class monster
{
...
   friend int steal_ammo(monster &);
   /* Класс hero должен быть определен ранее */
   friend void hero::kill(monster &);
};
int steal_ammo(monster &M){return --M.ammo;}
void hero::kill(monster &M){M.health = 0; M.ammo = 0;}
```

4.5.2 Дружественная класс

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого, весь класс объявляется дружественным с помощью ключевого слова `friend`. В приведенном ниже примере класс `mistress` объявляется дружественным классу `hero`:

```
class hero {
      ... friend class mistress;
};
class mistress{
      ... void f1();
      void f1();
};
```

Функции `f1` и `f2` являются дружественными по отношению к классу `hero` (хотя и описаны без ключевого слова `friend`) и имеют доступ ко всем его полям.

Объявление `friend` не является спецификатором доступа и не наследуется. Обратите внимание на то, что класс сам определяет, какие функции и классы являются дружественными, а какие нет.

4.6 Деструкторы

Деструктор – это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

– для локальных переменных – при выходе из блока, в котором они объявлены;

– для глобальных – как часть процедуры выхода из `main`;

– для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции `delete` (автоматический вызов деструктора при выходе указателя из области действия не производится).

При уничтожении массива деструктор вызывается для каждого элемента удаляемого массива. Для динамических объектов деструктор вызывается при уничтожении объекта операцией `delete`. При выполнении операции `delete[]` деструктор вызывается для каждого элемента удаляемого массива.

Имя деструктора начинается с тильды (`~`), непосредственно за которой следует имя класса. Деструктор:

– не имеет аргументов и возвращаемого значения;

– не может быть объявлен как `const` или `static`;

– не наследуется;

– может быть виртуальным;

– может вызываться явным образом путем указания полностью уточненного имени; это необходимо для объектов, которым с помощью `new` выделялся конкретный адрес.

Если деструктор явным образом не определен, компилятор создает его автоматически.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически – иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор для рассматриваемого примера должен выглядеть так:

```
monster::~monster() {delete [] name;}
```

5. ПЕРЕГРУЗКА ОПЕРАЦИЙ В ООП

5.1 Перегрузка операций

C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Эта дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением:

```
. .*   ?:   ::   #    ##   sizeof
```

Перегрузка операций осуществляется с помощью функций специального вида (функций-операций) и подчиняется следующим правилам:

– сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо) по сравнению с использованием в стандартных типах данных;

– нельзя переопределить операцию по отношению к стандартным типам данных;

– функция-операция не может иметь аргументов по умолчанию;

– функции-операции наследуются (за исключением =).

Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс (особый случай: функция-операция, первый параметр которой – стандартного типа, не может определяться как метод класса).

Функция-операция содержит ключевое слово `operator`, за которым следует знак переопределяемой операции:

```
тип operator операция ( список параметров) { тело функции }
```

5.2 Перегрузка унарных операций

Унарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например:

```
class monster
{...  monster & operator ++() {++health; return *this;}}
monster Vasia;
cout << (++Vasia).get_health();
```

Если функция определяется вне класса, она должна иметь один параметр типа класса:

```
class monster
{... friend monster & operator ++( monster &M);};
monster& operator ++(monster &M) {++M.health; return M;}
```

Если не описывать функцию внутри класса как дружественную, нужно учитывать доступность изменяемых полей (в данном случае поле `health` недоступно извне, так как описано со спецификатором `private`, поэтому для его изменения требуется использование соответствующего метода, не описанного в приведенном примере).

Операции постфиксного инкремента и декремента должны иметь первый параметр типа `int`. Он используется только для того, чтобы отличить их от префиксной формы:

```
class monster
{...  monster operator ++(int){monster M(*this); health++; return M;}};
monster Vasia;
cout << (Vasia++).get_health();
```

5.3 Перегрузка бинарных операций

Бинарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

```
class monster
{
  ...

  bool operator >( const monster &M)
  {
     if( health > M.get_health())
        return true;
     return false;
  }
};
```

Если функция определяется вне класса, она должна иметь два параметра типа класса:

```
bool operator >(const monster &M1, const monster &M2)
{
    if( M1.get_health() > M2.get_health()) return true;
    return false;
}
```

Бинарные арифметические операции, такие как `+`, `-` и `*`, возвращают новый экземпляр класса, помеченный ключевым словом `const`. Например:

```
const MyClass MyClass::operator+(const MyClass &other) const {
    MyClass result = *this;       // Make a copy of myself
    result.value += other.value;  // Use += to add other to the copy.
    return result;                // All done!
  }
```

Использование ключевого слова `const` необходимо для того, что бы было не возможно написать следующий код:

```
MyClass a, b, c;
(a + b) = c;
```

В случае отсутствия ключевого слова const, данный код будет успешно скомпилирован.

5.4 Перегрузка операции присваивания

Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля ссылок на динамически выделяемую память, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику операции, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент – ссылку на присваиваемый объект:

```
monster& operator = (const monster &M)
{
   if (&M == this)      // Проверка на самоприсваивание
      return *this;
   if (name)
      delete [] name;
   if (M.name) {
      name = new char [strlen(M.name) + 1];
      strcpy(name, M.name);
   }
   else name = 0;
   health = M.health; ammo = M.ammo; skin = M.skin;
   return *this;
}
```

Операцию присваивания можно определять только в теле класса. Она не наследуется. Можно заметить, что операция присваивания возвращает ссылку, что позволяет совершать «цепочки присваивания»:

```
int a, b, c, d;
a = b = c = d = 23;
```

В данной цепочке присваивания первой выполняется операция `d = 23`, возвращающая ссылку на переменную `d`, значение которой, в свою очередь, присваивается переменной `c` и т.д.

5.5 Перегрузка операции приведения типа

Можно определить функции-операции, которые будут осуществлять преобразование класса к другому типу. Формат:

```
operator имя_нового_типа ();
```

Тип возвращаемого значения и параметры указывать не требуется. Можно определять виртуальные функции преобразования типа.

Пример:

```
monster::operator int(){ return health; }
...
monster Vasia; cout << int(Vasia);
```

5.6 Особенности работы операторов new и delete

Переменная объектного типа в динамической памяти создаётся в два этапа:

1. Выделяется память с помощью оператора `new`.
2. Вызывается конструктор класса.

Удаляется такая переменная тоже в два этапа:

1. Вызывается деструктор класса.
2. Освобождается память с помощью оператора `delete`.

5.7 Перегрузка операторов new и delete для отдельных классов

Операторы `new` и `delete` можно перегрузить. Для этого есть несколько причин:

– можно увеличить производительность за счёт кеширования: при удалении объекта не освобождать память, а сохранять указатели на свободные блоки, используя их для вновь конструируемых объектов.

– можно выделять память сразу под несколько объектов.

– можно реализовать собственный «сборщик мусора» (`garbage collector`).

– можно вести лог выделения/освобождения памяти.

Операторы `new` и `delete` имеют следующие сигнатуры:

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

Оператор `new` принимает размер памяти, которую необходимо выделить, и возвращает указатель на выделенную память.

Оператор `delete` принимает указатель на память, которую нужно освободить.

```
class A {

public:
    void *operator new(size_t size);
    void operator delete(void *p);
};

void *A::operator new(size_t size) {
    printf("Allocated %d bytes\n", size);
    return malloc(size);
}

void A::operator delete(void *p) {
    free(p);
}
```

Вместо функций `malloc` и `free` можно использовать глобальные операторы `::new` и `::delete`.

Рекомендуется не производить в операторе `new` (особенно в глобальном) какие-либо операции с объектами, которые могут вызвать оператор `new`. Например, для вывода текста используется функция `printf`, а не объект `std::cout`.

Операторы `new` и `delete`, объявленные внутри класса, функционируют подобно статическим функциям и вызываются для данного класса и его наследников, для которых эти операторы не переопределены.

5.8 Переопределение глобальных операторов new и delete

В некоторых случаях может потребоваться перегрузить глобальные операторы `new` и `delete`. Они находятся не в пространстве имен `std`, а в глобальном пространстве имён.

Глобальные операторы `new` и `delete` вызываются для примитивных типов и для классов, в которых они не переопределены. Они имеют такие же сигнатуры, что и рассмотренные выше операторы `new` и `delete`.

```
// Для примитивных типов вызываются глобальные ::new и ::delete
int *i = new int;
delete i;

// Для класса A вызываются переопределённые A::new и A::delete
A *a = new A;
delete a;
```

```
// Для класса C операторы new и delete не переопределены,
// поэтому вызываются глобальные ::new и ::delete
C *c = new C;
delete c;
```

6. НАСЛЕДОВАНИЕ В ООП

При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

6.1 Виды наследования

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью модификаторов наследования `private`, `protected` и `public`.

Если базовых классов несколько, то они перечисляются через запятую. Перед каждым может стоять свой модификатор наследования. По умолчанию для классов установлен модификатор `private`, а для структур – `public`.

Если задан модификатор наследования `public`, оно называется открытым. Использование модификатора `protected` делает наследование защищенным, а модификатора `private` – закрытым. В зависимости от вида наследования классы ведут себя по-разному. Класс может наследовать от структуры, и наоборот.

Для любого элемента класса может также использоваться спецификатор `protected`, который для одиночных классов, не входящих в иерархию, равносилен `private`. Разница между ними проявляется при наследовании. Возможные сочетания модификаторов и спецификаторов доступа приведены в таблице ниже.

Как видно из таблицы ниже, `private` элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса.

Модификатор наследования	Спецификатор базового класса	Доступ в производном классе
private	private	нет
	protected	private
	public	private
protected	private	нет
	protected	protected
	public	protected
public	private	нет
	protected	protected
	public	public

Элементы protected при наследовании с ключом private становятся в производном классе private, в остальных случаях права доступа к ним не изменяются.

Доступ к элементам public при наследовании становится соответствующим ключу доступа.

Если базовый класс наследуется с ключом private, можно выборочно сделать некоторые его элементы доступными в производном классе, объявив их в секции public производного класса с помощью операции доступа к области видимости:

```
class Base {...
      public: void f();
};
class Derived : private Base {...
      public: using Base::f;
};
```

6.2 Простое наследование

Простым называется наследование, при котором производный класс имеет одного родителя. Для различных элементов класса существуют разные правила наследования. Рассмотрим наследование классов на примере.

Создадим производный от класса monster класс daemon, добавив «демону» способность думать:

```
enum color {red, green, blue};
// -------------  Класс monster -------------
class monster
{
    // --------- Скрытые поля класса: ------------
    int health, ammo;
    color skin;
    char *name;
public:
  // ------------- Конструкторы:
  monster(int he = 100, int am  = 10);
  monster(color sk);
```

```
  monster(char * nam);
  monster(monster &M);
  // ------------ Деструктор:
  ~monster() {delete [] name;}
  // ------------ Операции:
  monster& operator ++(){++health; return *this;}
  monster operator ++(int)
    {monster M(*this); health++; return M;}
  operator int(){return health;}
  bool operator >(monster &M)
  {
     if( health > M.get_health()) return true;
     return false;
  }
  monster& operator = (monster &M)
  {
     if (&M == this) return *this;
     if (name) delete [] name;
     if (M.name) {
       name = new char [strlen(M.name) + 1];
       strcpy(name, M.name);
     }
     else name = 0;
     health = M.health; ammo = M.ammo; skin = M.skin;
     return *this;
  }
  // ------------ Методы доступа к полям:
  int get_health()      const { return health; }
  int get_ammo()  const { return ammo; }
  // ------------ Методы, изменяющие значения полей:
  void set_health(int he){ health = he;}
  void draw(int x, int y, int scale, int position);
};
// ------------ Реализация класса monster ------------
monster::monster(int he, int am):
  health (he), ammo (am), skin (red), name (0){}
monster::monster(monster &M)
{
  if (M.name)
  {
    name = new char [strlen(M.name) + 1];
    strcpy(name, M.name);
  }
  else name = 0;
  health = M.health; ammo = M.ammo; skin = M.skin;
}
monster::monster(color sk)
{
  switch (sk)
  {
    case red: health = 100; ammo = 10; skin = red;  name = 0; break;
    case green: health = 100;ammo = 20;skin = green; name = 0; break;
    case blue: health = 100; ammo = 40; skin = blue; name = 0;break;
  }
}
monster::monster(char * nam)
{
  name = new char [strlen(nam)+1];
  strcpy(name, nam);
  health = 100; ammo = 10; skin = red;
}
```

```
void monster::draw(int x, int y, int scale, int position)
{ /* ... Отрисовка monster */ }
// -------------  Класс daemon -------------
class daemon : public monster
{
  int brain;
public:
  // ------------- Конструкторы:
  daemon(int br = 10){brain = br;};
  daemon(color sk) : monster (sk) {brain = 10;}
  daemon(char * nam) : monster (nam) {brain = 10;}
  daemon(daemon &M) : monster (M) {brain = M.brain;}
  // ------------- Операции:
  daemon& operator = (daemon &M)
  {
    if (&M == this) return *this;
    brain = M.brain;
    monster::operator = (M);
    return *this;
  }
  // ------------- Методы, изменяющие значения полей:
  void draw(int x, int y, int scale, int position);
  void think();
};
// ------------- Реализация класса daemon -------------
void daemon::draw(int x, int y, int scale, int position)
{ /* ... Отрисовка daemon */ }
void daemon:: think(){ /* ... */ }
```

В классе `daemon` введено поле `brain` и метод `think`, определены собственные конструкторы и операция присваивания, а также переопределен метод отрисовки `draw`. Все поля класса `monster`, операции (кроме присваивания) и методы `get_health`, `get_ammo` и `set_health` наследуются в классе `daemon`, а деструктор формируется по умолчанию.

6.2.1 Правила наследования различных методов

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров). Это использовано в первом из конструкторов класса `daemon`.

Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации (это продемонстрировано в трех последних конструкторах).

Не наследуется и операция присваивания, поэтому ее также требуется явно определить в классе `daemon`. Обратите внимание на запись функции-операции: в ее теле применен явный вызов функции-операции присваивания из базового класса. Чтобы лучше представить себе синтаксис вызова, ключевое слово `operator` вместе со знаком операции можно интерпретировать как имя функции-операции.

Вызов функций базового класса предпочтительнее копирования фрагментов кода из функций базового класса в функции производного. Кроме сокращения объема кода, этим достигается упрощение модификации программы: изменения требуется вносить только в одну точку программы, что сокращает количество возможных ошибок.

6.2.2 Правила для деструкторов при наследовании

Деструкторы не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.

Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

Поля, унаследованные из класса `monster`, недоступны функциям производного класса, поскольку они определены в базовом классе как `private`. Если функциям, определенным в `daemon`, требуется работать с этими полями, можно либо описать их в базовом классе как `protected`, либо обращаться к ним с помощью функций из `monster`, либо явно переопределить их в `daemon` так, как было показано в предыдущем разделе.

Добавляемые поля в наследнике могут совпадать и по имени, и по типу с полями базового класса. При этом поле предка будет скрыто.

Статические поля, объявленные в базовом классе, наследуются обычным образом. Все объекты базового класса и всех его наследников разделяют единственную копию статических полей базового класса.

Рассматривая наследование методов, обратите внимание на то, что в классе `daemon` описан метод `draw`, переопределяющий метод с тем же именем в классе `monster` (поскольку отрисовка различных персонажей, естественно, выполняется по-разному). Таким образом, производный класс может не только дополнять, но и корректировать поведение базового класса. Доступ к переопределенному методу базового класса для производного класса выполняется через уточненное с помощью операции доступа к области видимости имя.

Класс-потомок наследует все методы базового класса, кроме конструкторов, деструктора и операции присваивания. Не наследуются ни дружественные функции, ни дружественные отношения классов.

В классе-наследнике можно определять новые методы. В них разрешается вызывать любые доступные методы базового класса. Если имя метода в наследнике совпадает с именем метода базового класса, то метод производного класса скрывает все методы базового класса с таким именем. При этом прототипы методов могут не совпадать. Если в методе-наследнике требуется вызвать одноименный метод родительского класса, нужно задать его с префиксом класса. Это же касается и статических методов.

6.3 Виртуальные методы

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса, например:

```
monster *p; // Описывается указатель на базовый класс
p = new daemon;   /* Указатель ссылается на объект производного класса */
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например:

```
p -> draw(1, 1, 1, 1);
```

В результате будет вызван метод класса `monster`, а не класса `daemon`, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ранним связыванием. Чтобы вызвать метод класса `daemon`, можно использовать явное преобразование типа указателя:

```
((daemon * p)) -> draw(1, 1, 1, 1);
```

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен.

В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель любого производного класса. Другой пример – связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В C++ реализован механизм позднего связывания, когда разрешение ссылок на функцию происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего функцию. Этот механизм реализован с помощью виртуальных методов.

Для определения виртуального метода используется спецификатор `virtual`:

```
virtual void draw(int x, int y, int scale, int position);
```

Рассмотрим правила использования виртуальных методов:

– Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров – обычным.

– Виртуальные методы наследуются, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.

– Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.

– Виртуальный метод не может объявляться с модификатором `static`, но может быть объявлен как дружественная функция.

– Если производный класс содержит виртуальные методы, они должны быть определены в базовом классе хотя бы как чисто виртуальные.

Чисто виртуальный метод содержит признак = 0 вместо тела, например:

```
virtual void f(int) = 0;
```

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Если определить метод `draw` в классе `monster` как виртуальный, решение о том, метод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель:

```
monster *r, *p;
r = new monster;  // Создается объект класса monster
p = new daemon;               // Создается объект класса daemon
r -> draw(1,1,1,1);           // Вызывается метод monster::draw
p -> draw(1,1,1,1);           // Вызывается метод daemon::draw
p -> monster::draw(1,1,1,1);  // Обход механизма виртуальных методов
```

Если объект класса `daemon` будет вызывать метод `draw` не непосредственно, а косвенно (то есть из другого метода, который может быть определен только в классе `monster`), будет вызван метод `draw` класса `daemon`.

Итак, виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы (перевод красивого английского слова `virtual` – всего-навсего «фактический», то есть ссылка разрешается по факту вызова).

6.3.1 Механизм позднего связывания

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает таблицу виртуальных методов (`vtbl`), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в `vtbl` одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное поле ссылки на `vtbl`, называемое `vptr`. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к `vtbl` через `vptr` объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы, поэтому без необходимости делать методы виртуальными смысла не имеет.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта.

Четкого правила, по которому метод следует делать виртуальным, не существует. Можно только дать рекомендацию объявлять виртуальными ме-

тоды, для которых есть вероятность, что они будут переопределены в производных классах. Методы, которые во всей иерархии останутся неизменными или те, которыми производные классы пользоваться не будут, делать виртуальными нет смысла. С другой стороны, при проектировании иерархии не всегда можно предсказать, каким образом будут расширяться базовые классы, особенно при проектировании библиотек классов, а объявление метода виртуальным обеспечивает гибкость и возможность расширения.

Для пояснения последнего тезиса представим себе, что вызов метода `draw` осуществляется из метода перемещения объекта. Если текст метода перемещения не зависит от типа перемещаемого объекта (поскольку принцип перемещения всех объектов одинаков, а для отрисовки вызывается конкретный метод), переопределять этот метод в производных классах нет необходимости, и он может быть описан как невиртуальный. Если метод `draw` виртуальный, метод перемещения сможет без перекомпиляции работать с объектами любых производных классов – даже тех, о которых при его написании ничего известно не было.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется полиморфным. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

6.4 Виртуальный деструктор

В языке программирования C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

Рассмотрим следующий пример:

```
#include <iostream>
using namespace std;

// Вспомогательный класс
class Object
{
  public:
    Object() { cout << "Object::ctor()" << endl; }
   ~Object() { cout << "Object::dtor()" << endl; }
};

// Базовый класс
class Base
{
```

```
  public:
    Base() { cout << "Base::ctor()" << endl; }
    virtual ~Base() { cout << "Base::dtor()" << endl; }
    virtual void print() = 0;
};

// Производный класс
class Derived: public Base
{
  public:
    Derived() { cout << "Derived::ctor()" << endl; }
   ~Derived() { cout << "Derived::dtor()" << endl; }
    void print() {}
    Object  obj;
};

int main ()
{
    Base * p = new Derived;
    delete p;
    return 0;
}
```

В функции `main` указателю на базовый класс присваивается адрес динамически создаваемого объекта производного класса `Derived`. Затем через этот указатель объект разрушается. При этом наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Вывод программы с использованием виртуального деструктора в базовом классе будет следующим:

```
Base::ctor()
Object::ctor()
Derived::ctor()
Derived::dtor()
Object::dtor()
Base::dtor()
```

Уничтожение объекта производного класса через указатель на базовый класс с невиртуальным деструктором дает неопределенный результат. На практике это выражается в том, что будет разрушена только часть объекта, соответствующая базовому классу. Если в коде выше убрать ключевое слово `virtual` перед деструктором базового класса, то вывод программы будет уже иным. Обратите внимание, что член данных `obj` класса `Derived` также не разрушается.

```
Base::ctor()
Object::ctor()
Derived::ctor()
Base::dtor()
```

Когда же следует объявлять деструктор виртуальным? Существует правило – если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным. Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций `vptr`, что увеличивает его общий размер. Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

6.5 Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов – объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо иметь в виду следующее:

– Абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения.

– Допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект.

– Если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции, работающие с объектом любого типа в пределах одной иерархии.

7. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ В ООП

Множественное наследование означает, что класс имеет несколько базовых классов. При этом, если в базовых классах есть одноименные элементы, может произойти конфликт идентификаторов, который устраняется с помощью операции доступа к области видимости:

```
class monster
{
      public: int get_health();
      ...
};
class hero
{
      public: int get_health();
      ...
};
class being: public monster, public hero{ ... };
int main(){
      being A;
      cout << A.monster::get_health();
      cout << A.hero::get_health();}
```

Использование конструкции `A.get_health()` приведет к ошибке, поскольку компилятор не в состоянии разобраться, метод какого из базовых классов требуется вызвать.

Если у базовых классов есть общий предок, это приведет к тому, что производный от этих базовых класс унаследует два экземпляра полей предка, что чаще всего является нежелательным. Чтобы избежать такой ситуации, требуется при наследовании общего предка определить его как виртуальный класс:

```
class monster{ ... };
class daemon: virtual public monster{ ... };
class lady: virtual public monster{ ... };
class god: public daemon, public lady{ ... };
```

Класс `god` содержит только один экземпляр полей класса `monster`.

7.1 Альтернатива наследованию

Наследование – это очень сложная тема, и даже создатель языка C++ Б. Страуструп рекомендует везде, где это возможно, обходиться без него.

Альтернативой наследованию при проектировании классов является вложение, когда один класс включает в себя поля, являющиеся классами или указателями на них. Например, если есть класс «двигатель», а требуется

описать класс «самолет», логично сделать двигатель полем этого класса, а не его предком.

Вид вложения, когда в классе описано поле объектного типа, называют композицией. Если в классе описан указатель на объект другого класса, это обычно называют агрегацией. При композиции время жизни всех объектов (и объемлющего, и его полей) одинаково. Агрегация представляет собой более слабую связь между объектами, потому что объекты, на которые ссылаются поля-указатели, могут появляться и исчезать в течение жизни содержащего их объекта, кроме того, один и тот же указатель может ссылаться на объекты разных классов в пределах одной иерархии. Поле-указатель может также ссылаться не на один объект, а на неопределенное количество объектов, например быть указателем на начало линейного списка.

