Лекции по промышленному программированию на C++

Алексей Мартынов

26 августа 2015 г.

Версия 0.6

Оглавление

От автора

Этот сборник представляет собой курс лекций по программированию на C++, прочитанных в 2015 году в Санкт-Петербургском государственном политехническом университете на кафедре Информационных и Управляющих систем (ИУС). Содержание курса отражает мое мнение о том, что необходимо знать для разработки качественного промышленного программного обеспечения на языке C++.

Курс опирается на материалы предыдущего семестра, в котором традиционно изучаются основы C++. Первые лекции призваны осветить некоторые важные вопросы, которые обычно пропускаются в процессе изучения языка. С точки зрения академического преподавания они выглядят несущественно, но при разработке программного обеспечения вопросы физического дизайна системы, управления ресурсами и безопасности исключений являются неотъемлемой частью любой программы и должны рассматриваться с самого начала.

Одним из существенных моментов курса является упор на самостоятельную работу. В этот сборник включено только самое необходимое. Все подробности могут быть найдены в соответствующей литературе, список которой приведен в конце.

В курс входят также лабораторные работы, описанные в приложении B.

Благодарности

Я благодарю Котлярова В.П. (СПбГПУ, кафедра ИУС) за предоставленную возможность прочитать эти лекции, Саламатова М.А. (EMC^2) за настойчивую уверенность в необходимости данного курса, Тылика Д.Н. (EMC^2) и Толстого В.М. (EMC^2) за проведенную работу по рецензированию материалов лекций, Черкалову В.В. (EMC^2) за «голос разума» в планировании тем лекций и их содержания, а также за регулярное «приземление» полета мысли, Новожилова Е.А. (EMC^2) за демонстрацию восприятия студентов и рецензирование материалов, Михайлову А.А. как первую «жертву» моих педагогических способностей и Зыбина К.Г. и Robert Munsell (EMC^2) за поддержку в непростом процессе чтения лекций и ведения практических занятий в этом семестре.

Глава 1

Введение в промышленное программирование на C++

1.1 Применение C++

Развитие C++, история которого насчитывает уже более 30 лет, достигло уровня, на котором он стал применим во множестве областей. Среди них можно выделить:

- системное программирование, вне зависимости от смысла этого термина;
- встроенные системы;
- ресурсоограниченные системы;
- большие приложения.

Такая популярность и применимость C++ во многом связана с тем, что одним из его родителей является язык C. C настолько близок к оборудованию, что существует выражение «высокоуровневый ассемблер». C++ настолько же близок к оборудованию, как и C, но при этом вносит множество высокоуровневых концепций, таких как объектно-ориентированное и метапрограммирование.

1.2 Реализации C++

На сегодняшний день доступно несколько популярных реализаций C++:

- Microsoft® Visual Studio,
- GNU Compiler Collection,
- Intel C++ Compiler,
- Clang, построенный по технологии LLVM.

Все эти компиляторы предлагают несколько отличающийся уровень поддержки стандартов, но стандарт 2003 года поддерживается всеми компиляторами минимум на 98%.

1.3 История

Язык C++ был создан Бьярном Страуструпом (Bjarne Stroustrup) во время его работы в AT&T Bell Labs.

1.3.1 C с классами (1979-1983)

История C++ начинается в 1979 году, когда Бьярн Страуструп работал в Bell Labs над теорий очередей. Главным стимулом создания нового языка было то, что существовавшие тогда языки моделирования были малоэффективны, а машинные языки были слишком сложны для применения в моделировании из-за их низкой выразительности. Страуструп решил дополнить язык C различными возможностями, имевшимися в языке Симула.

Начало новому языку было положено путем добавления классов и их наследования, строгой проверки типов на этапе компиляции, встраиваемых функций и аргументов по умолчанию.

1.3.2 C++ (1983)

К 1983 году язык получил новые возможности:

- виртуальные функции,
- перегрузка функций и операторов,
- ссылки,
- константы.

Также язык получил улучшенный контроль типов и новый вид однострочных комментариев.

Совокупность всех доработок привела к тому, что язык обрел новое имя — C++. Реализация этой версии языка называлась Cfront и представляла собой транслятор из C++ в C.

1.3.3 C++ версия 2.0 (1989)

В 1985 году вышло первое издание «Языка программирования C++» [Str86], являвшееся по сути основным описанием языка на тот момент.

С выходом C++ версии 2.0 язык приобрел возможности множественного наследования, создания абстрактных классов, статических методов.

Дальнейшее развитие привнесло в язык шаблоны, исключения, пространства имен, новые способы приведения типов и булевский тип. В 1990 году вышло «Комментированное справочное руководство по C++» (ARM) [ES90], которое и легло впоследствии в основу стандарта.

Дальнейшее развитие языка шло по пути его стандартизации в ISO.

1.3.4 C++ 98 и C++ 03

В 1998 году вышел стандарт ISO/IEC 14882:1998 [C++ 98], обычно называемый C++ 98. В стандарт входили описания языка и стандартной библиотеки.

В рамках стандартной библиотеки язык получил набор контейнеров и алгоритмов для работы с ними, библиотеку потокового ввода-вывода и средства локализации.

В 2003 году вышел стандарт ISO/IEC 14882:2003 [C++ 03], которые исправил некоторые ошибки C++ 98. Этот стандарт пока является самым широко используемым в промышленной разработке.

1.3.5 C++ TR1 (2005)

Довольно скоро стало очевидно, что стандартная библиотека, описанная в стандартах 1998 и 2003 годов, не содержит множество вещей, которые требуются в повседневной работе. Поэтому в 2005 году был выпущен «Library Technical Report 1» (TR1). Он не является частью стандарта и описывает расширения стандартной библиотеки, которые предполагалось включить в будущие стандарты:

- умные указатели,
- функциональные объекты,
- средства метапрограммирования,
- генераторы псевдослучайных чисел,
- набор математических функций,
- контейнеры,
- регулярные выражения.

Поддержка TR1 различными реализациями C++ неуклонно повышается. Значительная часть этих расширений вошла в новый стандарт.

1.3.6 C++ 11

В 2009 году началась разработка нового стандарта. Изначально планировался выход новой версии в 2009 году, поэтому рабочим названием было C++09. Потом его изменили на C++0x, в итоге он получил название ISO/IEC 14882:2011 [C++ 11] или C++ 11.

В этом стандарте было внесено большое количество изменений как в сам язык, так и в библиотеку:

- ссылки на R-value и перемещающие конструкторы,
- константные выражения,
- вывод типов,
- шаблоны с переменным количеством параметров,
- лямбда-функции,
- строго типизированные перечисления,
- поддержка параллельного программирования,
- хэш-таблицы,
- регулярные выражения,
- умные указатели,
- средства метапрограммирования.

В 2014 году увидел свет ISO/IEC 14882:2014(E) или C++ 14. Этот стандарт содержит исправления ошибок и небольшие расширения, которые не попали в C++ 11.

1.4 Понятия стандарта C++

Стандарт C++ содержит некоторые понятия, существенно важные для функционирования программ.

1.4.1 Единица трансляции (translation unit)

Модулем программы на C++ является единица трансляции. Единицей трансляции называется файл исходного текста со всеми включенными в него заголовочными файлами и файлами исходных текстов. Включение заголовочных файлов и исходных текстов выполняет препроцессор языка C++ в процессе трансляции.

1.4.2 Поведение, определенное реализацией (Implementation-defined Behavior)

Стандарт описывает поведение некой абстрактной машины, которая исполняет код программы на C++. Некоторые аспекты выполнения данного кода не определены точно. Поведение является определяемым реализацией, если стандарт не оговаривает все детали функционирования корректной программы, оперирующей корректными данными, а передает их на усмотрение компилятора, компоновщика и платформы. Эти детали должны быть описаны в документации.

Например, стандарт 2003 года [C++ 03, параграф 1.7.1] говорит, что базовой единицей хранения данных является байт, представляющий собой последовательность бит. Количество бит в одном байте определяет реализация.

1.4.3 Неспецифицированное или неуточненное поведение (Unspecified Behavior)

Неуточненное поведение также не описывает детали для корректной программы и данных, но, в отличие от определенного реализацией поведения, стандарт не требует, чтобы реализация документировала это поведение.

Например, порядок вычисления аргументов функции не определен [C++ 03, параграф 5.2.2.8]. В то же время требуется, чтобы все побочные эффекты вычисления аргументов полностью наступили перед вызовом функции.

```
1 void foo(int, double);
2
3 ...
4
5 foo(2 * a, sqrt(b));
```

1.4.4 Неопределенное поведение (Undefined Behavior)

Следующим важным понятием является неопределенное поведение. Неопределенное поведение проявляется при использовании некорректной конструкции языка или некорректных данных. Стандарт не формулирует никаких требований к такому поведению, и, как следствие, может происходить все, что угодно.

Например, разыменование нулевого указателя является неопределенным поведением. Оно может вести к исключениям, аппаратным прерываниям или вообще не вызывать ничего необычного. Впрочем, результат в последнем случае тоже неизвестен и может проявиться позже, в самый неподходящий момент.

1.4.5 Правило одного определения (One Definition Rule, ODR)

Правило одного определения гласит [C++ 03, параграф 3.2.1]:

> Ни одна единица трансляции не должна содержать более одного определения любой одной и той же переменной, функции, класса, перечисления или шаблона.

Любая программа должна содержать ровно одно определение для каждой функции и каждого объекта, используемых в ней.

1.4.6 Классы памяти (Storage Classes)

Класс памяти определяет, где будет размещена переменная и каков будет тип ее связывания. Для функций класс памяти определяет тип связывания.

В языке присутствуют следующие классы памяти:

register Служит как подсказка компилятору, что переменная часто используется и может быть размещена в регистре процессора для оптимизации; на сегодняшний день применяется крайне редко.

static Указание на внутреннее связывание для переменных и функций в глобальном пространстве имен. Локальные переменные с этим спецификатором размещаются в особой области памяти и сохраняют свое значение между вызовами функции. Члены классов, отмеченные как **static**, разделяются всеми экземплярами этих классов. Статические методы классов не имеют доступа к нестатическим данным класса.

extern Указание на внешнее связывание.

auto Исторически, до вступления в силу стандарта 2011 года, данный класс памяти использовался для локальных переменных по умолчанию и не требовал указания. Так как это ключевое слово практически неиспользовалось, его смысл изменился с выходом стандарта 2011 года и оно более не обозначает класс памяти.

1.4.7 Типы памяти

Модель памяти C++ содержит следующие области или типы памяти:

Константные данные В области константных данных размещаются все строковые литералы и прочие данные, которые известны на этапе компиляции. Содержимое этой области доступно в течение всего времени работы программы и, зачастую, не имеет доступа на запись. Независимо от наличия такого доступа, любая модификация области константных данных представляет собой неопределенное поведение. Из этого следует, что данная область не содержит никаких объектов. Более того, компилятор имеет право на любые оптимизации данной области, например, он может объединять различные строковые литералы в один.

Глобальные и статические данные В этой области размещаются все глобальные переменные, статические поля классов и статические переменные. Время и порядок инициализации содержимого этой области не определены.

Куча Кучей называется область динамического распределения памяти при помощи функций malloc(), calloc () и free (). При промышленной разработке куча применяется только при взаимодействии с библиотеками, написанными на C и других языках.

Free Store Область Free Store — это область динамического распределения памяти при помощи операторов **new** и **delete**.

1.4.8 Точка следования (sequence point)

Точки следования присутствуют в стандарте 2003 года, но исключены из него в 2011 году. Тем не менее, концепция сохранилась.

Точкой следования называется такое место в программе, где все результаты, включая побочные эффекты, от предыдущих вычислений уже наступили, в то время как никаких эффектов от последующих вычислений еще нет [C++ 03, параграф 1.9.7].

Точками следования являются следующие места в программе:

- завершение вычисления полного выражения;
- вычисление всех аргументов при вызове функции;
- завершение копирования возвращаемого значения из вызова функции;
- завершение вычисления первого аргумента выражений a && b, a || b, a ? b : c и a, b; если эти выражения являются перегруженными операторами, точки следования определяются по правилам вызова функций;
- между инициализацией каждого базового класса и каждого поля в классе.

Модификация одного и того же объекта более одного раза между точками следования ведет к неопределенному поведению [C++ 03, параграф 5.4]:

```
1  i = v[i++];
2  i = ++i + 1;
```

Следующие выражения являются полностью корректными:

```
1  i = 7, i++, i++;
2  i = i + 1;
```

Первое из них корректно, так как приоритет присваивания выше, чем оператора «,», поэтому сначала происходит присваивание, после чего точка следования от первой запятой, потом инкремент, точка следования от второй запятой и затем финальный инкремент.

Второе выражение содержит только одну модификацию переменной — присваивание.

Важно отметить, что выражение i = ++i + 1; будет корректным в C++ 11 в случае, если используются встроенные операторы инкремента и декремента, так как они эквивалентны операторам += и −=, а для них выдвинуто требование:

> Присваивание выполняется после вычисления значений левого и правого операндов, но перед вычислением значения всего оператора [C++ 11, параграф 5.17.1].

Глава 2

Выражения и конструкции языка C++

2.1 Простые вычисления

Программирование на любом языке начинается с простых вычислений. Их проще всего изучать на примерах. Рассмотрим простую программу вычисления квадратных корней:

```
 1 #include <iostream>
 2 #include <cmath>
 3
 4 int main(int, char *[])
 5 {
 6   const double a = 2,
 7        b = 3,
 8        c = 4;
 9
10   double d = b * b - 4 * a * c;
11
12   std::cout << "Equation: " << a << " * x^2 + "
13             << b << " * x + " << c << "\n";
14
15   if (d > 0) {
16     double r1 = (-b + std::sqrt(d)) / (2 * a);
17     double r2 = (-b - std::sqrt(d)) / (2 * a);
18
19     std::cout << "Roots: " << r1 << ", " << r2 << "\n";
20   } else if (d == 0) {
21     double r = -b / (2 * a);
22
23     std::cout << "Root: " << r << "\n";
24   } else {
25     double r = -b / (2 * a);
26     double i = std::sqrt(std::abs(d)) / (2 * a);
27
28     std::cout << "Complex roots: "
29               << r << " + " << i << "i, "
30               << r << " - " << i << "i\n";
31   }
32
33   return 0;
34 }
```

На строках 1 и 2 расположены директивы включения заголовков. Заголовки содержат объявления различных символов и типов. В данном примере будут использоваться средства ввода-вывода из заголовка `iostream` и функция вычисления квадратного корня из `cmath`.

Далее, на строке 4 начинается определение функции main(). С этой функции начинается исполнение любой программы на языках C и C++, за исключением программ для графического пользовательского интерфейса Microsoft® Windows. Каноническая версия функции main() возвращает целое число, назы-

ваемое кодом выхода. Внешняя программа, запустившая эту, может дождаться окончания исполнения и узнать код выхода. По соглашению, успешное завершение программы соответствует коду выхода 0, а другие коды выхода могут сигнализировать о различных проблемах. Функция main() также принимает 2 аргумента, которые не используются в примере и поэтому для них не указаны имена параметров. Обычно, если эти параметры нужны, они имеют имена argc и argv:

```
int main(int argc, char * argv[])
```

Эти параметры будут содержать следующую информацию:

argc количество аргументов, переданных программе при запуске. На практике этот параметр всегда больше 0, так как программа принимает как минимум один аргумент — имя самой программы. Стандарт же утверждает, что значение argc неотрицательно.

argv значения аргументов программы. Это массив указателей на строки, количество этих строк передано в параметре argc.

