А.Б. Григорьев

Введение в программирование на языке С++

Учебное пособие

[Литература 4](#_Toc330302774)

[Литература для самостоятельного изучения 4](#_Toc330302775)

[Inline функции 5](#_Toc330302776)

[Недостатки макросов 5](#_Toc330302777)

[Встроенные функции 6](#_Toc330302778)

[Использование макрокоманд в С++ 7](#_Toc330302779)

[Константы 7](#_Toc330302780)

[Константы в С 7](#_Toc330302781)

[Константы в С++ 7](#_Toc330302782)

[Отличие константы в С++ от С 8](#_Toc330302783)

[Константные указатели 8](#_Toc330302784)

[Ссылки 9](#_Toc330302785)

[Указатели в С++ 9](#_Toc330302786)

[Ссылки в С++ 9](#_Toc330302787)

[Использование ссылок 10](#_Toc330302788)

[Параметры по умолчанию 10](#_Toc330302789)

[Явное приведение типов 11](#_Toc330302790)

[Операторы new и delete 12](#_Toc330302791)

[Пространства имен 13](#_Toc330302792)

[Операции ввода/вывода 14](#_Toc330302793)

[Консольный ввод/вывод 14](#_Toc330302794)

[Чтение и запись файлов 14](#_Toc330302795)

[Чтение и запись в память 14](#_Toc330302796)

[Перегрузка функций 15](#_Toc330302797)

[Аргументы 15](#_Toc330302798)

[Безопасное связывание 15](#_Toc330302799)

[Три шага разрешения перегрузки 15](#_Toc330302800)

[Перегрузка функций и параметры по умолчанию 17](#_Toc330302801)

[Объектно-ориентированное программирование 17](#_Toc330302802)

[Недостатки традиционного подхода 17](#_Toc330302803)

[Объектно-ориентированный подход 17](#_Toc330302804)

[Классы 18](#_Toc330302805)

[Композиция (повторное использование реализации) 18](#_Toc330302806)

[Наследование (повторное использование интерфейса) 19](#_Toc330302807)

[Полиморфизм 20](#_Toc330302808)

[Классы в С++ 21](#_Toc330302809)

[Определение класса 21](#_Toc330302810)

[Данные-члены класса 22](#_Toc330302811)

[Функции-члены класса 22](#_Toc330302812)

[Доступ к членам класса 23](#_Toc330302813)

[Друзья 24](#_Toc330302814)

[Константные функции-члены 24](#_Toc330302815)

[Неявный указатель this 25](#_Toc330302816)

[Перегрузка константных функций 25](#_Toc330302817)

[Статические члены класса 26](#_Toc330302818)

[Инициализация и уничтожение класса 26](#_Toc330302819)

[Конструктор класса 26](#_Toc330302820)

[Порядок вызова конструкторов 28](#_Toc330302821)

[Ограничение прав на создание объекта 28](#_Toc330302822)

[Конструктор копий 29](#_Toc330302823)

[Передача и возвращение по значению 29](#_Toc330302824)

[Деструктор класса 30](#_Toc330302825)

[Стековые и динамические объекты 30](#_Toc330302826)

[Операторы 31](#_Toc330302827)

[Потенциальные подводные камни 32](#_Toc330302828)

[Синтаксис 32](#_Toc330302829)

[Члены и не члены класса 32](#_Toc330302830)

[Ограничения 33](#_Toc330302831)

[Операторы инкремента и декремента 34](#_Toc330302832)

[Оператор [] 34](#_Toc330302833)

[Оператор -> 34](#_Toc330302834)

[Оператор вызова функции 35](#_Toc330302835)

[Операторы new и delete 35](#_Toc330302836)

[Операторы ввода/вывода 36](#_Toc330302837)

[Присваивание объектов 37](#_Toc330302838)

[Присваивание и инициализация 38](#_Toc330302839)

[Присваивание по умолчанию 38](#_Toc330302840)

[Перегрузка оператора = 38](#_Toc330302841)

[Присваивание для переменных класса 39](#_Toc330302842)

[Наследование и композиция 39](#_Toc330302843)

[Композиция 40](#_Toc330302844)

[Наследование 40](#_Toc330302845)

[Что лучше композиция или наследование 42](#_Toc330302846)

[Множественное наследование 44](#_Toc330302847)

[Полиморфизм и виртуальные функции 46](#_Toc330302848)

[Виртуальные функции 46](#_Toc330302849)

[Реализация позднего связывания в С++ 47](#_Toc330302850)

[Эффективность виртуальных функций 47](#_Toc330302851)

[Абстрактные базовые классы и чисто виртуальные функции 47](#_Toc330302852)

[Статический вызов виртуальной функции 48](#_Toc330302853)

[Отрезание объектов 48](#_Toc330302854)

[Виртуальные функции и конструкторы 48](#_Toc330302855)

[Виртуальные деструкторы 49](#_Toc330302856)

[Чисто виртуальные деструкторы 50](#_Toc330302857)

[Виртуальные функции и деструкторы 50](#_Toc330302858)

[Видимость перегруженных и виртуальных функций класса 50](#_Toc330302859)

[Определенные пользователем преобразования 51](#_Toc330302860)

[Конструктор как конвертер 51](#_Toc330302861)

[Конвертер 52](#_Toc330302862)

[Выбор преобразования 53](#_Toc330302863)

[Объектно-ориентированный дизайн 53](#_Toc330302864)

[Open-Close principle 53](#_Toc330302865)

[Liskov substitution principle 55](#_Toc330302866)

[Шаблоны 58](#_Toc330302867)

[Что такое шаблоны и зачем они нужны 58](#_Toc330302868)

[Шаблоны – усовершенствованные макросы 59](#_Toc330302869)

[Параметризованные функции 59](#_Toc330302870)

[Параметризованные типы 61](#_Toc330302871)

[Обобщенное программирование 65](#_Toc330302872)

[Статический полиморфизм 65](#_Toc330302873)

[Обобщенное программирование 66](#_Toc330302874)

[STL 66](#_Toc330302875)

[Контейнеры 66](#_Toc330302876)

[Итераторы 68](#_Toc330302877)

[Алгоритмы 69](#_Toc330302878)

[Итераторы адаптеры 69](#_Toc330302879)

[Итераторы вставки 69](#_Toc330302880)

[Итераторы потока 70](#_Toc330302881)

[Обратные итераторы 70](#_Toc330302882)

[auto\_ptr 70](#_Toc330302883)

[Классы свойств и стратегий 72](#_Toc330302884)

[Фиксированные свойства 73](#_Toc330302885)

[Числовые характеризующие шаблоны 74](#_Toc330302886)

[Параметризованные характеризующие шаблоны 75](#_Toc330302887)

[Стратегии и классы стратегий 75](#_Toc330302888)

[Обработка исключений 77](#_Toc330302889)

[Возбуждение исключения 77](#_Toc330302890)

[Try блок 78](#_Toc330302891)

[Обработчики исключений 78](#_Toc330302892)

[Объекты-исключения 78](#_Toc330302893)

[Прерывание или возобновление 80](#_Toc330302894)

[Повторное возбуждение исключения 80](#_Toc330302895)

[Перехват всех исключений 80](#_Toc330302896)

[Спецификация исключений 81](#_Toc330302897)

[Исключения и производительность 81](#_Toc330302898)

[Новый стандарт с++ 82](#_Toc330302899)

[<< при описании template-ов 82](#_Toc330302900)

[auto при описании типов 82](#_Toc330302901)

[Циклы по диапазону (range based loops) 82](#_Toc330302902)

[nullptr 83](#_Toc330302903)

[Лямбда выражения 83](#_Toc330302904)

[Using как typedef 83](#_Toc330302905)

[shared\_ptr 83](#_Toc330302906)

[weak\_ptr 84](#_Toc330302907)

[unique\_ptr 84](#_Toc330302908)

[Move semantic 84](#_Toc330302909)

[RValue ссылки 85](#_Toc330302910)

## Литература

Липпман “Язык программирования С++”

Эккель. “Философия C++. Введение в стандартный C++”

Страуструп “Язык программирования С++”

Макконнелл “Совершенный код

## Литература для самостоятельного изучения

Мейерс “Эффективное использование С++”,

 “Наиболее эффективное использование C++”

Элджер “С++ Библиотека программиста”

Коплиен “Програмирование на С++”

Александреску “Современное проектирование на С++”

Саттер “Решение сложных задач на С++”

## Inline функции

Одним из наиболее важных достоинств C++ унаследованных им от C является его производительность. Одним из способов достижения большей производительности в С является использование макросов.

### Недостатки макросов

В С++ макросы оставлены из соображений совместимости с С, но при этом в язык добавлены более эффективные механизмы. Основной недостаток макросов состоит в том, что препроцессор — это не компилятор, и все что он делает, это просто заменяет макрокоманду, встреченную в тексте на ее код. Кроме того, использование макросов может привести к появлению в программе ряда трудно выявляемых ошибок. Например,

#define F (x) (x + 1)

Заметьте, что между F и (x) стоит пробел. Тогда вызов

F(1)

Будет преобразован препроцессором в следующую конструкцию:

(x) (x + 1)(1)

Если мы уберем пробел, то все будет работать нормально. Еще пример:

#define F(x,b) x>=b?0:1

Рассмотрим следующую инструкцию:

if (F(a&0x0f,0x07))

Он будет преобразован в следующую конструкцию:

if (a&0x0f>=0x07?0:1)

Приоритет & ниже, нежели чем у >= так что результат этого выражения вас сильно удивит. Причем основная трудность состоит в том, что приоритет >= меньше большинства операций таких как +, /, -- и т.п. Так что в большинстве случаев эта макрокоманда будет работать правильно. Разумеется, эта проблема может быть легко решена заключением всех аргументов макрокоманды в скобки:

#define F(x,b) ((x)>=(b)?0:1)

Еще одна проблема заключается в том, что в отличие от обычной функции аргумент макрокоманды вычисляется каждый раз, при его использовании. Рассмотрим следующий пример:

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int a;

for(int i = 4; i < 6; i++) {

 a = i;

 printf("a = %d\n”, a);

 printf(“BAND(++a)=%d\n", BAND(++a));

 printf("a = %d\n”, a);

}

Результаты работы этого фрагмента кода будет следующие:

a = 4

BAND(++a)=0

a = 5

a = 5

BAND(++a)=8

a = 8

Таким образом, макрокоманда ведет себя по разному в зависимости от параметров.

### Встроенные функции

Отчасти для того, чтобы избежать возникновения этих проблем, отчасти в силу ряда внутренних причин в С++ было введено понятие встроенной (inline) функции. Функция может быть описана как inline с помощью соответствующего ключевого слова. В большинстве случаев inline функции помещаются в заголовочные файлы. При использовании этой функции компилятор выполняет проверки на соответствие параметров функции и подменяет вызов функции на ее тело.

### Использование макрокоманд в С++

Таким образом, при написании кода на C++ предпочтительнее использовать inline’ы нежели чем макрокоманды. Однако макрокоманды тоже могут быть использованы, там где это необходимо. Например, такая полезная директива как # может быть использована для преобразования идентификатора в строку, чего невозможно добиться с помощью inline функций.

Естественно макросы могут быть использованы для написания всякого рода конструкций, используемых в defensive programming’e – ASSERT, FATAL\_ERROR и т.п. В отладочной версии эти макрокоманды могут раскрываются в нечто осмысленное, а в окончательной версии продукта они просто опускаются.

## Ссылки

Помимо указателей в С++ существует такое понятие как ссылка. Рассмотрим сначала чем указатели в С++ отличаются от указателей в С

### Указатели в С++

Основное отличие состоит в том, что С++ является языком с гораздо более строгой типизацией нежели чем С. В основном это касается использования void\*. В С нельзя присвоить указателю на один тип указатель на другой тип, однако это ограничение не распространяется на void\*. В результате, можно написать следующий код:

int\* b;

float\* r;

void\* v;

v = r;

b = v;

Эта возможность оставляет место для появления огромного количество сложных ошибок, исправление которых может потребовать достаточно больших трудозатрат. В С++ при компиляции подобного кода мы получим ошибку компиляции, поэтому мы вынуждены поставить явное преобразование типов.

### Ссылки в С++

Ссылочный тип, иногда называемый псевдонимом, служит для задания объекту дополнительного имени. Ссылка позволяет косвенно манипулировать объектом, точно так же, как это делается с помощью указателя. Ссылочный тип обозначается символом & перед именем переменной.

int i = 1024;

int &ref = i;

В отличие от указателей ссылка обязательно должна быть инициализирована. Определив ссылку вы уже не можете изменить ее так чтобы работать с другим объектом (именно поэтому ссылка должна быть инициализирована в месте своего определения). В следующем примере оператор присваивания не меняет значения ref:

int j = 10;

ref = j;

Можно сказать, что все операции со ссылками реально воздействуют на адресуемые ими объекты (а не на сами ссылки, как это происходит в случае указателей). В том числе и операция взятия адреса. Так

ref += 2;

Добавляет 2 к i – переменной на которую ссылается r. Аналогично

int \*pi = &ref;

инициализирует pi адресом y.

### Использование ссылок

В реальных программах на С++ ссылки, как правило, используются в качестве параметров функций и возвращаемых ими значений. Естественно тоже самое может быть проделано с помощью указателей, но утверждается что использование ссылок обеспечивает лучшую читаемость кода.

Когда же стоит использовать указатели, а когда ссылки? Следует понимать, что не существует такой вещи как NULL ссылка. Ссылка должна ссылаться на какой-либо объект.

Тогда, если предполагается что параметр может быть NULL, или должен ссылаться на разные объекты в ходе выполнения функции, то следует использовать указатель.

## Параметры по умолчанию

C++ предоставляет механизм задания параметров функции по умолчанию. Выглядит это следующим образом:

void f (int i, float a = 0.f);

Значением по умолчанию называется величина, указываемая при объявлении функции, которая затем автоматически подставляется компилятором в случае, если при вызове функции не было указано соответственного значения.

В программе мы можем написать следующее:

main

{

 f(0); // вместо а будет подставлен 0.f

 f(0, 3.f); // a = 3.f

 return ;

}

Можно сказать, что значение параметра по умолчанию – это значение, которое по мнению разработчика является подходящим в большинстве случаев употребления функции, хотя и не во всех.

Следует учитывать, что нельзя определить аргумент по умолчанию для одного параметра функции и не определить таковых для последующих параметров:

int f (int i = 0, int a = 4, int j); // ошибка

должно быть:

int f (int i = 0, int a = 4, int j = 5);

Аргументы по умолчанию задаются только при объявлении функции.