7.2 Отличия структур и объединений от классов

Структуры (`struct`) и объединения (`union`) представляют собой частные случаи классов. Структуры отличаются от классов тем, что доступ к элементам, а также базовый класс при наследовании по умолчанию считаются `public`. Структуры предпочтительнее использовать для объектов, все элементы которых доступны.

Доступ в объединениях также устанавливается `public`, кроме того, в них вообще нельзя использовать спецификаторы доступа. Объединение не может участвовать в иерархии классов. Элементами объединения не могут быть объекты, содержащие конструкторы и деструкторы. Объединение может иметь конструктор и другие методы, только не статические. В анонимном объединении методы описывать нельзя.

7.3 Ромбовидное наследование

Ромбовидное наследование – ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса `B` и `C` наследуют от `A`, а класс `D` наследует от обоих классов `B` и `C`. При этой схеме наследования может возникнуть неоднозначность: если метод класса `D` вызывает метод, определенный в классе `A` (и этот метод не был переопределен), а классы `B` и `C` по-своему переопределили этот метод, то от какого класса его наследовать: `B` или `C`?

Проблема ромба получила свое название благодаря очертаниям диаграммы наследования классов в этой ситуации. В данной статье, класс `A` обозначается в виде вершины, классы `B` и `C` по отдельности указываются ниже, а `D` соединяется с обоими в самом низу, образуя ромб.

C++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно, в результате чего объект `D` будет на самом деле содержать два разных подобъекта `A`, и при использовании членов `A` потребуется указать путь наследования (`B::A` или `C::A`). Чтобы сгенерировать ромбовидную структуру наследования, необходимо воспользоваться виртуальным наследованием класса `A` на нескольких путях наследования: если оба наследования от `A` к `B` и от `A` к `C` помечаются спецификатором `virtual` (например, `class B : virtual public A`), C++ специальным образом проследит за созданием только одного подобъекта `A`, и использование членов `A` будет работать корректно. Если виртуальное и невиртуальное наследования смешиваются, то получается один виртуальный подобъект `A` и по одному невиртуальному подобъекту `A` для каждого пути невиртуального наследования к `A`. При виртуальном вызове метода виртуального базового класса используется так называемое правило доминирования: компилятор запрещает виртуальный вызов метода, который был перегружен на нескольких путях наследования.

Виртуальное наследование в C++ – один из вариантов наследования, который нужен для решения некоторых проблем, порождаемых наличием возможности множественного наследования (особенно «ромбовидного наследования»), путем разрешения неоднозначности того, методы которого из классов-предков необходимо использовать. Оно применяется в тех случаях, когда множественное наследование вместо предполагаемой полной композиции свойств классов-предков приводит к ограничению доступных наследуемых свойств вследствие неоднозначности. Базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слова `virtual`.

Рассмотрим следующий пример:

```
class Animal
{
public:
   virtual void eat();  // Метод определяется для данного класса.
   ...
};

class Mammal : public Animal
{
```

```
public:
   virtual Color getHairColor();
   ...
};

class WingedAnimal : public Animal
{
public:
   virtual void flap();
   ...
};

class Bat : public Mammal, public WingedAnimal
{};  // Метод eat() не переопределен в Bat.
```

В вышеприведенном коде вызов `bat.eat()` является неоднозначным. Он может относиться как к `Bat::WingedAnimal::Animal::eat()` так и к `Bat::Mammal::Animal::eat()`. У каждого промежуточного наследника (`WingedAnimal`, `Mammal`) метод `eat()` может быть переопределен (это не меняет сущность проблемы с точки зрения языка). Проблема в том, что семантика традиционного множественного наследования не соответствует моделируемой им реальности. В некотором смысле, сущность `Animal` единственна по сути; `Bat` – это `Mammal` и `WingedAnimal`, но свойство животности летучей мыши (`Bat`), оно же свойство животности млекопитающего (`Mammal`) и оно же свойство животности `WingedAnimal` – по сути это одно и то же свойство.

При наследовании классы предка и наследника просто помещаются в памяти друг за другом. Таким образом объект класса `Bat` это на самом деле последовательность объектов классов (`Animal`, `Mammal`, `Animal`, `WingedAnimal`, `Bat`), размещенных последовательно в памяти, при этом `Animal` повторяется дважды, что и приводит к неоднозначности.

Можно переопределить исходные классы следующим образом:

```
class Animal
{
public:
   virtual void eat();
   ...
};

// Two classes virtually inheriting Animal:
class Mammal : public virtual Animal        // <--- ключевое слово
virtual
{
public:
   virtual Color getHairColor();
   ...
};

class WingedAnimal : public virtual Animal  // <--- ключевое слово
```

```
virtual
{
public:
   virtual void flap();
 ...
};

class Bat : public Mammal, public WingedAnimal {};
```

Теперь, часть `Animal` объекта класса `Bat::WingedAnimal` та же самая, что и часть `Animal`, которая используется в `Bat::Mammal`, и можно сказать, что `Bat` имеет в своем представлении только одну часть `Animal` и вызов `Bat::eat()` становится однозначным.

Виртуальное наследование реализуется через добавление указателей на виртуальную таблицу `vtable` в классы `Mammal` и `WingedAnimal`, это делается в частности потому, что смещение памяти между началом `Mammal` и его `Animal` части неизвестно на этапе компиляции, а выясняется только во время выполнения. Таким образом, `Bat` представляется, как (`vtable*`, `Mammal`, `vtable*`, `WingedAnimal`, `Bat`, `Animal`). Два указателя `vtable` на объект увеличивают размер объекта на величину двух указателей, но это обеспечивает единственность `Animal` и отсутствие многозначности. Нужны два указателя `vtables`: по одному на каждого предка в иерархии, который виртуально наследуется от `Animal`: один для `Mammal` и один для `WingedAnimal`. Все объекты класса `Bat` будут иметь одни и те же указатели `vtable*`, но каждый отдельный объект `Bat` будет содержать собственную реализацию объекта `Animal`. Если какой-нибудь другой класс будет наследоваться от `Mammal`, например `Squirrel` (белка), то `vtable*` в объекте `Mammal` объекта `Squirrel` будет отличаться от `vtable*` в объекте `Mammal` объекта `Bat`, хотя в особом случае они по-прежнему могут быть одинаковы по сути: когда часть `Squirrel` объекта имеет тот же самый размер, что и часть `Bat`, поскольку, тогда расстояние от реализации `Mammal` до части `Animal` будет одинаковым. Но сами виртуальные таблицы `vtables` будут все же разными, в отличие от располагаемой в них информации о смещениях.

8. ШАБЛОНЫ ФУНКЦИЙ

8.1 Использование шаблонов функций

При создании функций иногда возникают ситуации, когда две функции выполняют одинаковую обработку, но работают с разными типами данных (например, одна использует параметры типа `int`, а другая типа `float`). Вы уже знаете, что с помощью механизма перегрузки функций можно использовать одно и то же имя для функций, выполняющих разные действия и имеющих разные типы параметров. Однако, если функции возвращают значения разных типов, вам следует использовать для них уникальные имена. Предположим, например, что у вас есть функция с именем `max`, которая возвращает максимальное из двух целых значений. Если позже вам потребуется подобная функция, которая возвращает максимальное из двух значений с плавающей точкой, вам следует определить другую функцию, например `fmax`.

Шаблон определяет набор операторов, с помощью которых ваши программы позже могут создать несколько функций.

Программы часто используют шаблоны функций для быстрого определения нескольких функций, которые с помощью одинаковых операторов работают с параметрами разных типов или имеют разные типы возвращаемых значений.

Шаблоны функций имеют специфичные имена, которые соответствуют имени функции, используемому вами в программе.

После того как ваша программа определила шаблон функции, она в дальнейшем может создать конкретную функцию, используя этот шаблон для задания прототипа, который включает имя данного шаблона, возвращаемое функцией значение и типы параметров.

В процессе компиляции компилятор C++ будет создавать в вашей программе функции с использованием типов, указанных в прототипах функций, которые ссылаются на имя шаблона.

Шаблоны функций имеют уникальный синтаксис, который может быть на первый взгляд непонятен. Однако после создания одного или двух шаблонов вы обнаружите, что реально их очень легко использовать.

8.2 Создание простого шаблона функции

С помощью такого шаблона ваши программы в дальнейшем могут определить конкретные функции с требуемыми типами. Например, ниже определен шаблон для функции с именем `max`, которая возвращает большее из двух значений:

```
template<class T> T max(T a, T b)
{
   if (a > b) return(a);
   else return(b);
}
```

Буква `T` данном случае представляет собой общий тип шаблона. После определения шаблона внутри вашей программы вы объявляете прототипы функций для каждого требуемого вам типа. В случае шаблона `max` следующие прототипы создают функции типа `float` и `int`.

```
float max(float, float);
int max(int, int);
```

Когда компилятор C++ встретит эти прототипы, то при построении функции он заменит тип шаблона `T` указанным вами типом. В случае с типом `float` функция `max` после замены примет следующий вид:

```
template<class T> T max(T a, T b)
{
   if (a > b) return(a) ;
   else return(b);
}

float max(float a, float b)
{
   if (a > b) return(a) ;
   else return(b);
}
```

Следующая программа MAX_TEMP.CPP использует шаблон `max` для создания функции типа `int` и `float`.

```
template< typename T >
// прототип: шаблон sort объявлен, но не определён
void sort( T array[], int size);
template< typename T >
void sort( T array[], int size )   // объявление и определение
{
  T t;
  for (int i = 0; i < size - 1; i++)
    for (int j = size - 1; j > i; j--)
      if (array[j] < array[j-1])
```

```
      {
        t = array[j];
        array[j] = array[j-1];
        array[j-1] = t;
      }
}

int i[5] = { 5, 4, 3, 2, 1 };
sort< int >( i, 5 );

char c[] = "бвгда";
sort< char >( c, strlen( c ) );

sort< int >( c, 5 ); // ошибка: у sort< int > параметр int[] а не char[]

int i[5] = { 5, 4, 3, 2, 1 };
sort( i, i + 5 );           // вызывается sort< int >

char c[] = "бвгда";
sort( c, c + strlen( c ) );         // вызывается sort< char >

#include <iostream.h>
template<class T> T max(T a, T b)
{
   if (a > b) return(a);
   else return(b);
}
float max(float, float);
int max(int, int);

void main(void)
{
   cout << "Максимум 100 и 200 равен " << max(100, 200) << endl;
   cout << "Максимум 5.4321 и 1.2345 равен "
        << max(5.4321, 1.2345) << endl;
}
```

В процессе компиляции компилятор C++ автоматически создает операторы для построения одной функции, работающей с типом `int`, и второй функции, работающей с типом `float`. Поскольку компилятор C++ управляет операторами, соответствующими функциям, которые вы создаете с помощью шаблонов, он позволяет вам использовать одинаковые имена для функций, которые возвращают значения разных типов. Вы не смогли бы это сделать, используя только перегрузку функций.

8.3 Использование шаблонов функций

По мере того как ваши программы становятся более сложными, возможны ситуации, когда вам потребуются подобные функции, выполняющие одни и те же операции, но с разными типами данных. Шаблон функции позволяет вашим программам определять общую, или типонезависимую, функцию. Когда программе требуется использовать функцию для определенного типа, например `int` или `float`, она указывает прототип функции, который

использует имя шаблона функции и типы возвращаемого значения и параметров. В процессе компиляции C++ создаст соответствующую функцию. Создавая шаблоны, вы уменьшаете количество функций, которые должны кодировать самостоятельно, а ваши программы могут использовать одно и то же имя для функций, выполняющих определенную операцию, независимо от возвращаемого функцией значения и типов параметров.

8.4 Шаблоны, использующие несколько типов

Предыдущее определение шаблона для функции max использовало единственный общий тип `T`. Очень часто в шаблоне функции требуется указать несколько типов. Например, следующие операторы создают шаблон для функции `show_array`, которая выводит элементы массива. Шаблон использует тип `T` для определения типа массива и тип `T1` для указания типа параметра `count`:

```
template<class T,class T1> void show_array(T *array,T1 count)
{
   T1 index;
   for (index =0; index < count; index++) cout << array[index] << ' ';
   cout << endl;
}
```

Как и ранее, программа должна указать прототипы функций для требуемых типов:

```
void show_array(int *, int);
void show_array(float *, unsigned);
```

Следующая программа SHOW_TEM.CPP использует шаблон для создания функций, которые выводят массивы типа `int` и типа `float`.

```
#include <iostream.h>
template<class T,class T1> void show_array( T *array,T1 count)
{
   T1 index;
   for (index = 0; index < count; index++) cout << array[index] << ' ';
   cout << endl;
}

void show_array(int *, int);
void show_array(float *, unsigned);
void main(void) {
   int pages[] = { 100, 200, 300, 400, 500 };
   float pricesH = { 10.05, 20.10, 30.15 };
   show_array(pages, 5);
   show_array(prices, 3);
}
```

8.5 Шаблоны и несколько типов

По мере того как шаблоны функций становятся более сложными, они могут обеспечить поддержку нескольких типов. Например, ваша программа может создать шаблон для функции с именем `array_sort`, которая сортирует элементы массива. В данном случае функция может использовать два параметра: первый, соответствующий массиву, и второй, соответствующий количеству элементов массива. Если программа предполагает, что массив никогда не будет содержать более 32767 значений она может использовать тип `int` для параметра размера массива. Однако более универсальный шаблон мог бы предоставить программе возможность указать свой собственный тип этого параметра, как показано ниже:

```
template<class T, class T1> void array_sort(T array[], T1 elements)
{
// операторы
}
```

С помощью шаблона `array_sort` программа может создать функции которые сортируют маленькие массивы типа `float` (менее 128 элементов) и очень большие массивы типа `int`, используя следующие прототипы:

```
void array_sort(float, char);
void array_sort(int, long);
template< class T1,                    // параметр-тип
          typename T2,                 // параметр-тип
          int I,                       // параметр обычного типа
          T1 DefaultValue,             // параметр обычного типа
          template< class > class T3,  // параметр-шаблон
          class Character = char       // параметр по умолчанию
        >
```

9. ШАБЛОНЫ КЛАССОВ

Шаблон класса позволяет задать класс, параметризованный типом данных. Передача классу различных типов данных в качестве параметра создает семейство родственных классов. Наиболее широкое применение шаблоны находят при создании контейнерных классов. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

9.1 Создание шаблонов классов

Рассмотрим процесс создания шаблона класса на примере двусвязного списка. Поскольку списки часто применяются для организации данных, удобно описать список в виде класса, а так как может потребоваться хранить данные различных типов, этот класс должен быть параметризованным.

Сначала рассмотрим непараметризованную версию класса. Список состоит из узлов, связанных между собой с помощью указателей. Каждый узел хранит целое число, являющееся ключом списка. Опишем вспомогательный класс для представления одного узла списка:

```
class Node
{
public:
    int d;              // Данные
    Node *next, *prev;//Указатели на предыдущий и последующий узлы
    Node(int dat = 0)
      { d = dat; next = 0; prev = 0; }    // Конструктор
};
```

Поскольку этот класс будет описан внутри класса, представляющего список, поля для простоты доступа из внешнего класса сделаны доступными (`public`). Это позволяет обойтись без функций доступа и изменения полей. Назовем класс списка `List`:

```
class List
{
  class Node{ ... };
  Node *pbeg, *pend;    // Указатели на начало и конец списка
public:
    List() { pbeg = 0; pend = 0; }  // Конструктор
    ~List();            // Деструктор
    void add(int d);          // Добавление узла в конец списка
    Node * find(int i); //    Поиск узла по ключу
    Node * insert(int key, int d); //Вставка узла d после узла с ключом
key
```

```
    bool remove(int key);      // Удаление узла
    void print();              // Печать списка в прямом направлении
    void print_back();         // Печать списка в обратном направлении
};
```

Рассмотрим реализацию методов класса. Метод `add` выделяет память под новый объект типа `Node` и присоединяет его к списку, обновляя указатели на его начало и конец:

```
void List::add(int d)
{
  Node *pv = new Node(d);         // Выделение памяти под новый узел
  if (pbeg == 0)
    pbeg = pend = pv;   // Первый узел списка
  else
  {
    pv->prev = pend;    // Связывание нового узла с предыдущим
    pend->next = pv;    pend = pv;
  }         // Обновление указателя на конец списка
}
```

Метод `find` выполняет поиск узла с заданным ключом и возвращает указатель на него в случае успешного поиска и 0 в случае отсутствия такого узла в списке:

```
Node * List::find( int d )
{
  Node *pv = pbeg;
  while (pv)
  {
    if(pv->d == d)
      break;
    pv=pv->next;
  }
  return pv;
}
```

Метод `insert` вставляет в список узел после узла с ключом `key` и возвращает указатель на вставленный узел. Если такого узла в списке нет, вставка не выполняется и возвращается значение 0:

```
Node * List::insert(int key, int d)
{
  if(Node *pkey = find(key))
  {
    // Поиск узла с ключом key
    /* Выделение памяти под новый узел и его инициализация */
    Node *pv = new Node(d);
    /* Установление связи нового узла с последующим */
    pv->next = pkey->next;
    // Установление связи нового узла с предыдущим
    pv->prev = pkey;
    // Установление связи предыдущего узла с новым
    pkey->next = pv;
    if( pkey != pend)
      (pv->next)->prev = pv;  /* Установление связи последующего узла с
```

```
новым */
    // Обновление указателя на конец списка, если узел вставляется в
конец
    else pend = pv;
    return pv;
  }
  return 0;
}
```

Метод `remove` удаляет узел с заданным ключом из списка и возвращает значение `true` в случае успешного удаления и `false`, если узел с таким ключом в списке не найден:

```
bool List::remove(int key) {
  if(Node *pkey = find(key)) {
    if (pkey == pbeg)
    {                    // Удаление из начала списка
      pbeg = pbeg->next; pbeg->prev = 0;
    }
    else if (pkey == pend)
    {     // Удаление из конца списка
      pend = pend->prev; pend->next = 0;
    }
    else
    {                   // Удаление из середины списка
      (pkey->prev)->next = pkey->next;
      (pkey->next)->prev = pkey->prev;
    }
    delete pkey; return true;
  }
  return false;
}
```

Методы печати списка в прямом и обратном направлении поэлементно просматривают список, переходя по соответствующим ссылкам:

```
void List::print() {
  Node *pv = pbeg;
  cout << endl << "list: ";
  while (pv) {
     cout << pv->d << ' ';
     pv=pv->next;}
  cout << endl;
}
void List::print_back() {
  Node *pv = pend;
  cout << endl << " list back: ";
  while (pv){
     cout << pv->d << ' ';
     pv=pv->prev;
  }
  cout << endl;
}
```

Деструктор списка освобождает память из-под всех его элементов:

```
List::~List(){
   if (pbeg != 0) {
```

```
      Node *pv = pbeg;
      while (pv)
        {pv = pv->next; delete pbeg; pbeg = pv;}
    }
}
```

Ниже приведен пример программы, использующей класс `List`. Программа формирует список из 5 чисел, выводит его на экран, добавляет число в список, удаляет число из списка и снова выводит его на экран:

```
int main()
{
List L;
for (int i = 2; i<6; i++) L.add(i);
L.print();  L.print_back();   L.insert(2,200);
if (!L.remove(5))cout << "not found";
L.print();  L.print_back();}
```

Класс `List` предназначен для хранения целых чисел. Чтобы хранить в нем данные любого типа, требуется описать этот класс как шаблон и передать тип в качестве параметра.

9.2 Синтаксис описания шаблона

Шаблон класса начинается с ключевого слова `template`. В угловых скобках записывают параметры шаблона. При использовании шаблона на место этих параметров шаблону передаются аргументы: типы и константы, перечисленные через запятую.

```
template <описание_параметров_шаблона> class имя
{ /* определение класса */ };
```

Типы могут быть как стандартными, так и определенными пользователем. Для их описания в списке параметров используется ключевое слово `class`. В простейшем случае одного параметра это выглядит как `<class T>`. Здесь `T` является параметром-типом. Имя параметра может быть любым, но принято начинать его с префикса `T`. Внутри класса-шаблона параметр может появляться в тех местах, где разрешается указывать конкретный тип, например:

```
template <class TData> class List {
  class Node{
      public:
      TData d;
      Node *next;
      Node *prev;
      Node(TData dat = 0) {d = dat; next = 0; prev = 0;}
  };
...
}
```

Класс `TData` можно рассматривать как параметр, на место которого при компиляции будет подставлен конкретный тип данных. Получившийся шаблонный класс имеет тип `List<TData>`.

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

```
template <описание_параметров_шаблона>
возвр_тип имя_класса <параметры_шаблона >:: имя_функции
(список_параметров функции)
```

Проще рассмотреть синтаксис описания методов шаблона на примере:

```
template <class Data> void List <Data>::print()
    { /*  тело функции */ }
```

– Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса.

– Локальные классы не могут иметь шаблоны в качестве своих элементов.

– Шаблоны методов не могут быть виртуальными.

– Шаблоны классов могут содержать статические элементы, дружественные функции и классы.

– Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.

– Внутри шаблона нельзя определять `friend`-шаблоны.

Если у шаблона несколько параметров, они перечисляются через запятую. Ключевое слово `class` требуется записывать перед каждым параметром, например:

```
template <class T1, class T2>
struct Pair { T1 first;  T2 second; };
```

Параметрам шаблонного класса можно присваивать значения по умолчанию, они записываются после знака =. Как и для обычных функций, задавать значения по умолчанию следует, начиная с правых параметров.

Ниже приведено полное описание параметризованного класса двусвязного списка `List`.

```
template <class TData> class List
{
  class Node
  {
  public:
    TData d;
```

```
    Node *next, *prev;
    Node(TData dat = 0){d = dat; next = 0; prev = 0;}
  };
  Node *pbeg, *pend;
public:
  List(){pbeg = 0; pend = 0;}
  ~List();
  void add(TData d);
  Node * find(TData i);
  Node * insert(TData key, TData d);
  bool remove(TData key);
  void print();
  void print_back();
};

template <class TData> List <TData>::~List()
{
  if (pbeg !=0)
  {
    Node *pv = pbeg;
    while (pv)
      {pv = pv->next; delete pbeg; pbeg = pv;}
  }
}

template <class TData> void List <TData>::print()
{
  Node *pv = pbeg;
  cout << endl << "list: ";
  while (pv)
  {
    cout << pv->d << ' ';
    pv = pv->next;
  }
  cout << endl;
}

template <class TData> void List <TData>::print_back()
{
  Node *pv = pend;
  cout << endl << " list back: ";
  while (pv)
  {
    cout << pv->d << ' ';
    pv = pv->prev;
  }
  cout << endl;
}

template <class TData> void List <TData>::add(TData d)
{
  Node *pv = new Node(d);
  if (pbeg == 0)pbeg = pend = pv;
  else
  {
    pv->prev = pend;
    pend->next = pv;
    pend = pv;
  }
}
template <class TData> Node * List <TData>::find( TData d)
```

```
{
  Node *pv = pbeg;
  while (pv)
  {
    if(pv->d == d) break;
    pv = pv->next;
  }
  return pv;
}

template <class TData> Node * List <TData>::insert(TData key, TData d)
{
  if(Node *pkey = find(key))
  {
    Node *pv = new Node(d);
    pv->next = pkey->next;
    pv->prev = pkey;
    pkey->next = pv;
    if( pkey != pend)(pv->next)->prev = pv;
    else pend = pv;
    return pv;
  }
  return 0;
}

template <class TData> bool List <TData>::remove(TData key)
{
  if(Node *pkey = find(key))
  {
    if (pkey == pbeg)
    {
      pbeg = pbeg->next; pbeg->prev = 0;
    }
    else if (pkey == pend)
    {
      pend = pend->prev; pend->next = 0;
    }
    else
    {
      (pkey->prev)->next = pkey->next;
      (pkey->next)->prev = pkey->prev;
    }
    delete pkey; return true;
  }
  return false;
}
```

Если требуется использовать шаблон `List` для хранения данных не встроенного, а определенного пользователем типа, в его описание необходимо добавить перегрузку операции вывода в поток и сравнения на равенство, а если для его полей используется динамическое выделение памяти, то и операцию присваивания.