Раздел 3.6.1 стандартов [C++ 03; C++ 11] описывает все требования к функции main(). В частности, там присутствует требование, чтобы все реализации допускали и такой вариант:

```
int main()
```

Также присутствуют следующие требования:

- Функция main() не может использоваться в программе.
- Функция main() должна быть определена только один раз.
- Программа, декларирующая main() как **static** или **inline** некорректна.

Также, в виде исключения, стандарт утверждает, что если функция main() завершается и при этом управляющая конструкция **return** не была выполнена, то это эквивалентно исполнению

```
return 0;
```

Пример явного возврата коды выхода можно найти на строке 33.

На строках 6 — 8 определяются коэффициенты квадратного уравнения в виде переменных. Слово **const** говорит о том, что эти переменные являются константами и не могут изменяться в программе. Тип этих переменных — **double**, что означает вещественное число с плавающей запятой двойной точности; a, b и c это имена переменных.

На строке 10 выполняется расчет дискриминанта квадратного уравнения.

Программа выводит решаемое уравнение на строке 12. std::cout представляет собой поток для стандартного вывода. Операция << выводит свой правый аргумент в поток слева.

Квадратное уравнение, в зависимости от знака дискриминанта, имеет 3 набора корней:

- если дискриминант больше 0 уравнение имеет 2 корня;
- если дискриминант равен 0 уравнение имеет один корень (или 2 совпадающих);
- при отрицательном дискриминанте уравнение имеет 2 комплексных корня.

Вычисление и вывод этих корней в зависимости от значения дискриминанта выполняется начиная со строки 15. Первое условие в **if** проверяет, что дискриминант положителен. Если это так, выполняются строки 16 — 19. В противном случае выполняется конструкция **else** на строке 20.

Строится и запускается эта программа примерно следующим образом (для платформ Mac OS X, Linux и FreeBSD, компиляторы GCC и Clang):

```
$ c++ -std=c++11 -Wall -Wextra -o square-roots square-roots.cpp
$ ./square-roots
Equation: 2 * x^2 + 3 * x + 4
Complex roots: -0.75 + 1.19896i, -0.75 - 1.19896i
$
```

2.2 Управляющие конструкции

В русском языке существует некоторая неоднозначность. Слово «оператор» может одновременно означать и «управляющая конструкция» (английский эквивалент «statement»), и «вычислительная операция» («operator»). Этот раздел посвящен полностью управляющим конструкциям. Все управляющие конструкции описаны в главе 6 стандартов [C++ 03; C++ 11].

2.2.1 Составной оператор

Составной оператор используется тех случаях, когда необходимо выполнить несколько операторов в том месте, где по условиям допустим только один, или необходимо сгруппировать какие-либо операторы вместе. Составной оператор представляется собор пару фигурных скобок, между которым заключены вычисления. Например:

```
1 {
2   double r = -b / (2 * a);
3
4   std::cout << "Root: " << r << "\n";
5 }
```

2.2.2 Пустой оператор

Пустой оператор применяется в тех случаях, когда язык требует какой-либо оператор, но при этом никаких действий или вычислений выполнять не надо. Существует 2 формы пустого оператора в C++: ; и {}

2.2.3 Условный оператор

Условный оператор предоставляет возможность ветвления программы в зависимости от условия.

```
if (condition)
  statement_1

if (condition)
  statement_1
else
  statement_2
```

Независимо от варианта, сначала вычисляется условие в скобках после **if**. Если результат вычисления приводится к **true**, то вычисляется `operator_1`. Если условие не является истинным (приводится к **false**), то в первом варианте не происходит ничего, во втором варианте вычисляется `operator_2`.

И `operator_1` и `operator_2` представляют собой один оператор. Если необходимо выполнить несколько операторов, то применяется составной оператор. В примере с квадратными корнями условный оператор применен как с составными операторами, так и с единственным оператором: в случае истинности условия на строке 15 выполняется составной оператор на строках 16 — 19. В противном случае срабатывает ветка **else**, в которой находится единственный оператор **if**.

Необходимо отметить, что по большинству соглашений об оформлении исходных текстов фигурные скобки вокруг операторов ставятся обязательно, а сами операторы сдвигаются на 1 отступ вправо, как в примере с квадратными корнями.

2.2.4 Оператор выбора

В случае большого количества веток исполнения, выбираемых путем сравнения целочисленного значения с эталонами конструкция в виде каскада **if** − **else if** − **else if** − ... − **else** выглядит довольно громоздко. Более того, она препятствует оптимизации исполнения кода. Поэтому в языке присутствует оператор выбора **switch**.

```
switch (integer_value) {
  case constant_1:
    statement_1_1;
    ...
    statement_1_n;
```

```
    case constant_2:
      ...
    case constant_m:
      statement_m_1;
      ...
      statement_m_k;

    default:
      statement_def_1;
      ...
      statement_def_l;
    }
```

В качестве `integer-value` может выступать целое число, перечисление или любой объект, непротиворечиво приводящийся к целочисленному значению или перечислению.

Порядок меток **case** и **default** стандартом не определяется и на порядок вычисления не влияет. После вычисления значения `integer-value` выполняется переход к метке с соответствующим значением и выполнение продолжается с оператора сразу за меткой и до конца всей управляющей конструкции. Если вычисленного значения нет, то выполняются операторы после **default**.

Так как операторы выполняются подряд без остановки, для выхода из оператора выбора до его завершения используется ключевое слово **break**.

Если предположить, что функция sign() возвращает знак числа (+1, −1 и 0), то условные операторы при вычислении корней можно переписать так:

```
1  switch (sign(d)) {
2  case 1:
3    {
4      double r1 = (-b + std::sqrt(d)) / (2 * a);
5      double r2 = (-b - std::sqrt(d)) / (2 * a);
6
7      std::cout << "Roots: " << r1 << ", " << r2 << "\n";
8      break;
9    }
10
11 case 0:
12   {
13     double r = -b / (2 * a);
14
15     std::cout << "Root: " << r << "\n";
16     break;
17   }
18
19 case -1:
20   {
21     double r = -b / (2 * a);
22     double i = std::sqrt(std::abs(d)) / (2 * a);
23
24     std::cout << "Complex roots: "
25               << r << " + " << i << "i, "
26               << r << " - " << i << "i\n";
27     break;
28   }
29 }
```

2.2.5 Цикл с параметром

Для повторения отдельных участков известным числом повторений используется оператор **for**.

```
  for (init; condition; step)
    statement
```

Например, для вывода квадратов первых 10 числе можно написать:

```
1 for (int i = 1; i <= 10; ++i) {
2   std::cout << i << "␣" << (i * i) << "\n";
3 }
```

Необходимо учитывать, что выражения в `init`, `condition` и `step` могут быть практически любыми или вообще отсутствовать. Важно лишь, чтобы `condition` приводился к булевскому типу. Если `condition` отсутствует, считается, что его вычисление дает **true**.

2.2.6 Циклы с пред- и пост-условием

Для повторения участков кода в соответствии с неким условием служат циклы с пред- и пост-условиями. Они выполняются до тех пор, пока вычисленное значение условия эквивалентно **true**.

```
  while (condition)
    statement

  do
    statement
  while (condition);
```

Например:

```
 1 double mean = 0;
 2 int n = 0;
 3 double value = 0.0;
 4
 5 do {
 6   value = read_value();
 7
 8   if (value != 0.0) {
 9     mean += n;
10     ++n;
11   }
12 } while (value != 0.0);
13
14 mean /= n;
```

2.2.7 Цикл по диапазону

Довольно частой задачей является проход по какой-либо коллекции, например, по массиву. В C++ 11 был добавлен специальный вариант оператора **for**, реализующий это:

```
1 int values[] = { 1, 2, 3, 4 };
2
3 for (int i: values) {
4   std::cout << i << "\n";
5 }
```

Переменная i на каждом проходе принимает значение очередного элемента из массива.

2.2.8 Операторы перехода

Одной из особенностей циклов является то, что часто возникает задача преждевременного выхода из них или перехода к следующей итерации. Для этого служат ключевые слова **break** и **continue**.

Ключевое слово **break** немедленно прерывает исполнение цикла и передает управление первому оператору сразу за циклом. Например, программа, выводящая квадратные корни для введенных значений:

```
1 double value = 0.0;
2
3 while ((value = read_value()) != 0.0) {
```

```
4     if (value < 0) {
5       break;
6     }
7
8     std::cout << value << " -> " << std::sqrt(value) << "\n";
9   }
```

В данном примере условием цикла является чтение числа, не равного 0.0. Если введенное число меньше нуля, то цикл немедленно прерывается.

Условия цикла также демонстрирует часто применяющуюся технику: чтобы избежать двойного написания кода по чтению чисел, переменная value присваивается внутри условия цикла.

Для немедленного перехода к следующей итерации служит оператор **continue**:

```
1   int i = 0;
2   do {
3     i++;
4     std::cout << "before the continue\n";
5
6     continue;
7
8     std::cout << "after the continue, should never print\n";
9   } while (i < 3);
```

В случае использования **continue** правильное написание вычислений, необходимых при переходе к следующей итерации, становится особенно важным, так как очень легко получить бесконечный цикл. Поэтому следует воспользоваться циклом **for** или техникой пересчета внутри условия.

Как было отмечено, операторы внутри **switch** равнозначны. Метки всего лишь определяют, с какого оператора продолжится выполнение. Для того, чтобы не выполнять лишние операторы из блока с другой меткой, используется оператор **break**. Пример его использования можно видеть в разделе, посвященном оператору **switch**.

2.3 Выражения и операторы

Слово «оператор» в данном разделе соответствует английскому слову «operator». Все описания операторов приведены для их встроенных версий. Сами операторы описаны в главе 5 стандартов [C++ 03; C++ 11].

2.3.1 Арифметические операторы

+a	
-a	Изменение знака аргумента
++a	Преинкремент
a++	Постинкремент
–a	Предекремент
a–	Постдекремент
a + b	Сложение
a - b	Вычитание
a * b	Умножение
a / b	Деление
a % b	Остаток от деления
~a	Побитовая инверсия
a & b	Побитовое И
a | b	Побитовое ИЛИ
a ^ b	Побитовое Исключающее ИЛИ
a << b	Арифметический сдвиг a влево на b битов
a >> b	Арифметический сдвиг a вправо на b битов

2.3.2 Сравнение

a == b	Равенство
a != b	Неравенство
a < b	Меньше
a > b	Больше
a <= b	Меньше или равно
a >= b	Больше или равно

2.3.3 Логические операторы

!a	Логическое отрицание
a && b	Логическое И
a || b	Логическое ИЛИ

Логические операторы И и ИЛИ вычисляются не полностью. Их вычисление останавливается в тот момент, когда результат всего выражения становится ясен, например, если в выражении a && b вычисление a дает **false**, то вычисление всего выражения останавливается и возвращается **false**. Аналогично и для ИЛИ.

2.3.4 Присваивание

Присваивание в C++ является не управляющей конструкцией, как в Pascal, а простым оператором. Поэтому оно может использоваться внутри более сложных выражений. Также в C++ существует понятие составного присваивания. Операторы составного присваивания существуют для всех арифметических операторов и работают так, как если бы левому аргументу присвоили результат арифметической операции между левым и правым аргументами. Например, для сложения:

```
a += b; // a = a + b;
```

2.3.5 Условный оператор

Оператор **if** представляет собой управляющую конструкцию и не имеет значения, как результата вычисления. Но иногда удобно сократить запись вида:

```
std::string name;

if (has_display_name) {
  name = display_name;
} else {
  name = "None";
}
```

Для этого служит тернарный (или условный) оператор/ Его общий вид a ? b :c. Например:

```
std::string name = has_display_name ? display_name : "None";
```

Есть важное условие: типы b и c должны быть одинаковы, а a должно приводиться к булевскому значению.

Этот оператор можно применять и в качестве первого аргумента присваивания:

```
double r = 0.0;
double i = 0.0;

((value >= 0.0) ? r : i) = std::sqrt(std::abs(value));
```

Этот фрагмент кода вычисляет мнимое или реальное значение квадратного корня в зависимости от знака числа.

2.3.6 Последовательное вычисление

Иногда требуется последовательное вычисление нескольких выражений в таком месте, где допустимо только одно. Например, в заголовке цикла. В этой ситуации на помощь приходит оператор последовательного вычисления «,». Например, разворот массива на месте может быть записан как:

```
for (int i = 0, j = array_size - 1; i < j; ++i, --j) {
  int value = array[i];

  array[j] = array[j];
  array[j] = value;
}
```

Необходимо учитывать, что запятая при определении переменных (в примере это выражение инициализации цикла) и при вызове функций не является самостоятельным оператором, а всего лишь элемент синтаксической конструкции. В выражении перехода к следующей итерации (++i, --j) используется именно оператор последовательного вычисления.

2.3.7 Приоритет операторов

В стандарте языка C++ нет таблицы приоритета операций. Порядок вычисления операция выводится из грамматики языка. Тем не менее, такую таблицу можно сформировать в порядке убывания приоритета:

1 ::
2 a++ a– () [] . ->
3 ++a –a +a -a ! ~(*type*) *a &a sizeof new new[] delete delete[]
4 .* ->*
5 * / %
6 + -
7 << >>
8 < <= >= >
9 == !=
10 &
11 ^
12 |
13 &&
14 ||
15 ?: = += -= *= /= %= <<= >>= &= ^= |=
16 **throw**
17 ,

Глава 3

Типы данных

Язык C++, будучи статически строго типизированным, ассоциирует с каждым элементом программы набор информации, называемой типом данных. Тип характеризует набор значений, которые может принимать любой объект, принадлежащий к этому типу, а также набор операции, которые можно осуществлять над этими данными.

3.1 Простые типы данных

3.1.1 Целые числа

Самыми простыми типами в C++ являются целочисленные типы. Целые числа со знаком представлены следующими типами:

- **signed char**
- **short int**
- **int**
- **long int**
- **long long int** (C++ 11).

Для этих типов справедливо утверждение (S обозначает оператор **sizeof**, возвращающий размер объекта или типа):

$$S_{signedchar} \leq S_{shortint} \leq S_{int} \leq S_{longint} \leq S_{longlongint} \tag{3.1}$$

Тип **signed char** должен иметь размер, достаточный для хранения одного символа из базового набора реализации C++. Традиционно, его размер составляет 8 бит, но возможны и другие реализации [C++ 03; C++ 11, параграф 3.9.1.1]. В соответствии с параграфом 1 раздела 5.3.3 [C++ 03; C++ 11] выражение **sizeof**(**signed char**) всегда имеет значение 1.

Размер **int** выбирается как имеющий «естественный» размер для платформы [C++ 03; C++ 11, параграф 3.9.1.2]. На практике это означает, что размер **int** эквивалентен размеру регистра процессора.