## Явное приведение типов

Достаточно известным является тот факт, что компилятор зачастую автоматически переводит один тип данных в другой в том случае если это необходимо. Например, если вы присвоите целочисленное значение дробной переменной, компилятор автоматически вставит в код фрагмент отвечающий за преобразование int в float:

float f = 5;

Приведение типов (casting) позволяет вам сделать подобное преобразование видимым или вынудить компилятор к его осуществлению, если в какой-то ситуации это не очевидно. В С приведение типов выглядит следующим образом:

int b = 200;

unsigned long a = (unsigned long)b;

В С++ существует альтернативная запись, напоминающая по синтаксису вызов функции, то есть скобки ставятся вокруг аргумента, а не вокруг имени функции:

float a = float(200);

// Что эквивалентно следующей записи:

float b = (float)200;

Кроме того, С++ предоставляет пользователю явные операторы приведения типов, которые призваны полностью заменить старый синтаксис, пришедший из языка С – это static\_cast, const\_cast, reiterpret\_cast, dynamic\_cast. Общая форма записи для этих операторов:

static\_cast<type>(variable);

Так предыдущее выражение можно переписать следующим образом:

float a = static\_cast<float>(200);

или

int i = 5;

float f = static\_cast<float>(i);

Рассмотрим операторы приведения типов более подробно:

* static\_cast – Обычно используется для “хорошего” приведения типов. Сюда входит приведение типов, которое и так будет сделано за вас компилятором, а так же менее очевидные, однако хорошо определенные приведения типов.
* const\_cast – для того чтобы привести константный тип к не константному.
* reinterpret\_cast – наименее безопасный способ приведения типов. Он работает с внутренним преобразованием объектов, причем правильность этого преобразования целиком зависит от программиста.
* dynamic\_cast – применяется для идентификации типа при выполнении run-time type identification (RTTI).

## Операторы new и delete

Прежде чем перейти к классам поговорим о работе с памятью в С++. Объекты в С++, точно так же как и в С могут размещаться либо статически во время компиляции, либо динамически во время выполнения программы, путем вызова соответствующих функций. Основная разница в использовании этих методов состоит в их эффективности и гибкости. Статическое выделение памяти более эффективно в том смысле что оно происходит до выполнения программы, однако оно гораздо менее гибко нежели чем динамическое, поскольку мы заранее должны знать тип и размер размещаемого объекта. Пример статического выделения памяти:

int pi[1024];

Для динамического размещения памяти в С++ используются операторы new и delete. Оператор new имеет две формы, первая выделяет память единичный объект определенного типа:

int \*pi = new int;

Здесь оператор new создает объект типа int, на который ссылается указатель pi. Выделенная память никак не инициализируется. Объект типа int из предидущего примера может быть инициализирован следующим образом:

int \*pi = new int(1024);

Константа в скобках задает начальное значение. Теперь pi ссылается на объект типа int имеющий значение 1024.

Вторая форма оператора new выделяет память под массив заданного размера состоящий из элементов определенного типа:

int \*pia = new int[4];

Память выделенная под динамический объект может быть освобождена с помощью оператора delete, имеющего как и new две формы для единичного объекта и для массива:

delete pi;

delete[] pia;

При этом язык C++ гарантирует что оператор delete нормально отработает, в случае

передачи ему нулевого указателя.

## Пространства имен

Одна из проблем языка С состоит в том, что при написании программ достаточно большого размера вы можете столкнуться с трудностями выбора имен функций, поскольку большая часть самоочевидных названий будет уже определена. В С++ для решения этой проблемы введено понятие пространства имен (namespace). Каждое множество определений в программе и библиотеке определяется в рамках собственного пространства имен, что позволяет избежать обычных для С конфликтных ситуаций:

namespace A {

 int f (void);

}

Ключевое слово namespace задает пространство имен, определяющее видимость нашей функции. Предположим у нас есть функции от других разработчиков, помещенные в собственные пространства имен:

namespace B {

 int f (void);

}

namespace C {

 int f (void);

}

По умолчанию в программе видны объекты, объявленные без явного указания пространства имен; они относятся к глобальному пространству имен. К объекту из другого пространства имен можно обратиться следующим образом:

A::f();

B::f();

Существует ключевое слово using, позволяющее указать компилятору что вы хотите использовать определения из того или иного пространства имен:

using namespace A;

f();

Пространство имен A становится видимым в программе. Можно сделать видимым не все пространство, а отдельные имена внутри него:

using A::f;

f();

Все стандартные С++ библиотеки находятся в пространстве имен std. Поэтому простого использования заголовочного файла не достаточно для того чтобы пользоваться ими напрямую. Необходимо использовать директиву using:

using namespace std;

## Перегрузка функций

Понятие перегрузки функции позволяет многократно использовать имена функций в пределах одной области видимости. Для чего это нужно? Во многих случаях полезным может оказаться набор функций, выполняющих одно и то же действие, но над параметрами различных типов. Например, функция min. Если бы не было перегрузки, то каждой такой функции пришлось бы присваивать индивидуальное имя.

### Аргументы

Две функции с одинаковыми именами считаются разными если они отличаются по количеству, порядку или типу аргументов.

void fn(void);

void fn(int);

int fn(int); // нельзя: отличается только тип возвращаемого

 // значения

int fn(char \*);

void fn(int, char \*);

void fn(char \*, int);

void fn(char\* s, int x, int y = 17); // Можно – три аргумента

 // вместо двух

fn(“hello”, 17); // Ошибка – совпадают две

 // сигнатуры

Пока аргументы отличаются, компилятор не обращает внимание на изменение возвращаемого значения.

### Безопасное связывание

На первый взгляд может показаться, что механизм перегрузки позволяет иметь несколько функций с одинаковыми именами, но разными списками параметров. Однако следует понимать, что перегрузка функций — это лишь лексическое свойство языка С++. Как правило, на этапе компиляции имя функции вместе с ее списком параметров декодируется таким образом, чтобы получить уникальное внутреннее имя, которое затем используется при линковке, оптимизации и т.п.

### Три шага разрешения перегрузки

Разрешением перегрузки функции называется процесс выбора той функции из множества перегруженных, которую следует вызвать.

Процесс разрешения перегрузки функций состоит из трех шагов:

* Выделяется множество перегруженных функций для данного вызова, а также свойства списка аргументов, переданных функций.
* Выбираются те из перегруженных функций, которые могут быть вызваны с данными аргументами, с учетом их количества и типов.
* Находится функция, которая лучше всего соответствует вызову.

Рассмотрим следующий пример:

void f(void);

void f(int);

void f(double, double = 3.4 );

void f(char \*, char \* );

void main()

{

 f( 5.6 );

 return 0;

}

На первом шаге необходимо идентифицировать множество перегруженных функций, которые будут рассматриваться при данном вызове. В нашем случае есть четыре кандидата : f(), f(int), f(double, double) и f(char\*, char\*).

После этого идентифицируются свойства переданного списка аргументов, т.е. их количество и типы. В нашем случае список состоит из одного аргумента типа double.

На втором шаге среди множества кандидатов отбираются устоявшиеся (viable) – такие, которые могут быть вызваны с данными аргументами. Устоявшаяся функция либо имеет столько же формальных параметров, сколько фактических аргументов передано, вызванной функции, либо больше, но тогда для каждого дополнительного параметра должно быть задано значение по умолчанию.

Чтобы функция считалась устоявшейся, для любого фактического аргумента, переданного при вызове обязано существовать преобразование к типу формального параметра переданного при вызове. В нашем случае есть две устоявшиеся функции, которые могут быть вызваны с приведенными аргументами:

* функция f(int) устояла, потому что у нее есть всего один параметр и существует преобразование double в int.
* функция f(double, double) устояла потому что для второго аргумента есть значение по умолчанию, а первый параметр имеет тип double, что в точности соответствует типу фактического аргумента.

Если после второго шага не было найдено устоявшихся функций, то вызов считается ошибочным. В таких случаях мы говорим, что имеет место отсутствие соответствия.

Третий шаг состоит в выборе функции лучше всего отвечающей контексту вызова. Такая функция называется наилучшей из устоявшихся или наиболее подходящей. На этом этапе проводится ранжирование преобразований, использованных для приведения типов фактических аргументов к типам формальных параметров. Ранжирование может дать один из следующих результатов:

* точное соответствие – тип аргумента точно соответствует типу параметра.
* соответствие с преобразованием типа – тип аргумента не соответствует типу параметра, но может быть преобразован в него.

Наиболее подходящей считается функция для которой выполняются следующие условия:

* Преобразования примененные к фактическим аргументам не хуже преобразований необходимых для вызова любой другой устоявшейся функции.
* Для некоторых аргументов примененные преобразования лучше, чем преобразования необходимые для приведения тех же аргументов в вызове других устоявшихся функций.

Для функции f(int) должно быть проведено преобразование фактического аргумента типа double к типу int. Для функции f(double, double) тип фактического аргумента double в точности соответствует типу формального параметра. Точное соответствие считается лучше стандартного преобразования, поэтому более подходящей для данного вызова является функция f(double, double).

Если на третьем шаге не удается отыскать такую функцию, которая подходила бы лучше остальных, то вызов считается не однозначным, то есть ошибочным.

## Объектно-ориентированное программирование

### Недостатки традиционного подхода

Любой язык программирования предоставляет пользователю какой-то уровень абстракции, на котором он может оперировать при решения своей конкретной задачи. Так можно сказать, что Ассемблер представляет достаточно примитивную абстракцию машины, для которой он реализован. Большая часть так называемых процедурных языков (С, Pascal, Fortran) представляют собой абстракцию языка ассемблера.

Альтернативой моделированию машины является моделирование проблемы, которую вы пытаетесь решить. Такие языки как LISP и Prolog были попытками продвинутся именно в этом направлении. Однако им не доставало общности из-за ограниченности их представления о природе решаемых задач. В результате, при попытке выйти за переделы некоторой четко-определенной для каждого из этих языков области возникали очень большие проблемы.

### Объектно-ориентированный подход

Объектно-ориентированный подход является логическим продолжением этой концепции, при котором пользователю предоставляется набор средств для представления решаемой проблемы. Это представление является достаточно общим для того, чтобы не быть ограниченным рамками какого-то определенного класса проблем. Элементы этого представления получили название объектов.

Основная идея состоит в том, что программа адаптируется для решения конкретной задачи путем добавления новых типов объектов. Таким образом решение проблемы записывается в терминах самой проблемы, а не абстрактного машинного языка.

### Классы

Объекты, идентичные друг другу за исключением их состояния в процессе работы программы, образуют так называемые “классы объектов”. Впервые понятие класса было введено в языке Simula-67.

Поскольку класс описывает множество объектов, которые имеют одинаковые характеристики и поведение, то класс на самом деле представляет собой тип данных, точно такой же как int, float и любой другой встроенный тип.

Создание абстрактных типов данных (классов) является одной из фундаментальных концепций объектно-ориентированного программирования. Вы можете создавать переменные такого типа, называемые объектами и манипулировать ими. Таким образом вы расширяете язык программирования, путем добавления в него новых типов данных для описания вашей конкретной задачи.

Но каким образом мы можем заставить объект выполнять нужные вам действия? Должна существовать возможность отправить объекту требование по получении которого будут выполнены определенные действия. Требования, которые вы можете послать каждому объекту определяются интерфейсом этого объекта. Рассмотрим, например, представление лампочки:



Light lt;

lt.on();

Интерфейс определяет какие требования или запросы могут быть предъявлены конкретному объекту. Однако должен существовать какой-то код для удовлетворения этих запросов. Этот код, наряду с внутренними данными называется реализацией (implementation).

### Композиция (повторное использование реализации)

После того, как класс был создан и оттестирован, он в идеале представляет собой отдельную единицу кода. На практике, задача написания хорошо структурированного класса, пригодного для дальнейшего использования в программе представляет собой достаточно трудную задачу. Но предположим, что это все же сделано, в таком случае класс может быть повторно использован в программе.

Наиболее простым примером такого использования является прямое использование в программе объектов этого класса. Однако объект класса может быть помещен внутрь объекта другого класса. Таким образом новый класс может быть составлен из множества объектов произвольного типа, в той комбинации, которая требуется для решения конкретной задачи. Такой подход получил название композиции (composition) или в более общем виде агрегации (aggregation). Композиция используется для представления так называемого ‘has a’ отношения (car has an engine).

### Наследование (повторное использование интерфейса)

Сама по себе идея объекта уже является достаточным средством для решения большинства задач. Однако очень неудобно, создав один класс, создавать другой обладающий схожей, но все же слегка отличной функциональностью. Намного более удобно было, бы сделать копию существующего класса и наделить ее недостающими свойствами. Это как раз то, что мы получаем в результате использования наследования, с тем исключением, что изменение первичного класса (его называют родительским или базовым иди суперклассом) влечет за собой соответствующее изменение клона (его еще называют производным, унаследованным или дочерним классом).

Отдельный тип не только описывает ряд ограничений накладываемых на множество объектов ему принадлежащих, он так же может находится в определенной зависимости от других типов. Два типа могут иметь какие-то общие свойства, но один из них может определять большее число свойств и принимать большее число запросов (или по другому их обрабатывать). Наследование определяет общие свойства типов путем выделения концепции базового и производных классов. Базовый тип характеризует свойства и поведение, являющиеся общими для всех производных классов.

Рассмотрим пример наследования, для геометрических фигур. Базовым типом является фигура, она характеризуется размером, цветом, позицией и т.п. Фигура может быть нарисована, стерта, передвинута, окрашена и т.п. Все геометрические фигуры, такие как круг, прямоугольник, треугольник являются производными от базового типа фигура. Подобная иерархия может быть отображена с помощью следующей диаграммы:



Когда вы наследуетесь от существующего типа, вы тем самым создаете новый тип. Этот новый тип содержит не только все данные базового типа, но что более важно – интерфейс базового типа. Таким образом все те запросы, которые могут быть посланы представителям базового типа, могут посланы и представителям производных типов, а это означает что, производный класс имеет тот же тип что и базовый класс. Это так называемое ‘is a’ отношение “a circle is a shape”.

У вас есть два основных способа изменения производного класса: первый и наиболее простой из них состоит в добавлении новой функциональности, второй – состоит в том, что вы изменяете поведение функций базового класса. Этот подход носит название переопределения (override) функций. Рассмотрим следующую диаграмму:



Для того, чтобы переопределить функцию, вы просто создаете новое определение функции в производном классе, тем самым как бы говоря, что вы используете тот же самый интерфейс, но сама функция работает по-другому, нежели чем в базовом классе.