При определении синтаксиса шаблона было сказано, что в него, кроме типов, могут передаваться константы. Соответствующим параметром шаблона может быть:

– переменная целого, символьного, булевского или перечислимого типа;

– указатель на объект или указатель на функцию;

– ссылка на объект или ссылка на функцию;

– указатель на элемент класса.

В теле шаблона такие параметры могут применяться в любом месте, где допустимо использовать константное выражение.

В качестве примера создадим шаблон класса, содержащего блок памяти определенной длины и типа:

```
template <class Type, int kol> class Block
{
public:
  Block(){p = new Type [kol];}
  ~Block(){delete [] p;}
  operator Type *();
protected:
  Type * p;
};
template <class Type, int kol>
Block <Type, kol>:: operator Type *()
{
  return p;
}
```

У класса-шаблона могут быть друзья, и шаблоны тоже могут быть друзьями. Класс может быть объявлен внутри шаблона, а шаблон – внутри как класса, так и шаблона. Единственным ограничением является то, что шаблонный класс нельзя объявлять внутри функции. В любом классе, как в обычном, так и в шаблоне, можно объявить метод-шаблон. После создания и отладки шаблоны классов удобно помещать в заголовочные файлы.

9.3 Использование шаблонов классов

Чтобы создать при помощи шаблона конкретный объект конкретного класса, при описании объекта после имени шаблона в угловых скобках перечисляются его фактические параметры:

```
имя_шаблона <фактические параметры> имя_объекта
(параметры_конструктора);
```

Процесс создания конкретного класса из шаблона путем подстановки аргументов называется инстанцированием шаблона. Имя шаблона вместе с фактическими параметрами можно воспринимать как уточненное имя класса. Примеры создания объектов по шаблонам:

```
List <int> List_int;  // список целых чисел
List <double> List_double; // список вещественных чисел
```

```
List <monster> List_monster; // список объектов класса monster
Block <char, 128> buf;       // блок символов
Block <monstr, 100> stado;   // блок объектов класса monster
Pair<int, int> a;            // объявление пары целых
Pair<int, double> b;         // объявление пары "целый, вещественный"
Pair<int, double> b = { 1, 2.1 };  // объявление с инициализацией
Pair<string, Date> d;        // аргументы - пользовательские классы
```

При использовании параметров шаблона по умолчанию список аргументов может оказаться пустым, при этом угловые скобки опускать нельзя:

```
template<class T = char> class String;
String<>* p;
```

Для каждого инстанцированного класса компилятор создает имя, отличающееся и от имени шаблона, и от имен других инстанцированных классов. Тем самым каждый инстанцированный класс определяет отдельный тип. В классе-шаблоне разрешено объявлять статические методы и статические поля, однако следует учесть, что каждый инстанцированный класс обладает собственной копией статических элементов.

После создания объектов с ними можно работать так же, как с объектами обычных классов, например:

```
for (int i = 1; i<10; i++)List_double.add(i*0.08);
List_double.print();
//---------------------------------
for (int i = 1; i<10; i++)List_monster.add(i);
List_monster.print();
//---------------------------------
strcpy(buf, "Очень важное сообщение");
cout << buf << endl;
```

Для упрощения использования шаблонов классов можно применить переименование типов с помощью `typedef`:

```
typedef List <double> Ldbl;
Ldbl List_double;
```

9.4 Явная специализация шаблонов

Специализация шаблонов является одной из нетривиальных возможностей языка C++ и используется в основном при создании библиотек. Предположим, что мы хотим изменить шаблон класса `List` только для параметров типа `int`, тогда необходимо задать явную специализацию шаблона:

```
template<> class List<int> { ... }; // int – аргумент шаблона.
```

Теперь можно писать методы и поля специальной реализации для `int`. Такая специализация обычно называется *полной специализацией* (full

specialization или explicit specialization). Для большинства практических задач большего не требуется.

Согласно ISO стандарту C++, создав специализированный шаблонный класс мы создаем новый шаблонный класс. Специализированный шаблонный класс может содержать методы, поля или объявления типов которых нет в шаблонном классе который мы специализируем. Удобно, когда нужно чтобы метод шаблонного класса работал только для конкретной специализации – достаточно объявить метод только в этой специализации, остальное сделает компилятор:

```
template<> class List<int> {
   void removeAllLessThan(int n);  // метод доступен только для
List<int>
};
```

Из того, что специализированный шаблонный класс это совсем-совсем новый и отдельный класс, следует, что он может иметь отдельные, никак не связанные с неспециализированным шаблонным классом параметры:

```
template< typename T, typename S > class B {};
template< typename U > class B< int, U > {};
```

Такая специализация шаблона, при которой задается новый список параметров и через эти параметры задаются аргументы для специализации называется *частичной специализацией* (partial specialization).

9.5 Достоинства и недостатки шаблонов

Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое можно назвать параметрическим полиморфизмом, обеспечивают безопасное использование типов, в отличие от макросов препроцессора, и являются вкупе с шаблонами функций средством реализации идей обобщенного программирования и метапрограммирования. Однако следует иметь в виду, что эти средства предназначены для грамотного использования и требуют знания многих тонкостей. Программа, использующая шаблоны, содержит код для каждого порожденного типа, что может увеличить размер исполняемого файла. Кроме того, с одними типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию шаблона.

Стандартная библиотека C++ предоставляет большой набор шаблонов для различных способов организации хранения и обработки данных.

10. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Обработка исключений – это механизм, позволяющий двум независимо разработанным программным компонентам взаимодействовать в аномальной ситуации, называемой исключением. В этой главе мы расскажем, как генерировать, или возбуждать, исключение в том месте программы, где имеет место аномалия. Затем мы покажем, как связать `catch`-обработчик исключений с множеством инструкций программы, используя `try`-блок. Потом речь пойдет о спецификации исключений – механизме, с помощью которого можно связать список исключений с объявлением функции, и функция не сможет возбудить никаких других исключений. Закончится эта глава обсуждением решений, принимаемых при проектировании программы, в которой используются исключения.

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить, например: деление на 0, выход за границы массива или истощение свободной памяти. Такие исключения нарушают нормальный ход работы программы, и на них нужно немедленно отреагировать. В C++ имеются встроенные средства для их возбуждения и обработки. С помощью этих средств активизируется механизм, позволяющий двум несвязанным (или независимо разработанным) фрагментам программы обмениваться информацией об исключении.

Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение. Чтобы понять, как это происходит, реализуем по-новому класс `iStack`, используя исключения для извещения об ошибках при работе со стеком. Определение класса:

```
#include
class iStack {
public:
   iStack( int capacity )
         : _stack( capacity ), _top( 0 ) { }

   bool pop( int &top_value );
   bool push( int value );

   bool full();
   bool empty();
   void display();

   int size();
private:
   int _top;
   vector< int > _stack;
};
```

Стек реализован на основе вектора из элементов типа `int`. При создании объекта класса `iStack` его конструктор создает вектор из `int`, размер которого (максимальное число элементов, хранящихся в стеке) задается с помощью начального значения. Например, следующая инструкция создает объект `myStack`, который способен содержать не более 20 элементов типа `int`:

```
iStack myStack(20);
```

При манипуляциях с объектом `myStack` могут возникнуть две ошибки:

– запрашивается операция `pop()`, но стек пуст;

– операция `push()`, но стек полон.

Вызвавшую функцию нужно уведомить об этих ошибках посредством исключений. С чего же начать?

Во-первых, мы должны определить, какие именно исключения могут быть возбуждены. В C++ они чаще всего реализуются с помощью классов. Мы определим два из них, чтобы использовать их как исключения для класса `iStack`. Эти определения мы поместим в заголовочный файл stackExcp.h:

```
// stackExcp.h
class popOnEmpty { /* ... */ };
class pushOnFull { /* ... */ };
```

Затем надо изменить определения функций-членов `pop()` и `push()` так, чтобы они возбуждали эти исключения. Для этого предназначена инструкция `throw`, которая во многих отношениях напоминает `return`. Она состоит из ключевого слова `throw`, за которым следует выражение того же типа, что и тип возбуждаемого исключения. Как выглядит инструкция `throw` для функции `pop()`? Попробуем такой вариант:

```
// увы, это не совсем правильно
throw popOnEmpty;
```

К сожалению, так нельзя. Исключение – это объект, и функция `pop()` должна генерировать объект класса соответствующего типа. Выражение в инструкции `throw` не может быть просто типом. Для создания нужного объекта необходимо вызвать конструктор класса. Инструкция `throw` для функции `pop()` будет выглядеть так:

```
// инструкция является вызовом конструктора
throw popOnEmpty();
```

Эта инструкция создает объект исключения типа `popOnEmpty`.

Напомним, что функции-члены `pop()` и `push()` были определены как возвращающие значение типа `bool`: `true` означало, что операция завершилась успешно, а `false` – что произошла ошибка. Поскольку теперь для извещения о неудаче `pop()` и `push()` используют исключения, возвращать значение необязательно. Поэтому мы будем считать, что эти функции-члены имеют тип `void`:

```
class iStack
{
public:
   // ...
   // больше не возвращают значения
   void pop( int &value );
   void push( int value );
private:
   // ...
};
```

Теперь функции, пользующиеся нашим классом `iStack`, будут предполагать, что все хорошо, если только не возбуждено исключение; им больше не надо проверять возвращенное значение, чтобы узнать, как завершилась операция. В двух следующих разделах мы покажем, как определить функцию для обработки исключений, а сейчас представим новые реализации функций-членов `pop()` и `push()` класса `iStack`:

```
#include "stackExcp.h"

void iStack::pop( int &top_value )
{
   if ( empty() )
      throw popOnEmpty();

   top_value = _stack[ --_top ];

   cout << "iStack::pop(): " << top_value " endl;
}

void iStack::push( int value )
{
   cout << "iStack::push( " << value << " )\n";

   if ( full() )
      throw pushOnFull( value );

   _stack[ _top++ ] = value;
}
```

Хотя исключения чаще всего представляют собой объекты типа класса, инструкция `throw` может генерировать объекты любого типа. Например, функция `mathFunc()` в следующем примере возбуждает исключение в виде объекта-перечисления . Это корректный код C++:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

int mathFunc( int i ) {
   if ( i == 0 )
      throw zeroOp;  // исключение в виде объекта-перечисления

   // в противном случае продолжается нормальная обработка
}
```

В нашей программе тестируется определенный в предыдущем разделе класс `iStack` и его функции-члены `pop()` и `push()`. Выполняется 50 итераций цикла `for`. На каждой итерации в стек помещается значение, кратное 3: 3, 6, 9 и т.д. Если значение кратно 4 (4, 8, 12...), то выводится текущее содержимое стека, а если кратно 10 (10, 20, 30...), то с вершины снимается один элемент, после чего содержимое стека выводится снова. Как нужно изменить функцию `main()`, чтобы она обрабатывала исключения, возбуждаемые функциями-членами класса `iStack`?

```
#include
#include "iStack.h"

int main() {
   iStack stack( 32 );

   stack.display();
   for ( int ix = 1; ix < 51; ++ix )
   {
        if ( ix % 3 == 0 )
          stack.push( ix );

        if ( ix % 4 == 0 )
           stack.display();

        if ( ix % 10  == 0 ) {
           int dummy;
           stack.pop( dummy );
           stack.display();
        }
   }
   return 0;
}
```

Инструкции, которые могут возбуждать исключения, должны быть заключены в `try`-блок. Такой блок начинается с ключевого слова `try`, за которым идет последовательность инструкций, заключенная в фигурные скобки, а после этого – список обработчиков, называемых `catch`-предложениями. `Try`-блок группирует инструкции программы и ассоциирует с ними обработчики исключений. Куда нужно поместить `try`-блоки в функции `main()`, чтобы были обработаны исключения `popOnEmpty` и `pushOnFull`?

```
for ( int ix = 1; ix < 51; ++ix ) {
```

```
    try {   // try-блок для исключений pushOnFull
        if ( ix % 3 == 0 )
          stack.push( ix );
    }
    catch ( pusOnFull ) { ... }

    if ( ix % 4 == 0 )
       stack.display();
    try {   // try-блок для исключений popOnEmpty
        if ( ix % 10  == 0 ) {
           int dummy;
           stack.pop( dummy );
           stack.display();
        }
    }
    catch ( popOnEmpty ) { ... }
}
```

В таком виде программа выполняется корректно. Однако обработка исключений в ней перемежается с кодом, использующимся при нормальных обстоятельствах, а такая организация несовершенна. В конце концов, исключения – это аномальные ситуации, возникающие только в особых случаях. Желательно отделить код для обработки аномалий от кода, реализующего операции со стеком. Мы полагаем, что показанная ниже схема облегчает чтение и сопровождение программы:

```
try {
   for ( int ix = 1; ix < 51; ++ix )
   {
      if ( ix % 3 == 0 )
         stack.push( ix );

      if ( ix % 4 == 0 )
          stack.display();

      if ( ix % 10  == 0 ) {
          int dummy;
          stack.pop( dummy );
          stack.display();
      }
   }
}
catch ( pushOnFull ) { ... }
catch ( popOnEmpty ) { ... }
```

С `try`-блоком ассоциированы два `catch`-предложения, которые могут обработать исключения `pushOnFull` и `popOnEmpty`, возбуждаемые функциями-членами `push()` и `pop()` внутри этого блока. Каждый `catch`-обработчик определяет тип «своего» исключения. Код для обработки исключения помещается внутрь составной инструкции (между фигурными скобками), которая является частью `catch`-обработчика. (Подробнее `catch`-предложения мы рассмотрим в следующем разделе.)

Исполнение программы может пойти по одному из следующих путей:

– если исключение не возбуждено, то выполняется код внутри try-блока, а ассоциированные с ним обработчики игнорируются. Функция main() возвращает 0;

– если функция-член push(), вызванная из первой инструкции if внутри цикла for, возбуждает исключение, то вторая и третья инструкции if игнорируются, управление покидает цикл for и try-блок, и выполняется обработчик исключений типа pushOnFull;

– если функция-член pop(), вызванная из третьей инструкции if внутри цикла for, возбуждает исключение, то вызов display() игнорируется, управление покидает цикл for и try-блок, и выполняется обработчик исключений типа popOnEmpty.

Когда возбуждается исключение, пропускаются все инструкции, следующие за той, где оно было возбуждено. Исполнение программы возобновляется в catch-обработчике этого исключения. Если такого обработчика не существует, то управление передается в функцию terminate(), определенную в стандартной библиотеке C++. Try-блок может содержать любую инструкцию языка C++: как выражения, так и объявления. Он вводит локальную область видимости, так что объявленные внутри него переменные недоступны вне этого блока, в том числе и в catch-обработчиках. Например, функцию main() можно переписать так, что объявление переменной stack окажется в try-блоке. В таком случае обращаться к этой переменной в catch-обработчиках нельзя:

```
int main() {
   try {
      iStack stack( 32 );    // правильно: объявление внутри try-блока

      stack.display();
      for ( int ix = 1; ix < 51; ++ix )
      {
          // то же, что и раньше
      }
   }
   catch ( pushOnFull ) {
      // здесь к переменной stack обращаться нельзя
   }
   catch ( popOnEmpty ) {
      // здесь к переменной stack обращаться нельзя
   }

   // и здесь к переменной stack обращаться нельзя
   return 0;
}
```

Можно объявить функцию так, что все ее тело будет заключено в try-блок. При этом не обязательно помещать try-блок внутрь определения

функции, удобнее заключить ее тело в функциональный try-блок. Такая организация поддерживает наиболее чистое разделение кода для нормальной обработки и кода для обработки исключений. Например:

```
int main() {
   try {
      iStack stack( 32 );    // правильно: объявление внутри try-блока
      stack.display();
      for ( int ix = 1; ix < 51; ++ix )
      {
          // то же, что и раньше
      }

      return 0;
   }
   catch ( pushOnFull ) {
      // здесь к переменной stack обращаться нельзя
   }
   catch ( popOnEmpty ) {
      // здесь к переменной stack обращаться нельзя
   }
}
```

10.1 Перехват исключений

В языке C++ исключения обрабатываются в предложениях catch. Когда какая-то инструкция внутри try-блока возбуждает исключение, то просматривается список последующих предложений catch в поисках такого, который может его обработать.

Catch-обработчик состоит из трех частей: ключевого слова catch, объявления одного типа или одного объекта, заключенного в круглые скобки (оно называется объявлением исключения), и составной инструкции. Если для обработки исключения выбрано некоторое catch-предложение, то выполняется эта составная инструкция. Рассмотрим catch-обработчики исключений pushOnFull и popOnEmpty в функции main() более подробно:

```
catch ( pushOnFull ) {
   cerr << "trying to push value on a full stack\n";
   return errorCode88;
}
catch ( popOnEmpty ) {
   cerr << "trying to pop a value on an empty stack\n";
   return errorCode89;
}
```

В обоих catch-обработчиках есть объявление типа класса; в первом это pushOnFull, а во втором – popOnEmpty. Для обработки исключения выбирается тот обработчик, для которого типы в объявлении исключения и в возбужденном исключении совпадают. Например, когда функция-член pop()

класса `iStack` возбуждает исключение `popOnEmpty`, то управление попадает во второй обработчик. После вывода сообщения об ошибке в `cerr`, функция `main()` возвращает код `errorCode89`.

А если `catch`-обработчики не содержат инструкции `return`, с какого места будет продолжено выполнение программы? После завершения обработчика выполнение возобновляется с инструкции, идущей за последним `catch`-обработчиком в списке. В нашем примере оно продолжается с инструкции `return` в функции `main()`. После того как `catch`-обработчик `popOnEmpty` выведет сообщение об ошибке, `main()` вернет 0.

```
int main() {
   iStack stack( 32 );

   try {
      stack.display();
      for ( int x = 1; ix < 51; ++ix )
      {
         // то же, что и раньше
      }
   }
   catch ( pushOnFull ) {
      cerr << "trying to push value on a full stack\n";
   }
   catch ( popOnEmpty ) {
      cerr << "trying to pop a value on an empty stack\n";
   }

   // исполнение продолжается отсюда
   return 0;
}
```

Говорят, что механизм обработки исключений в C++ невозвратный: после того как исключение обработано, управление не возобновляется с того места, где оно было возбуждено. В нашем примере управление не возвращается в функцию-член `pop()`, возбудившую исключение.

10.2 Объекты-исключения

Объявлением исключения в `catch`-обработчике могут быть объявления типа или объекта. В каких случаях это следует делать? Тогда, когда необходимо получить значение или как-то манипулировать объектом, созданным в выражении `throw`. Если классы исключений спроектированы так, что в объектах-исключениях при возбуждении сохраняется некоторая информация и если в объявлении исключения фигурирует такой объект, то инструкции внутри `catch`-обработчика могут обращаться к информации, сохраненной в объекте выражением `throw`.

Изменим реализацию класса исключения pushOnFull, сохранив в объекте-исключении то значение, которое не удалось поместить в стек. Catch-обработчик, сообщая об ошибке, теперь будет выводить его в cerr. Для этого мы сначала модифицируем определение типа класса pushOnFull следующим образом:

```
// новый класс исключения:
// он сохраняет значение, которое не удалось поместить в стек
class pushOnFull {
public:
   pushOnFull( int i ) : _value( i ) { }
   int value { return _value; }
private:
   int _value;
};
```

Новый закрытый член _value содержит число, которое не удалось поместить в стек. Конструктор принимает значение типа int и сохраняет его в члене _data. Вот как вызывается этот конструктор для сохранения значения из выражения throw:

```
void iStack::push( int value )
{
   if ( full() )
      // значение, сохраняемое в объекте-исключении
      throw pushOnFull( value );

   // ...
}
```

У класса pushOnFull появилась также новая функция-член value(), которую можно использовать в catch-обработчике для вывода хранящегося в объекте-исключении значения:

```
catch ( pushOnFull eObj ) {
   cerr << "trying to push value << "eObj.value()
        << "on a full stack\n";
}
```

Обратите внимание, что в объявлении исключения в catch-обработчике фигурирует объект eObj, с помощью которого вызывается функция-член value() класса pushOnFull.

Объект-исключение всегда создается в точке возбуждения, даже если выражение throw – это не вызов конструктора и, на первый взгляд, не должно создавать объекта.

Например:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;
```

```
int mathFunc( int i ) {
   if ( i == 0 ) {
      state = zeroOp;
      throw state;    // создан объект-исключение
   }
   // иначе продолжается обычная обработка
}
```

В этом примере объект `state` не используется в качестве объекта-исключения. Вместо этого выражением `throw` создается объект-исключение типа `EHstate`, который инициализируется значением глобального объекта `state`. Как программа может различить их? Для ответа на этот вопрос мы должны присмотреться к объявлению исключения в `catch`-обработчике более внимательно.

Это объявление ведет себя почти так же, как объявление формального параметра. Если при входе в `catch`-обработчик исключения выясняется, что в нем объявлен объект, то он инициализируется копией объекта-исключения. Например, следующая функция `calculate()` вызывает определенную выше `mathFunc()`. При входе в `catch`-обработчик внутри `calculate()` объект `eObj` инициализируется копией объекта-исключения, созданного выражением `throw`.

```
void calculate( int op ) {
   try {
      mathFunc( op );
   }
   catch ( EHstate eObj ) {
      // eObj - копия сгенерированного объекта-исключения
   }
}
```

Объявление исключения в этом примере напоминает передачу параметра по значению. Объект `eObj` инициализируется значением объекта-исключения точно так же, как переданный по значению формальный параметр функции – значением соответствующего фактического аргумента.

Как и в случае параметров функции, в объявлении исключения может фигурировать ссылка. Тогда `catch`-обработчик будет напрямую ссылаться на объект-исключение, сгенерированный выражением `throw`, а не создавать его локальную копию:

```
void calculate( int op ) {
try {
      mathFunc( op );
   }
   catch ( EHstate &eObj ) {
      // eObj ссылается на сгенерированный объект-исключение
   }
}
```

Для предотвращения ненужного копирования больших объектов применять ссылки следует не только в объявлениях параметров типа класса, но и в объявлениях исключений того же типа.