Реализация также может предоставлять и другие знаковые типы, называемые расширенными.

Для каждого знакового типа, включая расширенными, реализация должна предоставить беззнаковый тип соответствующей размерности. Положительные значения соответствующего знакового типа должны входить в диапазон беззнакового типа и иметь такое же представление. Эти типы образуются путем добавления префикса **unsigned**: **unsigned char**, **unsigned short int**, **unsigned int**, **unsigned long int** и **unsigned long long int**. Все вычисления с беззнаковыми числами выполняются по модулю 2^n, где n это количество бит в представлении типа [C++ 03; C++ 11, параграф 3.9.1.4]

Непосредственно представленные в программе значения называются литералами. Для представления целочисленных значений разных типов используются суффиксы и префиксы числа. Десятичные значения представлены как обычно. Для шестнадцатеричных литералов используются префиксы `0x` или `0X`, восьмеричные числа начинаются с 0:

```
int a = 15;    // 15
int b = 0x15; // 21
int c = 015;   // 13
```

Для указания, что литерал является беззнаковым используются суффиксы `u` или `U`. За указание размера отвечают суффиксы `l`, `L` для **long int**, `ll` и `LL` для **long long int**. В стандарте C++ 11 правила немного поменялись и конкретный тип литерала будет выбран так, чтобы он имел минимальный размер, вмещающий значение. Детальную информацию можно найти в таблице 6 [C++ 11]. Например:

Литерал	C++ 03	C++ 11
15	`int`	`int`
15u	`unsigned int`	`unsigned int`
15l	`long int`	`int`

3.1.2 Вещественные числа

Вещественные числа представлены в языке 3 типами:

- **float**
- **double**
- **long double**.

Формат представления этих типов определяется реализацией. Как и в случае со знаковыми типами, размеры и диапазоны представления этих типов связаны аналогичными нестрогим отношением:

$$S_{float} \leq S_{double} \leq S_{longdouble} \tag{3.2}$$

На практике, реализация вещественной математики соответствует стандарту «IEEE 754» или эквиваленту «ISO/IEC/IEEE 60599:2001».

Вещественные литералы содержат десятичную точку в значении мантиссы и необязательную часть указания порядка, идущую после символа `e` или `E`. На размер указывают суффиксы `f` и `F` для **float** и `l` и `L` для **long double**:

```
15.0
-25.3
15.5f
123.0e-5l
```

3.1.3 Символы

За символы (characters) в C++ отвечают следующие типы:

- **signed char**
- **unsigned char**
- **char**
- **wchar_t**
- **char16_t** (C++ 11)
- **char32_t** (C++ 11)

Тип **char** является самостоятельным типом, но при этом он эквивалентен либо **signed char**, либо **unsigned char**. Это соответствие определяется реализацией [C++ 03; C++ 11, параграф 3.9.1.1].

Типы **char16_t** и **char32_t** являются беззнаковыми и содержат как минимум 16 и 32 бита соответственно.

Символьные литералы представляют собой символ или служебную последовательность (escape sequence), заключенную в апострофы. Символы апострофа, обратной косой черты и новой строки не могут быть использованы непосредственно, вместо них должны использоваться служебные последовательности. Стандарт определяет следующие служебные последовательности:

`\n`	NL(LF)	перевод строки
`\t`	HT	табуляция
`\v`	VT	вертикальная табуляция
`\b`	BS	«забой»

\r	CR	возврат каретки
\f	FF	конец страницы
\a	BEL	звуковой сигнал
\\	\	обратная косая черта
\?	?	знак вопроса
\'	'	апостроф
\"	"	кавычка
\0ooo	*ooo*	символ с восьмеричным кодом *ooo*
\xhhh	*hhh*	символ с восьмеричным кодом *hhh*

Литерал также может иметь префиксы:

u C++ 11, тип **char16_t**

U C++ 11, тип **char32_t**

L тип **wchar_t**

3.1.4 Булевский тип

Булевские значения в C++ представлены ключевыми словами **true** и **false**, которые имеют тип **bool**. Размер этого типа и конкретные значения, используемые в качестве **true** и **false**, определяются реализацией. Использование неинициализированных переменных как булевских значений ведет к неопределенному поведению, например, значение может и не соответствовать ни **true**, ни **false** [C++ 03, сноска 42][C++ 11, сноска 47].

3.2 Составные типы

Фундаментальные типы дают основу для представления данных, но для серьезной программы этого недостаточно. Пользовательские типы данных могут быть построены путем формирования составных типов данных (compound types).

3.2.1 Перечисления

Простейшим составным типом данных являются перечисления. Перечислением (enumeration) называется набор констант, объединенных вместе. Создаются перечисления при помощи ключевого слова **enum**:

```
enum color_t {
  COLOR_RED,
  COLOR_GREEN,
  COLOR_BLUE
};
```

Каждая константа, указанна в перечислении, получает целочисленное значение. Первая константа будет равно 0, каждая последующая формируется путем прибавления 1. Размер типа перечисления будет выбран так, чтобы вмещать максимальное значение. Если автоматически созданные значения не устраивают, можно задавать свои, например, те же константы для цвета можно задать соответствующими модели RGB:

```
enum color_t {
  COLOR_RED   = 0x0000FF,
  COLOR_GREEN = 0x00FF00,
  COLOR_BLUE  = 0xFF0000
};
```

Использовать перечисления можно везде, где ожидается целое число, например:

```
1 color_t color = COLOR_RED;
2 ...
3 switch (color) {
4 case COLOR_RED:
5   ...
6
```

```
 7  case COLOR_GREEN:
 8    ...
 9
10  case COLOR_BLUE:
11    ...
12  }
```

Хорошие компилятор выдаст предупреждение, если в операторе **switch** используется перечисление и при этом не все значения перечисления покрыты метками **case** и отсутствует метка **default**.

Как видно из примера, имена констант автоматически вносятся в то же пространство имен, в котором находит тип перечисления. Это может приводить к конфликтам имен. Также точный размер перечисления неизвестен, что не дает возможности его просто объявлять. В связи с этим стандарт C++ 11 расширил возможности перечислений.

Первая возможность относится к устранению конфликтов. После ключевого слова **enum** может идти одно из ключевых слов **class** или **struct**. Таким образом будет создано scoped enum:

```
enum class color_t {
  RED   = 0x0000FF,
  GREEN = 0x00FF00,
  BLUE  = 0xFF0000
};
```

Ссылаются на такие константы с указанием типа перечисления:

```
 1  color_t color = color_t::RED;
 2  ...
 3  switch (color) {
 4  case color_t::RED:
 5    ...
 6
 7  case color_t::GREEN:
 8    ...
 9
10  case color_t::BLUE:
11    ...
12  }
```

В этом примере префикс «COLOR» у констант исключен, так как они должны быть использованы только вместе с именем перечисления.

Вторая возможность относится к размеру целочисленного типа, представляющего данные. Этот тип может быть указан перед перечислением через двоеточие:

```
enum color_t: int {
  ...
};
```

Для scoped перечислений будет использован **int** в случае, если он не задан явно.

3.2.2 Массивы

Для хранения множества значений одинакового типа в языке существуют массивы. Массив задается как идентификатор, за которым следуют квадратные скобки с размером массива внутри:

```
int arr[5] = { 1, 2, 3 };
```

В примере создается массив из 5 целых чисел, инициализированный значениями 1, 2 и 3. Так как массив содержит 5 значений, оставшиеся без инициализаторов элементы будут инициализированы по умолчанию, для фундаментальных типов это значения, эквивалентные 0. Значения массива располагаются в памяти последовательно:

1	2	3	0	0

Для доступа к элементам массива также используются квадратные скобки с индексом внутри, индексация начинается с 0:

```
1  arr[0]; // 1
2  arr[2]; // 3
3  arr[5]; // Undefined behavior
```

Массив может быть многомерным:

```
double matrix[2][3] = {
  { 11, 12, 13 },
  { 21, 22, 23 }
};
```

matrix — это массив из 2-х массивов по 3 вещественных элемента. Первым меняется самый правый индекс, потом второй справа и так далее:

11	12	13	21	22	23

При наличии инициализатора самый левый размер может быть опущен, его значение будет вычислено по инициализатору:

```
int data[][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
```

3.2.3 Структуры

Массивы неплохо работают, когда необходимо представить в программе коллекцию, но они неудобны, когда одно значение состоит из нескольких элементов, например, координаты точки в пространстве, и совсем неприменимы, когда эти элементы разных типов. В этой ситуации на помощь приходят структуры. Структуры создаются при помощи ключевого слова **struct**:

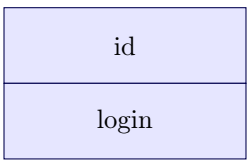
```
struct point_t {
  double x;
  double y;
  double z;
};

struct line_t {
  point_t begin;
  point_t end;
};

struct user_t {
  int id;
  std::string login;
};
```

В примере выше определяются 3 структуры: point_t определяет точку в 3-мерном пространстве, line_t — отрезок прямой, заданный началом и концом, user_t — пользователя некоей системы, который обладает целочисленным идентификатором (поле user_t::id) и именем (поле user_t::login).

Данные структуры хранятся в памяти последовательно:



Создание экземпляра структуры чем-то похоже на создание массива:

```
point_t pt = { 1.5, 2.5, 15.25 };

line_t lines[2] = {
```

```
    { { 0, 0, 0 }, { 1, 1, 1 } },
    { pt, { -1, -1, -1 } }
};
```

pt является точкой с координатами $(1.5, 2.5, 15.25)$. lines — это массив из 2-х отрезков, один из них от начала координат до точки $(1, 1, 1)$, второй от точки $(1.5, 2.5, 15.25)$ до точки $(-1, -1, -1)$.

Для доступа к элементам структуры используется оператор «.»:

```
double length = std::sqrt(pt.x * pt.x
    + pt.y * pt.y + pt.z * pt.z);
```

Допустимо определять анонимные структуры, если они сразу инициализируются, например, для преобразования цветов в названия и обратно можно сделать следующий массив:

```
1 const struct {
2   color_t color;
3   char name[50];
4 } COLOR_MAPPING[] = {
5   { color_t::RED, "red" },
6   { color_t::GREEN, "green" },
7   { color_t::BLUE, "blue" }
8 };
```

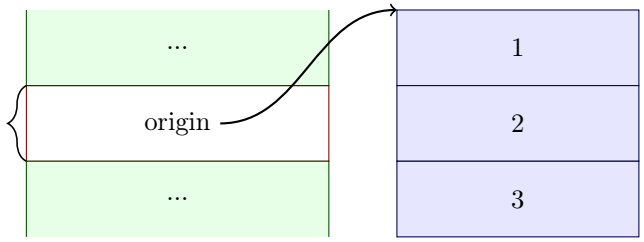
Возможности структур этим не ограничиваются. Язык также поддерживает возможность хранить внутри структур набор битов, так называемые битовые поля [C++ 03; C++ 11, раздел 9.6]. Также существует возможность определить тип, который может хранить данные разных типов, но в каждый момент только один из них. Эту возможность предоставляют объединения [C++ 03; C++ 11, раздел 9.5]. Эти возможности, чаще всего применяемые при низкоуровневом программировании, остаются для самостоятельного изучения.

3.3 Указатели

В тех случаях, когда требуется большая гибкость в доступе к данным, невозможно обойтись без указателей. Указатели хранят адрес какого-либо объекта в памяти. Например:

```
1 struct point_t {
2   double x;
3   double y;
4   double z;
5 };
6
7 point_t pt = { 1.0, 2.0, 3.0 };
8
9 const point * origin = &pt;
```

origin — это переменная-указатель, которая указывает на данные переменной pt.



Для получения адреса любого объекта (переменной или функции) используется унарный оператор «&», как в строке 9. Для получения объекта по адресу применяется оператор «*», а в случае структур, может использоваться оператор «->»:

```
double length = std::sqrt((*pt).x * (*pt).x + pt->y * pt->y
    + (*pt).z * pt->z);
```

Для указателей возможен вариант, когда они не указывают ни на какой объект. Такие указатели называются пустыми или нулевыми. В стандарте C++ 03 за пустой указатель отвечает литерал 0. Каждый раз, когда такой литерал встречается в контексте указателя, этот указатель будет иметь специальное значение. Необходимо отметить, что на большинстве платформ в памяти, где хранится значение указателя, пустой указатель будет представлен значением из всех нулевых байтов. Но стандарт этого не гарантирует, и существует возможность существования такой платформы, где пустой указатель и не равен физически 0.

Использование 0 в качестве литерала пустого указателя ведет к неоднозначности, поэтому в стандарте C++ 11 введена отдельная константа пустого указателя:

```
point_t * start = nullptr;
```

Для указателей существует адресная арифметика. Адреса допустимо менять и сравнивать. Например:

```
1  point_t points[10] = { ... };
2
3  point_t * ptr = &points[5];
4
5  *(ptr - 1);          // points[4]
6  *(ptr + 2);          // points[7]
7  ++ptr;               // points[6]
8
9  ptr > &points[0]; // true
10 ptr > &points[9]; // false
```

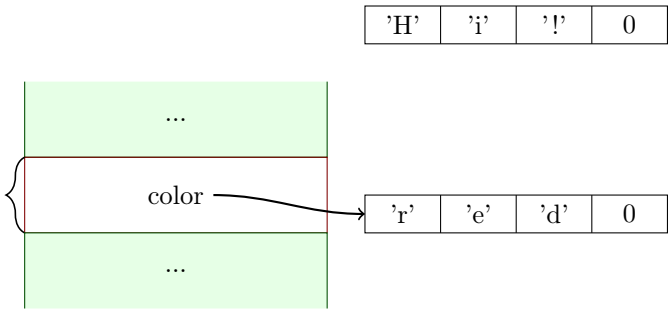
При прибавлении n к указателю добавляется размер объекта, на который он указывает, умноженный на n.

Сравнение указателей это сравнение адресов, хранящихся в них. Поэтому сравнение 2-х указателей, относящихся к разным массивам, представляет собой неуточненное поведение.

3.3.1 Строки

Особым случаем указателей и массивов являются строки и строковые литералы. Строки в C++ представляют собой последовательность символов, заканчивающуюся символом с кодом 0. Компилятор умеет преобразовывать строковые литералы в массивы символов и указатели на константные символы:

```
char greeting[] = "Hi!";
const char * color = "red";
```



Внутри строковых литералов можно использовать служебные последовательности для указания тех символов, которые невозможно ввести напрямую, например, конец строки.

Для указания типа символа в строке также используются префиксы:

u8 C++ 11, тип **char**, кодировка строки UTF-8

u C++ 11, тип **char16_t**

U C++ 11, тип **char32_t**

L тип **wchar_t**

3.4 Ссылки

Работа с указателями требует аккуратности. Указатели могут быть пустыми, а разыменование пустого указателя есть неопределенное поведение. В то же время косвенность при доступе к данным крайне удобна, поэтому для упрощения жизни разработчиков язык содержит ссылки. Ссылки обозначаются символом «&» при объявлении переменной и должны быть инициализированы. Объект, на который ссылается ссылка, изменять можно, саму ссылку — нет. Ссылка подобна указателю, который автоматически разыменовывается:

```
1 point_t pt = { ... };
2
3 point_t & ref = pt;
4
5 const point_t & const_ref = pt;
```

Наиболее часто ссылки применяются в качестве аргументов функций.