### Полиморфизм

При работе с иерархиями классов во многих случаях удобно трактовать объекты принадлежащих к разным классам в пределах одной иерархии, как имеющие один базовый тип. Таким образом, удается написать код, независящий от типов объектов. Например, в случае приведенной выше иерархии фигур, функция рисования может оперировать именно обобщенными фигурами, а не отдельными окружностями, треугольниками и квадратами.

void f (Shape &s)

{

 s.erase();

 s.draw();

}

Circle c;

f(c);

Truangle t;

f(t);

Подобный код ни коим образом не зависит от добавления новых типов данных в иерархию. Однако при таком подходе компилятору не известно, в каком случае, какой конкретный фрагмент кода должен будет выполнятся. Как же обеспечить вызов именно того фрагмента кода, который необходим в данном случае? Ответ состоит в том, что на самом деле компилятор не делает вызова функции в традиционном смысле этого слова.

В случае не объектно-ориентированных языков, компилятор производит так называемое раннее связывание (early binding). Это означает что компилятор создает вызов функции с соответствующим именем, а линкер ‘разрешает’ этот вызов путем подстановки абсолютного адреса кода, который должен быть выполнен.

При объектно-ориентированном подходе, абсолютный адрес кода, который должен быть вызван, может быть определен, только в ходе работы программы. Таким образом необходим, какой-то иной механизм, для передачи сообщения конкретному объекту.

Для решения этой проблемы объектно-ориентированные языки используют концепцию так называемого позднего связывания (late binding). Когда, вы посылаете сообщение объекту, на этапе компиляции не происходит определения адреса вызываемого кода. Компилятор просто проверяет, что такая функция существует и проверяет типы ее входных и возвращаемых параметров. Вместо явного вызова функции компилятор создает кусок кода, который вычисляет адрес функции, используя информацию, которая хранится в объекте. Таким образом для данного конкретного объекта вызывается именно та функция, которая нужна.

## Классы в С++

Рассмотрим более подробно механизм реализации классов в С++.

### Определение класса

В С++ oопределение класса состоит из двух частей: заголовка, включающего ключевое слово class, за которым следует имя класса, и тела, заключенного в фигурные скобки.

class A { /\* ... \*/ };

class A { /\* ... \*/ } a, b;

### Данные-члены класса

Данные-члены класса объявляются так же, как переменные. Объявления данных-членов очень похожи на объявления переменных в области видимости блока или пространства имен.

class A {

 int a;

};

Однако их, за исключением статических членов, нельзя явно инициализировать в теле класса:

class A {

 int a = 0; // ошибка

};

### Функции-члены класса

Функции-члены класса объявляются в его теле. Это объявление выглядит точно так же,

как объявление функции в области видимости пространства имен:

class A {

 int m1;

 float m2;

public:

 float GetM2 (void);

};

Определение функции члена вне тела класса начинается с имени класса с добавлением ::.

float A::GetM2 (void)

{

 return m2;

}

Функции-члены отличаются от обычных функций следующим:

* функция-член объявлена в области видимости своего класса, следовательно, ее имя не видно за пределами этой области. К функции-члену можно обратиться с помощью одного из операторов доступа к членам класса (.) или (->).
* функции-члены имеют право доступа как к открытым, так и к закрытым членам класса, тогда как обычным функциям доступны лишь открытые.

Определение функции-члена также можно поместить внутрь тела класса. Функция определенная таким образом, автоматически становится inline.

class A {

 int m1;

 float m2;

public:

 float GetM2 (void) {return m2; }

};

### Доступ к членам класса

В С++ существует три уровня доступа к членам класса – public, private, и protected. Члены, объявленные в секции public называются открытыми, а объявленные в секциях private и protected – соответственно закрытыми и защищенными.

class foo {

public:

 void f () { \_cursor = 0; }

protected:

 char f1 ( int, int );

private:

 int m1, m2;

};

* открытый член доступен из любого места программы. Класс, скрывающий информацию, оставляет открытыми только функции-члены, определяющие операции, с помощью которых внешняя программа может манипулировать его объектами;
* закрытый член доступен только функциям-членам и друзьям класса. Класс, который хочет скрыть информацию, объявляет свои данные-члены закрытыми;
* защищенный член ведет себя как открытый по отношению к производному классу и как закрытый по отношению к остальной части программы.

Если спецификатор доступа не указан, то секция, следующая непосредственно за открывающейся скобкой по умолчанию, считается private:

class foo {

 int m;

};

В программе доступ к открытым членам и функциям класса осуществляется с помощью операторов . и ->:

class foo {

public:

 int m;

public:

 void func (void);

}

foo f;

f.m++;

f.func();

foo \*f = new foo;

f->m++;

f->func();

### Друзья

Иногда бывает удобно разрешить некоторым функциям или классам доступ к закрытым членам класса, это может быть сделано с помощью так называемого механизма друзей. Объявление друга начинается с ключевого слова friend и может встречаться только внутри определения класса:

class foo {

 friend void func(foo &f);

 friend class A;

};

Теперь функция func может напрямую обращаться к закрытым членам foo.

### Константные функции-члены

Функции-члены класса могут быть объявлены как константные.

class foo {

public:

 void func (void) const;

};

Что это означает? Для того, чтобы понять это рассмотрим для начала концепцию константных объектов.

Если вы определите функцию член как константную, вы тем самым дадите компилятору понять, что эта функция может быть вызвана для константного объекта. При попытке вызова любой не константной функции-члена для подобного объекта, компилятор выдаст ошибку, поскольку по умолчанию подобные функции рассматриваются, как способные изменить данные-члены. Рассмотрим следующий пример:

class A {

private:

 int x;

public:

 void f(void) const;

 void g(void);

};

void h(int\*);

void A::f() const

{

 x = 17; // Нельзя: изменяется переменная класса

 h(&x); // Нельзя: h может изменить x

}

Первая ошибка – попытка изменить переменную класса, вторая – попытка вызвать не константную функцию.

Cледует помнить, что наличие спецификатора const в объявлении функции члена не предотвращает все возможные изменения. Например,

class Text {

public:

 void bad(char \*text, int n const;

private:

 char \*txt;

};

void Text::bad(char \*text, int n) const

{

 txt = text; // ошибка: нельзя модифицировать \_text

 for (int i = 0; i < n; i++)

 txt[i] = text[i]; // плохой стиль, но не ошибка

}

### Неявный указатель this

Рассмотрим следующий пример:

class foo {

public:

 int GetI (void) const;

 int GetJ (void) const;

private:

 int i, j;

};

foo f1, f2;

У каждого объекта класса foo есть собственная копия данных-членов. У f1 свои члены i и j, у f2 – свои. Однако каждая функция член класса существует в единственном экземпляре. Их и вызывают f1 и f2. В то же время функция член может обратится к членам своего класса, не используя оператора доступа. Как же она определяет для какого объекта она была вызвана. Дело в том, что каждой функции члену передается указатель на объект, для которого она была вызвана - this. В не константной функции – это указатель на тип класса, в константной – константный указатель на тот же тип.

### Перегрузка константных функций

Константная функция, аргументы которой совпадают с аргументами не константной функции, тем не менее считается другой функцией. Компилятор вызывает константную или не константную версии в зависимости от типа переменной, указывающей или ссылающейся на объект.

class foo {

public:

 void fn();

 void fn() const; // Другая функция!

};

foo\* f = new foo;

f->fn(); // Вызывается не константная версия

const foo\* f1 = f;

f1->fn(); // Вызывается константная версия

###

### Статические члены класса

Иногда нужно, чтобы все объекты данного класса имели доступ к единственному глобальному объекту. Это может быть сделано с помощью статических членов класса. В отличие от обычных данных-членов, статические члены существуют в единственном экземпляре и связаны с самим типом, а не с конкретным объектом. По сравнению с глобальными объектами, статические члены имеют следующие преимущества:

* статический член не находится в глобальном пространстве имен программы, следовательно, уменьшается вероятность случайного конфликта имен с другими глобальными объектами;
* остается возможность сокрытия информации, так как статический член может быть закрытым, а глобальный объект – никогда.

Чтобы сделать член статическим необходимо поместить в начале его объявления в теле класса ключевое слово static. Как правило, статический член инициализируется вне определения класса, при этом его имя должно быть специализировано именем класса:

class foo {

private:

 static int i;

};

int foo::i = 0;

В программе может быть только одно определение статического члена. Внутри функций членов класса к его статическим членам можно обращаться точно так же, как и к обычным членам. Для всей остальной программы обращение к статическому члену может быть осуществлено либо с помощью операторов доступа либо с добавлением имени класса и ::.

Функции-члены также могут быть объявлены как статические, при этом для них справедливы все свойства статических членов, описанные выше:

class foo {

protected:

 static void func (void);

};

Следует отметить, что статической функции-члену указатель this не предается, поэтому явное или неявное обращение к нему вызовет ошибку компиляции. В частности, попытка обращения к нестатическому члену класса неявно требует наличия указателя this и, следовательно, запрещена.

## Инициализация и уничтожение класса

Для поддержки инициализации объектов класса служит конструктор - специальная функция, определяемая проектировщиком и автоматически применяемая к каждому объекту класса при его создании. Парная к каждому конструктору функция – называется деструктор и предназначена для освобождения ресурсов, захваченных либо в конструкторе класса, либо на протяжении его жизни.

### Конструктор класса

Среди других функций-членов конструктор выделяется тем, что его имя совпадает с именем класса.

class foo {

public:

 foo (void);

 foo (int i);

};

Единственное синтаксическое ограничение, налагаемое на конструктор, состоит в том, что не должен иметь тип возвращаемого значения – даже void. Количество конструкторов у одного класса может быть любым, лишь бы все они имели разные списки формальных параметров. Конструктор без параметров, носит название конструктора по умолчанию.

На этапе компиляции во все определения объектов вставляется вызов соответствующих конструкторов. Решение о том, какой конструктор вызвать принимается на основе анализа списка формальных аргументов, указанных при определении объекта.

foo f1; // вызывается foo()

foo f2(3); // вызывается foo(int i)

Определять объект класса, не указывая списка фактических аргументов можно в том случае, если в нем либо объявлен конструктор по умолчанию, либо вообще нет никаких конструкторов. Если в классе объявлены какие-то конструкторы, то объект класса можно определять, только с помощью одного из них. В частности, если в классе объявлен конструктор, принимающий какие-то параметры, но не объявлен конструктор по умолчанию, то в каждом определении объекта этого класса должны присутствовать соответствующие аргументы:

class foo {

 foo (int i);

};

foo f1; // ошибка

foo f1(3); // все правильно вызывается foo(int i)

Подобно другим функциям-членам тело конструктора может быть объявлено как в теле класса, так и вне его. Например,

class foo {

 int i;

public:

 foo (void) { i = 5; }

};

или:

foo::foo (void)

{

 i = 5;

}

### Порядок вызова конструкторов

Если класс не содержит собственных конструкторов, то он инициализируется так, словно компилятор создал конструктор без аргументов за вас. Этот конструктор вызывает конструкторы базовых классов и переменных классов. Четко определенный порядок вызова конструкторов, не зависит от того какие именно конструкторы используются:

* Сначала вызываются конструкторы базовых классов в порядке их перечисления в списке наследования.
* Затем вызываются конструкторы переменных класса в порядке их объявления классе;
* После того, как будут сконструированы все базовые классы и переменные, выполняется тело конструктора.

Описанный порядок применяется рекурсивно, то есть первым конструируется первый базовый класс первого базового класса и т.п. Он не зависит от порядка указанного в списке инициализации членов.

### Конструктор копий

Конструктор копий (copy constructor) определяется специальной сигнатурой:

class foo {

public:

 foo(const foo &f);

};

Конструктор копий предназначен для создания копий объекта. Эта задача может возникнуть в самых разных обстоятельствах, например,

foo f;

foo f1 = f; // конструирование, а не присваивание

foo f1(f); // то же самое

f1 = f; // присваивание

Несмотря на, то что эта строка выглядит как присваивание, на самом деле это не что иное как альтернативный вариант вызова конструктора копий. Чтобы понять, чем присваивание отличается от инициализации, необходимо задать себе вопрос “Был ли объект сконструирован заранее или его создание является частью команды?”. В первом случае мы имеем дело с присваиванием, а во втором с использованием конструктора копий.

Если конструктор копий не определен, то компилятор создаст его за вас. Для этого конструктора используется строго определенная последовательность вызова конструкторов копий базовых классов и переменных класса:

* Конструкторы копий базовых классов вызываются в том порядке, в котором они объявлены в списке наследования.
* Конструкторы копий переменных вызываются в том порядке, в котором они объявлены внутри класса.

Описанный порядок применяется рекурсивно. То есть аналогично любому другому конструктору.

### Передача и возвращение по значению

При передаче объекта в функцию по значению (а не по ссылке или указателю) сначала создается копия этого объекта, которая затем передается в функцию.

foo func (foo f) { return f; }

foo f;

func(f);

То же самое происходит и при возвращении объекта по значению. Способ копирования определяется конструктором копий (по умолчанию это просто побитовое копирование). С учетом всего вышесказанного передача параметра по значению может оказаться очень дорогой операцией.

Для того чтобы избежать этой ситуации необходимо передавать параметры по ссылке или по указателю:

foo& func (foo &f) { return f; }

### Деструктор класса

Деструктор – это специальная определяемая пользователем функция член, которая автоматически вызывается при уничтожении объекта. Имя этой функции образовано из имени класса и предшествующего ему символа ~. Деструктор не возвращает значения и не принимает никаких параметров, а, следовательно, не может быть перегружен.

class foo {

public:

 ~foo();

};

Деструкторы гарантировано вызываются в порядке обратном порядку вызову конструкторов. Это означает, что сначала вызывается тело деструктора объекта, затем деструкторы переменных класса в порядке обратном порядку их объявления и наконец деструкторы базовых классов, начиная с последнего в списке наследования.

## Операторы

Операторы в С++ можно трактовать как обычные функции. Однако существуют определенные различия между вызовом функции и использованием оператора. В первую очередь это относится к синтаксису. Оператор вызывается путем размещения его между или после, а бывает и до аргументов. Кроме того, в случае операторов компилятор сам определяет какую “функцию” он должен вызвать. Так если вы вызываете оператор + для сложения двух целых аргументов, то будет вызвана функция целочисленного сложения. В том случае, если оператор + используется для сложения целого и дробного аргументов, сначала будет вызвана функция, преобразующее целочисленный аргумент в дробный и затем уже функция сложения аргументов с плавающей точкой.