В последнем случае `catch`-обработчик сможет модифицировать объект-исключение. Однако переменные, определенные в выражении `throw`, остаются без изменения. Например, модификация `eObj` внутри `catch`-обработчика не затрагивает глобальную переменную `state`, установленную в выражении `throw`:

```
void calculate( int op ) {
   try {
      mathFunc( op );
   }
   catch ( EHstate &eObj ) {
      // исправить ошибку, вызвавшую исключение
      eObj = noErr;  // глобальная переменная state не изменилась
   }
}
```

`Catch`-обработчик переустанавливает `eObj` в `noErr` после исправления ошибки, вызвавшей исключение. Поскольку `eObj` – это ссылка, можно ожидать, что присваивание модифицирует глобальную переменную `state`. Однако изменяется лишь объект-исключение, созданный в выражении `throw`, поэтому модификация `eObj` не затрагивает `state`.

10.3 Раскрутка стека

Поиск `catch`-обработчикадля возбужденного исключения происходит следующим образом. Когда выражение `throw` находится в `try`-блоке, все ассоциированные с ним предложения `catch` исследуются с точки зрения того, могут ли они обработать исключение. Если подходящее предложение `catch` найдено, то исключение обрабатывается. В противном случае поиск продолжается в вызывающей функции. Предположим, что вызов функции, выполнение которой прекратилось в результате исключения, погружен в `try`-блок; в такой ситуации исследуются все предложения `catch`, ассоциированные с этим блоком. Если один из них может обработать исключение, то процесс заканчивается. В противном случае переходим к следующей по порядку вызывающей функции. Этот поиск последовательно проводится во всей цепочке вложенных вызовов. Как только будет найдено подходящее предложение, управление передается в соответствующий обработчик.

В нашем примере первая функция, для которой нужен `catch`-обработчик, – это функция-член `pop()` класса `iStack`. Поскольку выражение `throw` внутри `pop()` не находится в `try`-блоке, то программа покидает `pop()`, не

обработав исключение. Следующей рассматривается функция, вызвавшая `pop()`, то есть `main()`. Вызов `pop()` внутри `main()` находится в `try`-блоке, и далее исследуется, может ли хотя бы одно ассоциированное с ним предложение `catch` обработать исключение. Поскольку обработчик исключения `popOnEmpty` имеется, то управление попадает в него.

Процесс, в результате которого программа последовательно покидает составные инструкции и определения функций в поисках предложения `catch`, способного обработать возникшее исключение, называется раскруткой стека. По мере раскрутки прекращают существование локальные объекты, объявленные в составных инструкциях и определениях функций, из которых произошел выход. C++ гарантирует, что во время описанного процесса вызываются деструкторы локальных объектов классов, хотя они исчезают из-за возбужденного исключения.

Если в программе нет предложения `catch`, способного обработать исключение, оно остается необработанным. Но исключение – это настолько серьезная ошибка, что программа не может продолжать выполнение. Поэтому, если обработчик не найден, вызывается функция `terminate()` из стандартной библиотеки C++. По умолчанию `terminate()` активизирует функцию `abort()`, которая аномально завершает программу. (В большинстве ситуаций вызов `abort()` оказывается вполне приемлемым решением. Однако иногда необходимо переопределить действия, выполняемые функцией `terminate()`.

Вы уже, наверное, заметили, что обработка исключений и вызов функции во многом похожи. Выражение `throw` ведет себя аналогично вызову, а предложение `catch` чем-то напоминает определение функции. Основная разница между этими двумя механизмами заключается в том, что информация, необходимая для вызова функции, доступна во время компиляции, а для обработки исключений – нет. Обработка исключений в C++ требует языковой поддержки во время выполнения. Например, для обычного вызова функции компилятору в точке активизации уже известно, какая из перегруженных функций будет вызвана. При обработке же исключения компилятор не знает, в какой функции находится `catch`-обработчик и откуда возобновится выполнение программы. Функция `terminate()` предоставляет механизм времени выполнения, который извещает пользователя о том, что подходящего обработчика не нашлось.

10.4 Повторное возбуждение исключений

Может оказаться так, что в одном предложении `catch` не удалось полностью обработать исключение. Выполнив некоторые корректирующие действия, `catch`-обработчик может решить, что дальнейшую обработку следует поручить функции, расположенной «выше» в цепочке вызовов. Передать исключение другому `catch`-обработчику можно с помощью повторного возбуждения исключения. Для этой цели в языке предусмотрена конструкция `throw`, которая вновь генерирует объект-исключение. Повторное возбуждение возможно только внутри составной инструкции, являющейся частью `catch`-обработчика:

```
catch ( exception eObj ) {
   if ( canHandle( eObj ) )
      // обработать исключение
      return;
   else
      // повторно возбудить исключение, чтобы его перехватил другой
      // catch-обработчик
      throw;
}
```

При повторном возбуждении новый объект-исключение не создается. Это имеет значение, если `catch`-обработчик модифицирует объект, прежде чем возбудить исключение повторно. В следующем фрагменте исходный объект-исключение не изменяется. Почему?

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

void calculate( int op ) {
   try {
      // исключение, возбужденное mathFunc(), имеет значение zeroOp
      mathFunc( op );
   }
   catch ( EHstate eObj ) {
      // что-то исправить

      // пытаемся модифицировать объект-исключение
      eObj = severeErr;

      // предполагалось, что повторно возбужденное исключение будет
      // иметь значение severeErr
      throw;
   }
}
```

Так как `eObj` не является ссылкой, то `catch`-обработчик получает копию объекта-исключения, так что любые модификации `eObj` относятся к локальной копии и не отражаются на исходном объекте-исключении, передаваемом при повторном возбуждении. Таким образом, переданный далее объект по-прежнему имеет тип `zeroOp`.

Чтобы модифицировать исходный объект-исключение, в объявлении исключения внутри `catch`-обработчика должна фигурировать ссылка:

```
catch ( EHstate &eObj ) {
   // модифицируем объект-исключение
   eObj = severeErr;

   // повторно возбужденное исключение имеет значение severeErr
   throw;
}
```

Теперь `eObj` ссылается на объект-исключение, созданный выражением `throw`, так что все изменения относятся непосредственно к исходному объекту. Поэтому при повторном возбуждении исключения далее передается модифицированный объект.

Таким образом, другая причина для объявления ссылки в `catch`-обработчике заключается в том, что сделанные внутри обработчика модификации объекта-исключения в таком случае будут видны при повторном возбуждении исключения

10.5 Обработка исключений в C

В эпоху расцвета процедурного программирования синтаксис работы с ошибками был тривиален и основывался на том, что вернула функция. Если функция возвращала `TRUE` – все хорошо, если же `FALSE` – то произошла ошибка. При этом сразу выделились два подхода к работе с ошибками:

– Подход *два в одном* – функция возвращает `FALSE` или нулевой указатель как для ожидаемой, так и для неожиданной ошибки. Такой подход как правило применялся в API общего назначения и коде пользовательских программ, когда большую часть ошибок можно было смело считать фатальными и падать. Для тех редких случаев когда делить было все же нужно использовалась некая дополнительная машинерия вида `GetLastError()`. Фрагмент кода того времени, копирующего данные из одного файла в другой и возвращающего ошибку в случае возникновения любых проблем:

```
BOOL Copy( CHAR* sname, CHAR* dname ) {
   FILE *sfile = 0, *dfile = 0;
   void* mem = 0;
   UINT32 size = 0, written = 0;
   BOOL ret = FALSE;

   sfile = fopen( sname, "rb" );
   if( ! sfile ) goto cleanup;

   dfile = fopen( dname, "wb" );
```

```
   if( ! dfile ) goto cleanup;

   mem = malloc( F_CHUNK_SIZE );
   if( ! mem ) goto cleanup;

   do
   {
      size = fread( sfile, mem, F_CHUNK_SIZE );
      written = fwrite( dfile, mem, size );
      if( size != written ) goto cleanup;
   }
   while( size )
   ret = TRUE;

cleanup: // Аналог деструктора.
   if( sfile) fclose( sfile );
   if( dfile) fclose( dfile );
   if( mem ) free( mem );

   return ret; // Ожидаемая ошибка.
}
```

– Подход *разделения ошибок*, при котором функция возвращает `FALSE` в случае неожиданной ошибки, а ожидаемую ошибку возвращает отдельным возвращаемым значением (в примере это error), если нужно. Такой подход применялся в более надежном коде, например apache, и подразумевал разделение на ожидаемые ошибки (файл не получилось открыть потому что его нет) и неожиданные (файл не получилось открыть потому, что закончилась память и не получилось выделить 20 байт чтобы скопировать строку с именем). Фрагмент того же код, но уже разделяющего неожиданную ошибку (возврат `FALSE`) и ожидаемую (возврат `HANDLE`).

```
BOOL Copy( CHAR* sname, CHAR* dname, OUT HANDLE* error ) {
   HANDLE sfile = 0, dfile = 0, data = 0;
   UINT32 size = 0;
   ENSURE( PoolAlloc() ); // Обработка неожиданной ошибки.

   ENSURE( FileOpen( sname, OUT& sfile, OUT error ) );
   REQUIRE( SUCCESS( error ) ); // Обработка ожидаемой ошибки.

   ENSURE( FileOpen( dname, OUT& dfile, OUT error ) );
   REQUIRE( SUCCESS( error ) );

   ENSURE( MemAlloc( OUT& data ) );
   REQUIRE( SUCCESS( error ) );

   do
   {
      ENSURE( FileRead( sfile, F_CHUNK_SIZE, OUT& data, OUT error ) );
      REQUIRE( SUCCESS( error ) );
      ENSURE( FileWrite( dfile, & data ) );
      REQUIRE( SUCCESS( error ) );
      ENSURE( MemGetSize( OUT& size ) )
   }
   while( size );
```

```
   ENSURE( PoolFree() ); // Пул обеспечивает аналог деструкторов и RAII.
   return TRUE;
}
```

Через некоторое время разработчики заметили, что большинство успешных решений использует ООП и решили, что неплохо бы его вынести в синтаксис языка, дабы писать больше кода по делу и меньше – повторяющегося кода для поддержки архитектуры.

Давайте возьмем код выше и посмотрим, как он трансформировался после добавления ООП в синтаксис языков программирования. Конструирование и уничтожение объектов (`fopen`, `fclose`) стало конструкторами и деструкторами. Переброс неожиданной ошибки (`BOOL ret` в первом примере, макрос `ENSURE` во втором) однозначно стал исключением.

А вот с ожидаемой ошибкой случилось самое интересное – случился выбор. Можно было использовать возвращаемое значение – теперь, когда заботу о неожиданных ошибках взяли на себя исключения, возвращаемое значение снова стало в полном распоряжении программиста. А можно было использовать исключения другого типа – если функции копирования файлов самой не нужно обрабатывать ожидаемые ошибки то логично вместо `if` и `REQUIRE` просто ничего не делать – и оба типа ошибок уйдут вверх по стеку. Соответственно, у программистов снова получилось два варианта:

– Подход *только исключения* – ожидаемые и неожиданные ошибки – это разные типы исключений.

```
void Copy( string sname, string dname ) {
   file source( sname );
   file destination( sname );
   source.open( "rb" );
   destination.open( "wb" );
   data bytes;
   do
   {
      bytes = source.read( F_CHUNK_SIZE );
      destination.write( bytes )
   }
   while( bytes.size() )
}
```

– *Комбинированный подход* – использование исключений для неожиданных ошибок и кодов возврата / `nullable` типов для ожидаемых:

```
bool Copy( string sname, string dname ) {
   file source( sname );
   file destination( sname );
   if( ! source.open( "rb" ) || ! destination.open( "wb" ) ) return
false;
```

```
    data bytes;
    do
    {
       bytes = source.read( F_CHUNK_SIZE );
       if( bytes.isValid() )
       {
          if( ! destination.write( bytes ) ) return false;
       }
    }
    while( bytes.isValid() && bytes.size() )
}
```

Итак, если внимательно посмотреть на два приведенных выше фрагмента кода то становится не совсем понятно почему выжил второй. Кода в нем объективно больше. Выглядит менее красиво. Если функция возвращает объект – то использовать коды возврата совсем неудобно. Вопрос – почему коды возврата вообще выжили в языках с поддержкой объектно-ориентированного программирования и исключений на уровне синтаксиса? Что я могу по этому поводу сказать:

Первые реализации исключений, особенно в C++, были не очень удобны для ежедневного использования. Например, бросание исключения во время обработки другого исключения приводил к завершению программы. Или же бросание исключения в конструкторе приводило к тому, что деструктор не вызывался.

Разработчикам API забыли объяснить для чего нужны исключения. В результате первое время не было даже деления на ожидаемые (`checked`) и неожиданные (`unchecked`), а API комбинировали как исключения, так и коды возврата.

В большинстве языков для исключений забыли добавить семантику «игнорировать ожидаемую ошибку». В результате на практике код, использующий исключения как для ожидаемых так и для неожиданных ошибок, с невероятной скоростью обрастал `try` и `catch` везде, где только можно.

10.6 Исключительные ситуации в конструкторах и деструкторах

Конструкторы не могут возвращать коды ошибок, соответственно исключения – это единственный метод, чтобы понять, что в конструкторе что-то пошло не так. Однако, необходимо правильно обрабатывать исключительные ситуации в таких случаях. Особенно если в конструкторе формируются объекты в динамической памяти.

Пример исключительной ситуации в конструкторе:

```
class Test {
public:
```

```
   Test() {
      std::cout << "Test::Test()" << std::endl;
      // Здесь, в соответствии с RAII, захватили ресурсы
      if ( 1 ) {
         throw std::runtime_error( "AAAAAAAA" );
      } else {}
   }
   ~Test() {
      std::cout << "Test::~Test()" << std::endl;
      // А здесь мы освобождаем те самые важные ресурсы...
   }
};

int main() {
   Test* t = 0;
   try {
      t = new Test();    // Вроде бы создали...
   } catch ( const std::exception& exc ) {
        std::cout << exc.what() << std::endl;
   }
   delete t;     // Удалили? Ну-ну...
   return 0;
}
```

Не стоит пытаться передавать исключения за тело конструктора, т.к. иначе, даже если мы не забываем вызвать `delete`, он не будет вызывать деструктор неправильно созданного объекта. В этом случае, никакие ресурсы, которые были инициализированы в конструкторе до вызова исключения (связи с БД, открытые файлы, и т.п.) уже никогда не будут освобождены (если они создавались в динамической памяти).

Вместо указателей на объекты в динамической памяти можно использовать один из множества типов «умных указателей», которые удаляются автоматически при выходе из области видимости (как локальные объекты):

```
// С указателями на объекты в
// динамической памяти:
class Cnt {
private:
   X *xa;
   X *xb;
public:
   Cnt(int a, int b) {
      cout << "Cnt::Cnt" << endl;
      xa = new X(a);
      xb = new X(b);
   }
   ~Cnt() {
      cout << "Cnt::~Cnt" << endl;
      delete xa;
      delete xb;
  }
};
```

```
// С умными указателями:
class Cnt {
private:
   auto_ptr<X> ia;
   auto_ptr<X> ib;
public:
   Cnt(int a, int b) : ia(new
X(a)), ib(new X(b)) {
      cout << "Cnt::Cnt" << endl;
   }
   ~Cnt() {
      cout << "Cnt::~Cnt" << endl;
   }
};
```

`std::auto_ptr<>` – реализует семантику владения. Так называемое разрушающее копирование – при присваивании, объект передается от одного указателя другому, удаляясь у первого, чтобы не удалять объект дважды при выходе из области видимости.

Пример исключительной ситуации в деструкторе:

```
class test
{
public:
    test() { }
    ~test(){
        throw std::runtime_error("Game over!");
    }
};

int main() {
    try {
        test t;
        throw std::runtime_error("Error!");
    }
    catch(std::exception const&)
    { }
    return 0;
}
```

Когда исключение покидает блок, все локальные объекты, созданные в этом блоке, уничтожаются. Если деструктор объекта, уничтожаемого во время развертки стека, генерирует исключение, то программа будет завершена досрочно, и ее уже ничего не спасет – вызывается функция `terminate`.

Исключение при раскрутке стека всегда будет инициализировать функцию `terminate()` – обрабатывайте все ошибки деструктора внутри деструктора! Ни в коем случае не инициализируйте исключение!

11. ИДЕНТИФИКАЦИЯ ТИПОВ ВО ВРЕМЯ ВЫПОЛНЕНИЯ

RTTI – идентификация типов во время выполнения (Run-time Type Identification), позволяет программам, которые манипулируют объектами через указатели или ссылки на базовые классы, получить истинный производный тип адресуемого объекта. Для поддержки RTTI в языке C++ есть два оператора:

– оператор `dynamic_cast` поддерживает преобразования типов во время выполнения, обеспечивая безопасную навигацию по иерархии классов. Он позволяет трансформировать указатель на базовый класс в указатель на производный от него, а также преобразовать l-значение, ссылающееся на базовый класс, в ссылку на производный, но только в том случае, если это завершится успешно;

– оператор `typeid` позволяет получить фактический производный тип объекта, адресованного указателем или ссылкой.

Однако для получения информации о типе производного класса операнд любого из операторов `dynamic_cast` или `typeid` должен иметь тип класса, в котором есть хотя бы одна виртуальная функция. Таким образом, операторы RTTI – это события времени выполнения для классов с виртуальными функциями и события времени компиляции для всех остальных типов. В данном разделе мы более подробно познакомимся с их возможностями. Использование RTTI оказывается необходимым при реализации таких приложений, как отладчики или объектные базы данных, когда тип объектов, которыми манипулирует программа, становится известен только во время выполнения путем исследования RTTI-информации, хранящейся вместе с типами объектов. Однако лучше пользоваться статической системой типов C++, поскольку она безопаснее и эффективнее.

11.1 Оператор dynamic_cast

Оператор `dynamic_cast` можно применять для преобразования указателя, ссылающегося на объект типа класса в указатель на тип класса из той же иерархии. Его также используют для трансформации l-значения объекта типа класса в ссылку на тип класса из той же иерархии. Приведение типов с помощью оператора `dynamic_cast`, в отличие от других имеющихся в C++ способов, осуществляется во время выполнения программы. Если указатель или l-значение не могут быть преобразованы в целевой тип, то `dynamic_cast` завершается неудачно. В случае приведения типа указателя признаком неудачи служит возврат нулевого значения. Если же l-значение

нельзя трансформировать в ссылочный тип, возбуждается исключение. Ниже мы приведем примеры неудачного выполнения этого оператора.

Прежде чем перейти к более детальному рассмотрению `dynamic_cast`, посмотрим, зачем его нужно применять. Предположим, что в программе используется библиотека классов для представления различных категорий служащих компании. Входящие в иерархию классы поддерживают функции-члены для вычисления зарплаты:

```
class employee {
public:
   virtual int salary();
};

class manager : public employee {
public:
   int salary();
};

class programmer : public employee {
public:
   int salary();
};

void company::payroll( employee *pe ) {
   // используется pe->salary()
}
```

В компании есть разные категории служащих. Параметром функции-члена `payroll()` класса `company` является указатель на объект `employee`, который может адресовать один из типов `manager` или `programmer`. Поскольку `payroll()` обращается к виртуальной функции-члену `salary()`, то вызывается подходящая замещающая функция, определенная в классе `manager` или `programmer`, в зависимости от того, какой объект адресован указателем.

Допустим, класс `employee` перестал удовлетворять нашим потребностям, и мы хотим его модифицировать, добавив еще одну функцию-член `bonus()`, используемую совместно с `salary()` при расчете платежной ведомости. Для этого нужно включить новую функцию-член в классы, составляющие иерархию `employee`:

```
class employee {
public:
   virtual int salary();
   virtual int bonus();
};

class manager : public employee {
public:
   int salary();
};
```

```
class programmer : public employee {
public:
   int salary();
   int bonus();
};

void company::payroll( employee *pe ) {
   // eniieucoaony pe->salary() e pe->bonus()
}
```

Если параметр `pe` функции `payroll()` указывает на объект типа `manager`, то вызывается виртуальная функция-член `bonus()` из базового класса `employee`, поскольку в классе `manager` она не замещена. Если же `pe` указывает на объект типа `programmer`, то вызывается виртуальная функция-член `bonus()` из класса `programmer`.

После добавления новых виртуальных функций в иерархию классов придется перекомпилировать все функции-члены. Добавить `bonus()` можно, если у нас есть доступ к исходным текстам функций-членов в классах `employee`, `manager` и `programmer`. Однако если иерархия была получена от независимого поставщика, то не исключено, что в нашем распоряжении имеются только заголовочные файлы, описывающие интерфейс библиотечных классов и объектные файлы с их реализацией, а исходные тексты функций-членов недоступны. В таком случае перекомпиляция всей иерархии невозможна.

Если мы хотим расширить функциональность библиотеки классов, не добавляя новые виртуальные функции-члены, можно воспользоваться оператором `dynamic_cast`.

Этот оператор применяется для получения указателя на производный класс, чтобы иметь возможность работать с теми его элементами, которые по-другому не доступны. Предположим, что мы расширяем библиотеку за счет добавления новой функции-члена `bonus()` в класс `programmer`. Ее объявление можно включить в определение `programmer`, находящееся в заголовочном файле, а саму функцию определить в одном из своих исходных файлов:

```
class employee {
public:
   virtual int salary();
};

class manager : public employee {
public:
   int salary();
};

class programmer : public employee {
```

```
public:
   int salary();
   int bonus();
};
```

Напомним, что `payroll()` принимает в качестве параметра указатель на базовый класс `employee`. Мы можем применить оператор `dynamic_cast` для получения указателя на производный `programmer` и воспользоваться им для вызова функции-члена `bonus()`:

```
void company::payroll( employee *pe )
{
   programmer *pm = dynamic_cast< programmer* >( pe );

   // anee pe oeacuaaao ia iauaeo oeia programmer,
   // oi dynamic_cast auiieieony oniaoii e pm aoaao
   // oeacuaaou ia ia?aei iauaeoa programmer
   if ( pm ) {
      // eniieuciaaou pm aey auciaa programmer::bonus()
   }
   // anee pe ia oeacuaaao ia iauaeo oeia programmer,
   // oi dynamic_cast auiieieony iaoaa?ii
   // e pm aoaao niaa??aou 0
   else {
      // eniieuciaaou ooieoee-?eaiu eeanna employee
   }
}
```

Оператор `dynamic_cast< programmer* >( pe )` приводит свой операнд `pe` к типу `programmer*`. Преобразование будет успешным, если `pe` ссылается на объект типа `programmer`, и неудачным в противном случае: тогда результатом `dynamic_cast` будет 0.

Таким образом, оператор `dynamic_cast` осуществляет сразу две операции. Он проверяет, выполнимо ли запрошенное приведение, и если это так, выполняет его. Проверка производится во время работы программы. `dynamic_cast` безопаснее, чем другие операции приведения типов в C++, поскольку проверяет возможность корректного преобразования.

Если в предыдущем примере `pe` действительно указывает на объект типа `programmer`, то операция `dynamic_cast` завершится успешно и `pm` будет инициализирован указателем на объект типа `programmer`. В противном случае `pm` получит значение 0. Проверив значение `pm`, функция `company::payroll()` может узнать, указывает ли `pm` на объект `programmer`. Если это так, то она вызывает функцию-член `programmer::bonus()` для вычисления премии программисту. Если же `dynamic_cast` завершается неудачно, то `pe` указывает на объект типа `manager`, а значит, необходимо применить более общий алгоритм расчета, не использующий новую функцию-член `programmer::bonus()`.

Оператор dynamic_cast употребляется для безопасного приведения указателя на базовый класс к указателю на производный. Такую операцию часто называют понижающим приведением (downcasting). Она применяется, когда необходимо воспользоваться особенностями производного класса, отсутствующими в базовом. Манипулирование объектами производного класса с помощью указателей на базовый обычно происходит автоматически, с помощью виртуальных функций. Однако иногда использовать виртуальные функции невозможно. В таких ситуациях dynamic_cast предлагает альтернативное решение, хотя этот механизм в большей степени подвержен ошибкам, чем виртуализация, и должен применяться с осторожностью.

Одна из возможных ошибок – это работа с результатом dynamic_cast без предварительной проверки на 0: нулевой указатель нельзя использовать для адресации объекта класса. Например:

```
void company::payroll( employee *pe )
{
   programmer *pm = dynamic_cast< programmer* >( pe );

   // iioaioeaeuiay ioeaea: pm eniieucoaony aac i?iaa?ee cia?aiey
   static int variablePay = 0;
   variablePay += pm->bonus();
   // ...
}
```

Результат, возвращенный dynamic_cast, всегда следует проверять, прежде чем использовать в качестве указателя. Более правильное определение функции company::payroll() могло бы выглядеть так:

```
void company::payroll( employee *pe )
{
   // auiieieou dynamic_cast e i?iaa?eou ?acoeuoao
   if ( programmer *pm = dynamic_cast< programmer* >( pe ) ) {
      // eniieuciaaou pm aey auciaa programmer::bonus()
   }
   else {
      // eniieuciaaou ooieoee-?eaiu eeanna employee
   }
}
```

Результат операции dynamic_cast используется для инициализации переменной pm внутри условного выражения в инструкции if. Это возможно, так как объявления в условиях возвращают значения. Ветвь, соответствующая истинности условия, выполняется, если pm не равно нулю: мы знаем, что операция dynamic_cast завершилась успешно и pe указывает на объект programmer. В противном случае результатом объявления будет 0 и выполняется ветвь else. Поскольку теперь оператор и проверка его результата

находятся в одной инструкции программы, то невозможно случайно вставить какой-либо код между выполнением dynamic_cast и проверкой, так что pm будет использоваться только тогда, когда содержит правильный указатель.

В предыдущем примере операция dynamic_cast преобразует указатель на базовый класс в указатель на производный. Ее также можно применять для трансформации l-значения типа базового класса в ссылку на тип производного. Синтаксис такого использования dynamic_cast следующий:

```
dynamic_cast< Type& >( lval )
```

Здесь Type& – это целевой тип преобразования, а lval – l-значение типа базового класса. Операнд lval успешно приводится к типу Type& только в том случае, когда lval действительно относится к объекту класса, для которого один из производных имеет тип Type.

Поскольку нулевых ссылок не бывает, то проверить успешность выполнения операции путем сравнения результата (т.е. возвращенной оператором dynamic_cast ссылки) с нулем невозможно. Если вместо указателей используются ссылки, тогда следующее условие:

```
if ( programmer *pm = dynamic_cast< programmer* >( pe ) )
```

Нельзя переписать в виде:

```
if ( programmer &pm = dynamic_cast<  programmer& >( pe ) )
```

Для извещения об ошибке в случае приведения к ссылочному типу оператор dynamic_cast возбуждает исключение. Следовательно, предыдущий пример можно записать так:

```
#include < typeinfo>
void company::payroll( employee &re )
{
   try {
      programmer &rm = dynamic_cast< programmer & >( re );
      // eniieuciaaou rm aey auciaa programmer::bonus()
   }
   catch ( std::bad_cast ) {
      // eniieuciaaou ooieoee-?eaiu eeanna employee
   }
}
```

В случае неудачного завершения ссылочного варианта dynamic_cast возбуждается исключение типа bad_cast. Класс bad_cast определен в стандартной библиотеке; для ссылки на него необходимо включить в программу заголовочный файл. (Исключения из стандартной библиотеки мы будем рассматривать в следующем разделе.)

Когда следует употреблять ссылочный вариант `dynamic_cast` вместо указательного? Это зависит только от желания программиста. При его использовании игнорировать ошибку приведения типа и работать с результатом без проверки (как в указательном варианте) невозможно; с другой стороны, применение исключений увеличивает накладные расходы во время выполнения программы.

11.2 Оператор typeid

Второй оператор, входящий в состав RTTI, – это `typeid`, который позволяет выяснить фактический тип выражения. Если оно принадлежит типу класса и этот класс содержит хотя бы одну виртуальную функцию-член, то ответ может и не совпадать с типом самого выражения. Так, если выражение является ссылкой на базовый класс, то `typeid` сообщает тип производного класса объекта:

```
#include < typeinfo>

programmer pobj;
employee &re = pobj;

// n ooieoeae name() iu iiciaeiieiny a iia?acaaea, iinayuaiiii type_info
// iia aica?auaao C-no?ieo "programmer"
coiut <<typeid( re ).name() <<endl;
```

Операнд `re` оператора `typeid` имеет тип `employee`. Но так как `re` – это ссылка на тип класса с виртуальными функциями, то `typeid` говорит, что тип адресуемого объекта – `programmer` (а не `employee`, на который ссылается `re`). Программа, использующая такой оператор, должна включать заголовочный файл, что мы и сделали в этом примере.

Где применяется `typeid`? В сложных системах разработки, например при построении отладчиков, а также при использовании устойчивых объектов, извлеченных из базы данных. В таких системах необходимо знать фактический тип объекта, которым программа манипулирует с помощью указателя или ссылки на базовый класс, например для получения списка его свойств во время сеанса работы с отладчиком или для правильного сохранения или извлечения объекта из базы данных. Оператор `typeid` допустимо использовать с выражениями и именами любых типов. Например, его операндами могут быть выражения встроенных типов и константы. Если операнд не принадлежит к типу класса, то `typeid` просто возвращает его тип:

```
int iobj;
cout << typeid( iobj ).name() << endl;   // ia?aoaaony: int
```

```
cout << typeid( 8.16 ).name() <<endl;   // печатается: double
```

Если операнд имеет тип класса, в котором нет виртуальных функций, то `typeid` возвращает тип операнда, а не связанного с ним объекта:

```
class Base {  /* нет виртуальных функций */ };
class Derived : public Base { };
Derived dobj;
Base *pb = &dobj;
cout <<typeid( *pb ).name() << endl;  // печатается: Base
```

Операнд `typeid` имеет тип `Base`, т.е. тип выражения `*pb`. Поскольку в классе `Base` нет виртуальных функций, результатом `typeid` будет `Base`, хотя объект, на который указывает `pb`, имеет тип `Derived`.

Результаты, возвращенные оператором `typeid`, можно сравнивать. Например:

```
#include <typeinfo>

employee *pe = new manager;
employee& re = *pe;
if ( typeid( pe ) == typeid( employee* ) )
   // ?oi-oi naaeaou
/*
if ( typeid( pe ) == typeid( manager* ) )
if ( typeid( pe ) == typeid( employee ) )
if ( typeid( pe ) == typeid( manager )  )
*/
```

Условие в инструкции `if` сравнивает результаты применения `typeid` к операнду, являющемуся выражением, и к операнду, являющемуся именем типа. Обратите внимание, что сравнение возвращает истину:

```
typeid( pe ) == typeid( employee* )
```

Это удивит пользователей, привыкших писать:

```
// вызов виртуальной функции
pe->salary();
```

Это приводит к вызову виртуальной функции `salary()` из производного класса `manager`. Поведение `typeid(pe)` не подчиняется данному механизму. Это связано с тем, что `pe` – указатель, а для получения типа производного класса операндом `typeid` должен быть тип класса с виртуальными функциями. Выражение `typeid(pe)` возвращает тип `pe`, т.е. указатель на `employee`. Это значение совпадает со значением `typeid(employee*)`, тогда как все остальные сравнения дают ложь.

Только при употреблении выражения `*pe` в качестве операнда `typeid` результат будет содержать тип объекта, на который указывает `pe`:

```
typeid( *pe ) == typeid( manager )     // истинно
typeid( *pe ) == typeid( employee )    // ложно
```

В этих сравнениях `*pe` – выражение типа класса, который имеет виртуальные функции, поэтому результатом применения `typeid` будет тип адресуемого операндом объекта `manager`.

Такой оператор можно использовать и со ссылками:

```
typeid( re ) == typeid( manager )      // истинно
typeid( re ) == typeid( employee )     // ложно
typeid( &re ) == typeid( employee* )   // истинно
typeid( &re ) == typeid( manager* )    // ложно
```

В первых двух сравнениях операнд `re` имеет тип класса с виртуальными функциями, поэтому результат применения `typeid` содержит тип объекта, на который ссылается `re`. В последних двух сравнениях операнд `&re` имеет тип указателя, следовательно, результатом будет тип самого операнда, т.е. `employee*`.

На самом деле оператор `typeid` возвращает объект класса типа `type_info`, который определен в заголовочном файле `<typeinfo>`. Интерфейс этого класса показывает, что можно делать с результатом, возвращенным `typeid`.

11.3 Иерархия классов исключений

В начале этого раздела мы определили иерархию классов исключений, с помощью которой наша программа сообщает об аномальных ситуациях. В стандартной библиотеке C++ есть аналогичная иерархия, предназначенная для извещения о проблемах при выполнении функций из самой стандартной библиотеки. Эти классы исключений вы можете использовать в своих программах непосредственно или создать производные от них классы для описания собственных специфических исключений.

Корневой класс исключения в стандартной иерархии называется `exception`. Он определен в стандартном заголовочном файле и является базовым для всех исключений, возбуждаемых функциями из стандартной библиотеки. Класс `exception` имеет следующий интерфейс:

```
namespace std {
   class exception
   public:
      exception() throw();
      exception( const exception & ) throw();
      exception& operator=( const exception & ) throw();
      virtual ~exception() throw();
      virtual const char* what() const throw();
   };
```

```
}
```

Как и всякий другой класс из стандартной библиотеки C++, `exception` помещен в пространство имен `std`, чтобы не засорять глобальное пространство имен программы.

Первые четыре функции-члена в определении класса – это конструктор по умолчанию, копирующий конструктор, копирующий оператор присваивания и деструктор. Поскольку все они открыты, любая программа может свободно создавать и копировать объекты-исключения, а также присваивать им значения. Деструктор объявлен виртуальным, чтобы сделать возможным дальнейшее наследование классу `exception`.

Самой интересной в этом списке является виртуальная функция `what()`, которая возвращает C-строку с текстовым описанием возбужденного исключения. Классы, производные от `exception`, могут заместить `what()` собственной версией, которая лучше характеризует объект-исключение.

Отметим, что все функции в определении класса `exception` имеют пустую спецификацию `throw()`, т.е. не возбуждают никаких исключений. Программа может манипулировать объектами-исключениями (к примеру, внутри `catch`-обработчиков типа `exception`), не опасаясь, что функции создания, копирования и уничтожения этих объектов возбудят исключения.

Помимо корневого `exception`, в стандартной библиотеке есть и другие классы, которые допустимо использовать в программе для извещения об ошибках, обычно подразделяемых на две больших категории: логические ошибки и ошибки времени выполнения.

Логические ошибки обусловлены нарушением внутренней логики программы, например логических предусловий или инвариантов класса. Предполагается, что их можно найти и предотвратить еще до начала выполнения программы. В стандартной библиотеке определены следующие такие ошибки:

```
namespace std {
   class logic_error : public exception {  // логическая ошибка
   public:
      explicit logic_error( const string &what_arg );
   };
   class invalid_argument : public logic_error { // неверный аргумент
   public:
      explicit invalid_argument( const string &what_arg );
   };
   class out_of_range : public logic_error { // вне диапазона
   public:
      explicit out_of_range( const string &what_arg );
   };
   class length_error : public logic_error { // неверная длина
   public:
```

```
      explicit length_error( const string &what_arg );
   };
   class domain_error : public logic_error { // вне допустимой области
   public:
      explicit domain_error( const string &what_arg );
   };
}
```

Функция может возбудить исключение `invalid_argument`, если получит аргумент с некорректным значением; в конкретной ситуации, когда значение аргумента выходит за пределы допустимого диапазона, разрешается возбудить исключение `out_of_range`, а `length_error` используется для оповещения о попытке создать объект, длина которого превышает максимально возможную.

Ошибки времени выполнения, напротив, вызваны событием, с самой программой не связанным. Предполагается, что их нельзя обнаружить, пока программа не начала работать. В стандартной библиотеке определены следующие такие ошибки:

```
namespace std {
   class runtime_error : public exception {  // ошибка времени
выполнения
   public:
      explicit runtime_error( const string &what_arg );
   };
   class range_error : public runtime_error { // ошибка диапазона
   public:
      explicit range_error( const string &what_arg );
   };
   class overflow_error : public runtime_error { // переполнение
   public:
      explicit overflow_error( const string &what_arg );
   };
   class underflow_error : public runtime_error { // потеря значимости
   public:
      explicit underflow_error( const string &what_arg );
   };
}
```

Функция может возбудить исключение `range_error`, чтобы сообщить об ошибке во внутренних вычислениях. Исключение `overflow_error` говорит об ошибке арифметического переполнения, а `underflow_error` – о потере значимости.

Класс `exception` является базовым и для класса исключения `bad_alloc`, которое возбуждает оператор `new()`, когда ему не удается выделить запрошенный объем памяти, и для класса исключения `bad_cast`, возбуждаемого в ситуации, когда ссылочный вариант оператора `dynamic_cast` не может быть выполнен

Определим оператор `operator[]` в шаблоне `Array` так, чтобы он возбуждал исключение типа `range_error`, если индекс массива `Array` выходит за границы:

```
#include <stdexcept>
#include <string>

template <class elemType>
class Array {
public:
   // ...
   elemType& operator[]( int ix ) const
   {
      if ( ix < 0 || ix >= _size )
      {
         string eObj =
         "ошибка: вне диапазона в Array<elemType>::operator[]() ";
         throw out_of_range( eObj );
      }
      return _ia[ix];
   }
   // ...
private:
   int _size;
   elemType *_ia;
};
```

Для использования предопределенных классов исключений в программу необходимо включить заголовочный файл. Описание возбужденного исключения содержится в объекте `eObj` типа `string`. Эту информацию можно извлечь в обработчике с помощью функции-члена `what()`:

```
int main()
{
   try {
      // функция main() такая же, как в разделе 16.2
   }
   catch ( const out_of_range &excep ) {
      // печатается:
      // ошибка: вне диапазона в Array>elemType>::operator[]()
      cerr << excep.what() <<  "\n ";
      return -1;
   }
}
```

В данной реализации выход индекса за пределы массива в функции `try_array()` приводит к тому, что оператор взятия индекса `operator[]()` класса `Array` возбуждает исключение типа `out_of_range`, которое перехватывается в `main()`.

12. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Паттерны проектирования предназначены для:

– эффективного решения характерных задач проектирования;

– обобщенного описания решения задачи, которое можно использовать в различных ситуациях;

– указания отношения и взаимодействия между классами и объектами.

Алгоритмы не являются паттернами, т.к. решают задачу вычисления, а не программирования. Они описывают решение задачи по шагам, а не общий подход к ее решению.

Паттерны проектирования являются инструментами, призванными помочь в решении широкого круга задач стандартными методами. Что положительное несет использование паттернов при проектировании программных систем:

– каждый паттерн описывает решение целого класса проблем;

– каждый паттерн имеет известное имя;

– имена паттернов позволяют абстрагироваться от конкретного алгоритма, а решать задачу на уровне общего подхода. Это позволяет облегчить взаимодействие программистов работающих даже на разных языках программирования;

– правильно сформулированный паттерн проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова;

– шаблоны проектирования не зависят от языка программирования (объектно-ориентированного), в отличие от идиом.

Идиома (программирование) – низкоуровневый шаблон проектирования, характерный для конкретного языка программирования.

Программная идиома – выражение, обозначающее элементарную конструкцию, типичную для одного или нескольких языков программирования.

12.1 Порождающие паттерны проектирования

Абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Инстанцирование – создание экземпляра класса. В отличие от слова «создание», применяется не к объекту, а к классу. То есть, говорят: «создать

экземпляр класса или инстанцировать класс (в виртуальной среде)». Порождающие шаблоны используют полиморфное инстанцирование.

Эти шаблоны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе. Они скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда.

Иногда допустимо выбирать между тем или иным порождающим шаблоном. Например, есть случаи, когда с пользой для дела можно использовать как прототип, так и абстрактную фабрику. В других ситуациях порождающие шаблоны дополняют друг друга. Так, применяя строитель, можно использовать другие шаблоны для решения вопроса о том, какие компоненты нужно строить, а прототип часто реализуется вместе с одиночкой. Порождающие шаблоны тесно связаны друг с другом, их рассмотрение лучше проводить совместно, чтобы лучше были видны их сходства и различия.

К порождающим паттернам проектирования относятся следующие:

- абстрактная фабрика (abstract factory);
- строитель (builder);
- фабричный метод (factory method);
- прототип (prototype);
- одиночка (singleton).

12.1.1 Абстрактная фабрика

Абстрактная фабрика (Abstract factory) – шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса `Factory`, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса, он может создавать окна и

кнопки). Затем пишутся наследующиеся от него классы, реализующие этот интерфейс.

Назначение:

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

Плюсы:

- изолирует конкретные классы;
- упрощает замену семейств продуктов;
- гарантирует сочетаемость продуктов.

Минусы:

- сложно добавить поддержку нового вида продуктов.

Применение:

Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты; Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения; Система должна конфигурироваться одним из семейств составляющих ее объектов; требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

```
#include <iostream>
// AbstractProductA
class ICar
{
public:
   virtual void info() = 0;
};
// ConcreteProductA1
class Ford : public ICar
{
public:
   virtual void info()
   {
      std::cout << "Ford" << std::endl;
   }
};
//ConcreteProductA2
class Toyota : public ICar
{
public:
   virtual void info()
   {
      std::cout << "Toyota" << std::endl;
   }
};
// AbstractProductB
class IEngine
```

```
{
public:
   virtual void getPower() = 0;
};
// ConcreteProductB1
class FordEngine : public IEngine
{
public:
   virtual void getPower()
   {
      std::cout << "Ford Engine 4.4" << std::endl;
   }
};
//ConcreteProductB2
class ToyotaEngine : public IEngine
{
public:
   virtual void getPower()
   {
      std::cout << "Toyota Engine 3.2" << std::endl;
   }
};
// AbstractFactory
class CarFactory
{
public:
   ICar* getNewCar()
   {
      return createCar();
   }
   IEngine* getNewEngine()
   {
      return createEngine();
   }
protected:
  virtual ICar*createCar()= 0;
  virtual IEngine*createEngine()= 0;
};
// ConcreteFactory1
class FordFactory : public CarFactory
{
protected:
   // from CarFactory
   virtual ICar* createCar()
   {
      return new Ford();
   }
   virtual IEngine* createEngine()
   {
      return new FordEngine();
   }
};
// ConcreteFactory2
class ToyotaFactory : public CarFactory
{
protected:
   // from CarFactory
   virtual ICar* createCar()
   {
      return new Toyota();
   }
```

```
    virtual IEngine* createEngine()
    {
        return new ToyotaEngine();
    }
};
int main()
{
    CarFactory* curFactory = NULL;
    ICar* myCar = NULL;
    IEngine* myEngine = NULL;
    ToyotaFactory toyotaFactory;
    FordFactory fordFactory;
    curFactory = &toyotaFactory;
    myCar = curFactory->getNewCar();
    myCar->info();
    myEngine = curFactory->getNewEngine();
    myEngine->getPower();
    delete myCar;
    delete myEngine;
    curFactory = &fordFactory;
    myCar = curFactory->getNewCar();
    myCar->info();
    myEngine = curFactory->getNewEngine();
    myEngine->getPower();
    delete myCar;
    delete myEngine;
    return 0;
}
```

12.1.2 Строитель (Builder)

Строитель (Builder) – шаблон проектирования, порождающий объекты.

Назначение:

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

Плюсы:

– позволяет изменять внутреннее представление продукта;
– изолирует код, реализующий конструирование и представление;
– дает более тонкий контроль над процессом конструирования.

Применение:

– алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
– процесс конструирования должен обеспечивать различные представления конструируемого объекта.