3.5 Размеры встроенных типов

Стандарт не определяет размеры встроенных типов, но есть несколько популярных платформ, данные по которым сведены в таблицу:

Платформа	`short`	`int`	`long`	`long long`	`void *`
Win32, 32bit UNIX	16	32	32	64	32
LLP64/IL32P64	16	32	32	64	64
LP64/I32LP64 (UNIX)	16	32	64	64	64

Глава 4

Функции

4.1 Определение функций

Программа на C++ представляет собой выполнение кода внутри функции main(). В теории, ничто не мешает всю программу разместить внутри этой функции, на практике же такую программу невозможно сопровождать. Простейшим способом структурирования кода программы является создание функций. В отличие от C, язык C++ требует обязательного наличия либо определения, либо объявления функции в точке ее вызова.

Простейшим примером может служить функция, вычисляющая факториал числа:

```
1  unsigned long long int factorial(unsigned int n)
2  {
3    unsigned long long int result = 1;
4
5    while (n > 1) {
6      result *= n--;
7    }
8
9    return result;
10 }
```

В строке 1 находится сигнатура функции, состоящая из имени функции (в данном случае «factorial») и ее параметров, заключенных в круглые скобки. Также в этой строке указан тип возвращаемого значения (**unsigned long long int**). Далее идет тело функции, выполняющее расчет.

Зачастую, в точке вызова функции совершено излишне знать полное ее определение. Этого не требуется для вызова и только замедляет компиляцию, а также ведет к тому, что в результате функция может быть определена более одного раза. Поэтому заголовочные файлы обычно содержат только объявление функции, которое выглядит как сигнатура функции с типом возвращаемого значения, после которой идет точка с запятой:

```
unsigned long long int factorial(unsigned int);
```

Так как объявление функции всего лишь декларирует ее наличие, оно может встречаться внутри единицы трансляции неоднократно. Более того, так как имена параметров могут и не нести особого смысла, в объявлениях функций их часто пропускают, как в примере выше.

4.1.1 Аргументы, параметры и возвращаемое значение

В определении функции в круглых скобках после имени располагаются ее аргументы, разделенные запятыми. Если в силу каких-либо причин внутри функции аргумент не используется, его имя следует пропустить, так как в противном случае компилятор будет жаловаться на неиспользуемый аргумент.

В точке вызова функции каждому ее аргументу сопоставляется фактический параметр вызова. Все значения передаются путем копирования, что позволяет изменять их внутри функции, как это, например, сделано в строке 6 с аргументом `n`. Его изменение никоим образом не отражается в точке вызова функции. Копирование данных может оказаться крайне неэффективным в случае использования больших структур данных. В этой ситуации на помощь приходят указатели и ссылки. Например, прототип функции, выводящей информацию о пользователе может выглядеть так:

```
1 void print_user(const user_t & /*user*/);
```

Информация о пользователе передается по ссылке, что предотвращает ее копирование. Так как предполагается, что функция print_user() не модифицирует переданные ей данные, они отмечены квалификатором **const**. У этой функции есть один недостаток: она способна выводить данные только на терминал. В то же время полезно иметь возможность выводить эти же данные в файл, например, в протокол работы программы. В этом случае поток вывода необходимо передать в качестве параметра:

```
1 void print_user(std::ostream & /*out*/,
2     const user_t & /*user*/);
```

std::ostream представляет собой обобщенный тип потока для вывода данных. Это поток также передается по ссылке, но она не отмечена как ссылка на константу. Это связано с тем, что вывод чего-либо в поток меняет его состояние, поэтому print_user() меняет свой параметр и эти изменения должны быть доступны за пределами функции.

В некоторых случаях для аргумента существует некое значение, которое обычно передается качестве параметра. Например, вычисление факториала можно написать рекурсивно:

```
1 unsigned long long int factorial(unsigned int n)
2 {
3   if (n > 1) {
4     return n * factorial(n - 1);
5   } else {
6     return 1;
7   }
8 }
```

Или даже так:

```
1 unsigned long long int factorial(unsigned int n)
2 {
3   return (n > 1) ? (n * factorial(n - 1)) : 1ull;
4 }
```

Но такое решение обладает одним недостатком: чем больше n, тем больше в стеке будет вызовов, что, в конце концов может привести к его переполнению (впрочем, в случае факториала, переполнение целочисленной переменной наступит раньше). Можно разрешить компилятору оптимизировать рекурсивный вызов, переписав рекурсию так, чтобы не было вычислений после рекурсивного вызова:

```
1 unsigned long long int factorial(unsigned int n,
2     unsigned long long int acc)
3 {
4   return (n > 1) ? factorial(n - 1, acc * n) : acc;
5 }
6 ...
7 factorial(5, 1);
```

Эта реализация разрешает компилятору заменить рекурсивный вызов на простой переход в начало функции, что решает проблему с переполнением стека. Но у такого решения есть другой недостаток: каждый вызывающий должен передать 1 в качестве второго аргумента. Неудобно нагружать клиентский код такими подробностями, поэтому язык предлагает так называемые аргументы по умолчанию. Каждому аргументу, начиная с последнего, можно сопоставить некое значение, которое будет использовано если клиентский код не передает этого аргумента:

```
1 unsigned long long int factorial(unsigned int n,
2     unsigned long long int /*acc*/ = 1);
3
4 unsigned long long int factorial(unsigned int n,
5     unsigned long long int acc)
6 {
7   return (n > 1) ? factorial(n - 1, acc * n) : acc;
8 }
```

Аргументы по умолчанию должны быть указаны в объявлении функции и пропущены в определении, если есть объявление.

Оператор **return** (строка 9 первого примера с факториалом, стр. 33) прерывает выполнение функции и возвращает свой параметр в качестве результата вычисления функции. Этот оператор может встречаться в теле функции неоднократно. Допустимо возвращать ссылки из функции, например, зачастую удобно связывать вывод в цепочки, для этого из функции print_user() следует вернуть ссылку на поток:

```
1 std::ostream & print_user(std::ostream & out,
2     const user_t & user);
3 {
4   ...
5   return out;
6 }
```

Необходимо помнить, что возвращать ссылки и указатели на автоматические переменные недопустимо, так как после выхода из функции они перестают существовать.

4.1.2 Временные значения и копирование данных

Вызов функции часто сопряжен с созданием и разрушением временных значений. Для каждого параметра внутри создается автоматическая переменная. В случае аргумента-ссылки эта автоматическая переменная ссылается на настоящий объект, определенный за пределами функции. В случае, если типы аргумента-ссылки и параметра не совместимы, компилятор делает одно из двух:

- если аргумент ссылается на неконстантный объект, компилятор выдает ошибку;
- если аргумент ссылается на константный объект и существует способ преобразования типа параметра в тип аргумента, компилятор создает временный объект типа аргумента и передает ссылку на него в качестве параметра.

Причина, по которой компилятор не создает временных объектов в случае ссылки на неконстанту, довольно проста: раз аргумент неконстантен, значит функция его изменяет, а потеря сделанных изменений будет крайне неожиданна для разработчика.

Аргументы по умолчанию также является временными значениями.

Возврат значения из функции также создает временный объект, поэтому нельзя создать ссылку на результат вызова функции, если функция возвращает не ссылку. Из этого правила есть одно исключение: допустимо создать ссылку на константный объект, возвращенный из функции. В этом, и только в этом случае время жизни временного объекта будет продлено до конца времени жизни ссылки:

```
1 void foo(const char * name)
2 {
3   session_t login_user(const char * name);
4
5   const session_t & session = login_user(name);
6   ...
7 }
```

В примере временный объект, возвращенный из login_user() будет существовать до конца функции foo()

Дополнительную информацию можно найти в книге Скотта Мейерса [Mey96, совет 19].

4.1.3 Типы связывания (linkage specification)

Типы связывания в C++ непосредственно следуют из процесса построения результирующей программы. Упрощенно, тип связывания определяет степень видимости символа между единицами трансляции. Существует 3 типа связывания:

Внутреннее Символ с внутренним связыванием доступен только внутри той же единицы трансляции.

Внешнее Внешнее связывание позволяет использовать символ в других единицах трансляции.

Внешнее со спецификатором «C» Этот вариант позволяет использовать и экспортировать символы, совместимые с языком C.

Например:

```
1  int i = 1;
2  static int j = 2;
3  const int k = 3;
4  extern const int l = 4;
5  extern int m;
6
7  void foo();
8  extern void bar();
9  static void baz();
10 extern "C" int rand();
```

В строке 1 показан вариант с внешним связыванием. Переменная i будет размещена и приниициализирована в данной единице трансляции. Строка 2 демонстрирует переменную с внутренним связыванием, только функции в данной единице трансляции будут иметь к ней доступ. Переменная k в строке 3 является неизменяемой. Это также подразумевает внутреннее связывание. Если требуется константа с внешним связыванием, применение ключевого слова **extern** позволяет добиться требуемого результата (строка 4). Вопреки расхожему мнению, строка 5 всего лишь декларирует переменную с внешним связыванием. Ее определение находится в другой единице трансляции.

Правила для функций аналогичны. В строках 7 и 8 приведены объявления для функций с внешним связыванием. Строка 9 содержит объявление функции с внутренним связыванием, а строка 10 — объявление функции, реализованной на языке C.

Типы связывания тесно перекликаются с понятием классов памяти.

4.1.4 Сокрытие деталей реализации

Решение с аккумулятором довольно эффективно, но при этом оно обладает существенным недостатком: невозможно запретить пользовательскому коду передать вторым аргументом не 1, а это ведет к неправильной работе функции. На самом деле, то, что функция работает рекурсивно и использует аккумулятор является классическим примером деталей реализации. Хорошие компоненты скрывают детали реализации от клиентского кода.

Решить эту задачу в C++ можно 2-мя способами. Первый работает и в C. Как следует из предыдущего раздела, функции с внутренним связыванием недоступны за пределами единицы трансляции. Таким образом, вычисление факториала можно разделить на 2 части:

```
1  static
2  unsigned long long int factorial_impl(unsigned int n,
3      unsigned long long int acc)
4  {
5    return (n > 1) ? factorial_impl(n - 1, acc * n) : acc;
6  }
7
8  unsigned long long int factorial(unsigned int n)
9  {
10   return factorial_impl(n, 1);
11 }
```

Второй способ использует пространства имен. Пространство имен представляет собой некий именованный регион, созданный при помощи ключевого слова **namespace**, содержащий объявления и определения. Для обращения к сущностям, объявленным и/или определенным в нем, используется оператор «::»:

```
1  namespace lab {
2    unsigned long long int factorial(unsigned int n);
3  }
4  ...
5  lab::factorial(5);
```

Возможен вариант, когда пространство имен не имеет имени. В этом варианте компилятор генерирует для него уникальное имя так, чтобы доступ к содержимому был только в той же единице трансляции, где оно содержится. Все элементы пространства имен по умолчанию будут иметь внешнее связывание. Для сокрытия деталей реализации можно поступить так:

```
1  namespace {
2    unsigned long long int factorial_impl(unsigned int n,
3        unsigned long long int acc)
4    {
5      return (n > 1)
6          ? factorial_impl(n - 1, acc * n)
7          : acc;
8    }
9  }
10
11 unsigned long long int lab::factorial(unsigned int n)
12 {
13   return factorial_impl(n, 1);
14 }
```

4.2 Перегрузка функций и операторов

При реализации различных задач часто возникают ситуации, когда одно и то же выполняется немного разными способами в зависимости от того, какая информация доступна. Например, поиск информации о конкретном пользователе системы может выполнятся по его идентификатору или по имени учетной записи (логину). Логически эти операции должны называться одинаково, но в C и Pascal это невозможно, поэтому появляются функции getUserById() и getUserByLogin(). C++ предлагает эффективное решение этой проблемы, называемое перегрузкой (overload) функций.

4.2.1 Функции

Перегруженной называется функция, которая имеет 2 и более определения, отличающиеся сигнатурой, т.е. количеством, типами или порядком аргументов. Рассматриваются только типы аргументов. Например, для получения информации о пользователе могут быть объявлены следующие перегруженные функции:

```
1 // I
2 const user_t * getUser(int id);
3 // II
4 const user_t * getUser(const char * login);
```

Встретив вызов функции, компилятор выберет подходящий по типам параметров вариант и разместит его вызов:

```
1 char ADMIN_USER[] = "root";
2
3 getUser(15);          //  I
4 getUser("web");       // II
5 getUser(ADMIN_USER); // II
6 getUser(0);           //  I
```

Необходимо отметить, что добавить вариант перегруженной функции можно в любой единице трансляции, в том числе и с внутренним связыванием. С другой стороны, определение в разных единицах трансляции перегруженных функций с одинаковой сигнатурой является нарушением правила одного определения (стр. 14). Компилятор выбирает вызываемую функцию, рассматривая известные перегрузки в точке вызова. Все варианты, которые не объявлены к этой точке, в качестве кандидатов не рассматриваются.

Некоторые типы при определении перегрузки считаются эквивалентными:

- Указатели и массивы:

```
void foo(int * bar);
void foo(int bar[]);
```

- Функции и указатели на такую же функцию:

```
void foo(int bar());
void foo(int (* bar)());
```

- Аргументы, отличающиеся только квалификаторами **const** и **volatile**:

```
void foo(int bar);
void foo(const int bar);
void foo(volatile int bar);
```

Но следующие объявления относятся к разным перегрузкам:

```
void foo(int & bar);
void foo(const int  & bar);
```

4.2.2 Операторы

Помимо функций стандарт разрешает перегружать операторы. Допустимо перегружать все операторы за исключением:

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

Для перегрузки оператора необходимо, чтобы хотя бы один его аргумент был пользовательским типом. Некоторые операторы обретают или теряют специфичные особенности. Это будет рассмотрено в соответствующих разделах.

Унарные операторы могут быть перегружены только как метод класса, не принимающий никаких аргументов. Примеры такой перегрузки будут приведены в разделах, посвященных классам и управлению ресурсами.

Бинарные операторы могут перегружены и как методы класса, и как свободные функции, принимающие 2 аргумента. Например, для библиотеки матричных операций было бы логично определить операции сложения и вычитания матриц:

```
1  namespace math {
2    struct matrix_t {
3      ...
4    };
5
6    matrix_t createMatrix(unsigned int cols,
7        unsigned int rows);
8
9    matrix_t operator +(const matrix_t & lhs,
10       const matrix_t & rhs);
11   matrix_t & operator +=(matrix_t & lhs,
12       const matrix_t & rhs);
13   matrix_t operator -(const matrix_t & lhs,
14       const matrix_t & rhs);
15   matrix_t & operator -=(matrix_t & lhs,
16       const matrix_t & rhs);
17 }
```

Важно, чтобы семантика перегруженного оператора была очевидной (для сложения и вычитания матриц это правило соблюдается), а также чтобы парные операторы давали согласованное поведение. Для этого операторы обычно реализуются друг через друга:

```
1  math::matrix_t & math::operator +=(math::matrix & lhs,
2      const math::matrix_t & rhs)
3  {
4    ...
5    return lhs;
6  }
7
```

```
 8 math::matrix_t math::operator +(
 9     const math::matrix_t & lhs,
10     const math::matrix_t & rhs)
11 {
12   matrix_t result = lhs;
13
14   result += rhs;
15
16   return result;
17 }
```

4.2.3 Тонкие моменты

Перегрузка функций вовлекает в процесс множество тонких материй стандарта, таких как встроенные преобразования типов. Зачастую это ведет к интересным эффектам, которых разработчик совершенно не ожидает.