С++ предоставляет возможность определения операторов для работы с классами, определенными пользователем. Такое определение выглядит как определение обычной функции, за исключением того, что оно начинается с ключевого слова operator:

class foo {

 foo& operator = (const foo&); // перегруженный оператор =

};

В остальном она ведет себя точно так же, как и любая другая функция.

### Синтаксис

Определение перегруженного оператора представляет собой определение обычной функции, с тем исключением, что имя функции должно выглядеть следующим образом operator#, где # - представляет собой перегружаемый оператор. Рассмотрим следующий пример:

class Int {

 int i;

public:

 Int(int ii) : i(ii) {}

 const Int operator + (const Int& rv) const

 {

 return Int(i + rv.i);

 }

 Integer& operator +=(const Int& rv)

 {

 i += rv.i;

 return \*this;

 }

};

Общей для большинства операторов, за исключением условных, является необходимость возвращения объекта или ссылки на объект оперируемого типа. Выполнение этого условия позволяет конструировать сложные выражения:

kk += ii + jj;

Здесь operator+ возвращает временный объект типа Int, ссылка на который используется в качестве параметра rv для operator+=. Этот объект уничтожается сразу же, когда в нем отпадает необходимость.

### Члены и не члены класса

Рассмотрим следующий пример:

class String {

public:

 String (void);

 String (const char \*ps);

 BOOL operator == (const String &s);

 BOOL operator == (const char \*s);

};

String s;

if (s == “test”) { // правильно

} else if (“mark” == s) { // ошибка

}

При первом сравнении вызывается перегруженный operator == (const char \*) класса String. А во втором случае компилятор выдает сообщение об ошибке. Дело в том, что перегруженный оператор, являющийся членом некоторого класса, применяется только тогда, когда левым операндом служит объект этого класса. Поскольку во втором случае левый операнд не принадлежит к классу String, компилятор пытается найти такой оператор, для которого левым операндом является С строка, а правым объект типа String и выдает ошибку.

Решение этой проблемы состоит в определении перегруженного оператора не являющегося членом класса String. В С++ существует такая возможность:

BOOL operator== (const String& s1, const String& s2);

BOOL operator== (const String& s, const char \*pS);

Какой же оператор вызовется для выражения “mark” == s? Мы ведь не определили оператора:

BOOL operator== (const char \*pS, const String &s);

Однако это необязательно. Когда оператор определен в глобальном пространстве имен, то как для первого, так и для второго его параметра рассматриваются возможные преобразования. В результате наше выражение интерпретируется следующим образом:

operator == (String(“mark”, s)

и вызывается:

BOOL operator== (const String& s1, const String& s2);

Итак, на основе чего принимается решение о том, делать ли оператор членом класса или нет:

* если перегруженный оператор является членом класса, то он вызывается лишь при условии, что левым операндом служит член этого класса. Если же левый операнд имеет другой тип, оператор обязан быть глобальным.
* язык требует, чтобы операторы присваивания (“=”), взятия индекса (“[]”), вызова (“()”) и доступа к членам (“->”) были определены как члены класса. В противном случае компилятор выдает ошибку.

В остальных случаях решение принимает проектировщик класса.

### Ограничения

Следует отметить тот факт, что несмотря на возможность переопределения практически всех операторов, предоставляемых языком С, существуют некоторые ограничения. В частности, вы не можете менять количество аргументов и приоритет операторов или комбинировать операторы, таким образом, который не определен в языке С (например, использовать \*\* для экспоненты и т.п.) Существует сравнительно небольшое подмножество стандартных операторов, которые не могут быть перегружены. Основной причиной подобного запрета является безопаcность. Перечислим эти операторы:

* Operator. – оператор доступа к члену класса;
* Operator.\* – dereference оператор;
* Operator\*\* – оператор экспоненты из Фортрана;
* Operator :: - оператор разрешения глобальной области видимости.

### Операторы инкремента и декремента

Перегрузка этих операторов таит в себе потенциальную неоднозначность, поскольку оба они могут быть использованы как в префиксной, так и в постфиксной форме. Реализованное в С++ решение является достаточно простым, хотя многие находят его не интуитивным. Если компилятор встречает выражение ++a, он вызывает operator++(a):

 foo& operator++(foo &f);

 foo& operator—(foo &f);

В случае же a++ – operator++(a, int)

 foo& operator++(foo &f, int i);

 foo& operator—(foo &f, int i);

В случае определения оператора внутри класса это будут operator++() и operator++(int):

 class foo {

 foo& operator++(void);

 foo& operator—(void);

 foo& operator++(int i);

 foo& operator—(int i);

 };

Параметр int никак не используется при определении оператора и служит лишь для того, чтобы различать префиксную и постфиксную формы

### Оператор []

Оператор индексирования – operator[] (можно определять для классов представляющих абстракцию контейнера, из которого извлекаются отдельные элементы). Оператор взятия индекса обязан быть определен как член класса и иметь один аргумент. Этот оператор может появляться, как справа, так и слева от оператора присваивания. Чтобы быть в левой части – он должен возвращать ссылку на индексируемый элемент:

class Array {

public:

 int& operator [] (int i) const { return arr[i]; }

private:

 int arr[5]

};

Array a;

a[0] = 1;

### Оператор ->

Оператор operator-> обеспечивающий доступ к членам класса, так же может быть перегружен. Он должен быть определен как функция член и обеспечивать семантику указателя:

class foo {

public:

 int i;

};

class FooPtr {

public:

 FooPtr (const foo&) : pFoo(&foo) {}

protected:

 foo \*pFoo;

};

Для того чтобы этот класс вел себя как настоящий указатель необходимо определить некоторые перегруженные операторы:

class FooPtr {

 Foo& operator \*() { return \*pFoo; }

 Foo\* operator ->() { return pFoo; }

};

Теперь мы можем использовать FooPtr для доступа к foo:

foo f;

FooPtr fp(f);

int i = fp->i;

### Оператор вызова функции

Оператор вызова функции – operator() может быть переопределен для объектов типа класса. Если определен класс, представляющий некоторую операцию, то для ее вызова перегружается соответствующий оператор. Например, для получения модуля целого числа можно определить следующий класс:

class absInt {

public:

 int operator() (int i) {

 int res = i < 0 ? -i : i;

 return res;

 }

};

Перегруженный оператор operator() должен быть объявлен как функция член с произвольным числом параметров. Параметры и возвращаемое значение могут иметь любые типы допустимые для функции. Применяется operator() следующим образом:

absInt func;

int i = func(-5);

### Операторы new и delete

C++ позволяет перегрузить операторы new и delete. С их помощью класс может реализовать собственную стратегию управления памятью. Если операторы переопределены в пределах класса, то они будут вызываться вместо глобальных операторов с целью выделения и освобождения памяти для объектов этого класса:

class foo {

public:

 void\* operator new(size\_t size);

 void operator delete(void \*ptr);

};

Точно так же можно переопределить операторы new[] и delete[] для работы с массивами.

class foo {

public:

 void\* operator new[](size\_t size);

void operator delete[](void \*ptr);

}

 Когда для создания объекта используется new(), компилятор проверяет определен ли в этом классе такой оператор. Если да, то для выделения памяти под объект вызывается именно он, в противном случае – глобальный оператор new(). Параметр size автоматически инициализируется значением, равным размеру объекта в байтах.

### Операторы ввода/вывода

Если мы хотим, чтобы наш класс поддерживал операции ввода/вывода, то необходимо перегрузить оба соответствующих оператора. Оператор вывода можно перегрузить так:

class A {

 friend ostream& operator << (ostream&, const A&);

 int i;

};

ostream& operator << (ostream& os, const A& a)

{

 os << a.i;

return os;

}

Аналогично для оператора ввода

class A {

 friend ostream& operator >> (ostream&, const A&);

 int i;

};

ostream& operator >> (ostream& os, const A& a)

{

 os >> a.i;

return os;

}

### Присваивание объектов

В С++ присваивание одного объекта другому представляет собой достаточно сложный процесс. Для присвоения одного оператора другому используется оператор operator=.

foo f;

foo f1;

f1 = f;

Если бы f и f1 были бы целыми, то смысл этой строки был бы предельно ясен: содержимое области памяти, на которую ссылается f, копируется в область памяти, на которую ссылается f1. Однако если foo относится к нетривиальному классу, то все существенно усложняется. В приведенном примере, компилятор использует оператор = объявленный по умолчанию, который вызывается для выполнения фактического копирования. Однако иногда версии по умолчанию может оказаться недостаточно. Рассмотрим следующий пример:

class Str {

private:

 char\* s;

public:

 Str(char\*);

 ~Str();

 void Dump(void);

};

Str::Str(char\* str) : s(NULL)

{

 if (str == NULL) {

 s = new char[1];

 \*s = ‘\0’;

 } else {

 s = new char[strlen(str) + 1];

 strcpy(s, str);

 }

}

Str::~Str()

{

 delete s;

}

void Str::Dump(void)

{

 printf(“%s\n”, s);

}

Str s1 = Str(“Hello”);

Str s2 = Str(“Goodbye”);

s2 = s1;

delete s1; // Память освободилась

s2->Dump(); // Ошибка !

delete s2; // Ошибка !

По умолчанию компилятор копирует содержимое s2->s поверх s1->s. При этом копируется значение указателя, а не символы, поэтому после присваивания два разных объекта начинают ссылаться на одну и ту же строку. При удалении s1 деструктор освобождает память, однако s2 продолжает на нее ссылаться. Естественно удаление s2 ведет к непредсказуемым результатам.

Та же проблема возникает и при создании копий, поэтому обычно конструктор копирования и оператор присваивания обычно перегружаются одновременно.

### Присваивание по умолчанию

Оператор = по умолчанию, как и конструктор копий по умолчанию, ведет себя четко определенным образом. Как и конструктор копий, который рекурсивно вызывает другие конструкторы копий, оператор = по умолчанию не ограничивается простым копированием битов из одного объекта в другой, последовательность его действий выглядит следующим образом:

* Присваивание для базовых классов выполняется в порядке их перечисления в списке наследования. При этом используются перегруженные операторы = базовых классов, или в случае их отсутствия – оператор = по умолчанию.
* Присваивание переменных класса выполняется в порядке их перечисления в объявлении класса. При этом используются соответствующие перегруженные операторы =, или в случае их отсутствия – оператор = по умолчанию.

Эти правила применяются рекурсивно, как и в случае с конструкторами.

### Перегрузка оператора =

Перегрузка оператора = практически не отличается от перегрузки всех остальных операторов. Его сигнатура выглядит следующим образом: X& X::operator=(const X&). Рассмотрим следующий пример:

class Str {

private:

 char\* s;

public:

 Str(char\*);

 ~Str();

 Str(const Str&);

 Str& operator=(const Str&);

 void Dump(void);

};

Str::Str(const Str& s1) : s(NULL)

{

 s = new char[strlen(s1.s) + 1];

 strcpy(s, s1.s);

}

String& Str::operator=(const Str& s1)

{

 if (this == &s1) return \*this;

 delete s;

 s = new char[strlen(s1.s) + 1];

 strcpy(s, s1.s);

 return \*this;

}

Конструктор копий и оператор = вместо простого копирования адреса теперь создают копию новой строки. Деструктор теперь стал безопасным. Структура обобщенного оператора = должна выглядеть следующим образом:

* Убедится, что не выполняется присваивание типа x=x. Если левая и правая части ссылаются на один и тот же объект, то делать ничего не надо. Если не перехватить этот особый случай, то следующий шаг уничтожит значение до того, как оно будет скопировано.
* Удалить предыдущие данные.
* Скопировать значение.
* Возвратить указатель \*this.

## Наследование и композиция

Одним из наиболее значимых свойств С++ является повторное использование кода. В С повторное использование кода, представляет собой обычное копирование и модификацию существующих фрагментов. В С++ повторное использование кода тесно связано с понятием класса. Оно достигается путем создания новых классов, причем классы создаются не с нуля, а с использованием уже существующих и отлаженных классов.

Это может быть достигнуто двумя путями. Первый из них достаточно прост и очевиден – вы просто создаете объекты существующего класса в рамках вашего нового класса.

Такой подход носит название композиции (composition), поскольку новый класс представляет собой композицию объектов существующих классов. Второй способ состоит в том, что вы создаете новый класс, того же типа что и существующий, вносите в него изменения, не затрагивая при этом кода базового класса. Этот подход называется наследование (inheritance). Естественно композиция и наследования могут использоваться вместе. Более того, во многих случаях это является наиболее правильным решением поставленной задачи.

### Композиция

На самом деле до сих пор мы использовали композицию для создания классов. Только в качестве элементов композиции выступали встроенные типы. Точно также можно использовать композицию для типов, определенных пользователем:

class foo {

 int i;

public:

 foo (void) : i(0) { }

 int get (void) { return i; }

 void set (int ii) { i = ii; }

};

class foo1 {

 int i;

public:

 foo f;

public:

 foo1 (void) : i(0) { }

 int get (void) { return i; }

 void set (int ii) { i = ii; }

};

foo1 f1;

f1.set(10);

f1.f.set(10);

### Наследование

При использовании наследования между именем класса и открывающейся скобкой необходимо после: указать имя базового класса. Когда вы делаете это, вы автоматически включаете все данные и функции-члены базового класса в ваш новый класс.

class A {

 int i;

public:

 A (void) : i(0) {}

};

class B : public A {

 int i1;

 public:

 B (void) : i1(0) { }

};

Таким образом класс foo1 является наследником foo. Атрибут public после ‘:’ определяет уровень доступа к членам базового класса в производном классе.

#### Доступ к членам базового класса

Объект произвольного класса фактически формируется из нескольких частей. Каждый их предшествующих ему классов в иерархии вносит свой вклад в виде подобъекта, составленного из нестатических членов этого класса. Таким образом получается, что объект производного класса состоит из подобъектов, соответствующих каждому из базовых классов, а также из части, включающей нестатические члены самого класса.

Внутри произвольного класса к членам, унаследованным из базового класса можно обращаться напрямую как своим собственным (если не принимать во внимание уровни доступа, о которых мы поговорим позднее). Тоже самое можно сказать и про функции базового класса.

class A {

public:

 int i;

 void func();

};

class B : public A {

public:

 void f() { i = 0; func(); }

};

Однако доступ из производного класса к члену базового запрещен, если имя последнего скрыто в производном классе:

class A {

public:

 int i;

};

class B : public A {

public:

 char \*i; // скрывает A::i

};

Для того, чтобы обратиться к члену класса A из функции класса B надо явно указать его принадлежность с помощью оператора разрешения области видимости:

void B::f (void)

{

 i = NULL;

 A::i = 1;

}

Еще раз рассмотрим уровни доступа, теперь применительно к наследованию:

* **private** – закрытые члены не видны никому кроме функций класса и друзей, соответственно мы не можем достучаться до закрытых членов класса в производных классах.
* **protected –** защищенные члены класса доступны в пределах иерархии во всех производных классах
* **public –** члены доступны везде в программе

#### Открытое, закрытое и защищенное наследование

Открытое наследование, еще называют наследованием типа. Производный класс в этом случае является подтипом базового. В C++ открытое наследование реализовано с использованием модификатора public:

class B : public A {}

В данном случае, все открытые и защищенные члены будут доступны в производном классе, а все закрытые – соответственно доступны не будут.