```
#include <iostream>
#include <memory>
#include <string>
```

```
// Product
class Pizza
{
private:
   std::string dough;
   std::string sauce;
   std::string topping;
public:
   Pizza() { }
   ~Pizza() { }
   void SetDough(const std::string& d) { dough = d; };
   void SetSauce(const std::string& s) { sauce = s; };
   void SetTopping(const std::string& t) { topping = t; }
   void ShowPizza()
   {
      std::cout << " Yummy !!!" << std::endl
                << "Pizza with Dough as " << dough
                << ", Sauce as " << sauce
                << " and Topping as " << topping
                << " !!! " << std::endl;
   }
};
// Abstract Builder
class PizzaBuilder
{
protected:
   std::auto_ptr<Pizza> pizza;
public:
   PizzaBuilder() {}
   virtual ~PizzaBuilder() {}
   std::auto_ptr<Pizza> GetPizza() { return pizza; }
   void createNewPizzaProduct() { pizza.reset (new Pizza); }
   virtual void buildDough()=0;
   virtual void buildSauce()=0;
   virtual void buildTopping()=0;
};
// ConcreteBuilder
class HawaiianPizzaBuilder : public PizzaBuilder
{
public:
   HawaiianPizzaBuilder() : PizzaBuilder() {}
   ~HawaiianPizzaBuilder(){}
   void buildDough() { pizza->SetDough("cross"); }
   void buildSauce() { pizza->SetSauce("mild"); }
   void buildTopping() { pizza->SetTopping("ham and pineapple"); }
};
// ConcreteBuilder
class SpicyPizzaBuilder : public PizzaBuilder
{
public:
   SpicyPizzaBuilder() : PizzaBuilder() {}
   ~SpicyPizzaBuilder() {}
   void buildDough() { pizza->SetDough("pan baked"); }
   void buildSauce() { pizza->SetSauce("hot"); }
   void buildTopping() { pizza->SetTopping("pepperoni and salami"); }
};

// Director
class Waiter
{
private:
```

```
    PizzaBuilder* pizzaBuilder;
public:
    Waiter() : pizzaBuilder(NULL) {}
    ~Waiter() { }
    void SetPizzaBuilder(PizzaBuilder* b) { pizzaBuilder = b; }
    std::auto_ptr<Pizza> GetPizza() { return pizzaBuilder->GetPizza(); }
    void ConstructPizza()
    {
        pizzaBuilder->createNewPizzaProduct();
        pizzaBuilder->buildDough();
        pizzaBuilder->buildSauce();
        pizzaBuilder->buildTopping();
    }
};
// Клиент заказывает две пиццы.
int main()
{
    Waiter waiter;
    HawaiianPizzaBuilder hawaiianPizzaBuilder;
    waiter.SetPizzaBuilder (&hawaiianPizzaBuilder);
    waiter.ConstructPizza();
    std::auto_ptr<Pizza> pizza = waiter.GetPizza();
    pizza->ShowPizza();
    SpicyPizzaBuilder spicyPizzaBuilder;
    waiter.SetPizzaBuilder(&spicyPizzaBuilder);
    waiter.ConstructPizza();
    pizza = waiter.GetPizza();
    pizza->ShowPizza();
    return EXIT_SUCCESS;
}
```

12.1.3 Фабричный метод (Factory Method)

Фабричный метод (Factory Method) – шаблон проектирования, реализующий идею «виртуального конструктора», то есть создания объектов без явного указания их типа. Относится к порождающим шаблонам проектирования.

Назначение:

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать. Фабричный метод позволяет классу делегировать создание подклассам.

Плюсы:

– позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (`ConcreteProduct`), а оперируя лишь общим интерфейсом (`Product`);

– позволяет установить связь между параллельными иерархиями классов.

Минусы:

Необходимо создавать наследника `Creator` для каждого нового типа продукта (`ConcreteProduct`):

– продукт (`Product`) определяет интерфейс объектов, создаваемых абстрактным методом;

– конкретный продукт (`ConcreteProduct`) реализует интерфейс `Product`;

– создатель (`Creator`) объявляет фабричный метод, который возвращает объект типа `Product`. Может также содержать реализацию этого метода «по умолчанию»; может вызывать фабричный метод для создания объекта типа `Product`;

– конкретный создатель (`ConcreteCreator`) переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса `ConcreteProduct`.

Применение:

– классу заранее неизвестно, объекты каких подклассов ему нужно создавать.

– класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.

– класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

```
#include<iostream>
#include<string>
using namespace std;
// "Product"
class Product{
public:
   virtual string getName() = 0;
};
// "ConcreteProductA"
class ConcreteProductA : public Product {
public:
   string getName() {
      return "ConcreteProductA";
   }
};
// "ConcreteProductB"
class ConcreteProductB : public Product {
public:
   string getName() {
      return "ConcreteProductB";
   }
};
class Creator{
public:
   virtual Product* FactoryMethod() = 0;
```

```
};
// "ConcreteCreatorA"
class ConcreteCreatorA : public Creator {
public:
   Product* FactoryMethod() {
      return new ConcreteProductA();
   }
};
// "ConcreteCreatorB"
class ConcreteCreatorB : public Creator {
public:
   Product* FactoryMethod() {
      return new ConcreteProductB();
   }
};
int main() {
   const int size = 2;
   // An array of creators
   Creator* creators[size];
   creators[0] = new ConcreteCreatorA();
   creators[1] = new ConcreteCreatorB();
   // Iterate over creators and create products
   for(int i=0;i<size;i++) {
      Product* product = creators[i]->FactoryMethod();
      cout<<product->getName()<<endl;
      delete product;
   }
   int a;
   cin >> a;
   for (int i=0;i<size;i++) {
      delete creators[i];
   }
   return 0;
}
```

12.1.4 Прототип (Prototype)

Прототип (Prototype) – шаблон проектирования, порождающий объекты.

Назначение:

Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Паттерн Prototype (прототип) можно использовать в следующих случаях:

– система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения `new` в коде приложения считается нежелательным (подробнее об этом в разделе Порождающие паттерны);

– необходимо создавать объекты, точные классы которых становятся известными уже на стадии выполнения программы.

Плюсы:

– для создания новых объектов клиенту необязательно знать их конкретные классы;

– возможность гибкого управления процессом создания новых объектов за счет возможности динамических добавления и удаления прототипов в реестр.

Минусы:

– Каждый тип создаваемого продукта должен реализовывать операцию клонирования `clone()`. В случае, если требуется глубокое копирование объекта (объект содержит ссылки или указатели на другие объекты), это может быть непростой задачей.

Применение:

Для создания новых объектов паттерн Prototype использует прототипы. Прототип – это уже существующий в системе объект, который поддерживает операцию клонирования, то есть умеет создавать копию самого себя. Таким образом, для создания объекта некоторого класса достаточно выполнить операцию `clone()` соответствующего прототипа.

Паттерн Prototype реализует подобное поведение следующим образом: все классы, объекты которых нужно создавать, должны быть подклассами одного общего абстрактного базового класса. Этот базовый класс должен объявлять интерфейс метода `clone()`. Также здесь могут объявляться виртуальными и другие общие методы, например, `initialize()` в случае, если после клонирования нужна инициализация вновь созданного объекта. Все производные классы должны реализовывать метод `clone()`. В языке C++ для создания копий объектов используется конструктор копирования, однако, в общем случае, создание объектов при помощи операции копирования не является обязательным.

Используйте этот шаблон проектирования, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:

– инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки;

– для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;

– экземпляры класса могут находиться в одном из нескольких различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

```
#include <iostream>
#include <vector>
#include <map>

// Идентификаторы всех родов войск
enum Warrior_ID { Infantryman_ID, Archer_ID, Horseman_ID };

class Warrior;  // Опережающее объявление
typedef map<Warrior_ID, Warrior*> Registry;

// Реестр прототипов определен в виде Singleton Мэйерса
Registry& getRegistry()
{
  static Registry _instance;
  return _instance;
}

// Единственное назначение этого класса - помощь в выборе нужного
// конструктора при создании прототипов
class Dummy { };

// Полиморфный базовый класс. Здесь также определен статический
// обобщенный конструктор для создания боевых единиц всех родов войск
class Warrior
{
  public:
    virtual Warrior* clone() = 0;
    virtual void info() = 0;
    virtual ~Warrior() {}
    // Параметризированный статический метод для создания воинов
    // всех родов войск
    static Warrior* createWarrior( Warrior_ID id ) {
      Registry& r = getRegistry();
      if (r.find(id) != r.end())
        return r[id]->clone();
      return 0;
    }
  protected:
    // Добавление прототипа в множество прототипов
    static void addPrototype( Warrior_ID id, Warrior * prototype ) {
      Registry& r = getRegistry();
      r[id] = prototype;
    }
    // Удаление прототипа из множества прототипов
    static void removePrototype( Warrior_ID id ) {
      Registry& r = getRegistry();
      r.erase( r.find( id));
    }
};

// В производных классах различных родов войск в виде статических
// членов-данных определяются соответствующие прототипы
class Infantryman: public Warrior
{
  public:
    Warrior* clone() {
      return new Infantryman( *this);
    }
    void info() {
      cout << "Infantryman" << endl;
```

```
    }
  private:
    Infantryman( Dummy ) {
      Warrior::addPrototype( Infantryman_ID, this);
    }
    Infantryman() {}
    static Infantryman prototype;
};

class Archer: public Warrior
{
  public:
    Warrior* clone() {
      return new Archer( *this);
    }
    void info() {
      cout << "Archer" << endl;
    }
  private:
    Archer(Dummy) {
      addPrototype( Archer_ID, this);
    }
    Archer() {}
    static Archer prototype;
};

class Horseman: public Warrior
{
  public:
    Warrior* clone() {
      return new Horseman( *this);
    }
    void info() {
      cout << "Horseman" << endl;
    }
  private:
    Horseman(Dummy) {
      addPrototype( Horseman_ID, this);
    }
    Horseman() {}
    static Horseman prototype;
};

Infantryman Infantryman::prototype = Infantryman( Dummy());
Archer Archer::prototype = Archer( Dummy());
Horseman Horseman::prototype = Horseman( Dummy());


int main()
{
  vector<Warrior*> v;
  v.push_back( Warrior::createWarrior( Infantryman_ID));
  v.push_back( Warrior::createWarrior( Archer_ID));
  v.push_back( Warrior::createWarrior( Horseman_ID));

  for(int i=0; i<v.size(); i++)
    v[i]->info();
  // ...
}
```

12.1.5 Одиночка (Singleton)

Одиночка (Singleton) – паттерн, контролирующий создание единственного экземпляра некоторого класса и предоставляющий доступ к нему

Назначение:

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа.

Плюсы:

– Класс сам контролирует процесс создания единственного экземпляра.

– Паттерн легко адаптировать для создания нужного числа экземпляров.

– Возможность создания объектов классов, производных от Singleton.

Минусы:

– В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

Применение:

Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

1. Такая переменная доступна всегда. Время жизни глобальной переменной – от запуска программы до ее завершения.

2. Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Однако, использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения единственности экземпляра, а именно, возможно создание нескольких переменных того же самого типа (например, стековых).

Для решения этой проблемы паттерн Singleton возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес. Для обеспечения уникальности объекта, конструкторы и оператор присваивания объявляются закрытыми.

```
// Singleton.h
class Singleton
{
  private:
    static Singleton * p_instance;
    // Конструкторы и оператор присваивания недоступны клиентам
    Singleton() {}
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
  public:
    static Singleton * getInstance() {
        if(!p_instance)
            p_instance = new Singleton();
        return p_instance;
    }
};

// Singleton.cpp
#include "Singleton.h"

Singleton* Singleton::p_instance = 0;
```

Клиенты запрашивают единственный объект класса через статическую функцию-член `getInstance()`, которая при первом запросе динамически выделяет память под этот объект и затем возвращает указатель на этот участок памяти. В последствии клиенты должны сами позаботиться об освобождении памяти при помощи оператора `delete`.

Последняя особенность является серьезным недостатком классической реализации шаблона Singleton. Так как класс сам контролирует создание единственного объекта, было бы логичным возложить на него ответственность и за разрушение объекта. Этот недостаток отсутствует в реализации Singleton, впервые предложенной Скоттом Мэйерсом.

12.2 Структурные паттерны проектирования

Структурные паттерны рассматривают вопросы о компоновке системы на основе классов и объектов. При этом могут использоваться следующие механизмы:

– наследование, когда базовый класс определяет интерфейс, а подклассы – реализацию. Структуры на основе наследования получаются статичными;

– композиция, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

К структурным паттернам проектирования относятся следующие:

– адаптер (adapter);

– мост (bridge);

– компоновщик (composite);
– декоратор (decorator);
– прокси (proxy).

12.2.1 Адаптер (Adapter)

Адаптер (Adapter) – структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Назначение:

Часто в новом программном проекте не удается повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью, но иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн Adapter.

Паттерн Adapter, представляющий собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть мы разрабатываем систему климат-контроля, предназначенной для автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта. Нужен адаптер, преобразующий температуру в шкалу Цельсия.

Плюсы:

Паттерн Adapter позволяет повторно использовать уже имеющийся код, адаптируя его несовместимый интерфейс к виду, пригодному для использования.

Минусы:

Задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

Применение:

Пусть класс, интерфейс которого нужно адаптировать к нужному виду, имеет имя `Adaptee`. Для решения задачи преобразования его интерфейса паттерн Adapter вводит следующую иерархию классов:

– виртуальный базовый класс `Target`. Здесь объявляется пользовательский интерфейс подходящего вида. Только этот интерфейс доступен для пользователя.

– производный класс `Adapter`, реализующий интерфейс `Target`. В этом классе также имеется указатель или ссылка на экземпляр `Adaptee`. Паттерн Adapter использует этот указатель для перенаправления клиентских вызовов в `Adaptee`. Так как интерфейсы `Adaptee` и `Target` несовместимы между собой, то эти вызовы обычно требуют преобразования.

Приведем реализацию паттерна Adapter. Для примера выше адаптируем показания температурного датчика системы климат-контроля, переведя их из градусов Фаренгейта в градусы Цельсия (предполагается, что код этого датчика недоступен для модификации).

```
#include <iostream>

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
      // ... какой то код
      return t;
    }
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {
    }
   ~Adapter() {
      delete p_fsensor;
    }
    float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
  private:
    FahrenheitSensor* p_fsensor;
};
```

```
int main()
{
  Sensor* p = new Adapter( new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

12.2.2 Мост (Bridge)

Мост (Bridge) – шаблон проектирования, используемый в проектировании программного обеспечения чтобы «разделять абстракцию и реализацию так, чтобы они могли изменяться независимо». Шаблон bridge (от англ. – мост) использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

Назначение:

При частом изменении класса, преимущества объектно-ориентированного подхода становятся очень полезными, позволяя делать изменения в программе, обладая минимальными сведениями о реализации программы. Шаблон Bridge является полезным там, где не только сам класс часто меняется, но и то, что класс делает.

Плюсы:

– проще расширять систему новыми типами за счет сокращения общего числа родственных подклассов;

– возможность динамического изменения реализации в процессе выполнения программы;

– паттерн Bridge полностью скрывает реализацию от клиента. В случае модификации реализации пользовательский код не требует перекомпиляции.

Применение:

Когда абстракция и реализация разделены, они могут изменяться независимо. Рассмотрим такую абстракцию как фигура. Существует множество типов фигур, каждая со своими свойствами и методами. Однако есть что-то, что объединяет все фигуры. Например, каждая фигура должна уметь рисовать себя, масштабироваться и т.п. В то же время рисование графики может отличаться в зависимости от типа ОС, или графической библиотеки. Фигуры должны иметь возможность рисовать себя в различных графических средах, но реализовывать в каждой фигуре все способы рисования или модифицировать фигуру каждый раз при изменении способа рисования непрактично. В этом случае помогает шаблон Bridge, позволяя создавать но-

вые классы, которые будут реализовывать рисование в различных графических средах. При использовании такого подхода очень легко можно добавлять, как новые фигуры, так и способы их рисования.

Приведем реализацию логгера с применением паттерна Bridge:

```
// Logger.h – Абстракция
#include <string>

// Опережающее объявление
class LoggerImpl;

class Logger
{
  public:
    Logger( LoggerImpl* p );
    virtual ~Logger( );
    virtual void log( string & str ) = 0;
  protected:
    LoggerImpl * pimpl;
};

class ConsoleLogger : public Logger
{
  public:
    ConsoleLogger();
    void log( string & str );
};

class FileLogger : public Logger
{
  public:
    FileLogger( string & file_name );
    void log( string & str );
  private:
    string file;
};

class SocketLogger : public Logger
{
  public:
    SocketLogger( string & remote_host, int remote_port );
    void log( string & str );
  private:
    string host;
    int    port;
};


// Logger.cpp – Абстракция
#include "Logger.h"
#include "LoggerImpl.h"

Logger::Logger( LoggerImpl* p ) : pimpl(p)
{ }

Logger::~Logger( )
{
    delete pimpl;
}
```

```
ConsoleLogger::ConsoleLogger() : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    )
{ }

void ConsoleLogger::log( string & str )
{
    pimpl->console_log( str);
}

FileLogger::FileLogger( string & file_name ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), file(file_name)
{ }

void FileLogger::log( string & str )
{
    pimpl->file_log( file, str);
}

SocketLogger::SocketLogger( string & remote_host,
                            int remote_port ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), host(remote_host), port(remote_port)
{ }

void SocketLogger::log( string & str )
{
    pimpl->socket_log( host, port, str);
}


// LoggerImpl.h - Реализация
#include <string>

class LoggerImpl
{
  public:
    virtual ~LoggerImpl( ) {}
    virtual void console_log( string & str ) = 0;
    virtual void file_log(
                    string & file, string & str ) = 0;
    virtual void socket_log(
           tring & host, int port, string & str ) = 0;
};

class ST_LoggerImpl : public LoggerImpl
{
```

```
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};

class MT_LoggerImpl : public LoggerImpl
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};

// LoggerImpl.cpp – Реализация
#include <iostream>
#include "LoggerImpl.h"


void ST_LoggerImpl::console_log( string & str )
{
  cout << "Single-threaded console logger" << endl;
}

void ST_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Single-threaded file logger" << endl;
}

void ST_LoggerImpl::socket_log(
                      string & host, int port, string & str )
{
  cout << "Single-threaded socket logger" << endl;
};

void MT_LoggerImpl::console_log( string & str )
{
  cout << "Multithreaded console logger" << endl;
}

void MT_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Multithreaded file logger" << endl;
}

void MT_LoggerImpl::socket_log(
                     string & host, int port, string & str )
{
  cout << "Multithreaded socket logger" << endl;
}


// Main.cpp
#include <string>
#include "Logger.h"

int main()
{
  Logger * p = new FileLogger( string("log.txt"));
```

```
  p->log( string("message"));
  delete p;
  return 0;
}
```

Отметим несколько важных моментов приведенной реализации паттерна Bridge:

1. При модификации реализации клиентский код перекомпилировать не нужно. Использование в абстракции указателя на реализацию (идиома pimpl) позволяет заменить в файле Logger.h включение `include "Logger-Impl.h"` на опережающее объявление `class LoggerImpl`. Такой прием снимает зависимость времени компиляции файла Logger.h (и, соответственно, использующих его файлов клиента) от файла LoggerImpl.h.

2. Пользователь класса `Logger` не видит никаких деталей его реализации.

12.2.3 Компоновщик (Composite)

Компоновщик (Composite) – шаблон проектирования, объединяет объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

Назначение:

– необходимо объединять группы схожих объектов и управлять ими;

– объекты могут быть как примитивными (элементарными), так и составными (сложными). Составной объект может включать в себя коллекции других объектов, образуя сложные древовидные структуры. Пример: директория файловой системы состоит из элементов, каждый их которых также может быть директорией;

– код клиента работает с примитивными и составными объектами единообразно.

Плюсы:

– в систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component;

– код клиента имеет простую структуру – примитивные и составные объекты обрабатываются одинаковым образом;

– паттерн Composite позволяет легко обойти все узлы древовидной структуры.

Минусы:

Неудобно осуществить запрет на добавление в составной объект Composite объектов определенных типов. Так, например, в состав римской армии не могут входить боевые слоны.

Применение:

Управление группами объектов может быть непростой задачей, особенно, если эти объекты содержат собственные объекты.

Для военной стратегической игры «Пунические войны», описывающей военное противостояние между Римом и Карфагеном, каждая боевая единица (всадник, лучник, пехотинец) имеет свою собственную разрушающую силу. Эти единицы могут объединяться в группы для образования более сложных военных подразделений, например, римские легионы, которые, в свою очередь, объединяясь, образуют целую армию. Как рассчитать боевую мощь таких иерархических соединений?

Паттерн Composite предлагает следующее решение. Он вводит абстрактный базовый класс `Component` с поведением, общим для всех примитивных и составных объектов. Для случая стратегической игры – это метод `getStrength()` для подсчета разрушающей силы. Подклассы `Primitive` и `Composite` являются производными от класса `Component`. Составной объект `Composite` хранит компоненты-потомки абстрактного типа `Component`, каждый из которых может быть также `Composite`.

```
#include <iostream>
#include <vector>
#include <assert.h>

// Component
class Unit
{
  public:
    virtual int getStrength() = 0;
    virtual void addUnit(Unit* p) {
      assert( false);
    }
    virtual ~Unit() {}
};

// Primitives
class Archer: public Unit
{
  public:
    virtual int getStrength() {
      return 1;
    }
```

```
};

class Infantryman: public Unit
{
  public:
    virtual int getStrength() {
      return 2;
    }
};

class Horseman: public Unit
{
  public:
    virtual int getStrength() {
      return 3;
    }
};


// Composite
class CompositeUnit: public Unit
{
  public:
    int getStrength() {
      int total = 0;
      for(int i=0; i<c.size(); ++i)
        total += c[i]->getStrength();
      return total;
    }
    void addUnit(Unit* p) {
        c.push_back( p);
    }
    ~CompositeUnit() {
      for(int i=0; i<c.size(); ++i)
        delete c[i];
    }
  private:
   std::vector<Unit*> c;
};


// Вспомогательная функция для создания легиона
CompositeUnit* createLegion()
{
  // Римский легион содержит:
  CompositeUnit* legion = new CompositeUnit;
  // 3000 тяжелых пехотинцев
  for (int i=0; i<3000; ++i)
    legion->addUnit(new Infantryman);
  // 1200 легких пехотинцев
  for (int i=0; i<1200; ++i)
    legion->addUnit(new Archer);
  // 300 всадников
  for (int i=0; i<300; ++i)
    legion->addUnit(new Horseman);

  return legion;
}

int main()
{
```

```
  // Римская армия состоит из 4-х легионов
  CompositeUnit* army = new CompositeUnit;
  for (int i=0; i<4; ++i)
    army->addUnit( createLegion());

  cout << "Roman army damaging strength is "
       << army->getStrength() << endl;
  // …
  delete army;
  return 0;
}
```

Следует обратить внимание на один важный момент. Абстрактный базовый класс `Unit` объявляет интерфейс для добавления новых боевых единиц `addUnit()`, несмотря на то, что объектам примитивных типов (`Archer`, `Infantryman`, `Horseman`) подобная операция не нужна. Сделано это в угоду прозрачности системы в ущерб ее безопасности. Клиент знает, что объект типа `Unit` всегда будет иметь метод `addUnit()`. Однако его вызов для примитивных объектов считается ошибочным и небезопасным.

12.2.4 Декоратор (Decorator)

Декоратор (Decorator) – структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике созданию подклассов с целью расширения функциональности.

Назначение:

– Паттерн Decorator динамически добавляет новые обязанности объекту. Декораторы являются гибкой альтернативой порождению подклассов для расширения функциональности.

– Рекурсивно декорирует основной объект.

– Паттерн Decorator использует схему «обертываем подарок, кладем его в коробку, обертываем коробку».

Плюсы

– нет необходимости создавать подклассы для расширения функциональности объекта;

– возможность динамически подключать новую функциональность до или после основной функциональности объекта `ConcreteComponent`.

Применение:

Клиент всегда заинтересован в функциональности `CoreFunctionality.doThis()`. Клиент может или не может быть заинтересован в методах `OptionalOne.doThis()` и `OptionalTwo.doThis()`. Каждый

из этих классов переадресует запрос базовому классу `Decorator`, а тот направляет его в декорируемый объект.

Паттерн Decorator динамически добавляет новые обязанности объекту. Украшения для новогодней елки являются примерами декораторов. Огни, гирлянды, игрушки и т.д. вешают на елку для придания ей праздничного вида. Украшения не меняют саму елку, а только делают ее новогодней.

Хотя картины можно повесить на стену и без рамок, рамки часто добавляются для придания нового стиля.

```
class I {
  public:
    virtual ~I(){}
    virtual void do_it() = 0;
};

class A: public I {
  public:
    ~A() {
        cout << "A dtor" << '\n';
    }
    /*virtual*/
    void do_it() {
        cout << 'A';
    }
};

class D: public I {
  public:
    D(I *inner) {
        m_wrappee = inner;
    }
    ~D() {
        delete m_wrappee;
    }
    /*virtual*/
    void do_it() {
        m_wrappee->do_it();
    }
  private:
    I *m_wrappee;
};

class X: public D {
  public:
    X(I *core): D(core){}
    ~X() {
        cout << "X dtor" << "    ";
    }
    /*virtual*/
    void do_it() {
        D::do_it();
        cout << 'X';
    }
};

class Y: public D {
```

```
  public:
    Y(I *core): D(core){}
    ~Y() {
        cout << "Y dtor" << "    ";
    }
    /*virtual*/
    void do_it() {
        D::do_it();
        cout << 'Y';
    }
};

class Z: public D {
  public:
    Z(I *core): D(core){}
    ~Z() {
        cout << "Z dtor" << "    ";
    }
    /*virtual*/
    void do_it() {
        D::do_it();
        cout << 'Z';
    }
};

int main() {
  I *anX = new X(new A);
  I *anXY = new Y(new X(new A));
  I *anXYZ = new Z(new Y(new X(new A)));
  anX->do_it();
  cout << '\n';
  anXY->do_it();
  cout << '\n';
  anXYZ->do_it();
  cout << '\n';
  delete anX;
  delete anXY;
  delete anXYZ;
}
```

Вывод программы:

```
AX
AXY
AXYZ
X dtor    A dtor
Y dtor    X dtor    A dtor
Z dtor    Y dtor    X dtor    A dtor
```

12.2.5 Прокси (Proxy)

Прокси (Proxy) – структурный шаблон проектирования, предназначенный для замещения другого объекта для контроля доступа к нему.