Крайне не рекомендуется перегружать функцию по указателю и целому числу, а также крайне осторожно подходить к перегрузке по целым и вещественным типам:

```
 1 //   I
 2 void foo(char *);
 3 //  II
 4 void foo(int);
 5 // III
 6 void foo(long long int);
 7 //  IV
 8 void foo(float);
 9 ...
10 char str[] = "bar";
11 const char cstr[] = "baz";
12
13 foo(str);      //   I
14 foo(cstr);     // Error
15 foo("quux");   // Error
16 foo(0);        //  II
17 foo(nullptr); //   I
18 foo(15ll);     // III
19 foo(1.5);      // Error
```

Еще более опасным является перегрузка по **bool**. Встроенное преобразование в **bool** определено для многих типов и очень часто доминирует:

```
 1 // I
 2 void foo(char *);
 3 // II
 4 void foo(int);
 5 // V
 6 void foo(bool);
 7 ...
 8 char str[] = "bar";
 9 const char cstr[] = "baz";
10
11 foo(str);      //  I
12 foo(cstr);     //  V
13 foo("quux");   //  V
14 foo(0);        // II
```

Перегрузки операторов «&&», «||» и «,» также следует избегать [Mey96, совет 7]. Это связано с тем, что перегруженные версии этих операторов работают по правилам вызова функций. Это приводит к тому, что для «&&» и «||» вычисляются оба аргумента в произвольном порядке [C++ 03, сноска 12]. То же самое относится и оператору «,».

Глава 5

Препроцессор

5.1 Стадии обработки программы

Программа, написанная на языке C++, проходит несколько стадий преобразования прежде чем превратится в исполняемый файл.

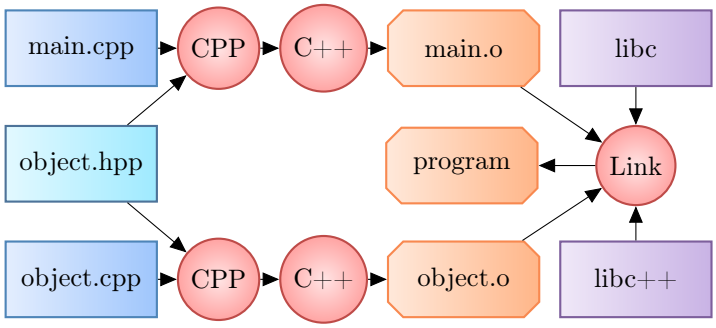


Рис. 5.1: Построение программы на C++

Первым в игру вступает препроцессор. Его задачами являются простые преобразования текста программы:

- формирование единой единицы трансляции для компилятора;
- включение и исключение фрагментов кода;
- макроподстановка;
- удаление комментариев.

Результат работы препроцессора передается компилятору C++. В результате компиляции для каждой единицы трансляции создается так называемый объектный файл. Объектный файл содержит машинные коды для функций, определенных внутри единицы трансляции, переменные с внутренним связыванием, определенные переменные с внешним связыванием. Все используемые, но не определенные символы с внешним связыванием при этом вносятся в специальные таблицы.

Набор объектных файлов далее передается компоновщику. Его задача из объектных файлов и библиотек сформировать исполняемый файл. Компоновщик связывает различные символы, подставляя их расположение в точки использования.

Для простейшей программы, состоящей из одной-двух единиц трансляции, все эти стадии проходят незаметно внутри одного вызова компилятора:

```
$ c++ -std=c++11 -Wall -Wextra -o program main.cpp object.cpp
$ ./program
```

Для крупных программ неразумно так поступать, так как от одной компиляции к другой меняется только малая часть исходных текстов. В этом случае скорость сборки программы растет, если использовать объектные файлы неизменившихся единиц трансляции с предыдущей сборки. В этом случае стадия компоновки выделяется отдельным вызовом:

```
$ c++ -std=c++11 -Wall -Wextra -c -o main.o main.cpp
$ c++ -std=c++11 -Wall -Wextra -c -o object.o object.cpp
$ c++ -std=c++11 -Wall -Wextra -o program main.o object.o
$ ./program
```

Обычно эту работу поручают специальным программам, которые умеют определять зависимости и перекомпилировать и перекомпоновывать только реально изменившиеся файлы. Типичным примером такой программы является Make.

С другой стороны, стадию препроцессора редко отделяют от компиляции, так как особой ценности результат его работы не несет. Отдельный вызов препроцессора применяется только для отладочных целей.

5.2 Директивы препроцессора

Препроцессор читает исходный текст программы и выполняет директивы, размещенные в нем. Все директивы препроцессора начинаются с символа «#» и делятся на несколько категорий.

5.2.1 Включение текста

За включение текста отвечает директива #**include**. Эта директива существует в 2-х видах:

```
1 #include <Object.hpp>
2
3 #include "Impl.hpp"
```

Первый вариант на строке 1 выполняет поиск в определенных реализацией C++ местах заголовочного файла, указанного в угловых скобках (параграф 16.2.2 [C++ 03; C++ 11]). Имя заголовочного файла и его связь с реальным содержимым, а также определение мест поиска зависит от реализации.

Второй вариант на строке 3 в соответствии с параграфом 16.2.3 [C++ 03; C++ 11] ищет исходный файл в соответствии с реализацией языка. Если такой файл не найден, директива выполняется так, как будто вместо кавычек использованы угловые скобки.

Таким образом, #**include** <string> не обязана включать файл `string`. Например, его содержимое может и не присутствовать в файловой системе совсем.

Стандартной практикой является применение следующих техник:

- Все заголовочные файлы от стандартных библиотек C и C++ включаются только при помощи угловых скобок, так как они могут быть и не файлами (хотя вероятность этого крайне низка).
- Все заголовочные файлы внутри компонента включаются при помощи кавычек.
- Заголовочные файлы внешних компонентов включаются при помощи угловых скобок после того, как каталог с ними должным образом добавлен в список путей поиска.
- Заголовочные файлы компонента, составляющие внешний интерфейс, внутри компонента также включаются при помощи угловых скобок.

5.2.2 Символы препроцессора

Директива #**define** вместе с обратной директовой #**undef** определяет и, соответственно, уничтожает определение символа препроцессора:

```
1 #define SYMBOL
2 #define OBJECT_COUNT 10
3
4 ...
5
6 #undef SYMBOL
```

Листинг 5.1: Директива #define

Чаще всего символы используются для условной компиляции и макроподстановки.

По соглашению, все символы препроцессора, определяемые в программе, состоят из заглавных букв, отдельные слова в имени символа разделяются подчеркиваниями («_»).

Язык определяет некоторое количество стандартных символов [C++ 03; C++ 11, раздел 16.8]:

`__cplusplus`	Определен при компиляции компилятором C++, имеет значение `199711L` если компилятор соответствует C++ 98 или C++ 03, и `201103L` если компилятор соответствует C++ 11.
`__DATE__`	Дата единицы трансляции.
`__FILE__`	Имя текущего исходного файла в виде строкового литерала.
`__LINE__`	Номер строки в текущем исходном файле.
`__TIME__`	Время единицы трансляции.

5.2.3 Условная компиляция

Условная компиляция позволяет включать и исключать фрагменты исходных файлов из компиляции. Чаще всего это используется для предотвращения повторного определения различных объектов языка из-за повторного включения заголовочных файлов. На втором месте по использованию условной компиляции стоит написание платформо-зависимых частей кода. Условной компиляцией управляют следующие директивы:

#**ifdef**	Включает последующий код, если определен символ препроцессора, переданный параметром директивы.
#**ifndef**	Обратный вариант к #**ifdef**: включает последующий код, если символ не определен.
#**if**	Включает последующий код, если выражение не 0. Выражение может включать в себя простые вычисления, сравнения и оператор #defined(), проверяющий, что символ определен.
#**else**	Заканчивает блок, начатый любым оператором #**if** и начинает новый блок кода. Если предыдущий блок включался в компиляцию, новый будет исключен и наоборот.
#**elif**	Заканчивает блок и начинает новый. Если новый блок должен быть включен, он начинается с вычисления условия, эта директива эквивалентна последовательности #**if** - #**else** - #**if** - #**endif** - #**endif**
#**endif**	Завершает блок условной компиляции. Каждой директиве #**if**/#**ifdef**/#**ifndef** должна соответствовать директива #**endif**.

Например, библиотека, написанная на языке C, для правильной работы в C++ должна указывать тип связывания «C». Так как это **extern** "C" в языке C не определен, его необходимо исключить при использовании компилятора C и включить при использовании C++:

```
1 #ifdef __cplusplus
2  extern "C" {
3 #endif
4
5  double sqrt(double);
6
7 #if defined(__cplusplus)
8  }
9 #endif
```

В примере на строках 1 и 7 проверяется, что определен символ `__cplusplus` и в этом случае добавляется указание типа связывания. В данном примере символ проверяется разными способами, но обычно используется только один либо в соответствии со стандартами кодирования проекта, либо по вкусу автора.

В программах, состоящих из нескольких единиц трансляций важно правильно использовать объявления и определения различных объектов языка. Обычно все объявления функций и определения типов данных сводят в отдельный файл заголовка, который включают в единицу трансляции, определяющую функции из него. Этот же заголовочный файл включают единицы трансляции, использующие объявленные в нем объекты языка. Так как каждый заголовочный файл должен быть самодостаточным (клиентский код не должен знать о его зависимостях), в заголовочные файлы включают другие заголовочные файлы, получая направленный граф. В этом случае возможно, что в одну единицу трансляции один и тот же заголовочный файл будет включен более одного раза, что приведет к ошибке компиляции, так как типы из этого файла будут определены более одного раза. Для избежания этого используется техника, называемая «header guard».

«Header guards» представляют собой связку из директив #**ifndef** и #**define**. Все содержимое заголовочного файла помещается внутрь директивы #**ifndef**. Первой строкой внутри директивы определятся символ, проверяемый в условии. Например:

```
1 #ifndef TEST_HEADER
```

```
2 #define TEST_HEADER
3
4 ...
5
6 #endif // TEST_HEADER
```

Соответственно, при повторном включении этого файла препроцессор увидит, что символ `TEST_HEADER` определен и не будет включать в единицу трансляции весь текст до #**endif**.

Имя guard'а (имя символа препроцессора, определяемого через #**define**) видно препроцессору во всей единице трансляции. Поэтому если 2 заголовочных файла используют один и тот же символ, результат компиляции может разочаровать даже самого самоуверенного разработчика, а искать эту проблему довольно сложно, так как все ошибки компиляции носят косвенный характер. Вследствие этого рекомендуется добавлять в имя guard'а не только имя файла, но и дополнительную информацию, например, имя компонента. Применение для этого GUID'ов обычно выглядит чрезмерно, хотя и допустимо.

5.2.4 Макроподстановка

Препроцессор также способен проводить простые манипуляции с текстом программы, выполняя замену одних фрагментов текста на другие. Каждый символ, определенный через #**define** участвует в макроподстановке. Для того, чтобы придать ему особенный смысл, необходимо в директиве после имени символа указать выражение, на которое символ должен быть заменен. Например, в листинге 5.1 `OBJECT_COUNT` будет заменяться на значение 10. Эта замена будет выполнятся для всех вхождений `OBJECT_COUNT`, как самостоятельных символов, но не частей других, например, `TEST_OBJECT_COUNT`. Таким образом, программа:

```
1  #define SYMBOL
2  #define OBJECT_COUNT 10
3
4  ...
5
6     int TEST_OBJECT_COUNT = 15;
7
8     std::cout << OBJECT_COUNT << "\n"
9               << TEST_OBJECT_COUNT << "\n";
10
11 ...
12
13 #undef SYMBOL
```

Листинг 5.2: Простая макроподстановка

Выводит

```
10
15
```

Такой простой подстановкой многого не сделать, поэтому макросы могут иметь аргументы, указываемые после имени в круглых скобках:

```
1  #define LOG_MESSAGE(severity, message)      \
2     std::cerr << __FILE__                    \
3     << ":" << __LINE__                       \
4     << "[" << severity << "] " << message << "\n"
5
6  ...
7
8  LOG_MESSAGE("DEBUG", "Debug message");
```

Листинг 5.3: Макроподстановка с параметрами

Определение макроса `LOG_MESSAGE` содержит использование 2-х стандартных символов `__FILE__` и `__LINE__`. Поэтому при использовании макроса вместо них будут использованы актуальные значения для точки использования. Например, в качестве строки будет выведено 8. Сам макрос имеет довольно большое

определение, поэтому он разбит на несколько строк, каждая из которых, за исключением последней, заканчивается на символ обратной косой черты «\», который говорит препроцессору объединить эту строку с последующей.

На строке 4 отсутствует завершающая точка с запятой. Это сделано для того, чтобы каждое использование `LOG_MESSAGE` завершалось точкой с запятой. С точки зрения компилятора не важно, где именно эта точка с запятой находится, но редакторы правильно выравнивают код только в том случае, если каждое выражение заканчивается точкой с запятой, так как они не выполняют макроподстановку самостоятельно.

Внутри определения макроса 2 оператора выполняют особые преобразования:

- # преобразует следующий токен в строковый литерал.
- ## объединяет 2 токена в единый токен.