Закрытое наследование, называют также наследование реализации. Производный класс напрямую не поддерживает открытый интерфейс базового, но пользуется его реализацией, предоставляя свой собственный открытый интерфейс. Такое наследование реализуется с помощью модификатора private

сlass B : private A {}

или

сlass B : A {}

В этом случае, все даже открытые элементы базового класса будут закрыты в производном классе.

Существует еще понятие защищенного наследования (protected inheritance). При его использовании все открытые и защищенные члены базового класса становятся защищенными членами производного класса, т.е. доступными только в пределах иерархии классов, но не во всей остальной программе. Реализовано с использованием модификатора protected:

сlass B : protected A {}

### Что лучше композиция или наследование

В принципе, композиция и наследование позволяют достичь во многом одного и того же, а именно размещения объектов существующих классов в вашем новом классе. В чем же разница между ними и какой из подходов, когда следует применять.

Композиция обычно используется, когда вы хотите включить новый объект, для того чтобы получить какие-то его особенности внутри вашего класса, но не его интерфейс. В этом случае пользователь получает интерфейс, который вы определяете для нового класса, а не интерфейс существующего класса. Хорошим примером использования композиции является пример машины:

class Engine {

public:

 void start();

 void stop();

};

class Wheel {

public:

 void inflate(int psi);

};

class Window {

public:

 void rollup();

 void rolldown();

};

class Door {

public:

 Window window;

 void open();

 void close();

};

class Car {

public:

 Engine engine;

 Wheel wheel[4];

 Door left, right; // 2-door

};

#### Оператор присваивания и наследование

При перегрузке оператора = вся ответственность за выполнение присваивания для переменных базовых классов ложится на вас, по умолчанию базовые классы и переменные левостороннего объекта остаются без изменений. Рассмотрим следующий пример:

class Base {

public:

 Base(int initialValue = 0): x(initialValue) {}

private:

 int x;

};

class Derived: public Base {

public:

 Derived(int initialValue)

 : Base(initialValue), y(initialValue) {}

 Derived& operator=(const Derived& rhs);

private:

 int y;

};

Вполне логичной кажется следующая версия оператора = для класса Derived:

Derived& Derived::operator=(const Derived& rhs)

{

 if (this == &rhs) return \*this;

 y = rhs.y;

 return \*this;

}

К сожалению этот вариант неправильный, из-за того, что переменная x остается неизменной. Правильная версия выглядит следующим образом:

Derived& Derived::operator=(const Derived& rhs)

{

 if (this == &rhs) return \*this;

 Base::operator=(rhs);

 y = rhs.y;

 return \*this;

}

#### Конструктор копирования и наследование

Предположим, что вы перегрузили конструктор копий для какого-то класса. При отсутствии явных вызовов, конструкторов копий базовых классов и переменных класса в списке инициализации членов, компилятор будет использовать конструктор без аргументов для инициализации базовых классов и переменных. Рассмотрим следующий пример:

class B {

public:

 B (int i = 0): x(i) {}

 B (const B& rB): x(rB.x) {}

private:

 int x;

};

class D: public B {

public:

 D (int i) : B(i), y(i) {}

 D (const D& rD) : y(rD.y) {} // Ошибка !!!

private:

 int y;

};

В данном случае мы имеем дело с одной из наиболее трудно обнаружимых ошибок в С++. Дело в том, что здесь базовая часть класса не копируется. Естественно она конструируется, однако конструируется с помощью конструктора по умолчанию, в результате переменная класса x – инициализируется нулем, независимо от ее значения в копируемом объекте.

Для предотвращения этой ситуации необходимо, явно вызвать конструктор копий для Base внутри конструктора копий для D.

class D: public B {

public:

 D(const D& rD): B(rD), y(rD.y) {}

 ...

};

### Множественное наследование

В некоторых случаях бывает удобно наследовать свойства сразу нескольких базовых классов. Такой подход носит название множественного наследования. Целесообразность его использования до сих пор является предметом для яростных споров среди программистов на С++. Единственно в чем сходятся сторонники и противники использования множественного наследования, так это в том, что использовать его стоит лишь только в том случае если ваш опыт работы с С++ уже достаточно велик. Дело в том, что на первый взгляд множественное наследование не несет никакой угрозы. Действительно, это ведь просто указание не одного, а нескольких классов после ‘:’ в объявлении класса. однако его использование связано с пониманием ряда не вполне очевидных моментов. Рассмотрим некоторые из них.

#### Дублирование подобъекты

Когда вы наследуетесь от некоторого базового класса, вы получаете копию всех его данных-членов внутри производного класса. Такую копию обычно называют подобъектом (subobject). В случае наследования от двух базовых классов производный класс содержит два таких подобъекта.



Теперь рассмотрим что происходит если d1 и d2 являются производными от одного и того же базового класса Base.



Легко видеть, что в этом случае mi содержит два подобъекта base. Подобная форма наследования носит название ромба. Если не принимать во внимание случай ромбов (diamond) идея множественного наследования вполне проста и понятна. Однако ромбы несут в себе большой запас различного рода трудностей и неоднозначностей.

#### Виртуальные базовые классы

Что произойдет если мы попытаемся привести указатель на mi к указателю на base. В пределах mi есть два подобъекта base, какой же из них мы получим в результате такого приведения? На самом деле такое приведение типов просто невозможно.

Для решения этой проблемы используется так называемое виртуальное наследование. Если вы наследуете базовый класс как виртуальный, то только один его подобъект появится в производном классе. Виртуальные базовые классы реализуются компилятором с помощью специального механизма, напоминающего механизм, используемый для обычных виртуальных функций. На самом деле при использовании механизма виртуального наследования в d1 и d2 размещается указатель на базовый класс, а не сам класс, как в случае обычного наследования.

## Полиморфизм и виртуальные функции

Полиморфизм реализованный в С++ посредством, так называемых, виртуальных функций является третьим важнейшим столпом объектно-ориентированного программирования, наряду с абстракцией данных и наследованием.

### Виртуальные функции

Для того чтобы задействовать механизм позднего связывания для конкретной функции в С++ необходимо использовать ключевое слово virtual при определении функции в теле класса:

class foo {

public:

 virtual void func(void);

};

Позднее связывание осуществляется только для виртуальных функций и только когда вы используете указатель на базовый класс, в котором определены эти функции (хотя они могут быть определены и в базовых классах, расположенных выше по иерархии). Функция, объявленная виртуальной в базовом классе, является таковой для всех производных классов. Переопределение виртуальной функции в производном классе называется overriding.

### Реализация позднего связывания в С++

Рассмотрим несколько подробнее как в С++ реализован механизм позднего связывания. Обычно компилятор создает таблицу виртуальных функций для каждого класса. Адреса виртуальных функций для данного конкретного класса помещаются в эту таблицу. В каждом таком классе компилятор размещает специальный указатель на таблицу виртуальных функций. Когда вы вызываете виртуальную функцию используя указатель на базовый класс, компилятор вставляет в этом месте код, который вызывает нужную функцию, используя соответствующую таблицу. Все это происходит автоматически, так что вам нет нужны ни о чем беспокоится, компилятор все сделает за вас.

### Эффективность виртуальных функций

К этому моменту уже может возникнуть вопрос – если виртуальные функции настолько полезны и удобны в использовании, то почему они выделены в отдельное понятие. Почему бы нам не сделать все функции виртуальными? Ответ - просто это недостаточно эффективно. Действительно, в случае позднего связывания вместо простого вызова функции с помощью ассемблерной инструкции call, происходит поиск ее адреса в таблице и уже последующий вызов, что гораздо медленнее и в критических случаях может оказать драматический эффект на общую производительность всей программы.

### Абстрактные базовые классы и чисто виртуальные функции

Во многих случаях удобно, если базовый класс представляет только интерфейс для производных классов. То есть, вы предполагаете, что никто не будет создавать объекты этого класса, а сам класс будет использоваться только для наследования. Это достигается использованием, так называемых, абстрактных классов. Абстрактным называется класс, который имеет хотя бы одну чисто виртуальную функцию (pure virtual function). Подобные функции описываются следующим образом:

class foo {

public:

 virtual func (void) = 0;

};

При наследовании от абстрактного класса, все чисто виртуальные функции должны быть определены, иначе класс так же считается абстрактным. Компилятор не позволит вам создать экземпляр абстрактного класса.

### Отрезание объектов

При использовании полиморфизма существует существенная разница между передачей объекта в функцию по ссылке и по значению. В случае передачи объекта по значению он будет “срезан” таким образом, чтобы удовлетворить типу соответствующего параметра функции. В результате, в функцию будет передан объект базового типа и ее поведение будет не совсем таким как бы вам этого хотелось.

class A {

public:

virtual void f() { printf(“A”); }

};

class B : public A {

public:

 virtual void f () { printf(“B”); }

};

void f1(A a) { a.f();}

void f2(A &a) { a.f();}

B b;

f1(b); // A

f2(b); // B

###

### Виртуальные функции и конструкторы

При создании объекта, содержащего хотя бы одну виртуальную функцию, производится его дополнительная инициализация в ходе которой заполняется таблица виртуальных функции. Разумеется, этот процесс должен быть завершен до вызова виртуальных функций. Поэтому компилятор вставляет соответствующий код в тело конструктора. Этот факт следует учитывать и помнить, что простая на вид функция, которой, как правило, является ваш конструктор на самом деле скрывает за собой некоторый дополнительный объем кода.

Как уже было сказано выше, конструкторы всех базовых классов вызываются в конструкторе производного класса. Что произойдет если мы вызовем виртуальную функцию внутри конструктора. В случае обычной функции будет использован механизм позднего связывания. Однако в случае конструктора будет использована локальная версия функции. Таким образом механизм виртуальных функции не работает внутри конструкторов.

### Виртуальные деструкторы

Ключевое слово virtual не может быть использовано применительно к конструкторам, в то же время деструкторы могут и очень часто должны быть виртуальными.

Обычно механизм деструкторов справляется со своей задачей, однако проблема возникает в случае если вы хотите работать с объектом используя указатель на базовый класс. Как правило это происходит при удалении объектов, которые были созданы с помощью оператора new. При вызове delete для указателя на базовый класс будет вызван деструктор базового класса. Для решения подобной проблемы и были введены виртуальные функции. Этот механизм работает и для деструкторов. Таким образом все что необходимо сделать это объявить деструктор виртуальным.

### Виртуальные функции и деструкторы

При вызове виртуальной функции внутри деструктора используется локальная версия функции и позднее связывание игнорируется. Причины такого ограничения вполне очевидны. Дело в том, что при разрешении вызова функции с помощью механизма позднего связывания, он вполне мог бы привести к вызову функции из производного класса, для которого уже был вызван деструктор, в силу того, что деструкторы вызываются в обратном порядке. Что естественно может привести к не вполне предсказуемым последствиям.

### Видимость перегруженных и виртуальных функций класса

Если в базовом классе функция объявлена не виртуальной, то превращать ее в виртуальную в производном классе не рекомендуется.

class A {

public:

 void f();

};

class B : public A {

public:

 virtual void f();

};

Она поведет себя не так как виртуальная и существенно осложнит понимание вашей программы. Однако на ситуацию можно взглянуть и под другим углом. На самом деле ключевое слово virtual обязано присутствовать только в базовом классе. Если оно пропущено в производном классе, компилятор интерпретирует версию функции так, как будто она и там была объявлена виртуальной. В следующем примере для обоих указателей будет вызвана функция B::f():

class A {

public:

virtual void f();

};

class B : public A {

public:

void f(); // Все равно считается виртуальной

};

B\* b = new B;

b->f(); // Вызывает B::f()

A\* a = b;

a->f(); // Также вызывает B::f()

Подобные ситуации не желательны в силу того, что их наличие существенно затрудняет понимание программы, поэтому их следует избегать.

Если в производном классе создается функция с тем же именем, но с другой сигнатурой, она скрывает все сигнатуры базового класса для данной функции, но только в области действия производного класса. Рассмотрим следующий пример:

class A {

public:

virtual void f();

virtual void f(char\*);

};

class B : public A {

public:

virtual void f(int); // Можно, но не желательно

};

Попробуем разобраться что происходит в данном случае:

* При попытке вызова f() через B\* доступной будет лишь сигнатура void f(int). Обе версии базового класса скрыты и недоступны через B\*
* При преобразовании B\* в A\* становятся доступными обе сигнатуры, добавленные в A но не сигнатура void f(int). Более того, это не переопределение, поскольку сигнатура B::f отличается от версии базового класса. Другими словами, ключевое слово virtual никак не влияет на работу этого фрагмента.

На самом деле подобные вещи делать не рекомендуется из-за их неочевидности и сложности в понимании.

## Определенные пользователем преобразования

В С++ можно сделать так чтобы пользовательский тип автоматически преобразовывался в другие типы. Сделать это можно несколькими способами.

### Конструктор как конвертер

Набор конструкторов класса, принимающих один параметр определяет множество неявных преобразований для этого класса. Пусть у нас есть класс A с двумя конструкторами из int-а и float-а.

class A {

public:

 A (int i) {}

};

Тогда в программе мы можем написать:

void f(A a);

int i = 0;

f(i);

При вызове функций аргумент автоматически преобразуется в класс А при наличии соответствующего конструктора. Затем копия этого объекта передается внутрь функции.

Возникает вопрос, если конструктор используется для неявного преобразования, должен ли тип его параметра точно соответствовать преобразуемому типу? Например, будет ли в следующем примере вызван конструктор A(int):

float fl = 0.f;

f(fl);

Ответ да. Компилятор выполняет неявное преобразование float к int-у и затем вызывает конструктор A(int).