Назначение:

– Паттерн Proxy является суррогатом или заместителем другого объекта и контролирует доступ к нему.

– Предоставляя дополнительный уровень косвенности при доступе к объекту, может применяться для поддержки распределенного, управляемого или интеллектуального доступа.

– Являясь «оберткой» реального компонента, защищает его от излишней сложности.

Применение:

Суррогат или заместитель это объект, интерфейс которого идентичен интерфейсу реального объекта. При первом запросе клиента заместитель создает реальный объект, сохраняет его адрес и затем отправляет запрос этому реальному объекту. Все последующие запросы просто переадресуются инкапсулированному реальному объекту.

Существует четыре ситуации, когда можно использовать паттерн Proxy:

– Виртуальный proxy является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту.

– Удаленный proxy предоставляет локального представителя для объекта, который находится в другом адресном пространстве («заглушки» в RPC и CORBA).

– Защитный proxy контролирует доступ к основному объекту. «Суррогатный» объект предоставляет доступ к реальному объекту, только вызывающий объект имеет соответствующие права.

– Интеллектуальный proxy выполняет дополнительные действия при доступе к объекту.

– Вот типичные области применения интеллектуальных proxy:

– Подсчет числа ссылок на реальный объект. При отсутствии ссылок память под объект автоматически освобождается (известен также как интеллектуальный указатель или smart pointer).

– Загрузка объекта в память при первом обращении к нему.

– Установка запрета на изменение реального объекта при обращении к нему других объектов.

Паттерн Proxy для доступа к реальному объекту использует его суррогат или заместитель. Банковский чек является заместителем денежных средств на счете. Чек может быть использован вместо наличных денег для совершения покупок и, в конечном счете, контролирует доступ к наличным деньгам на счете чекодателя.

Особенности паттерна Proxy:

– Adapter предоставляет своему объекту другой интерфейс. Proxy предоставляет тот же интерфейс. Decorator предоставляет расширенный интерфейс.

– Decorator и Proxy имеют разные цели, но схожие структуры. Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.

```
class RealImage
{
    int m_id;
  public:
    RealImage(int i)
    {
        m_id = i;
        cout << "   $$ ctor: " << m_id << '\n';
    }
    ~RealImage()
    {
        cout << "   dtor: " << m_id << '\n';
    }
    void draw()
    {
        cout << "   drawing image " << m_id << '\n';
    }
};

// 1. Класс-обертка с "дополнительным уровнем косвенности"
class Image
{
    // 2. Класс-обертка содержит указатель на реальный класс
    RealImage *m_the_real_thing;
    int m_id;
    static int s_next;
  public:
    Image()
    {
        m_id = s_next++;
        // 3. Инициализируется нулевым значением
        m_the_real_thing = 0;
    }
    ~Image()
    {
        delete m_the_real_thing;
    }
    void draw()
    {
        // 4. Реальный объект создается при поступлении
        //    запроса "на первом использовании"
        if (!m_the_real_thing)
          m_the_real_thing = new RealImage(m_id);
        // 5. Запрос всегда делегируется реальному объекту
        m_the_real_thing->draw();
    }
};
int Image::s_next = 1;

int main()
```

```
{
  Image images[5];

  for (int i; true;)

  {
    cout << "Exit[0], Image[1-5]: ";
    cin >> i;
    if (i == 0)
      break;
    images[i - 1].draw();
  }
}
```

12.3 Поведенческие паттерны проектирования

Паттерны поведения рассматривают вопросы о связях между объектами и распределением обязанностей между ними. Для этого могут использоваться механизмы, основанные как на наследовании, так и на композиции.

К поведенческим паттернам проектирования относятся следующие:

- команда (command);
- итератор (iterator);
- наблюдатель (observer);
- стратегия (strategy);
- шаблонный метод (template method);
- посетитель (visitor).

12.3.1 Команда (Command)

Команда (Command) – шаблон проектирования, используемый при объектно-ориентированном программировании, преобразующий запрос на выполнение действия в отдельный объект-команду.

Назначение:

– Система управляется событиями. При появлении такого события (запроса) необходимо выполнить определенную последовательность действий.

– Необходимо параметризировать объекты выполняемым действием, ставить запросы в очередь или поддерживать операции отмены (undo) и повтора (redo) действий.

– Нужен объектно-ориентированный аналог функции обратного вызова в процедурном программировании.

Плюсы:

Придает системе гибкость, отделяя инициатора запроса от его получателя, позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Применение:

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Интерфейс командного объекта определяется абстрактным базовым классом `Command` и в самом простом случае имеет единственный метод `execute()`. Производные классы определяют получателя запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод `execute()` подклассов `Command` просто вызывает нужную операцию получателя.

В паттерне Command может быть до трех участников:

- клиент, создающий экземпляр командного объекта;
- инициатор запроса, использующий командный объект;
- получатель запроса.

Сначала клиент создает объект `ConcreteCommand`, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод `execute()`. Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

```
#include<iostream>
#include<vector>
#include<string>

class Game {
  public:
    void create( ) {
      cout << "Create game " << endl;
    }
    void open( string file ) {
      cout << "Open game from " << file << endl;
    }
    void save( string file ) {
```

```
      cout << "Save game in " << file << endl;
    }
    void make_move( string move ) {
      cout << "Make move " << move << endl;
    }
};

string getPlayerInput( string prompt ) {
  string input;
  cout << prompt;
  cin >> input;
  return input;
}

// Базовый класс
class Command {
  public:
    virtual ~Command() {}
    virtual void execute() = 0;
  protected:
    Command( Game* p ): pgame( p) {}
    Game * pgame;
};

class CreateGameCommand: public Command {
  public:
    CreateGameCommand( Game * p ) : Command( p) {}
    void execute() {
      pgame->create( );
    }
};

class OpenGameCommand: public Command {
  public:
    OpenGameCommand( Game * p ) : Command( p) {}
    void execute() {
      string file_name;
      file_name = getPlayerInput( "Enter file name:");
      pgame->open( file_name);
    }
};

class SaveGameCommand: public Command {
  public:
    SaveGameCommand( Game * p ) : Command( p) {}
    void execute( ) {
      string file_name;
      file_name = getPlayerInput( "Enter file name:");
      pgame->save( file_name);
    }
};

class MakeMoveCommand: public Command {
  public:
    MakeMoveCommand( Game * p) : Command( p) {}
    void execute() {
      // Сохраним игру для возможного последующего отката
      pgame->save("TEMP_FILE");
      string move;
      move = getPlayerInput( "Enter your move:");
      pgame->make_move(move);
```

```
    }
};

class UndoCommand: public Command {
  public:
    UndoCommand( Game * p ) : Command( p) {}
    void execute() {
      // Восстановим игру из временного файла
      pgame->open( "TEMP_FILE");
    }
};


int main()
{
  Game game;
  // Имитация действий игрока
  vector<Command*> v;
  // Создаем новую игру
  v.push_back( new CreateGameCommand( &game));
  // Делаем несколько ходов
  v.push_back( new MakeMoveCommand( &game));
  v.push_back( new MakeMoveCommand( &game));
  // Последний ход отменяем
  v.push_back( new UndoCommand( &game));
  // Сохраняем игру
  v.push_back( new SaveGameCommand( &game));

  for (size_t i=0; i<v.size(); ++i)
    v[i]->execute();

  for (size_t i=0; i<v.size(); ++i)
    delete v[i];

  return 0;
}
```

12.3.2 Итератор (Iterator)

Итератор (Iterator) – Шаблон проектирования, предоставляющий механизм обхода элементов составных объектов (коллекций) не раскрывая их внутреннего представления.

Назначение:

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

– Абстракция в стандартных библиотеках C++ и Java, позволяющая разделить классы коллекций и алгоритмов.

– Придает обходу коллекции «объектно-ориентированный статус».

– Полиморфный обход.

Применение:

Составной объект, такой как список, должен предоставлять способ доступа к его элементам без раскрытия своей внутренней структуры. Более того, иногда нужно перебирать элементы списка различными способами, в зависимости от конкретной задачи. Но вы, вероятно, не хотите раздувать интерфейс списка операциями для различных обходов, даже если они необходимы. Кроме того, иногда нужно иметь несколько активных обходов одного списка одновременно. Было бы хорошо иметь единый интерфейс для обхода разных типов составных объектов (т.е. полиморфная итерация).

Паттерн Iterator позволяет все это делать. Ключевая идея состоит в том, чтобы ответственность за доступ и обход переместить из составного объекта на объект Iterator, который будет определять стандартный протокол обхода.

Абстракция Iterator имеет основополагающее значение для технологии, называемой «обобщенное программирование». Эта технология четко разделяет такие понятия как «алгоритм» и «структура данных». Мотивирующие факторы: способствование компонентной разработке, повышение производительности и снижение расходов на управление.

Рассмотрим пример. Если вы хотите одновременно поддерживать четыре вида структур данных (массив, бинарное дерево, связанный список и хэш-таблица) и три алгоритма (сортировка, поиск и слияние), то традиционный подход потребует 12 вариантов конфигураций (четыре раза по три), в то время как обобщенное программирование требует лишь 7 (четыре плюс три).

```
#include <iostream>
using namespace std;

class Stack {
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack()
    {
        sp = -1;
    }
    void push(int in)
    {
        items[++sp] = in;
    }
    int pop()
    {
        return items[sp--];
    }
    bool isEmpty()
    {
        return (sp == -1);
    }
};
```

```
class StackIter {
    const Stack &stk;
    int index;
  public:
    StackIter(const Stack &s): stk(s)
    {
        index = 0;
    }
    void operator++()
    {
        index++;
    }
    bool operator()()
    {
        return index != stk.sp + 1;
    }
    int operator *()
    {
        return stk.items[index];
    }
};

bool operator == (const Stack &l, const Stack &r)
{
  StackIter itl(l), itr(r);
  for (; itl(); ++itl, ++itr)
    if (*itl !=  *itr)
      break;
  return !itl() && !itr();
}

int main()
{
  Stack s1;
  int i;
  for (i = 1; i < 5; i++)
    s1.push(i);
  Stack s2(s1), s3(s1), s4(s1), s5(s1);
  s3.pop();
  s5.pop();
  s4.push(2);
  s5.push(9);
  cout << "1 == 2 is " << (s1 == s2) << endl;
  cout << "1 == 3 is " << (s1 == s3) << endl;
  cout << "1 == 4 is " << (s1 == s4) << endl;
  cout << "1 == 5 is " << (s1 == s5) << endl;
}
```

12.3.3 Наблюдатель (Observer)

Наблюдатель (Observer) – поведенческий шаблон проектирования. Также известен как «подчинённые» (Dependents), «издатель-подписчик» (Publisher-Subscriber).

Назначение:

– Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

– Инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.

– Определяет часть View в модели Model-View-Controller (MVC).

Применение:

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

При реализации шаблона «наблюдатель» обычно используются следующие классы:

– Observable – интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей.

– Observer – интерфейс, с помощью которого наблюдаемый объект оповещает наблюдателей.

– ConcreteObservable – конкретный класс, который реализует интерфейс Observable.

– ConcreteObserver – конкретный класс, который реализует интерфейс Observer.

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

– существует, как минимум, один объект, рассылающий сообщения;

– имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают с предоставленной им информацией получатели.

```
class Observer
{
  public:
    virtual void update(int value) = 0;
};

class Subject {
    int m_value;
    vector m_views;
  public:
    void attach(Observer *obs)
    {
        m_views.push_back(obs);
```

```
    }
    void set_val(int value)
    {
        m_value = value;
        notify();
    }
    void notify()
    {
        for (int i = 0; i < m_views.size(); ++i)
          m_views[i]->update(m_value);
    }
};

class DivObserver: public Observer
{
    int m_div;
  public:
    DivObserver(Subject *model, int div)
    {
        model->attach(this);
        m_div = div;
    }
     /* virtual */void update(int v)
    {
        cout << v << " div " << m_div << " is " << v / m_div << '\n';
    }
};

class ModObserver: public Observer
{
    int m_mod;
  public:
    ModObserver(Subject *model, int mod)
    {
        model->attach(this);
        m_mod = mod;
    }
     /* virtual */void update(int v)
    {
        cout << v << " mod " << m_mod << " is " << v % m_mod << '\n';
    }
};

int main() {
  Subject subj;
  DivObserver divObs1(&subj, 4);
  DivObserver divObs2(&subj, 3);
  ModObserver modObs3(&subj, 3);
  subj.set_val(14);
}
```

Вывод программы:

```
14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2
```

12.3.4 Стратегия (Strategy)

Стратегия (Strategy) – поведенческий шаблон проектирования. Переносит семейство алгоритмов в отдельную иерархию классов, что позволяет заменять один алгоритм другим в ходе выполнения программы.

Назначение:

Существуют системы, поведение которых может определяться согласно одному алгоритму из некоторого семейства. Все алгоритмы этого семейства являются родственными: предназначены для решения общих задач, имеют одинаковый интерфейс для использования и отличаются только реализацией (поведением). Пользователь, предварительно настроив программу на нужный алгоритм (выбрав стратегию), получает ожидаемый результат. Как пример, – приложение, предназначенное для компрессии файлов использует один из доступных алгоритмов: zip, arj или rar.

Объектно-ориентированный дизайн такой программы может быть построен на идее использования полиморфизма. В результате получаем набор родственных классов с общим интерфейсом и различными реализациями алгоритмов.

Представленному подходу свойственны следующие недостатки:

– Реализация алгоритма жестко привязана к его подклассу, что затрудняет поддержку и расширение такой системы.

– Система, построенная на основе наследования, является статичной. Заменить один алгоритм на другой в ходе выполнения программы уже невозможно.

Применение паттерна Strategy позволяет устранить указанные недостатки.

Применение:

Паттерн Strategy переносит в отдельную иерархию классов все детали, связанные с реализацией алгоритмов. Для случая программы сжатия файлов абстрактный базовый класс `Compression` этой иерархии объявляет интерфейс, общий для всех алгоритмов и используемый классом `Compressor`. Подклассы `ZIP_Compression`, `ARJ_Compression` и `RAR_Compression` его реализуют в соответствии с тем или иным алгоритмом. Класс `Compressor` содержит указатель на объект абстрактного типа `Compression` и предназначен для переадресации пользовательских запросов конкретному алгоритму. Для замены одного алгоритма другим достаточно перенастроить этот указатель на объект нужного типа.

```
#include <iostream>
```

```
#include <string>

class Compression
{
  public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};

class ZIP_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ZIP compression" << endl;
    }
};

class ARJ_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ARJ compression" << endl;
    }
};

class RAR_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "RAR compression" << endl;
    }
};

class Compressor
{
  public:
    Compressor( Compression* comp): p(comp) {}
   ~Compressor() { delete p; }
    void compress( const string & file ) {
      p->compress( file);
    }
  private:
    Compression* p;
};

int main()
{
  Compressor* p = new Compressor( new ZIP_Compression);
  p->compress( "file.txt");
  delete p;
  return 0;
}
```

Плюсы:

– Систему проще поддерживать и модифицировать, так как семейство алгоритмов перенесено в отдельную иерархию классов.

– Паттерн Strategy предоставляет возможность замены одного алгоритма другим в процессе выполнения программы.

– Паттерн Strategy позволяет скрыть детали реализации алгоритмов от клиента.

Минусы:

– Для правильной настройки системы пользователь должен знать об особенностях всех алгоритмов.

– Число классов в системе, построенной с применением паттерна Strategy, возрастает.

12.3.5 Шаблонный метод (Template method)

Шаблонный метод (Template Method) – шаблон проектирования, определяющий основу алгоритма и позволяющий подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры.

Назначение:

Имеются два разных, но в тоже время очень похожих компонента. Вы хотите внести изменения в оба компонента, избежав дублирования кода. Базовый класс определяет шаги алгоритма с помощью абстрактных операций, а производные классы их реализуют.

Применение:

Проектировщик компонента решает, какие шаги алгоритма являются неизменными (или стандартными), а какие изменяемыми (или настраиваемыми). Абстрактный базовый класс реализует стандартные шаги алгоритма и может предоставлять (или нет) реализацию по умолчанию для настраиваемых шагов. Изменяемые шаги могут (или должны) предоставляться клиентом компонента в конкретных производных классах.

Проектировщик компонента определяет необходимые шаги алгоритма, порядок их выполнения, но позволяет клиентам компонента расширять или замещать некоторые из этих шагов.

Паттерн Template Method широко применяется в каркасах приложений (frameworks). Каждый каркас реализует неизменные части архитектуры в предметной области, а также определяет те части, которые могут или должны настраиваться клиентом. Таким образом, каркас приложения становится «центром вселенной», а настройки клиента являются просто «третьей планетой от Солнца». Эту инвертированную структуру кода ласково называют принципом Голливуда – «Не звоните нам, мы сами вам позвоним».

Паттерн Template Method определяет основу алгоритма и позволяет подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры. Строители зданий используют шаблонный метод при проектировании новых домов. Здесь могут использоваться уже существующие типовые планы, в которых модифицируются только отдельные части.

```
#include <iostream>
using namespace std;

class Base
{
    void a()
    {
        cout << "a  ";
    }
    void c()
    {
        cout << "c  ";
    }
    void e()
    {
        cout << "e  ";
    }
    // 2. Для шагов, требующих особенной реализации, определите
    //    "замещающие" методы.
    virtual void ph1() = 0;
    virtual void ph2() = 0;
  public:
    // 1. Стандартизуйте основу алгоритма в шаблонном методе
    //    базового класса
    void execute()
    {
        a();
        ph1();
        c();
        ph2();
        e();
    }
};

class One: public Base
{
   // 3. Производные классы реализуют "замещающие" методы.
     /*virtual*/void ph1()
    {
        cout << "b  ";
    }
     /*virtual*/void ph2()
    {
        cout << "d  ";
    }
};

class Two: public Base
{
     /*virtual*/void ph1()
    {
        cout << "2  ";
```

```
    }
     /*virtual*/void ph2()
    {
        cout << "4  ";
    }
};

int main()
{
  Base *array[] =
  {
     &One(), &Two()
  };
  for (int i = 0; i < 2; i++)
  {
    array[i]->execute();
    cout << '\n';
  }
}
```

Вывод программы:

```
a b c d e a 2 c 4 e
```

12.3.6 Посетитель (Visitor)

Посетитель (Visitor) – шаблон проектирования, определяющий операцию, выполняемую на каждом элементе из некоторой структуры без изменения классов этих объектов.

Назначение:

– Является классической техникой для восстановления потерянной информации о типе.

– Паттерн Visitor позволяет выполнить нужные действия в зависимости от типов двух объектов.

– Предоставляет механизм двойной диспетчеризации.

Применение:

Различные и несвязанные операции должны выполняться над узловыми объектами некоторой гетерогенной совокупной структуры. Вы хотите избежать «загрязнения» классов этих узлов такими операциями (то есть избежать добавления соответствующих методов в эти классы). И вы не хотите запрашивать тип каждого узла и осуществлять приведение указателя к правильному типу, прежде чем выполнить нужную операцию.

Основным назначением паттерна Visitor является введение абстрактной функциональности для совокупной иерархической структуры объектов «элемент», а именно, паттерн Visitor позволяет, не изменяя классы `Element`,

добавлять в них новые операции. Для этого вся обрабатывающая функциональность переносится из самих классов `Element` (эти классы становятся «легковесными») в иерархию наследования `Visitor`.

При этом паттерн Visitor использует технику «двойной диспетчеризации». Обычно при передаче запросов используется «одинарная диспетчеризация» – то, какая операция будет выполнена для обработки запроса, зависит от имени запроса и типа получателя. В «двойной диспетчеризации» вызываемая операция зависит от имени запроса и типов двух получателей (типа `Visitor` и типа посещаемого элемента `Element`).

Реализуйте паттерн Visitor следующим образом. Создайте иерархию классов `Visitor`, в абстрактном базовом классе которой для каждого подкласса `Element` совокупной структуры определяется чисто виртуальный метод `visit()`. Каждый метод `visit()` принимает один аргумент – указатель или ссылку на подкласс `Element`.

Каждая новая добавляемая операция моделируется при помощи конкретного подкласса `Visitor`. Подклассы `Visitor` реализуют `visit()` методы, объявленные в базовом классе `Visitor`.

Добавьте один чисто виртуальный метод `accept()` в базовый класс иерархии `Element`. В качестве параметра `accept()` принимает единственный аргумент – указатель или ссылку на абстрактный базовый класс иерархии `Visitor`.

Каждый конкретный подкласс `Element` реализует метод `accept()` следующим образом: используя полученный в качестве параметра адрес экземпляра подкласса `Visitor`, просто вызывает его метод `visit()`, передавая в качестве единственного параметра указатель `this`.

Теперь «элементы» и «посетители» готовы. Если клиенту нужно выполнить какую-либо операцию, то он создает экземпляр объекта соответствующего подкласса `Visitor` и вызывает `accept()` метод для каждого объекта `Element`, передавая экземпляр `Visitor` в качестве параметра.

При вызове метода `accept()` ищется правильный подкласс `Element`. Затем, при вызове метода `visit()` программное управление передается правильному подклассу `Visitor`. Таким образом, двойная диспетчеризация получается как сумма одинарных диспетчеризаций сначала в методе `accept()`, а затем в методе `visit()`.

Паттерн Visitor позволяет легко добавлять новые операции – нужно просто добавить новый производный от `Visitor` класс. Однако паттерн Visitor следует использовать только в том случае, если подклассы `Element` совокупной иерархической структуры остаются стабильными (неизменяемыми).

В противном случае, нужно приложить значительные усилия на обновление всей иерархии `Visitor`.

```
class Color
{
  public:
    virtual void accept(class Visitor*) = 0;
};

class Red: public Color
{
  public:
     /*virtual*/void accept(Visitor*);
    void eye()
    {
        cout << "Red::eye\n";
    }
};
class Blu: public Color
{
  public:
     /*virtual*/void accept(Visitor*);
    void sky()
    {
        cout << "Blu::sky\n";
    }
};

class Visitor
{
  public:
    virtual void visit(Red*) = 0;
    virtual void visit(Blu*) = 0;
};

class CountVisitor: public Visitor
{
  public:
    CountVisitor()
    {
        m_num_red = m_num_blu = 0;
    }
     /*virtual*/void visit(Red*)
    {
        ++m_num_red;
    }
     /*virtual*/void visit(Blu*)
    {
        ++m_num_blu;
    }
    void report_num()
    {
        cout << "Reds " << m_num_red << ", Blus " << m_num_blu << '\n';
    }
  private:
    int m_num_red, m_num_blu;
};

class CallVisitor: public Visitor
{
```

```
  public:
     /*virtual*/void visit(Red *r)
    {
        r->eye();
    }
     /*virtual*/void visit(Blu *b)
    {
        b->sky();
    }
};

void Red::accept(Visitor *v)
{
  v->visit(this);
}

void Blu::accept(Visitor *v)
{
  v->visit(this);
}

int main()
{
  Color *set[] =
  {
    new Red, new Blu, new Blu, new Red, new Red, 0
  };
  CountVisitor count_operation;
  CallVisitor call_operation;
  for (int i = 0; set[i]; i++)
  {
    set[i]->accept(&count_operation);
    set[i]->accept(&call_operation);
  }
  count_operation.report_num();
}
```

Вывод программы:

```
Red::eye Blu::sky Blu::sky Red::eye Red::eye Reds 3, Blus 2
```

13. ИСПОЛЬЗОВАНИЕ STL В C++

Механизм шаблонов был встроен в компилятор C++ с целью дать возможность программистам C++ создавать эффективные и компактные библиотеки. Через некоторое время была создана одна из библиотек, которая впоследствии и стала стандартной частью C++. STL это самая эффективная библиотека для C++, существующая на сегодняшний день.