Эти 2 оператора часто используются для формирования идентификаторов и названий. Например, с помощью макросов можно сформировать функцию-преобразователь перечисления в строку:

```
1  #include <iostream>
2
3  enum struct color_t {
4    red,
5    green,
6    blue
7  };
8
9  #define MAKE_ENTRY(type, value) { type##_t::value, \
10         #value }
11 #define MAKE_MAPPING(type) \
12   const struct {           \
13   type##_t value;          \
14   const char * string;     \
15 } type##_MAPPING[] = {
16 #define END_MAPPING(type) };                          \
17   const char * get_##type##_name(type##_t value) \
18   {                                                  \
19     for (size_t i = 0;                               \
20          i < sizeof(type##_MAPPING) / sizeof(type##_MAPPING[0]); \
21          ++i) {                                      \
22       if (type##_MAPPING[i].value == value) {       \
23         return type##_MAPPING[i].string;            \
24       }                                             \
25     }                                               \
26     return nullptr;                                 \
27   }
28
29 MAKE_MAPPING(color)
30   MAKE_ENTRY(color, red),
31   MAKE_ENTRY(color, green),
32   MAKE_ENTRY(color, blue)
33 END_MAPPING(color);
34
35 int main(int, char *[])
36 {
37   std::cout << get_color_name(color_t::red) << "\n"
38             << get_color_name(color_t::green) << "\n"
39             << get_color_name(color_t::blue) << "\n";
40
41   return 0;
42 }
```

Листинг 5.4: Генерация конвертера

TODO: Описание примера

Результатом обработки будет (начало единицы трансляции со стандартными заголовками удалено):

```
 1 enum struct color_t {
 2  red,
 3  green,
 4  blue
 5 };
 6 #line 27 "convert-enum.cpp"
 7 const struct { color_t value; const char * string; } color_MAPPING[] = {
 8  { color_t::red, "red" },
 9  { color_t::green, "green" },
10  { color_t::blue, "blue" }
11 }; const char * get_color_name(color_t value) { for (size_t i = 0; i <
       sizeof(color_MAPPING) / sizeof(color_MAPPING[0]); ++i) { if
       (color_MAPPING[i].value == value) { return color_MAPPING[i].string; } }
       return nullptr; };
12
13 int main(int, char *[])
14 {
15  std::cout << get_color_name(color_t::red) << "\n"
16            << get_color_name(color_t::green) << "\n"
17            << get_color_name(color_t::blue) << "\n";
18
19  return 0;
20 }
```

Листинг 5.5: Результат генерации конвертера

5.2.5 Вспомогательные директивы

Помимо описанных, препроцессор также поддерживает некоторые вспомогательные директивы, пример которых можно увидеть в листинге 5.5.

Первая из этих директив #**line**. Эта директива предоставляет контроль за именем исходного файла и номерами строк в сообщениях компилятора. Она принимает 1 или 2 параметра. Первый параметр указывает, какой номер строки должен использоваться далее. Компилятор начнет считать последующие строки начиная с этого номера. Второй параметр отвечает за имя исходного файла. Все сообщения компилятора начиная с этой директивы начнут относится в указанному во втором параметре файлу. Если второго параметра нет, имя файла не меняется. Эта директива широко применяется различными генераторами кода. Пример ее использования можно видеть в листинге 5.5 на строке 6.

Вторая директива, #**error**, позволяет выдать сообщение об ошибке и остановить компиляцию. Она принимает в качестве параметра сообщение.

5.3 Подводные камни препроцессора

Препроцессор работает на самом раннем этапе обработки программы. Он не знает никаких синтаксических правил языка C++ и выполняет разбиение исходного текста на токены по самым простейшим правилам. Как результат, неудачно определенный символ препроцессора может испортить не один день неаккуратному разработчику. Типичным примером таких макросов являются `min` и `max`, автоматически определяемые при подключении заголовочного файла `windows.h` на платформе Microsoft Windows. Стандартная библиотека C++ определяет некоторое количество функций с именами min() и max(). Как следствие конфликта, препроцессор заменяет вызовы этих функций на раскрытые макросы, приводя код в синтаксически неверное состояние. Наиболее правильным способом борьбы с этим является определение символа `WIN32_LEAN_AND_MEAN` перед включением *windows.h. Еще более непредсказуемо работающие макросы находятся в заголовочном файле *stddlgs.h на той же платформе.

Проблемы с макросами не исчерпываются конфликтами имен. Результат раскрытия макроса тоже может принести сюрпризы. Например:

```
1 #include <iostream>
2 #define ADD_TWO(x) x + 2
3
4 int main(int, char *[])
5 {
```

```
 6    int i = 5;
 7    int j = ADD_TWO(i << 2);
 8
 9    std::cout << j << "\n";
10
11    return 0;
12  }
```

Ожидаемый результат $22 = 5 * 4 + 2$. В действительности, результатом вычисления будет 80, так как результатом раскрытия макроса будет выражение $5 << 2 + 2$, а приоритет сдвига ниже чем сложения. Для решения этой проблемы можно добавить скобки вокруг аргумента макроса:

```
 1  #include <iostream>
 2
 3  #define ADD_TWO(x) (x) + 2
 4
 5  int main(int, char *[])
 6  {
 7    int i = 5;
 8    int j = ADD_TWO(i << 2);
 9
10    std::cout << j << "\n";
11
12    return 0;
13  }
```

Но в другом выражении опять проблемы:

```
 1  #include <iostream>
 2
 3  #define ADD_TWO(x) (x) + 2
 4
 5  int main(int, char *[])
 6  {
 7    int i = 5;
 8    int j = ADD_TWO(i << 2);
 9
10    std::cout << j << "\n";
11
12    j = ADD_TWO(i << 2) * 3;
13    std::cout << j << "\n";
14
15    return 0;
16  }
```

Ожидаемое значение 66, но результат всего лишь 26, так как результат раскрытия $(5 << 2) + 2 * 3$. Эта проблема также решается добавлением скобок вокруг всего макроса:

```
 1  #include <iostream>
 2  #define ADD_TWO(x) ((x) + 2)
 3
 4  int main(int, char *[])
 5  {
 6    int i = 5;
 7    int j = ADD_TWO(i << 2);
 8
 9    std::cout << j << "\n";
10    j = ADD_TWO(i << 2) * 3;
11    std::cout << j << "\n";
12
13    return 0;
14  }
```

Увы, но и это еще не конец. Рассмотрим следующий пример:

```
 1 #include <iostream>
 2
 3 #define MIN_M(a, b) ((a) <= (b) ? (a) : (b))
 4
 5 int min_f(int a, int b)
 6 {
 7   return a <= b ? a : b;
 8 }
 9
10 int main(int, char *[])
11 {
12   int a = 5;
13   int b = 15;
14
15   std::cout << min_f(++a, ++b) << "\n";
16
17   a = 5;
18   b = 15;
19   std::cout << MIN_M(++a, ++b) << "\n";
20
21   return 0;
22 }
```

Результатом вычисления будет

```
6
7
```

Второе число равно 7 по той причине, что инкремент вычисляется дважды при раскрытии макроса как результат подстановки, в то время как при вызове функции вес параметры вычисляются один раз перед вызовом. Проблема множественных вычислений в макросах не решаема в текущей реализации препроцессора.

Глава 6

Классы

6.1 Классы и объекты

Классы в C++ – это сущности, объединяющие данные и функции для их обработки. Классы создаются при помощи ключевых слов **class**, **struct** и **union**. Каждый класс, также как и структуры, создает новый тип. Экземпляры классов называются объектами. Создание программы как набора взаимодействующих объектов составляет суть объектно-ориентированного программирования.

6.1.1 Поля и методы

Каждый класс содержит данные и функции, работающие с этими данными. Данные обычно называют полями. Функции, предназначенные для обработки данных, называют методами и они определяются внутри классов также как и поля. Например, класс, описывающий прямоугольник, умеющий себя рисовать и перемещать, может выглядеть так:

```
1  struct Rectangle {
2    double x;
3    double y;
4    double width;
5    double height;
6
7    void draw();
8    void move(double dx, double dy);
9    void setWidth(double width);
10 };
```

2 метода draw() и move() могут быть реализованы следующим образом:

```
1  void Rectangle::draw()
2  {
3    moveTo(x, y);
4    lineTo(x + width, y);
5    lineTo(x + width, y + height);
6    lineTo(x, y + height);
7    lineTo(x, y);
8  }
9
10 void Rectangle::move(double dx, double dy)
11 {
12   x += dx;
13   y += dy;
14 }
15
16 void Rectangle::setWidth(double width)
17 {
18   this->width = width;
19 }
```

Из этого примера видно, что поля класса доступны внутри его методов непосредственно по имени, как если бы они были локальными переменными. Тем не менее, возможна ситуация, когда локальная переменная имеет такое же имя, как и поле. В этой ситуации локальная переменная скрывает поле. Для обращения к полю каждому методу передается неявный параметр **this**, который указывает на экземпляр класса, метод которого вызван.

Обычные, или свободные, функции могут быть перегружены для указателей на константные объекты таким образом, что их поведение будет отличаться. Так как указатель на самого себя передается неявно, прямого контроля за его типом нет, что может приводить к разного рода трудностям, например:

```
1 #include "rectangle.hpp"
2
3 void handleRectangle(const Rectangle & r)
4 {
5   ...
6   r.draw();
7   ...
8 }
```

```
$ c++ -std=c++11 -Wall -Wextra draw-const-rectangle.cpp
draw-const-rectangle.cpp:6:2: error: member function 'draw' not viable: 'this'
      argument has type 'const Rectangle', but function is not marked const
        r.draw();
        ^
./rectangle.hpp:7:7: note: 'draw' declared here
        void draw();
             ^
1 error generated.
```

Clang в данном случае многословен, подсказывая в чем проблема, другие компиляторы предоставят меньше информации, но результат будет тем же. Это связано с тем, что компилятор предполагает, что метод draw() модифицирует экземпляр класса, что невозможно для константного объекта. Для того, чтобы подсказать компилятору, что метод не модифицирует свой объект, необходимо объявить константный метод, указав в качестве суффикса **const**:

```
1   ...
2   void draw() const;
3   ...
```

Необходимости определять второй, неконстантный метод, нет, так как компилятор автоматически добавит квалификатор **const** к неконстантному объекту в случае, если есть только константный метод.

Как обычно, все сказанное про **const** относится и к **volatile**.

Любое поле или метод могут быть отмечены ключевым словом **static**. В этом случае оно не меняет типа связывания, вместо этого оно меняет поведение. Все поля и методы, отмеченные этим словом, являются общими для всех объектов этого класса. Для доступа к ним не требуется иметь экземпляр класса, и методы также не имеют неявного параметра **this**.

6.2 Специальные методы классов

Некоторое количество методов класса носят специальный характер. Компилятор попытается автоматически создать их для разработчика.

6.2.1 Конструкторы

Приведенный выше пример прямоугольника обладает одним недостатком: существует возможность создать его без инициализации полей. Для решения этой проблемы специальная функция-конструктор вызывается при создании объекта. Конструктор должен иметь имя, совпадающее с именем класса. При вызове конструктора срабатывают стандартные правила перегрузки, поэтому количество конструкторов не ограничено.

Для класса прямоугольника можно написать один конструктор, принимающий координаты левого нижнего угла, ширину и высоту:

```
 1 #include <iostream>
 2
 3 struct Rectangle {
 4   double x;
 5   double y;
 6   double width;
 7   double height;
 8
 9   Rectangle(double left, double bottom,
10       double w, double h):
11     x(left),
12     y(bottom),
13     width(w),
14     height(h)
15   {
16     if (width <= 0) {
17       std::cerr << "Invalid rectangle width" << std::endl;
18     }
19     if (height <= 0) {
20       std::cerr << "Invalid rectangle height" << std::endl;
21     }
22   }
23
24   void draw() const;
25   void move(double dx, double dy);
26 };
```

На строке 9 начинается определение конструктора, принимающего 4 значения типа **double**. После списка аргументов идет список инициализации, начинающийся с двоеточия и содержащий поля и выражения, их инициализирующие (строка 11). Поля инициализируются в том порядке, в котором они указаны в определении класса. Если список инициализации содержит поля в другом порядке, то компилятор переставит их так, чтобы соблюдался порядок в определении класса.

В теле конструктора происходит проверка, что ширина и высота имеют положительные значения. В этом примере сообщение об ошибке выводится в стандартный поток ошибок. В главе «Исключения» показан более правильный способ сообщения об ошибках.

Создать несколько прямоугольников при помощи конструктора можно следующим образом:

```
 1 #include "rectangle-with-constructor.hpp"
 2
 3 void handleRectangle(const Rectangle &);
 4
 5 void foo()
 6 {
 7   Rectangle rect1(1, 2, 2.5, 2.5);
 8   Rectangle rect2{ 2.5, 2.5, 10, 15 };
 9   Rectangle rect3 = { 5, -2, 3, 4 };
10
11   handleRectangle(Rectangle{ 5, 16, 2, 3 });
12 }
```

В строке 7 создаваемый объект будет проинициализирован как имеющий левый нижний угол в $(1, 2)$ и стороны по 2.5. Строки 8 и 9 содержат инициализацию в стиле стандарта C++ 11. Эти варианты были добавлены в стандарт чтобы синтаксически конструирование объекта выглядело также как и инициализация структур и массивов.

В строке 11 вызывается функция, которой передается временный объект типа Rectangle. Для инициализации временных объектов может использоваться как синтаксис C++ 11, так и C++ 03 с круглыми скобками.

Некоторые конструкторы имеют специальное название и назначение. Компилятор постарается создать их автоматически.

Первым таким конструктором является конструктор по умолчанию. Это конструктор, который может быть вызван без параметров [C++ 03; C++ 11, параграф 5, раздел 12.1]. Если класс не содержит ни

одного конструктора, конструктор по умолчанию будет создан компилятором автоматически с семантикой инициализации всех полей по умолчанию. Если хотя бы одно поле не имеет конструктора по умолчанию и при этом не является встроенным типом, то конструктор по умолчанию не будет создан автоматически.

Вторым специальным конструктором является конструктор копирования. Конструктор копирования принимает первым аргументом ссылку на объект того же типа. Другие аргументы отсутствуют или имеют значения по умолчанию [C++ 03; C++ 11, параграф 2, раздел 12.8]. Если все поля имеют доступные конструкторы копирования, конструктор копирования будет создан автоматически как функция, выполняющая копирование каждого поля. Конструкторы копирования вызываются автоматически для возвращаемых значений из функций, а также при передаче параметров по значению.

Третьим специальным конструктором является конструктор перемещения, введенный в стандарте C++ 11. Детальная информация о нем будет приведена в главе «Копирование и перемещение».

6.2.2 Деструктор

Часто классы требуют специфических действий при разрушении. Например, если класс содержит идентификатор открытого файла, при разрушении объекта этот идентификатор будет утерян и файл будет закрыт в лучшем случае при завершении программы. Для предотвращения этого специальная функция, называемая деструктором, вызывается при разрушении объекта.

Деструктор объявляется как функция не возвращающая значения, не принимающая аргументов и имеющая имя, совпадающее с именем класса с префиксом «~».

Объединяя все выше сказанное, можно реализовать следующую технику подсчета количества экземпляров прямоугольников в программе:

- статическое целочисленное поле отвечает за хранение количества объектов;
- каждый конструктор увеличивает счетчик;
- деструктор уменьшает счетчик.

```
1  #include <iostream>
2
3  struct Rectangle {
4    double x;
5    double y;
6    double width;
7    double height;
8
9    static int objectCount;
10
11   Rectangle(double left, double bottom,
12       double w, double h):
13     x(left),
14     y(bottom),
15     width(w),
16     height(h)
17   {
18     ++objectCount;
19     if (width <= 0) {
20       std::cerr << "Invalid rectangle width" << std::endl;
21     }
22     if (height <= 0) {
23       std::cerr << "Invalid rectangle height" << std::endl;
24     }
25   }
26
27   Rectangle(const Rectangle & rhs):
28     x(rhs.x),
29     y(rhs.y),
30     width(rhs.width),
31     height(rhs.height)
32   {
33     ++objectCount;
```

```
34      }
35
36      ~Rectangle()
37      {
38          --objectCount;
39      }
40
41      void draw() const;
42      void move(double dx, double dy);
43  };
```

```
1   int Rectangle::objectCount = 0;
2
3   void Rectangle::draw() const
4   {
5       moveTo(x, y);
6       lineTo(x + width, y);
7       lineTo(x + width, y + height);
8       lineTo(x, y + height);
9       lineTo(x, y);
10  }
11
12  void Rectangle::move(double dx, double dy)
13  {
14      x += dx;
15      y += dy;
16  }
```

6.2.3 Операторы присваивания и перемещения

Компилятор также может создать еще 2 специальных функции, которые являются перегрузкой оператора «=».