### Конвертер

В С++ в пределах класса можно определить конвертер. Конвертер – это особый случай функции-члена, реализующий преобразование объекта в некоторый другой тип. Конвертер объявляется в теле класса путем указания ключевого слова operator, за которым следует целевой тип преобразования.

class A {

public:

operator int() { return i; }

operator float() { return f; }

private:

 int i;

 float f;

};

Эти конвертеры используются для неявного приведения типов, например, пусть у нас есть функция принимающая на вход int и мы передаем в нее экземпляр класса A:

void f(int i);

void f1(float f);

A a;

f(a);

f1(a);

Общий вид конвертера следующий:

operator type()

Здесь type может быть встроенным типом, типом класса или именем typedef. Конвертеры, в которых в качестве типа используется тип массива или функции не поддерживаются. Конвертер должен быть функцией членом, в его объявлении не должны задаваться ни тип возвращаемого значения, ни список параметров.

### Выбор преобразования

Определенное пользователем преобразование реализуется в виде конвертера или конструктора. После преобразования, выполненного конвертером разрешается использовать стандартные преобразования, точно так же трансформации выполненной конструктором может предшествовать стандартное преобразование. Таким образом в общем случае последовательность преобразований выглядит следующим образом:

* Последовательность стандартных преобразований
* Определенное пользовательское преобразование
* Последовательность стандартных преобразований

Где определенное пользовательское преобразование реализуется конвертером либо конструктором.

Может случиться, что для трансформации существует две разных последовательности преобразований и тогда компилятор должен выбрать из них лучшую. Рассмотрим как это делается. Пусть у нас есть класс, в котором определены два конвертера:

class A {

 operator int();

 operator float();

};

 Оба эти конвертера подходят для преобразования в тип float. В случае использования float() имеет место точное соответствие. Точное соответствие всегда лучше и выбирается оно. Может быть, что в классе реализованы два конструктора преобразования к типу:

class A {

public:

A (int i);

 A (float f);

};

В этом случае, для преобразования в float анализируется последовательность стандартных преобразований, предшествующих вызову конструктора. В данном случае точное преобразование опять же имеет преимущество. Может оказаться так что все преобразования одинаково хороши и тогда преобразование является неоднозначным.

## Шаблоны

Определения шаблона можно сформулировать следующим образом – это предписание для создания класса, в котором один или несколько типов либо значений параметризированы.

### Что такое шаблоны и зачем они нужны

Рассмотрим следующий простой пример класса-коллекции:

class Node {

private:

Node\* next;

void\* data;

public:

Node(void\* d, Node\* n = NULL) : next(n), data(d) {}

~Node() { delete next; }

void\* Data() { return data; }

Node\* Next() { return next; }

};

Очевидным и бросающимся в глаза недостатком такого дизайна является чрезмерное использование void-указателей. В результате код, работающий с подобной коллекцией принимает следующий вид:

for (Node\* n = listHead; n != NULL; n = n->Next())

f((Foo\*)n->Data());

Иначе говоря, нам придется постоянно приводить void\* к какому-то конкретному типу, что разумеется далеко не безопасно. Ведь никто не может гарантировать, что в коллекцию не будет занесен объект другого типа. Вторая проблема состоит в том, что элементы списка сами не знают на какой тип они указывают, в результате они не могут быть просто уничтожены с использованием delete.

Одним из возможных решений является требование того, чтобы все объекты нашей коллекции происходили от общего предка. В этом случае void\* можно будет заменить на указатель на базовый класс. Таким образом будет решена проблема удаления элементов коллекции.

Другое решение состоит в том, чтобы создать список, рассчитанный на конкретный тип. Это, казалось бы, решит все наши проблемы, однако в результате мы получим множество классов двойников, отличающихся только типами, с которыми они работают. В языке С подобные проблемы решались с использованием макроса #define:

#define Node(Type) \

class Node##Type { \

private: \

Node##Type\* next; \

Type\* data; \

public: \

Node##Type(Type\* d, Node\* n = NULL) : next(n), data(d) {} \

~Node() { delete next; } \

void\* Data() { return data; } \

Node\* Next() { return next; } \

};

Такая методика обладает многочисленными недостатками. Если функции класса не являются inline, вам придется писать отдельные макросы для их определения и следить чтобы они были реализованы в одном модуле компиляции.

### Шаблоны – усовершенствованные макросы

На самом деле механизм шаблонов, представляет собой усовершенствованный макропроцессор для директив #define. Шаблоны представляют собой ничто иное как макросы, безо всех перечисленных выше ограничений. Они могут быть вложенными. При возникновении ошибки отладчик правильно укажет вам соответствующую строку шаблона. В С++ шаблоны используются для описания параметризованных функций и параметризованных типов.

### Параметризованные функции

Параметризованные функции позволяют нам описать целое семейство функций. Описание такой функции похоже на описание обычной функции с тем исключением, что некоторые ее элементы параметризуются. Рассмотрим следующий пример:

template <class T> inline T const& max (T const& a, T const& b)

{

 // if a < b then use b else use a

 return a<b?b:a;

}

Этот шаблон определяет семейство функций, возвращающих максимум двух величин, передаваемых в виде параметров, а и b.

#### Конкретизация шаблона

Обычно шаблонная функция не компилируется в некую единую сущность, которая может работать с параметрами любого типа. Вместо этого на основе шаблона создаются различные сущности при каждом его использовании совместно с новой комбинации типов. Процесс замены параметров шаблона на конкретные параметры носит название конкретизации (instantiation) шаблона.

Попытка конкретизации шаблона для типа, которые не поддерживает каких-то из используемых в рамках шаблона операции приведет к ошибке компиляции. Таким образом шаблоны компилируются дважды:

* Без конкретизации код шаблона проверяется на синтаксическую правильность.
* В момент конкретизации, код шаблона проверяется с целью убедится что все вызовы правильны.

#### Параметры шаблона

Когда мы вызываем шаблон max для каких-то аргументов, параметры шаблона определяются именно этими аргументами. Если мы передаем два целых значения, то компилятор делает вывод, что T – это целое. В случае шаблонов не применяется никакого автоматического приведения типов. Например:

template <typename T>

inline T const& max (const T &a, const T &b);

…

max(4,7) // OK: T is int for both arguments

max(4,4.2) // ERROR: first T is int, second T is double

Существует несколько способов избежать последней ошибки. Первый из них состоит в том, чтобы явно привести аргументы к одному типу:

max(static\_cast<double>(4),4.2);

Второй способ - явно указать тип T:

max<double>(4,4.2);

И наконец третий способ состоит в том, чтобы указать при определении шаблона, что параметры могут быть разных типов, например следующим образом:

template <typename T1, typename T2>

inline T1 max (const T1 &a, const T2 &b)

{

 return a < b ? b : a;

}

…

max(4, 4.2); // OK

Однако в таком, казалось бы очевидном, определении max() кроется одна ошибка. Дело в том что в данном случае тип возвращаемого значения будет зависеть от порядка параметров. Еще одним недостатком является то, что при приведении типов будет создаваться временый объект, как следствие вы не сможете вернуть результат по ссылке. Можно кончено ввести третий аргумент шаблона:

template <typename T1, typename T2, typename RT>

inline RT max (const T1 &a, const T2 &b);

Однако поскокльку тип резульата не указывается при вызове функции вы будете вынуждены использовать следующий формат вызова:

max<int,double,double>(4,4.2)

К счастью компилятор позволяет нам указать тип первого аргумента явно, а определение типа остальных оставить на его совести. Таким образом мы приходим к уже знакомой нам записи:

max<double>(4,4.2)

Таким образом легко видеть, что проще всего использовать однопараметорную версию шаблона.

#### Перегрузка параметризованных функций

Параметризованная функция может быть перегружена точно так же, как и обычная функция. Рассмотрим следующий пример:

inline int const& max (int const& a, int const& b)

{

 return a<b?b:a;

}

template <typename T> inline T const& max (T const& a, T const& b)

{

 return a<b?b:a;

}

template <typename T>

inline T const& max (T const& a, T const& b, T const& c)

{

 return max (max(a,b), c);

}

int main()

{

 max(7, 42, 68); // calls the template for three arguments

 max(7.0, 42.0); // calls max<double> (by argument deduction)

 max('a', 'b'); // calls max<char> (by argument deduction)

 max(7, 42); // calls the nontemplate for two ints

 max<>(7, 42); // calls max<int> (by argument deduction)

 max<double>(7, 42); // calls max<double> (no argument deduction)

 max('a', 42.7); // calls the nontemplate for two ints

}

Как видно из этого примера не параметризованные функции могут вполне естественно сочетаться со своими параметризованными аналогами. Общее правило состоит в том, что при прочих равных условиях предпочтение отдается в пользу не параметризованных функций. Что и происходит в четвертом вызове функции:

max(7, 42) // both int values match the nontemplate function

// perfectly

В то же время если на основе шаблона может быть получено лучшее соответствие предпочтение отдается шаблону:

max(7.0, 42.0) // calls the max<double> (by argument deduction)

max('a', 'b'); // calls the max<char> (by argument deduction)

Существует так же возможность явно указать что при разрешении данного вызова следует использовать только шаблоны. Для этого сразу после имени функции необходимо поместить <>.

Еще одна тонкость, которую следует отметить состоит в том, что при конкретизации шаблонов невозможно приведение типов, поэтому в последнем случае выбор делается в пользу не параметризованного варианта:

max('a', 42.7) // only the nontemplate function allows

// different argument types

### Параметризованные типы

Аналогично функциям классы так же могут быть параметризованы одним или несколькими параметрами. Определение шаблона класса во многим аналогично определению шаблона функции:

template <typename T>

class Node {

 …

};

#### Описание шаблона класса

В пределах шаблона класса параметра T может быть использован точно так же, как и любой другой класс при определении данных и функций класса.

template <typename T> class Node {

private:

Node<T> \*next;

T \*data;

public:

Node(T\* d, Node<T>\* n = NULL) : next(n), data(d) {}

~Node();

T\* Data();

Node<T>\* Next();

};

#### Описание функций параметризованного класса

Для того, чтобы описать функции члены шаблонного класса необходимо указать что это шаблон функции и использовать при этом полное именование шаблона класса:

template <class Type>

Node<Type>::Node(Type\* d, Node<Type>\* n = NULL)

: next(n), data(d)

{

}

template <class Type> Type\* Node<Type>::Data()

{

 return data;

}

Для того чтобы использовать объект шаблонного класса необходимо явно указать аргумент шаблона. Например,

Node<Foo> list = new Node<Foo> (new Foo);

Foo\* f = list->Data();

Следует отметить что компилятор создает код только для тех функций шаблона, которые используются в программе. Помимо того, что это свойство экономит место и время, оно позволяет нам использовать шаблон для тех классов, которые поддерживают не все множество функций, использованных в шаблоне. Естественно, что это возможно только при том условии что мы не будем вызывать соответствующих функций шаблона.

#### Специализация шаблонов классов

Вы можете специализировать шаблон класса для некоторых значений аргументов шаблона. Такая возможность позволяет вам, например, оптимизировать код для каких-то частных случаев или наоборот разрешить какие-то противоречия в основном коде шаблона. Однако если вы специализируете шаблон класса, то вы обязаны специализировать все его функции члены. Существует возможность специализации только функции-члена класса, однако в этом случае вы уже не можете специализировать весь класс.

Для того чтобы специализировать шаблон класса необходимо объявить его следующим образом:

template<> class Node<float> {

 …

};

Специализации функции-члена выглядит точно так же, как и объявление функции-члена шаблона, за тем исключением что каждое вхождение T заменяется на специализируемый тип:

template <float> Node<float>\* Node<float>::Next()

{

return next;

}

#### Частичная специализация

Шаблоны классов могут быть частично специализированы. Вы можете определить специальные варианты реализации для каких-то особых обстоятельств, однако в отличие от полной специализации, описанной выше, здесь часть параметров может быть определена пользователем. Рассмотрим следующий пример:

template <typename T1, typename T2> class MyClass {

 …

};

В данном случае возможны следующие варианты частичной специализации:

// partial specialization: both template parameters have same type

template <typename T> class MyClass<T,T> {

 …

};

// partial specialization: second type is int

template <typename T> class MyClass<T,int> {

 …

};

// partial specialization: both template parameters are pointer

// types

template <typename T1, typename T2> class MyClass<T1\*,T2\*> {

 …

};

Следующий пример показывает, в каких случаях при конкретизации используются приведенные выше шаблоны:

MyClass<int,float> mif; // uses MyClass<T1,T2>

MyClass<float,float> mff; // uses MyClass<T,T>

MyClass<float,int> mfi; // uses MyClass<T,int>

MyClass<int\*,float\*> mp; // uses MyClass<T1\*,T2\*>

Если при описании переменной два или более варианта подходят одинаково хорошо – определение признается неоднозначным, например,

MyClass<int,int> m; // ERROR: matches MyClass<T,T>

 // and MyClass<T,int>

MyClass<int\*,int\*> m; // ERROR: matches MyClass<T,T>

 // and MyClass<T1\*,T2\*>

Для разрешения последней неоднозначности можно предоставить дополнительную специализацию следующего вида:

template <typename T> class MyClass<T\*,T\*> {

 …

};

#### Параметры по умолчанию

При определении шаблона класса существует возможность задания параметров по умолчанию. Такие значения носят название параметры шаблона по умолчанию (default template arguments). Например,

template <typename T1, typename T2 = int> class MyClass {

 …

};

Следует отметить, что в роли параметров по умолчанию могут выступать другие шаблоны. Такие параметры носят название шаблонных шаблонных параметров (template template parameters):

template <typename T1, typename T2 = std::vector<T> > class MyClass {

 …

};

### Обобщенное программирование

Понятие статического полиморфизма тесно связано с дисциплиной обобщенного программирования (generic programming). К сожалению, на данный момент не существует общего определения обобщенного программирования, точно так же, как и не существует определения понятия объектно-ориентированного программирования. Можно сказать, что:

 Обобщенное программирование — это подраздел компьютерной науки, занимающийся поиском абстрактного представления эффективных алгоритмов, структур данных и других программных концепций, а также их систематизацией.

В контексте С++ под обобщенным программированием иногда понимают программирование с использованием шаблонов. Однако это не совсем так, подобно тому что объектно-ориентированное программирование – не есть просто программирование с использованием виртуальных функций.

## STL

На данный момент наиболее известным примером обобщенного программирования является STL (standard template library). STL предоставляет пользователю множество стандартных операций, называемых алгоритмами, работающих с множеством линейных структур данных представляющих коллекции объектов – контейнерами. Как алгоритмы, так и контейнеры реализованы с помощью шаблонов.

### Контейнеры

В STL можно выделить два вида контейнеров:

* последовательные (sequence containers) – упорядоченная коллекция элементов, в которой каждый элемент занимает определенную позицию. Позиция может зависеть от времени и места вставки, но не зависит от величины элемента. В STL можно выделить три таких контейнера – vector, list и deque
* ассоциативные (associative containers) – сортированная коллекция элементов, в которой положение элемента определяется на основе его значения в соответствие с какими-то критериями. STL содержит четыре таких контейнера – map, multimap, set и multiset.