Сегодня существует множество реализаций стандартной библиотеки шаблонов, которые следуют стандарту, но при этом предлагают свои расширения, что является с одной стороны плюсом, но, с другой, не очень хорошо, поскольку не всегда можно использовать код повторно с другим компилятором. Поэтому предпочтительно оставаться в рамках стандарта, даже если вы в дальнейшем очень хорошо разберетесь с реализацией вашей библиотеки.

13.1 Основные компоненты STL

Каждая STL коллекция имеет собственный набор шаблонных параметров, который необходим ей для того, чтобы на базе шаблона реализовать тот или иной класс, максимально приспособленный для решения конкретных задач. Какой тип коллекции использовать, зависит от задач, поэтому необходимо знать их внутреннее устройство для наиболее эффективного использования. Рассмотрим наиболее часто используемые типы коллекций.

`vector` – коллекция элементов `T`, сохраненных в массиве, увеличиваемом по мере необходимости. Для того, чтобы начать использование данной коллекции, включите `#include <vector>`.

`list` – коллекция элементов `T`, сохраненных, как двунаправленный связанный список. Для того, чтобы начать использование данной коллекции, включите `#include <list>`.

`map` – это коллекция, сохраняющая пары значений `pair<const Key, T>`. Эта коллекция предназначена для быстрого поиска значения `T` по ключу `const Key`. В качестве ключа может быть использовано все, что угодно, например, строка или `int` но при этом необходимо помнить, что главной особенностью ключа является возможность применить к нему операцию сравнения. Быстрый поиск значения по ключу осуществляется благодаря тому, что пары хранятся в отсортированном виде. Эта коллекция имеет соответственно и недостаток – скорость вставки новой пары обратно пропорциональна количеству элементов, сохраненных в коллекции, поскольку просто добавить новое значение в конец коллекции не получится. Еще одна

важная вещь, которую необходимо помнить при использовании данной коллекции – ключ должен быть уникальным. Для того, чтобы начать использование данной коллекции, включите `#include <map>`. Если вы хотите использовать данную коллекцию, чтобы избежать дубликатов, то вы избежите их только по ключу.

`set` – это коллекция уникальных значений `const Key`, каждое из которых является также и ключом – то есть, проще говоря, это отсортированная коллекция, предназначенная для быстрого поиска необходимого значения. К ключу предъявляются те же требования, что и в случае ключа для map. Естественно, использовать ее для этой цели нет смысла, если вы хотите сохранить в ней простые типы данных, по меньшей мере вам необходимо определить свой класс, хранящий пару ключ – значение и определяющий операцию сравнения по ключу. Очень удобно использовать данную коллекцию, если вы хотите избежать повторного сохранения одного и того же значения. Для того, чтобы начать использование данной коллекции, включите `#include <set>`.

`multimap` – это модифицированный `map`, в котором отсутствует требования уникальности ключа – то есть, если вы произведете поиск по ключу, то вам вернется не одно значение, а набор значений, сохраненных с данным ключом. Для того, чтобы начать использование данной коллекции включите `#include <map>`.

`multiset` – то же самое относится и к этой коллекции, требования уникальности ключа в ней не существует, что приводит к возможности хранения дубликатов значений. Тем не менее, существует возможность быстрого нахождения значений по ключу в случае, если вы определили свой класс. Поскольку все значения в `map` и `set` хранятся в отсортированном виде, то получается, что в этих коллекциях мы можем очень быстро отыскать необходимое нам значение по ключу, но при этом операция вставки нового элемента `T` будет стоить нам несколько дороже, чем например в `vector`. Для того, чтобы начать использование данной коллекции, включите `#include <set>`.

13.2 STL-строки

Не существует серьезной библиотеки, которая бы не включала в себя свой класс для представления строк или даже несколько подобных классов. STL – строки поддерживают как формат ASCII, так и формат Unicode.

`string` – представляет из себя коллекцию, хранящую символы `char` в формате ASCII. Для того, чтобы использовать данную коллекцию, вам необходимо включить `#include <string>`.

`wstring` – это коллекция для хранения двухбайтных символов `wchar_t`, которые используются для представления всего набора символов в формате Unicode. Для того, чтобы использовать данную коллекцию, вам необходимо включить `#include <xstring>`.

13.2.1 Строковые потоки

Используются для организации сохранения простых типов данных в STL строки в стиле C++. Ниже приведена простая программа, демонстрирующая возможности использования строковых потоков:

```
//stl.cpp: Defines the entry point for the console application

#include "stdafx.h"
#include <iostream>
#include <strstream>
#include <string>
using namespace std;

int _tmain (int argc, _TCHAR* argv [])
{
   strstream xstr;
   for (int i = 0; i < 10; i++)
   {
      xstr << "Demo " << i << endl;
   }
   cout << xstr.str ();
   string str;
   str.assign (xstr.str (), xstr.pcount ());
   cout << str.c_str ();
   return 0;
}
```

Строковый поток – это просто буфер, в конце которого установлен нуль терминатор, поэтому мы наблюдаем в конце строки мусор при первой распечатке, то есть реальный конец строки определен не посредством нуль терминатора, а с помощью счетчика, и его размер можно получить с помощью метода `pcount()`.

Далее мы производим копирование содержимого буфера в строку и печатаем строку второй раз. На этот раз она печатается без мусора.

Основные методы, которые присутствуют почти во всех STL коллекциях, приведены ниже.

- `empty` – определяет, является ли коллекция пустой.
- `size` – определяет размер коллекции.

– `begin` – возвращает прямой итератор, указывающий на начало коллекции.

– `end` – возвращает прямой итератор, указывающий на конец коллекции. При этом надо учесть, что реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним.

– `rbegin` – возвращает обратный итератор, указывающий на начало коллекции.

– `rend` – возвращает обратный итератор, указывающий на конец коллекции. При этом надо учесть, что реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним.

– `clear` – удаляет все элементы коллекции, при этом, если в вашей коллекции сохранены указатели, то вы должны не забыть удалить все элементы вручную посредством вызова `delete` для каждого указателя.

– `erase` – удаляет элемент или несколько элементов из коллекции.

– `capacity` – вместимость коллекции определяет реальный размер – то есть размер буфера коллекции, а не то, сколько в нем хранится элементов. Когда вы создаете коллекцию, то выделяется некоторое количество памяти. Как только размер буфера оказывается меньшим, чем размер, необходимый для хранения всех элементов коллекции, происходит выделение памяти для нового буфера, а все элементы старого копируются в новый буфер. При этом размер нового буфера будет в два раза большим, чем размер буфера, выделенного перед этим – такая стратегия позволяет уменьшить количество операций перераспределения памяти, но при этом очень расточительно расходуется память. Причем в некоторых реализациях STL первое выделение памяти происходит не в конструкторе, а как ни странно, при добавлении первого элемента коллекции. Фрагмент программы ниже демонстрирует, что размер и вместимость коллекции – две разные сущности:

```
vector<int> vec;
cout << "Real size of array in vector: " << vec.capacity () << endl;
for (int j = 0; j < 10; j++)
{
   vec.push_back (10);
}
cout << "Real size of array in vector: " << vec.capacity () << endl;
return 0;
```

13.3 Векторы

Наиболее часто используемая коллекция – это вектор. Как уже было отмечено выше, внутренняя реализация этой коллекции представляет из себя массив и счетчик элементов, сохраненных в нем.

Предположим, нам необходимо написать логику клиент – серверного приложения. Администратор сети посылает сообщения на сервер с определенным интервалом, где они сохраняются в одном общем массиве common, при этом каждое сообщение имеет поле `To`, однозначно идентифицирующее каждого клиента.

Каждый клиент также подключается к серверу, но с гораздо большим интервалом, чем приход сообщений от администратора, чтобы просмотреть сообщения, адресованные ему. При этом нам также необходимо знать хронологию прихода сообщений, адресованных разным пользователям (какое сообщение пришло раньше, а какое позже в любой момент времени). Для того, чтобы получить сообщения, клиент должен подключиться к серверу, просмотреть массив common для того, чтобы выбрать сообщения, адресованные ему, и после отключиться.

В нашем случае, три клиента подключаются к серверу и каждый просматривает общий массив сообщений, при этом мы должны сделать наше приложение поточно – безопасным, поэтому должны использовать код внутри критической секции. Все это рано или поздно приведет к тому, что с увеличением числа клиентов приложение станет очень медленным.

Для того, чтобы избежать этой ситуации, мы заведем массив сообщений для каждого клиента и вместо того, чтобы просматривать общий массив сообщений три раза, мы будем просматривать его всего лишь один раз с интервалом времени, адекватным периоду подключения одного клиента. При этом скопируем все сообщения в соответствующие массивы. Клиенты же будут просто забирать данные из своих массивов при подключении.

На самом деле это немного неправильный подход для решения этой задачи. Скорее всего, нам надо было бы сохранять сообщения в момент их прихода в оба массива, но наша цель – посмотреть возможности использования коллекции `vector`, поэтому воспользуемся этим подходом и представим упрощенную логику такого приложения:

```
#include "stdafx.h"
#include <iostream>
#include <strstream>
#include <string>
#include <vector>
#include <algorithm>
```

```
using namespace std;

class MyMessage
{
  private:
    string from;
    string to;
    string message;
    int id;
  public:
    MyMessage(string from, string to, string message)
    {
       this->from = from;
       this->to = to;
       this->message = message;
    }
    int GetId()
    {
       return this->id;
    }
    void SetId(int id)
    {
       this->id = id;
    }
    string GetMessage()
    {
       return this->message;
    }
    string GetFrom()
    {
       return this->from;
    }
    string GetTo()
    {
       return this->to;
    }
};

int _tmain (int argc, _TCHAR* argv [])
{
   vector<MyMessage> common;
   // create pool of messages for 3 users:
   for (int user = 0; user < 3; user++)
   {
      for (int i = 0; i < 10; i++)
      {
         strstream messagex;
         messagex << "Message " << i;
         string smessage;
         smessage.assign(messagex.str(), messagex.pcount());
         strstream userx;
         userx << "User " << user;
         string suser;
         suser.assign (userx.str (), userx.pcount ());
         MyMessage message ("Administrator", suser, smessage);
         message.SetId (user*10 + i);
         common.push_back (message);
      }
   }

   // create vector for each user:
```

```
    vector<MyMessage> user0;
    vector<MyMessage> user1;
    vector<MyMessage> user2;
    for (int x = 0; x < (int) common.size (); x++)
    {
       MyMessage message = common [x];
       if (message.GetTo () == "User 0")
       {
          user0.push_back (message);
       }
       else if (message.GetTo () == "User 1")
       {
          user1.push_back (message);
       }
       else if (message.GetTo () == "User 2")
       {
          user2.push_back (message);
       }
    }

    cout << "Messages for user 2: " << endl;
    for (int i = 0; i < (int) user2.size (); i++)
    {
       MyMessage message = user2[i];
       cout << message.GetTo () << endl;
    }
    cout << "Messages for user 1: " << endl;
    for (int i = 0; i < (int) user1.size (); i++)
    {
       MyMessage message = user1[i];
       cout << message.GetTo () << endl;
    }
    cout << "Messages for user 0: " << endl;
    for (int i = 0; i < (int) user0.size (); i++)
    {
       MyMessage message = user0[i];
       cout << message.GetTo () << endl;
    }

    cout << "Size of common vector: " << (int) common.size () << endl;
    return 0;
}
```

13.4 Итераторы

При перечислении основных методов коллекций упоминались итераторы, при этом не было дано определение этой сущности. Итератор – это абстракция, которая ведет себя, как указатель с некоторыми ограничениями или без них, то есть, сохраняет все свойства своего прародителя. Указатель – это тоже итератор. В действительности, итераторы, в большинстве случаев, это объектные обертки указателей. Вот как примерно может выглядеть внутреннее устройство итератора:

```
class Iterator
{
    T* pointer;
```

```
  public:
    T* GetPointer ()
    {
       return this->pointer;
    }
    void SetPointer (T* pointer)
    {
       this->pointer = pointer;
    }
};
```

Но итератор представляет собой более высокий уровень абстракции, чем указатель, поэтому утверждение, что итератор – это указатель в некоторых случаях может быть неверно. А вот обратное будет верно всегда.

Итераторы обеспечивают доступ к элементам в коллекции.

Итераторы для конкретного класса коллекции определяются внутри класса этой коллекции. В STL существует три типа итераторов: iterator, reverse_iterator, и random access iterator. Для обхода коллекции от меньшего индекса к большему, используются обычные или forward итераторы. Для обхода коллекции в обратном направлении используются reverse итераторы. Random access iterator являются итераторами, которые могут обходить коллекцию как вперед, так и назад. Ниже приведен пример использования итераторов для удаления половины элементов вектора:

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void printInt (int number);

int _tmain (int argc, _TCHAR* argv [])
{
   vector<int> myVec;
   vector<int>::iterator first, last;
   for (long i=0; i<10; i++)
   {
      myVec.push_back (i);
   }
   first = myVec.begin ();
   last = myVec.begin () + 5;
   if (last >= myVec.end ())
   {
      return -1;
   }
   myVec.erase (first, last);
   for_each (myVec.begin (), myVec.end (), printInt);
   return 0;
}
void printInt (int number)
{
   cout << number << endl;
}
```

Важно помнить, что когда вы получаете итератор к коллекции, а после этого модифицируете коллекцию, то этот итератор становится уже непригодным к использованию. Естественно, не все изменения приводят к непригодности итератора для дальнейшего использования, а только изменения структуры коллекции. В случае же, если вы просто измените значения, сохраненные в коллекции, то ничего страшного не произойдет и итератор не испортится.

Итерация по коллекции вперед происходит так:

```
for (iterator element = begin (); element < end (); element++)
{
   t = (*element);
}

// Итерация по коллекции назад происходит так:
for (reverse_iterator element = rbegin(); element < rend(); element++)
{
   t = (*element);
}
```

При работе и с random access iterator итератором синтаксис конструкции может быть, например, таким:

```
for (iterator element = begin(); element < end(); element += 2)
{
   t = (*element);
}
```

Для более эффективного использования контейнеров используйте `typedef` или наследуйте свой класс от класса коллекции.

Сделать это можно так:

```
typedef vector<int> myVector
typedef map<string, int> myMap
typedef deque<string> myQue

// Или вот такая техника в случае наследования:
class myVector: public vector<int> {};

// В случае с итератором применима предыдущая техника:
typedef myVector::iterator vectorIterator
typedef myVector::reverse_iterator revVectorIterator
```

13.5 Алгоритмы

До этого мы посмотрели основные приемы использования STL коллекций на примере использования вектора. Это основа STL, но для того, чтобы по-настоящему использовать всю мощь этой библиотеки, придется расширить наши знания. С использованием алгоритмов возможно создание очень мощных и эффективных программ. По компактности такой код превосходит

код, написанный на таких современных языках, как Java и C#, и в значительной степени эффективнее последнего.

STL-алгоритмы представляют набор готовых функций, которые могут быть применены к STL коллекциям и могут быть подразделены на три основных группы:

– функции для перебора всех членов коллекции и выполнения определенных действий над каждым из них;

– функции для сортировки членов коллекции;

– функции для выполнения определенных арифметических действий над членами коллекции.

К первой группе функций относятся:

`count`, `find`, `for_each`, `search`, `copy`, `swap`, `replace`, `transform`, `remove`, `unique`, `reverse`, `random_shuffle`, `partition` и др.

Интерфейсы функций `for_each` и `accumulate`:

```
for_each (iterator1, iterator2, function);

list<int> L(...)
int sum = accumulate (L.begin(), L.end(), 0, [plus<int>()]);
```

Пример использования функции `transform`:

```
// transform algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int op_increase(int i) { return ++i; }
int op_sum(int i, int j) { return i + j; }

int main()
{
  vector<int> first;
  vector<int> second;
  vector<int>::iterator it;

  // set some values:
  for (int i = 1; i < 6; i++) first.push_back(i*10);
                                  // first: 10 20 30 40 50

  // allocate space
  second.resize(first.size());
  transform(first.begin(), first.end(), second.begin(), op_increase);
                                  // second: 11 21 31 41 51

   transform(first.begin(), first.end(),
             second.begin(), first.begin(), op_sum);
                                  // first: 21 41 61 81 101
   return 0;
}
```

Интерфейсы функций заполнения коллекций:

```
// fill заполняет последовательность элементов от iterator1 до iterator2
fill (iterator1, iterator2, value);                    // значением
value

// fill_n заполняет n элементов последовательности, начиная с указанного
fill_n (iterator1, N, value);

// generate работает, как fill, но для генерации используется функция,
generate (iterator1, iterator2, func);  // которая не принимает
аргументов

generate_n (iterator1, N, function);
```

Пример использования функции `generate`:

```
// function generator:
int RandomNumber () { return (std::rand()%100); }

// class generator:
struct c_unique {
   int current;
   c_unique() {current=0;}
   int operator()() {return ++current;}
} UniqueNumber;

int main ()
{
   std::srand ( unsigned ( std::time(0) ) );
   std::vector<int> myvector (8);

   generate (myvector.begin(), myvector.end(), RandomNumber);

   std::cout << "myvector contains:";
   for (std::vector<int>::iterator it = myvector.begin();
                                   it != myvector.end(); ++it)
      std::cout << ' ' << *it;
   std::cout << '\n';

   std::generate (myvector.begin(), myvector.end(), UniqueNumber);

   std::cout << "myvector contains:";
   for (std::vector<int>::iterator it = myvector.begin();
                                   it != myvector.end(); ++it)
      std::cout << ' ' << *it;
   std::cout << '\n';

  return 0;
}
```

Интерфейсы функций обмена:

```
// swap осуществляет обмен двух элементов
swap (element1, element2);

// iter_swap осуществляет обмен элементов, на которые указывают
итераторы
iter_swap (iterator1, iterator2);
```

```
// swap_ranges – обмен элементами от iter1 до iter2 с элементами,
swap_ranges (iter1, iter2, iter3);                // начинающимися с
iter3
```

Вторая группа функций включает:

`sort`, `stable_sort`, `nth_element`, `binary_search`, `lower_bound`, `upper_bound`, `equal_range`, `merge`, `includes`, `min`, `max`, `min_element`, `max_element`, `lexographical_compare` и др.

Интерфейсы алгоритмов сортировки коллекций:

```
sort (begin, end);

// partial_sort выдает первые N элементов в отсортированном порядке, а
partial_sort (begin, begin + N, end)            // остальные – как
попало.

// nth_element выдает элемент на корректной N-й позиции, в случае, если
бы
nth_element (begin, begin + N, end)        // коллекция была
отсортирована

// partition разделяет коллекцию на 2 части, согласно function
partition (begin, end, function)
```

Интерфейсы функций сравнения коллекций:

```
// equal возвращает true, если последовательности равны (==)
equal(first1, last1, first2);

// mismatch возвращает указатель на первый элемент, который не равен
mismatch(first1, last1, first2); // соответствующему в
последовательности

// lexicographical_compare возвращает true, если первая
последовательность
lexicographical_compare(first1, last1, first2, last2); // меньше второй
```

К третьей группе функций относятся:

`accumulate`, `inner_product`, `partial_sum`, `adjacent_difference`.

Ранее мы уже использовали один из алгоритмов: `for_each()` для того, чтобы распечатать все значения из вектора. Важно отметить, что, кроме указателя на функцию в этом случае мы могли бы передать функтор – специальный класс с перегруженным оператором `operator()`.

```
#include "stdafx.h"
#include <iostream>
#include <strstream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

class MyFunctor
```

```
{
    string comment;
  public:
    MyFunctor ()
    {
       comment = "My comment";
    }
    MyFunctor (string comment)
    {
       this->comment = comment;
    }
    void operator ()(int test)
    {
       cout << test << comment << endl;
    }
};

int _tmain (int argc, _TCHAR* argv [])
{
   vector<int> test;
   // fill vector:
   for (int i = 0; i < 5; i++)
   {
      test.push_back (i);
   }
   // now use our functor:
   MyFunctor functor ("Test comment");
   for_each (test.begin (), test.end (), functor);
   return 0;
}
```

Преимущество такого подхода заключается в том, что если нам необходимо передать какие – либо параметры для того, чтобы произвести обработку каждого члена коллекции, то мы имеем возможность сделать это в конструкторе функтора или определить дополнительные функции в классе – функторе. Это позволит нам сократить количество переменных в области видимости функции – обработчика членов коллекции для хранения значений, которые впоследствии будут использованы внутри тела этой функции. Если же для работы над членами коллекции нам не нужно передавать параметры, то целесообразнее определить просто функцию.

Еще один небольшой пример использования алгоритмов приведен ниже, мы создаем две коллекции: женскую и мужскую, после чего заполняем каждую из них соответствующими именами мужчин и женщин. Если мы захотим поменять местонахождение членов обоих коллекций – то есть, женщин поместить в мужскую коллекцию и наоборот, то сделать это с использованием алгоритмов очень просто:

```
#include "stdafx.h"
#include <iostream>
#include <strstream>
#include <string>
#include <vector>
#include <algorithm>
```

```
using namespace std;

void printMan (string user);

int _tmain (int argc, _TCHAR* argv [])
{
   vector<string> maleRoom;
   vector<string> fimaleRoom;
   maleRoom.push_back ("Vasya");
   maleRoom.push_back ("Petya");
   maleRoom.push_back ("Sasha");

   fimaleRoom.push_back ("Nastya");
   fimaleRoom.push_back ("Alena");
   fimaleRoom.push_back ("Sveta");

   for_each (maleRoom.begin (), maleRoom.end (), printMan);
   reverse (maleRoom.begin (), maleRoom.end ());
   cout << "Males in reverse order " << endl;
   for_each (maleRoom.begin (), maleRoom.end (), printMan);
   maleRoom.swap (fimaleRoom);
   cout << "Now in male room are fimales: " << endl;
   for_each (maleRoom.begin (), maleRoom.end (), printMan);
   return 0;
}

void printMan (string man)
{
   cout << man << endl;
}
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. / Г. Буч и др. // Пер. с англ. – М.: «И.Д. Вильямс», 2010. – 720 с.

2. Павловская, Т.А. C/C++. Программирование на языке высокого уровня / Т.А. Павловская // СПб.: Питер, 2009. – 461 с.

3. Подбельский, В.В. Язык Си++ : Учеб. пособие для вузов по направлениям «Приклад. математика» и «Вычисл. машины, комплексы, системы и сети» / В.В. Подбельский // М.: Финансы и статистика , 2001 г. – 559 с.

4. Страуструп, Б. Язык программирования C++. Специальное издание. Пер. с англ. / Б. Страуструп // СПб., М.: «Невский диалект» – «Издательство БИНОМ», 2008 г. – 1104 с.

5. Майерс С. Эффективное использование C++ / С. Майерс // СПб.: Питер, 2006 г. – 240 с.

6. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования /С. Майерс // СПб.: Питер, 2006. –224 с.

ОГЛАВЛЕНИЕ