Первый из них называется оператором присваивания и принимает в качестве аргумента ссылку на объект своего класса.

Стандарт C++ 11 добавил также оператор перемещения, он принимает ссылку на неконстантный временный объект своего класса.

Подробная информация об этих операторах присутствует в главе «Копирование и перемещение».

6.3 Видимость и доступность

В стандарте четко разделяются понятия видимости и доступности. Все поля и методы полностью определенного класса являются видимыми. В то же время, полезно часть информации скрыть от клиентов. Для этого используется управление доступом.

Для управления доступом используются специальные ключевые слова, формирующие секции внутри определения класса. Таких секций может быть сколько угодно, но они могут быть только 3-х типов:

public открытые поля и методы, доступные всем;

private закрытая часть класса, доступная только самому классу;

protected защищенная часть, доступная самому классу и его наследникам.

Управление доступом используется для сокрытия деталей реализации класса и защиты от некорректных изменений данных. Например, в классе прямоугольника никто не мешает после создания изменить ширину и/или высоту на отрицательные значения. Для предотвращения этого все поля можно поместить в закрытую часть класса, а методы, предназначенные для работы с ними — в открытую. Обычно, по соглашениям, первой в определении класса идет открытая часть, потом защищенная и завершает определение класса закрытая часть. Например, для прямоугольника может быть использовано следующее определение:

```
#include <iostream>

class Rectangle {
public:
  Rectangle(double left, double bottom,
      double w, double h);

  double getX() const;
  double getY() const;
  double getCenterX() const;
  double getCenterY() const;
  void draw() const;
  void move(double dx, double dy);
  void setWidth(double width);
  void setHeight(double height);

private:
  double x_;
  double y_;
  double width_;
  double height_;
};
```

Методы могут быть реализованы, например, так:

```
Rectangle::Rectangle(double left, double bottom,
    double w, double h):
  x_(left),
  y_(bottom),
  width_(w),
  height_(h)
{
  if (width_ <= 0) {
    std::cerr << "Invalid rectangle width" << std::endl;
  }
  if (height_ <= 0) {
    std::cerr << "Invalid rectangle height" << std::endl;
  }
}

double Rectangle::getX() const
{
  return x_;
}

double Rectangle::getY() const
{
  return y_;
}

double Rectangle::getCenterX() const
{
  return x_ + width_ / 2;
}

double Rectangle::getCenterY() const
{
  return y_ + height_ / 2;
}

void Rectangle::draw() const
```

```
37 {
38   moveTo(x_, y_);
39   lineTo(x_ + width_, y_);
40   lineTo(x_ + width_, y_ + height_);
41   lineTo(x_, y_ + height_);
42   lineTo(x_, y_);
43 }
44
45 void Rectangle::move(double dx, double dy)
46 {
47   x_ += dx;
48   y_ += dy;
49 }
50
51 void Rectangle::setWidth(double width)
52 {
53   if (width <= 0) {
54     std::cerr << "Invalid rectangle width" << std::endl;
55   } else {
56     width_ = width;
57   }
58 }
59
60 void Rectangle::setHeight(double height);
61 {
62   if (height <= 0) {
63     std::cerr << "Invalid rectangle height" << std::endl;
64   } else {
65     height_ = height;
66   }
67 }
```

Такая реализация обладает еще одним достоинством: если в процессе развития системы станет ясно, что в прямоугольнике удобнее хранить не координаты левого нижнего угла, а координаты центра, то будет достаточно поменять реализации методов класса Rectangle. Весь остальной код, не зная деталей реализации, не будет затронут этим изменением.

При определении любого класса есть понятие доступа по умолчанию. Это тот доступ, который действует без указания ключевых слов. Для классов, определенных через **class** доступом по умолчанию является **private**. Для тех, кто определен через **struct** — **public**.

Расписывание примеров кода в ситуации, когда хочется донести какие-то особенности общего дизайна объектно-ориентированной программы, конечно, возможно, но обычно неудобно: долго писать, сложно понимать. Гораздо проще понять схему. Для отображения таких схем существует много нотаций. Самая распространенная система нотаций называется Unified Modelling Language (UML).

Правила отображения классов следующие (рис. 6.1):

- класс отображается прямоугольником, разделенным на 3 части;
- в верхней части указывается имя класса;
- в средней части перечисляются поля;
- в нижней части указываются методы;
- открытые поля и методы имеют префикс «+»;
- закрытые — «-»;
- защищенные — «#».

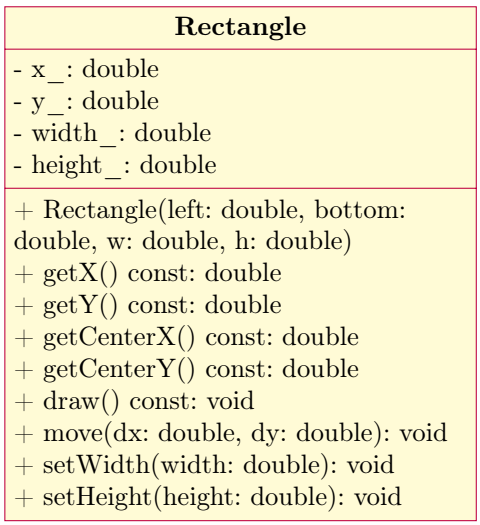


Рис. 6.1: Класс Rectangle в UML

6.4 Наследование

Продолжая пример с прямоугольниками, очевидно, что любая программа, работающая с геометрическими фигурами, неизбежно будет содержать и другие фигуры, например, окружности и треугольники. Но это объекты других типов. В то же время они разделяют с прямоугольниками некоторые общие свойства, например, способность рисовать и некие координаты, указывающие на «начало» фигуры. Поэтому было бы удобно иметь возможность это обобщать.

За подобное обобщение «вид-подвид» в объектно ориентированном программировании отвечает концепция наследования. Класс-наследник содержит все поля и методы базового класса.

В C++ при создании класса его базовые классы указываются после двоеточия, следующего за именем класса. Например, для геометрических фигур возможен такой вариант:

```
 1  class Shape {
 2  public:
 3    double getX() const;
 4    double getY() const;
 5
 6    void move(double dx, double dy);
 7    void draw() const;
 8
 9  protected:
10    double x_;
11    double y_;
12
13    Shape(double x, double y):
14      x_(x),
15      y_(y)
16    {}
17  };
18
19  class Rectangle:
20      public Shape {
21  public:
22    Rectangle(double x, double y,
23        double width, double height):
24      Shape(x, y),
25      width_(width),
26      height_(height)
27    {
28      ...
29    }
```

```
30
31    double getWidth() const;
32    double getHeight() const;
33    void draw() const;
34
35  private:
36    double width_;
37    double height_;
38  };
39
40  class Circle:
41      public Shape {
42  public:
43    Circle(double x, double y,
44        double radius):
45      Shape(x, y),
46      radius_(radius)
47    {
48      ...
49    }
50
51    double getRadius() const;
52    void draw() const;
53
54  private:
55    double radius_;
56  };
```

Конструктор Shape помещен в защищенную секцию потому что он необходим наследникам, но в то же время создавать экземпляры класса Shape нет необходимости. Он должен использоваться только как базовый класс.

Для наследования применимы те же правила доступа, что и для полей и методов.

Наследование в UML отображается стрелкой, направленной от наследника к базовому классу и заканчивающейся пустой треугольной стрелкой (рис. 9.1).

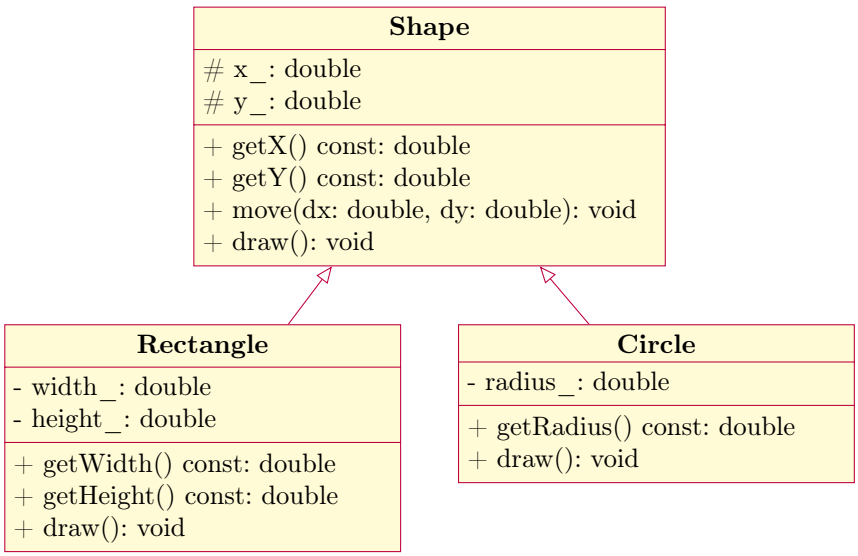


Рис. 6.2: Диаграмма наследования геометрических фигур

Рассмотрим простую программу:

```
1  #include <iostream>
2
3  #include "shapes.hpp"
4
```

```
 5 ...
 6
 7 void Shape::draw() const
 8 {
 9   std::cout << __PRETTY_FUNCTION__ << std::endl;
10 }
11
12 void Rectangle::draw() const
13 {
14   std::cout << __PRETTY_FUNCTION__ << std::endl;
15 }
16
17 void Circle::draw() const
18 {
19   std::cout << __PRETTY_FUNCTION__ << std::endl;
20 }
21
22 void drawShape(const Shape & shape)...
23 {
24   shape.draw();
25 }
26
27 int main(int, char *[])
28 {
29   const Rectangle rect{ 0, 0, 5, 10 };
30   const Circle circle{ 0, 0, 5 };
31
32   rect.draw();
33   circle.draw();
34
35   drawShape(rect);
36   drawShape(circle);
37 }
```

Ее вывод будет несколько неожиданным:

```
$ c++ -std=c++11 -Wall -Wextra -o shapes shapes.cpp
$ ./shapes
void Rectangle::draw() const
void Circle::draw() const
void Shape::draw() const
void Shape::draw() const
```

Это связано с тем, что переменная, определенная на строке 22, имеет статический тип «ссылка на Shape». Компилятор выполняет раннее связывание функции, поэтому вызывается функция Shape::draw() вместо переопределенных в наследниках функций.

6.5 Полиморфизм

Функция drawShape() представляет собой неудачную попытку создать полиморфную функцию: она должна для любого наследника Shape правильно рисовать его, не зная его истинного (или динамического) типа. Для решения этой проблемы необходимо иметь позднее связывание: вместо вызова функции компилятор должен подставить код, вычисляющий динамический тип указателя или ссылки и вызывающий соответствующую функцию. В C++ этот механизм связан с виртуальными функциями. Ключевое слово **virtual** перед объявлением метода заставляет компилятор определять конкретную вызываемую функцию во время исполнения программы. Такой вид полиморфизма называется «полиморфизм подтипов» или «полиморфизм включения».

Для примера с геометрическими фигурами решение будет выглядеть так:

```
1 class Shape {
2 public:
```

```
 3    double getX() const;
 4    double getY() const;
 5
 6    void move(double dx, double dy);
 7    virtual void draw() const;
 8
 9  protected:
10    double x_;
11    double y_;
12
13    Shape(double x, double y):
14      x_(x),
15      y_(y)
16    {}
17  };
18  /*{endShape}*/
19  class Rectangle:
20      public Shape {
21  public:
22    Rectangle(double x, double y,
23        double width, double height):
24      Shape(x, y),
25      width_(width),
26      height_(height)
27    {
28      ...
29    }
30
31    double getWidth() const;
32    double getHeight() const;
33    void draw() const override;
34
35  private:
36    double width_;
37    double height_;
38  };
39
40  class Circle:
41      public Shape {
42  public:
43    Circle(double x, double y,
44        double radius):
45      Shape(x, y),
46      radius_(radius)
47    {
48      ...
49    }
50
51    double getRadius() const;
52    void draw() const override;
53
54  private:
55    double radius_;
56  };
```

На строке 10 метод Shape::draw() объявляется виртуальным. На строках 33 и 52 она переопределяется. Ключевое слово override введено в C++ 11 для возможности явного указания на переопределение виртуальной функции.

Объекты типа Shape смысла не имеют, так как непонятно, что это за фигура. Фактически, Shape является абстрактным классом, который представляет собой некую концепцию фигуры, умеющей себя рисовать. Метод Shape::draw() также не имеет осмысленного содержания, он может быть реализован

только в наследниках. Для того, чтобы подчеркнуть это и заставить наследников его переопределить, C++ предлагает механизм чисто виртуальных функций. Чисто виртуальная функция содержит суффикс `= 0` после ее определения (строка 7):

```
 1  class Shape {
 2  public:
 3    double getX() const;
 4    double getY() const;
 5
 6    void move(double dx, double dy);
 7    virtual void draw() const = 0;
 8
 9  protected:
10    double x_;
11    double y_;
12
13    Shape(double x, double y):
14      x_(x),
15      y_(y)
16    {}
17  };
```

Класс, содержащий хотя бы одну чисто виртуальную функцию является абстрактным, создать объекты такого типа невозможно.

Глава 7

Объектно-ориентированное программирование

7.1 Очень краткий обзор ООП

Одна из самых распространенных парадигм программирования на текущий момент — объектно-ориентированное программирование. Его реализации в разных языках варьируются от необязательных механизмов (Common Lisp), предоставляющих возможность использовать объекты, до комплексов, в которых нет места сущностям необъектного типа (Java).

C++, как язык поддерживающий различные парадигмы программирования, не навязывает применение ООП. В то же время, возможности ООП в C++ очень велики и практически не уступают другим языкам.

Однако, различные принципы ООП требуют крайне осмотрительного применения.

7.2 Агрегация и наследование

ООП базируется на инкапсуляции и наследовании (и полиморфизме, конечно). При проектировании любой программы выбор между этими двумя вариантами зачастую неочевиден для начинающего разработчика. Более того, иногда и опытные разработчики ошибаются в выборе. На самом деле, обычно выбор не сложен, если руководствоваться некоторыми подсказками.