В данном случае сортированный не означает что именно эти контейнеры лучше использовать при сортировке. Сортировать можно элементы и в последовательных контейнерах, однако ассоциативные контейнеры во многих случаях луче подходят для поиска.

#### Vector

Можно сказать, что vector это динамический массив со всеми преимуществами и недостатками массива. Добавление и удаление элементов в вектор очень быстрое. Вставка в тоже время достаточно долгая операция потому что она связана с перемещением элементов в памяти. Вектор предоставляет прямой доступ к своим элементам. В программе вектор можно определить следующим образом:

vector<int> a;

Добавление элемента в вектор реализовано с помощью функции push\_back.

a.push\_back(0);

Доступ к элементам вектора осуществляется с помощью оператора индексирования:

a[1] = 0;

Размер вектора можно узнать используя функцию size(). Эта функция возвращает количество элементов в векторе. Реальны размер занимаемый в памяти можно узнать с помощью функции capacity(). С помощью функции reserve можно зарезервировать количество элементов в векторе, то есть заранее выделить память под какое-то количество элементов.

vector<int> a;

a.reserve(10)

#### Deque

Deque - это двусторонняя очередь (double ended queue). На самом деле это динамический массив реализованный таким образом, что он может расти в двух направлениях. Таким образом обеспечивается быстрое добавление элементов в начало и конец, при этом добавление элементов в середину достаточно длинная операция, поскольку она связана с копированием в памяти. В программе deque можно определить следующим образом:

deque<float> d;

Интерфейс deque практически не отличается от интерфейса вектора за исключением нескольких функций, например push\_front и pop\_front. Доступ к элементам так же как и в векторе осуществляется с помощью оператора индексирования.

#### List

List – это двунаправленный список со всеми его достоинствами и недостатками. Как-то быстрая вставка и удаление, но медленный случайный доступ. В программе list можно определить следующим образом:

list<float> l;

Добавление элементов реализовано с помощью функций push\_back и push\_front. У списка есть такие методы как sort и reverse, которые более эффективны с точки зрения работы с памятью нежели чем стандартные sort и reverse, которые применимы ко всем контейнерам.

#### Set и multiset

Set это сортированная коллекция уникальных элементов. В программе set определяется следующим образом:

set<int> l;

Добавление в множество осуществляется с помощью функции insert. Multiset это тоже самое что и set, только позволяет хранить повторяющиеся элементы.

#### Map и multimap

Map это ассоциативный контейнер хранящий пару ключ/значение. В программе map можно объявить следующим образом:

map<char, int> m;

Первый параметр — это ключ, второй – значение. Добавление в map осуществляется с помощью оператора индексации, например,

m[‘a’] = 10;

m[‘b’] = 20;

Кроме того, возможно добавление элементов с помощью функции insert и специальной функции make\_pair:

m.insert(make\_pair(‘d’, 30));

Multimap это map, который содержит дубликаты для одного и того же ключа. При этом в multimap не определен оператор индексирования, в отличие от map.

#### Контейнеры адаптеры

В дополнение к основным классам контейнеров C++ предоставляет специальные контейнеры, которые отличаются поведением и основаны на использовании основных контейнеров.

* **Stack** – LIFO контейнер. По умолчанию использует deque в качестве контейнера.
* **Queue** – FIFO контейнер. По умолчанию использует deque в качестве контейнера
* **Priority Queue** – очередь с приоритетами. Контейнер по умолчанию – vector. Оператор сортировки тоже может быть задан, по умолчанию используется оператор <.

На самом деле контейнер, используемый в пределах адаптера может быть задан в качестве параметра шаблона.

### Итераторы

Основной особенностью STL является тот факт, что алгоритмы не являются функциями-членами контейнеров. Вместо этого алгоритмы реализованы таким образом, чтобы они могли быть использованы для любого контейнера. Для этого в STL была предложена концепция итераторов, которые могут быть предоставлены для любой коллекции. В результате, например, вычисление максимума последовательности может быть реализовано не зависимо от внутренней структуры этой последовательности:

template <class Iterator>

Iterator max\_element (Iterator beg, Iterator end)

{

 …

}

Таким образом, вместо того чтобы определять каждую операцию для каждого контейнера, в STL каждый алгоритм определяется только один раз и может быть использован для любого контейнера.

Поведение итератора определяется следующими операторами:

* **operator\*** - возвращает элемент в данной позиции итератора
* **operator++** - перемещает итератор на следующий элемент
* **operators = и !=** - определяют если итераторы находятся на той же позиции
* **operator= -** присваивает один итератор другом (перемещает его в нужную позицию)

Все классы контейнеров предоставляют следующий интерфейс:

* **begin()** – возвращает итератор на начало контейнера
* **end()** – итератор на конец контейнера, технически это элемент следующий за последним.

Таким образом с использованием итераторов код прохода по контейнеру будет выглядеть следующим образом:

list<int> l;

for (int i = 0; i < 10; i++) {

 l.push\_back(i);

}

for (list<int>::iterator it = l.begin(); it != l.end(); ++it) {

 cout << \*it << endl;

}

Каждый контейнер определяет два типа итераторов – обычный итератор ::iterator и константный итератор ::iterator.

### Алгоритмы

STL предоставляет набор стандартных алгоритмов, для работы с контейнерами, таких как поиск, сортировка и т.п. Алгоритмы представляют собой отдельные функции и в теории могут быть использованы для любого контейнера. На практике оказывается, что не все алгоритмы могут быть использованы со всеми контейнерами, кроме того в некоторых случаях это приводит к каким-то не очевидным интерфейсам и ограничениям.

Алгоритмы работают с диапазонами элементов контейнера, которые задаются с помощью итераторов. Например,

list<int> l;

for (int i = 20; I < =40; ++i) {

l.push\_back(i);

}

list<int>::iterator pos = find (l.begin(), l.end(), 3);

В данном случае, такого элемента нет, поэтому pos будет равно l.end(). Алгоритмы описаны в модуле algorithm и их там очень много, в качетсве пример можно привести min\_element, max\_element, copy, find, sort и т.п.

### Итераторы адаптеры

Понятие итератора — это абстракция, поэтому вы можете написать нечто что ведет себя как итератор и это будет тоже итератор. Таким образом можно написать класс, который ведет себя как итератор, но делает нечто совершенно отличное от него. STL предоставляет несколько специальных итераторов:

* **insertion iterators** – итераторы вставки
* **stream iterators** – итераторы потока
* **reverse iterators** – обратные итераторы

### Итераторы вставки

Итераторы вставки используются для того чтобы перевести алгоритмы в режим вставки, а не перезаписи. Это делается следующим образом:

* если вы присваиваете значение такому итератору, то он вставляет это значение в тот контейнер, которому он принадлежит
* операция смещения на следующий элемент ничего не делает

Существует три типа итераторов вставки:

* **back inserters** – как следует из названия, добавление идет в конец контейнера с помощью функции push\_back. Соответственно, такие итераторы могут быть использованы только с контейнерами реализующими этот метод.

copy(src.begin(), src.end(), back\_inserter(dst));

* **front inserters** – добавление идет в начало контейнера с помощью метода push\_front.

copy(src.begin(), src.end(), front\_inserter(dst));

* **general inserters** – добавление идет с помощью функции insert которой передается итератор переданный в конструктор в качестве параметра.

copy(src.begin(),src.end(),front\_inserter(dst,dst.begin()));

### Итераторы потока

Итераторы потока, как понятно из их названия читают и пишут в поток. Тем самым они предоставляют нам абстракцию, позволяющую, например, рассматривать ввод с клавиатуры в виде коллекции из которой мы можем читать. Например,

vector<string> s;

copy(istream\_iterator<string>(cin), istream\_iterator<string>(), back\_inserter(s));

Здесь istream\_iterator<string> создает итератор потока, который читает из cin строки и передает его в алгоритм копирования. Каждый раз когда алгоритм копирования хочет перейти к следующему элементу это транслируется в cin >> string. Для вывода можно использовать ostream\_iterator.

unique\_copy(s.begin(),s.end(),

ostream\_iterator<string>(cout,"\n"));

Второй параметр в конструкторе оstream\_iterator-а позволяет задать разделительный символ.

### Обратные итераторы

Обратные итераторы как следует из их названия работают в обратную сторону. Грубо говоря, при вызове оператора инкремента, на самом деле происходит декремент и наоборот. Все контейнеры могут создавать обратные итераторы с помощью функций rbegin и rend.

vector<int> l;

for (int i = 1; I <= 9; ++i) {

l.push\_back(i);

}

copy(l.rbegin(), l.rend(),ostream\_iterator<int>(cout, " "));

### Строки

Все мы знаем, что работа со строками в C как с массивами char-ов сопряжена со множеством проблем. Работая с такими строками достаточно легко прописаться по памяти, часто требуется пере выделять память вручную, при этом можно ошибиться и т.п. Попробуем сформулировать минимальные требования к классу строк, для того чтобы с ним удобно бы было работать:

* инициализация массивом символов (строкой встроенного типа) или другим объектом типа string
* копирование одной строки в другую, для встроенного типа приходится использовать strcpy
* доступ к отдельным символам строки для чтения и записи
* сравнение двух строк на равенство, для встроенного типа используется strcmp
* конкатенация двух строк, для встроенного типа используется strcpy, strcat
* вычисление длинны строки, для встроенного типа это strlen
* возможность узнать пуста ли строка

Класс string стандартной библиотеки реализует все перечисленные выше операции, как и многие другие. Для того чтобы использовать объекты класса string необходимо включить заголовочный файл string.

#include <string>

string a(“test”);

string b; // пустая строка

Длину строки возвращает функция size(). Для того чтобы узнать пустая ли строка есть функция empty(). Сравнение двух строк осуществляется с помощью оператора сравнения:

string a(“test”);

string b(“test”);

if (a == b) …

Для конкатенации строк можно использовать оператор сложения:

string a(“test”);

string b(“more test”);

string s = a + “ ” + b;

Как видите операция сложения может складывать строки и со строками встроенного типа, то есть у класса string есть конструктор, принимающий на вход строку встроенного типа. Для того чтобы осуществить обратное преобразование необходимо использовать функцию c\_str().

const char \*c = a.c\_str();

К отдельным элементам строки можно обращаться с помощью оператора индексирования:

string a(“test”);

for (int i = 0; i < a.size(); i++) {

 cout << a[i];

}

Следует отметить что класс string можно рассматривать как контейнер и соответственно к нему применимо большинство алгоритмов, работающих с контейнерами (replace, find и т.п.).

## Обработка исключений

К сожалению при написании многих программ общепринитой практикой является игнорирование проверки ошибок. Код пишется таким образом, как будто он работает в идиальных условиях и никаких ошибок не может произойти в принципе. На самом деле это далеко не так, более того многие ошибки могут не зависеть от самой программы и могут быть вызваны внешними причинами, например нехватка памяти или ошибка во входных данных. Во многом такое игнорирование ошибок объясняется существенным повышением сложности кода всвязи с добавлением в него множества проверок и обработки особых случаев.

Например, функция printf возвращает количество напечатанных символов, однако практически не кто не проверяет это значение. Представьте во что превратилась бы простейшая программа при наличии такого рода проверок.

Основная проблема обработки исключительных ситуаций в пределах какой-либо С функции состоит в том, что код самой функции и соответствующих обработчиков ошибок становится слишком сильно связанными. Что существенно усложняет их дальнейшую поддержку и изменение.

Механизм обработки исключений (exception handling) является несомненно одним из наииболее полезных нововведений в языке С++. Строго говоря, обработка исключений – это механизм позволяющий двум независимым программным компонентам взаимодействовать в аномальной ситуации называемой исключением (exception).

### Возбуждение исключения

Исключение – это аномальная ситуация во время выполнения, которое программа может обнаружить (например деление на 0, выход за границы массива и т.п.). Как правило, определение такой аномальной ситуации происходит на уровне, на котором не хватает данных для принятия решения о том какие действия предпринять в этой ситуации. С++ позволяет вам передать эту информацию на более высокие уровни программной иерархии путем механизма возбуждения исключения (throwing exception). Например это может выглядеть следующим образом:

throw myerror(“something bad happened”);

Здесь myerror это некоторый класс, конструктор которого принимает char\* в качестве аргумента. В качестве параметра throw можно использовать объект произвольного типа, включая встроеннные. Однако в большинстве случаем для этого используются специальным образом разработанные типы данных.

Возбуждение исключения влечет за собой целый ряд действий. Сперва, происходит создание объекта, заданного в качестве аргумента для throw. Затем этот объект возвращается из функции, причем возвращается независимо от того, совпадает ли его тип с типом возвращаемого значения для функции внутри которой определено исключение или нет. Слово возвращается может быть употреблено в данном случае с большой натяжкой, поскольку механизм при этом задействованный отличается от механизма используемого инструкцией return. Кроме того возврат происходит не в то место откуда была вызвана функция, а в совершенно другой фрагмент кода. На самом деле после возбуждения исключения мы попадаем в тело соответствующего обработчика исключения. В добавок удаляются только те объекты которые были успешно созданы на момент возбуждения исключения, в отличие от обычного выхода из функции, при котором предполагается что все объекты из соответствующей области видимости должны быть удалены.

### Try блок

Если в теле функции возбуждается исключение, то происходит выход из этой функции. Если вы не хотите выдти из функции после возбуждения исключения, то код который может возбудить исключение должен быть помещен в специальный try-блок:

try {

 // Code that may generate exceptions

}

Такой блок начинается с ключевого слова try, за которым идет последовательность инструкций, заключенная в фигурные скобки. При написании обработки ошибок без использования исключений нам бы пришлось для каждой инструкции писать коди типа if-then-else. Использование исключений позволяет избежать этого заключив все потенциально-опасные участки кода в try-блоки.

### Обработчики исключений

Естественно возбужденное исключение надо где-то обрабатывать. Именно для этого и предназначены так называемые обработчики исключений (exception handlers). Обработчики исключений размещаются непосредственно за try-блоком и предваряется ключевым словом catch.

try {

 // code that may generate exceptions

} catch(type1 id1) {

 // handle exceptions of type1

} catch(type2 id2) {

 // handle exceptions of type2

}

 // etc...

Каждый обработчик исключений представляет собой как бы небольшую функцию с одним аргументом. Переменные id1, id2 могут быть использованы в теле обработчика, точно так же, как и обычные аргументы внутри функции. Хотя за частую это и не нужно, поскольку информации о типе исключения уже достаточно, для того чтобы отреагировать на него соответствующим образом.

В случае возбуждения исключения механизм обработки ищет первый обработчик тип которого совпадает с типом исключения. После этого управление передается в код обработчика и исключение считается обработанным и поиск обработчика прекращается. В противном случае поиск обработчика продолжается в вызывающей функции. Этот поиск последовательно проводится по всей цепочке вложенных вызовов. Такой процесс носит называние раскруткой стека. По мере раскрутки стека происходит удаление локальных объектов, объявленных в составных инструкциях и определениях функций, из которых произошел выход.

После завершения обработчика выполнение возобновляется с инструкции, идущей за последним catch обработчиком в списке (если только внутри тела обработчика не был вызван return).

Если в программе нет обработчика способного обработать возбужденное исключение, оно остается необработанным, после чего вызывается функция terminate() из стандартной библиотеки С++. По умолчанию terminate активизирует функцию abort(), которая аномально завершает программу.

### Объекты-исключения

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

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

enum EHstate state = noErr;

int mathFunc( int i ) {

 f ( i == 0 ) {

 tate = zeroOp;

 hrow 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.

Возникает закономерный вопрос как программа может различить эти два объекта. Для ответа на него необходимо более подробно к объявлению исключения в обработчике. Это объявление ведет себя почти также, как и объявление формального параметра. Если при входе в обработчик исключения выясняется, что в нем объявлен объект, то он инициализируется копией объекта-исключения. Например:

void calculate (int op) {

 try {

 mathFunc(op);

 }

 catch (EHstate eObj) {

 // eObj - копия сгенерированного объекта исключения

 }

}

При входе в обработчик исключения внутри calculate объект eObj инициализируется копией объекта-исключения, созданного выражением throw.

Объявления исключения в этом примере напоминает передачу параметра по значению. Как и в случае параметров функции в объявлении исключения может фигурировать ссылка. Тогда обработчик исключения будет на прямую ссылаться на объект-исключение:

void calculate (int op) {

 try {

 mathFunc(op);

 }

 catch (EHState &eObj) {

 // eObj – ссылается на сгенерированный объект-исключение

 }

}

В этом случае обработчик может модифицировать объект исключение, при этом переменные определенные в выражении throw остаются без изменения. Так модификация eObj внутри обработчика исключений не затрагивает глобальную переменную state использованную в выражении throw:

void calculate (int op) {

 try {

 mathFunc(op);

 }

 catch (EHState &eObj) {

 eObj = noErr; // глобальная переменная state не изменилась

 }

}

Следует понимать, что в данном случае в обработчике меняется глобальный объект-исключение созданный в результате выполнения throw и его изменение никак не затрагивает глобальную переменную state.

### Прерывание или возобновление

В теории обработки исключительных ситуаций существуют две основополагающих модели – прерывание (используется C++) и возобновление. В случае прерывания предполагается что ошибка на столько критична, что возобновление работы программы с того же самого места в котором произошло исключение не возможно.

Альтернативой прерыванию является возобновление. При таком подходе предполагается, что обработчик исключений каким-то образом может исправить ситуацию и работа программы может быть возобновлена с того же самого места. В С++ такая модель может быть реализована с помощью обычных функций или, например, путем помещения try-блока внутрь while цикла.

### Повторное возбуждение исключения

Может оказаться, что в пределах одного обработчика исключений не удается полностью обработать исключение. Выполнив некоторые корректирующие действия обработчик исключения может решить, что дальнейшую обработку следует перепоручить функции, расположенной выше в цепочке вызовов. Это может быть сделано с помощью повторного возбуждения исключения:

catch ( exception eObj ) {

 if ( canHandle( eObj ) )

 // обработать исключение

 return;

 else

 // повторно возбудить исключение, чтобы его перехватил // другой catch-обработчик

 throw;

Конструкция throw; вновь генерирует объект-исключение. Ее использование возможно только внутри обработчика исключений. При повторном возбуждении новый объект-исключение не создается. Это особенно удобно в том случае если обработчик исключений модифицирует объект, прежде чем возбудить исключение повторно.

### Перехват всех исключений

Иногда функции необходимо выполнить некоторое действие до того, как она завершит обработку исключения даже несмотря на то, что обработать его она не может. Например, в случае захвата функцией некоторого ресурса (открытие файла, выделение памяти и т.п.) этот ресурс необходимо освободить перед выходом. При возбуждении исключения управление не попадет на инструкцию освобождения ресурса. Для того чтобы все-таки освободить ресурс, можно воспользоваться специальной конструкцией catch (…):

void manip() {

 resource res;

 res.lock();

 try {

 // использование ресурса

 // действие, в результате которого возбуждено исключение

 }

 catch (...) {

 res.release();

 throw;

 }

 res.release(); // не выполняется, если возбуждено исключение

Эта конструкция используется в сочетании с повторным возбуждением исключения. Catch(…) может быть использовано в сочетании с другими обработчиками исключений. В этом случае следует озаботиться чтобы оно было последним в списке обработчиков, поскольку поиск соответствующего обработчика происходит последовательно.

### Спецификация исключений

В С++ существует возможность информировать пользователей функции о возбуждаемых ей исключениях. Эта может быть сделано с помощью, так называемой, спецификации исключений:

void f() throw(toobig, toosmall, divzero);

При этом гарантируется что другие исключения функция возбуждать не будет. Если при объявлении функции указаны типы исключений, то при ее повторном объявлении должны быть перечислены те же типы:

extern void f() throw(toobig, toosmall, divzero);

Спецификации исключений в разных объявлениях одной и той же функции не суммируются. Что произойдет, если функция возбудит исключение, не перечисленное в ее спецификации? Это вполне легальная ситуация, в этом случае будет вызвана функция unexpected() из стандартной библиотеки С++.

Необходимо уточнить тот факт, что unexpected() вызывается только в том случае если внутри функции было возбуждено исключение не определенное в спецификации и оно не было обработано внутри этой функции:

void f () throw(ExceptionType)

{

try {

 // ...

 throw string("we're in control");

}

 // обрабатывается возбужденное исключение

 catch ( string ) {

 // сделать все необходимое

}

Функция f возбуждает исключение типа string, несмотря на его отсутствие в спецификации. Поскольку это исключение обработано внутри функции unexpected не вызывается.

Существует возможность указать что функция вообще не возбуждает исключений:

void f() throw();

Если же в объявлении функции спецификация исключений отсутствует, то может быть возбуждено исключение любого типа.

Между типом исключения и типом исключения, указанного в спецификации не разрешается проводить никаких преобразований:

int f ( int parm ) throw(string)

{

 if ( somethingRather )

 // ошибка программы: f() не допускает исключения типа // const char\*

 throw "help!";

}

Выражение throw в функции f возбуждает исключение типа const char\*. Обычно выражения типа const char\* можно привести к типу string. Однако спецификация исключений не допускает преобразования типов, поэтому будет вызвано unexpected(). Исправить ошибку можно следующим образом:

throw string(“help!”);

### Исключения и производительность

Следует помнить, что процесс возбуждения исключения может существенно повлиять на производительность программы. Поэтому их использование должно быть обдуманным и тщательно взвешенным. Исключения в С++ разработаны таким образом, что они не влияют на производительность основной программы до своего возбуждения. Хотя этот факт конечно существенно зависит от используемого компилятора. По крайней мере в теории это должно быть именно так.

## Новый стандарт с++

В настоящее время уже существует новая версия языка C++, которая во многих аспектах существенно отличается от обычного C++. Стандарт был принят совсем недавно, и еще не вся функциональность поддерживается современными компиляторами.

### << при описании template-ов

Как вы помните в языке c++ при описании вложенных template-ов было необходимо ставить пробел между знаками меньше:

Template <typnate t = vector<int> >

Теперь можно этот пробел больше не ставить.

### auto при описании типов

Auto переменные имеют тот же тип, что и инициализирующее их выражение:

auto a = 10; // a : int

std::map<int, std::string> m;

auto i1 = m.begin(); // i1 : std::map<int, std::string>::iterator

Допустимо понятие константности и ссылок для auto переменных:

const auto \*a1 = &a; // a1 : const int \*

const auto &m1 = m; // m1 : const std::map<int, std::string>&

Auto может быть использовано для объявление нескольких переменных, однако все они должны принадлежать к одному и тому же типу:

auto a = 10, b = 11;

auto a = 10, b = 11.0; // error

### Циклы по диапазону (range based loops)

Появился новый способ итерироваться в пределах контейнера:

std::vector<int> v;

for (int i : v) std::cout << i;

Итерируемая переменная также может быть ссылкой:

for (int& i : v) std::cout << ++i;

Точно так же можно использовать auto:

for (auto i : v) std::cout << i;

for (auto& i : v) std::cout << ++i;

Такой способ итерации может быть использован для всех контейнеров, которые поддерживают понятие диапазона, то есть предоставляют функции begin и end, возвращающие соответствующие итераторы. Для обычных массивов он тоже применим:

int a[10];

for (auto i : a) std::cout << ++i;

Диапазонная форма применима только к циклам типа for, для while-ов все остается по старому.

### nullptr

Введено новое ключевое слово nullptr, обозначающее указатель на null. Эта величина может быть приведена к любому типу и к bool-у, но при этом не может по умолчанию как раньше использоваться как int.

const char \*p = nullptr; // ps is null

if (p)

int i = nullptr // error

При этом старое понятие null остается валидным:

int \*p1 = nullptr;

int \*p2 = 0;

int \*p3 = NULL;

if (p1 == p2 && p1 == p3)

### Лямбда выражения

По сути дела лямбда функции позволяют программисту определять функции на лету:

std::vector<int> v;

auto it = std::find\_if(v.begin(), v.end(), [](int i) { return i > 0 && i < 10; });

На самом деле, это разворачивается в что-то похожее на:

class A {

public:

 bool operator()(int i) const { return i > 0 && i < 10; }

};

auto it = std::find\_if(v.begin(), v.end(), A());

### Using как typedef

Теперь вместо typedef можно будет писать аналогичные конструкции, начинающиеся с using:

typedef std::vector<int> VECT;

using VECT = std::vector<int>;

или более сложный вариант, для указателя на функцию:

typedef void (\*callback)(int);

using callback = void(\*)(int);

### shared\_ptr

Указатель с подсчетом ссылок:

{

std::shared\_ptr<A> p1(new A);



std::shared\_ptr<A> p2(p1);



p1->doThis();

if (p2) p2->doThat();

p2 = nullptr;



} // reference counter = 0 and A is deleted

### weak\_ptr

Weak указатели похожи на обычные указатели, за исключением того, что они знают когда то на что они указывают уже удалено.

std::shared\_ptr<A> p(new A); // rc = 1

std::weak\_ptr<A> w(p); // rc = 1

if (!w.expired()) …

Weak\_ptr это на самом деле не указатель. У него нет операторов разименования (-> и \*), нет явной проверки на null. Для того чтобы использовать weak\_ptr как указатель, его надо явно привести к shared\_ptr.

std::weak\_ptr<A> w(p);

w->doThat(); // error

std::shared\_ptr<A> pw(w);

pw->doThat();

### unique\_ptr

Этот указатель является заменой auto\_ptr. Так же как и auto\_ptr выполняет move вместо копирования. Лучше чем auto\_ptr с точки зрения использования в контейнерах и т.п.

{

std::unique\_ptr<A> p(new A);

std::unique\_ptr<A> p1(p); // error

std::unique\_ptr<A> p2(std::move(p));

p2->doSomething();

} // delete p2.get()

### Move semantic

Известно что иногда C++ выполняет не нужное копирование. Например, при возвращении вектора из функции по значению:

typedef std::vector<T> TVec;

TVec createTVec();

TVec tv;

tv = createTVec(); // копирует объект в tv, потом удаляет временный объект



Move семантика позволяет избежать лишнего копирования, при использовании move семантики в данном случае должно получится



Рассмотрим более сложный пример – добавление элементов в vector:

std::vector<T> vt;

T t;

vt.push\_back(t); // предположим, что в результате push\_back происходит выделение дополнительной памяти



В случае использования move семантики должно получиться следующее:



Другие операции связанные с перевыделением памяти (insert, erace и т.п.) будут работать так же.

### RValue ссылки

Возникает вопрос как реализовать move семантику на уровне языка, к сожалению все вышеописанное не может происходить автоматически. Для этой цели в c++ вводится понятие rvalue ссылки. Для того чтобы понять что это такое давайте сначала разберемся что из себя представляют lvalue и rvalue величины.

Можно сказать, что lvalue это все от чего можно взять адрес – именнованные объекты и lvalue ссылки (про них поздже). Rvalue это объекты, для которых невозможна операция взятия адреса:

int x, \*p; // x, p, \*p - lvalues

int f(string s);// a – lvalue, возвращаемое значение для f – rvalue

f(“test”); // временная строка здесь rvalue

std::vector<int> vi; // vi – lvalue

vi[3] = 3; // vi[3] – lvaule потому что operator[] возвращает ссылку

Легко видеть что move семантика для lvalue не вполне безопасна. Дело в том, что где-то может присутствовать ссылка на lvalue переменную и тогда после выполнения перемещения она может перестать быть валидной:

TVec vt1;

…

TVec vt2(vt1);

…

использовать vt1

В данном случае автор ожидает, что vt1 будет скопирован в vt2 а не перемещен. При этом для rvalue move семантика как раз вполне безопасна:

TVec vt1;

vt1 = createTVec();

Соответственно для решения проблемы в C++ 11 предложено ввести понятие rvalue ссылки T&&. Те ссылки, о которых мы говорили раньше теперь называются lvalue ссылки. Rvalue ссылки сохраняют все свойства ссылок, то есть они должны быть инициализированы при объявлении, не могут быть переназначены на другой объект и т.п. Rvalue ссылка определяет что объект может быть перемещен в случае когда это нужно. Примеры:

void f1(const TVec&); // принимает lvalue

TVec vt;

f1(vt); // копирование как обычно

f1(createTVec()); // копирование как обычно

void f2(TVec &&);

f2(createTVec()); // происходит перемещение

Для того, чтобы различать на уровне компилятора когда делать копирование, а когда делать перемещение вводится понятие специального move конструктора и оператора перемещения.

class A {

public:

A(const A&); // copy constructor

A(A&&); // move constuctor

A& operator=(const A&); // copy assignment op

A& operator=(A&&); // move assignment op

…

};

A createA(); // factory function

A a1;

A a2 = a1; // lvalue src ⇒ copy req’d

a2 = createA(); // rvalue src ⇒ move okay

a1 = a2; // lvalue src ⇒ copy req’d

###