7.2.1 Инкапсуляция

Инкапсуляция подразумевает сокрытие деталей реализации. Например, стандартной практикой является объявление всех полей как закрытых. Это предотвращает несанкционированный доступ к ним. Вместо прямого доступа реализуется набор методов-аксессоров. На первый взгляд, это совершенно излишняя работа. С другой стороны, рассмотрим пример.

```
1  struct add_area;
2
3  class ComplexShape:
4      public Shape
5  {
6  public:
7    std::list< Shape * > elements_;
8    mutable double area_;
9
10   ComplexShape():
11     area_(-1)
12   {}
13
14   double getArea() const
15   {
16     if (area_ < 0) {
17       area_ =  std::accumulate(elements_.begin(), elements_.end(),
18           add_area());
```

```
19      }
20      return area_;
21    }
22  };
23
24  ...
25  ComplexShape shape;
26  ...
27  shape.elements_.clear();
28  shape.getArea();
```

В данном примере присутствует класс, реализующий некоторую композитную (составленную из других) фигуру. Элементы этой фигуры, как и она сама, реализуют интерфейс Shape, в который входит метод Shape::getArea(). Для композитной фигуры площадь определяется как сумма площадей ее элементов. Так как расчет полной площади фигуры требует обхода всех ее элементов, значение площади кэшируется в переменной area_.

Данный дизайн крайне уязвим, так как клиент может модифицировать список элементов, но единожды рссчитанное значение площади не изменится. Во многих языках эта проблема решается на уровне соглашений об использовании и именовании членов объектов. C++ предоставляет более удачный механизм — управление доступностью при помощи ключевых слов **private**, **protected** и **public**. Скрыв все поля в данном классе внутри секции **private**, эта проблема предотвращается.

Возможный вариант исправления:

```
1  struct add_area;
2
3  class ComplexShape:
4      public Shape
5  {
6  public:
7    ComplexShape():
8      area_(-1)
9    {}
10
11   double getArea() const
12   {
13     if (area_ < 0) {
14       area_ = std::accumulate(elements_.begin(), elements_.end(),
15           add_area());
16     }
17     return area_;
18   }
19
20   void add(Shape * shape)
21   {
22     assert(shape != nullptr);
23     elements_.push_back(shape);
24     area_ = -1;
25   }
26
27 private:
28   mutable double area_;
29   std::list< Shape * > elements_;
30 };
31
32 ...
33 ComplexShape shape;
34 ...
35 shape.add(...);
36 shape.getArea();
```

Возможно пойти дальше и пересчитывать значение площади при каждом изменении структуры:

```
1  class ComplexShape:
2      public Shape
3  {
4  public:
5      ComplexShape():
6          area_(0)
7      {}
8
9      double getArea() const
10     {
11         return area_;
12     }
13
14     void add(Shape * shape)
15     {
16       assert(shape != nullptr);
17       elements_.push_back(shape);
18       area_ += shape->getArea();
19     }
20
21 private:
22     double area_;
23     std::list< Shape * > elements_;
24 };
25
26 ...
27 ComplexShape shape;
28 ...
29 shape.add(...);
30 shape.getArea();
```

7.2.2 Наследование

Наследование – это второй принцип ООП. К сожалению, им слишком часто злоупотребляют. Например, на рис. 7.1 у нас есть класс ConfigurationParameters, который предоставляет доступ к некоей конфигурации. Он открыто унаследован от класса FileParser, реализующего логику чтения и разбора файла с конфигурационными параметрами.

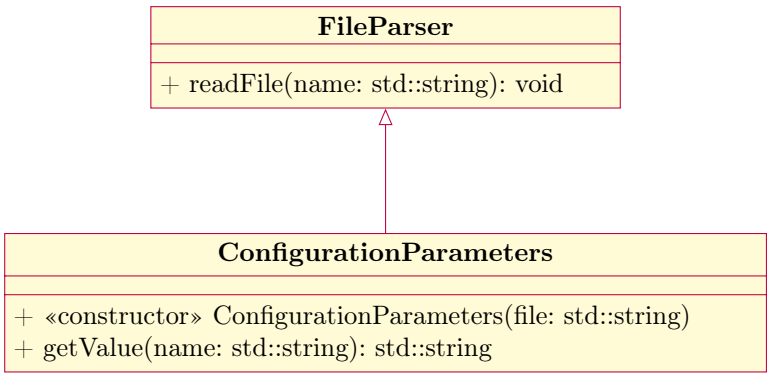


Рис. 7.1: Пример открытого наследования

Данный пример прекрасно демонстрирует неудачный вариант проектирования. С одной стороны, унаследовав контейнер конфигурационных параметров от анализатора файлов, были созданы ограничения на функциональность: параметры могут храниться только в файле только одного определенного формата. Это может как сыграть свою роль в процессе сопровождения системы, так и никогда не проявиться.

С другой стороны, открытое наследование дает клиенту класса ConfigurationParameters возможность узнать, что он использует FileParser в качестве источника информации. Также это дает возможность читать дополнительные файлы, пополняя информацию внутри экземпляра ConfigurationParameters.

Любое средство языка программирования моделирует что-либо. Наследование позволяет моделировать следующие варианты отношений:

открытое, public это отношение вида «является», например, «утка» является «птицей».

закрытое, private отношение «реализовано посредством». Обычно очень мало причин его использовать, потому что есть другие средства для решения проблем.

защищенное, protected это может быть как «реализовано посредством» для внешних классов, так и «является» для самого себя и друзей класса.

Возвращаясь к примеру, контейнер параметров не является анализатором, а всего лишь реализован с его помощью, поэтому тут нет места открытому наследованию. Правильное решение обычно зависит от требований. В общем случае, разумно предположить, что:

- пользователей конфигурационных параметров не волнует, как и откуда они были получены;
- разбор текста некоторого формата с параметрами слабо связан с тем, откуда именно получен этот текст: из файла, по сети и т. п.;
- потенциально, может поддерживаться несколько форматов текста с параметрами;
- прочитанные параметры не должны модифицироваться в процессе их использования.

Из этого следует, что класс ConfigurationParameters не должен быть унаследован от анализатора. Более того, анализатор не должен ничего знать о контейнере, так как вариантов формата может быть более одного, равно как и способов хранения. В качестве варианта, можно предложить дизайн с рис. 7.2.

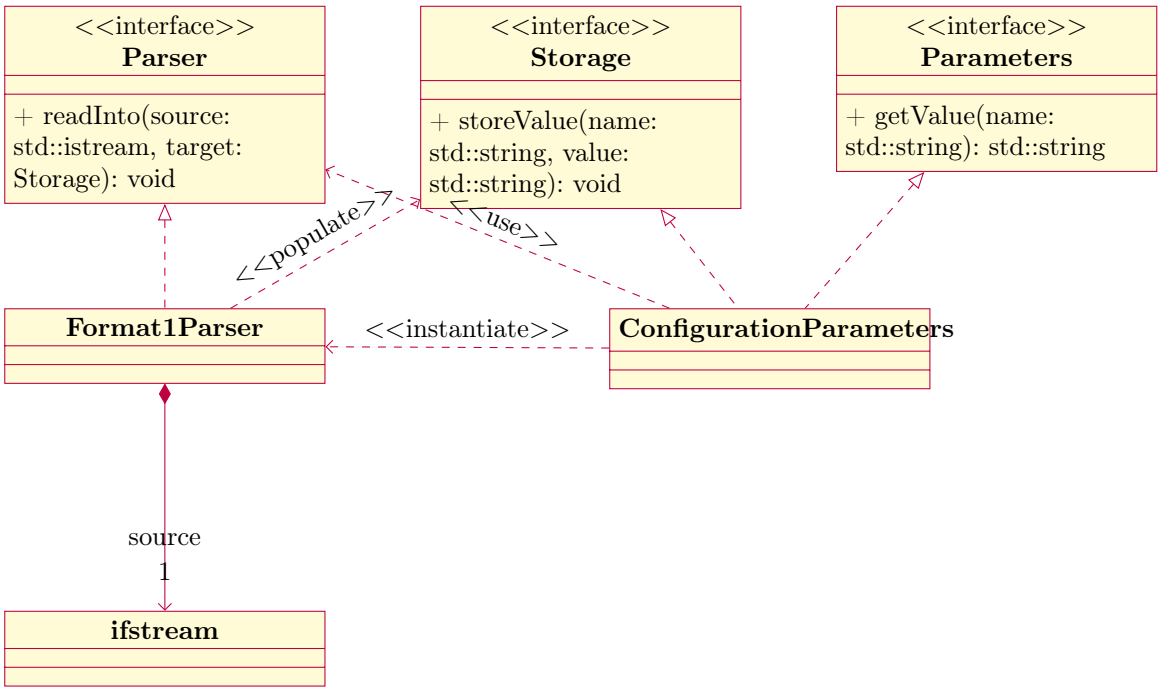


Рис. 7.2: Пример дизайна

Как видно из диаграммы, анализатор файлов разделен на 2 части: интерфейс Parser и его реализация Format1Parser. Это позволяет создать несколько реализаций анализатора для поддержки нескольких форматов. В то же время, Format1Parser использует ifstream в качестве источника данных для разбора.

Класс ConfigurationParameters реализует 2 интерфейса: Storage и Parameters. Интерфейс Storage представляет собой некое абстрактное хранилище параметров и используется анализаторами для сохранения результатов разбора.

Интерфейс Parameters предназначен для доступа к сохраненным параметрам.

В этой диаграмме самый тонкий момент в том, что наследование интерфейса Parameters открытое, так как он должен быть доступен для всех клиентов ConfigurationParameters, в то время как Storage используется только на этапе создания объекта самим объектом и потому имеет закрытое наследование.

В данном примере присутствует еще одна техника ООП — композиция. Экземпляр ifstream полностью принадлежит экземпляру Format1Parser. Строго говоря, это еще один вариант отношения «реализован посредством». Таким образом, существует 2 способа использовать какую-либо стороннюю функциональность в своем классе: закрытое наследование и композиция. Каким образом можно выбрать между ними?

Для правильного выбора между наследованием и композицией необходимо сформировать критерий выбора. Простейшим будет являться степень зависимости между 2-мя классами: чем теснее связь, тем хуже.

Наследование создает самую тесную связь между 2-мя классами: класс-наследник имеет доступ ко всем открытым и защищенным члена базового класса, причем доступ к ним не требует никаких дополнительных действий, потому что все доступные члены базового класса помещены в область видимости внутри всех методов наследника. Как следствие, малейшие изменения в базовом классе влекут за собой потенциальные изменения в наследнике.

В противоположность этому, композиция создает менее тесную связь: имеется доступ только к открытым членам базового класса, более того, доступ к ним требует обязательной квалификации их именем поля класса-наследника.

Их этого следует, что всегда предпочтательнее композиция чем наследование, за исключением тех случаев, когда это невозможно.

7.3 Полиморфизм

Третьим «китом» ООП является полиморфизм. Обычно полиморфизм в C++ рассматривается только как часть ООП, что, в общем-то, неверно. Помимо виртуальных функций, в C++ доступны и другие варианты.

7.3.1 Полиморфизм подтипов (включения)

В языке C++ за данный вид полиморфизма отвечают виртуальные функции и наследование: класс-наследник имеет возможность переопределить реализацию виртуальной функции из базового класса.

Реализация данного вида полиморфизма в C++ содержит 2 важных концепции:

позднее связывание вызываемая реализация виртуальной функции определяется во время выполнения программы по динамическому типу объекта, а не во время компиляции;

одиночная диспетчеризация вызываемая реализация определяется по первому аргументу (объекту), другие параметры функции не рассматриваются.

В качестве примера можно привести расчет площади геометрических фигур:

```
 1  class Shape
 2  {
 3  public:
 4    virtual ~Shape();
 5
 6    virtual double getArea() const = 0;
 7  };
 8
 9  class Circle: public Shape
10  {
11  public:
12    double getArea() const
13    {
14      ...
15    }
16  };
17
18  class Rectangle: public Shape
19  {
20  public:
21    double getArea() const
22    {
23      ...
24    }
```

```
25 };
26
27 void printArea(const Shape & shape)
28 {
29   std::cout << shape.getArea() << "\n";
30 }
31
32 ...
33 printArea(Circle(...));
34
35 printArea(Rectangle(...));
36 ...
```

Функция printArea() правильно вызывает реализацию метода Shape::getArea() в зависимости от того, каков истинный тип переданного объекта.

Позднее связывание работает здесь потому, что передается ссылка на объект. Также оно работает для указателей. Если же функция printArea() будет принимать объект по значению, то код либо не скомпилируется (как в данном случае, так как Shape — абстрактный класс), либо произойдет срезка объекта и вызовется неверная функция.

7.3.2 Ad-hoc полиморфизм

Ad-hoc или ситуативный полиморфизм возникает, когда доступно более одной реализации функции для различных комбинаций типов параметров. В отличие от полиморфизма подтипов, здесь может отсутствовать наследование и выбор реализации осуществляется компилятором.

Язык C++ реализует ad-hoc полиморфизм посредством перегрузки функций и операторов, например:

```
1  class A {
2    ...
3  };
4
5  class B {
6    ...
7  };
8
9  std::ostream & operator <<(std::ostream & out, const A & a)
10 {
11   ...
12   return out;
13 }
14
15 std::ostream & operator <<(std::ostream & out, const B & b)
16 {
17   ...
18   return out;
19 }
20
21 ...
22 std::cout << A() << "\n";
23
24 std::cout << B() << "\n";
25 ...
```

В данном примере осуществлена перегрузка операторов сдвига для двух классов в соответствии с требованиями библиотеки потоков ввода-вывода C++. Компилятор правильно выберет вызываемый оператор в зависимости от статического типа аргумента.

Необходимо отметить, что данный вид полиморфизма также задействован в примере с функцией max(), так как **operator** <() вызывается для сравнения объектов и должен быть перегружен для невстроенных в компилятор типов.

7.3.3 Параметрический полиморфизм

Упрощенно, параметрический полиморфизм возникает в языках программирования тогда, когда данные различных типов обрабатываются одинаково. Этот вариант полиморфизма будет рассмотрен в главе «Шаблоны и обобщенное программирование».

7.4 Идиомы

Идиомы играют важную роль в языках программирования. Правильное применение идиом позволяет решать достаточно общие проблемы.

Часть идиом призвана решить или облегчить какие-либо неудобные аспекты языка программирования. Например, RAII позволяет упростить и автоматизировать управление ресурсами в языках с детерминированным вызовом деструктора.

Другая часть идиом представляет собой обычный способ реализации паттернов проектирования. В данном разделе будут рассмотрены по одной идиоме каждого вида.

7.4.1 Указатель на реализацию

Одной из особенностей языка C++ является то, что внутреннее содержание класса, в том числе и закрытое, присутствует внутри определения класса. В некоторых случаях это вызывает некоторые трудности, так как детали реализации раскрываются:

- Время компиляции растет, так как все зависимости вынуждены быть включенными в заголовочный файл с классом. Для всех зависимостей их личные зависимости также включены в их заголовки и так далее.
- Для зависящих от платформы классов, например, для примитивов синхронизации, в заголовке возникают множественные участки условной компиляции, что тоже не облегчает понимание кода.
- Внутренние вспомогательные классы обретают не только внешнее связывание, но и вполне осмысленные имена символов, что может привести к коллизиям.
- Требования к изменениям, вносимым в класс и сохраняющим бинарную совместимость, резко растут.

В C++ существует классический способ решить данные проблемы путем некоторого падения производительности кода — идиома «Указатель на реализацию» (Pointer to Implementation, PIMPL).

Основная идея довольно проста и продемонстрирована на рис. 7.3.

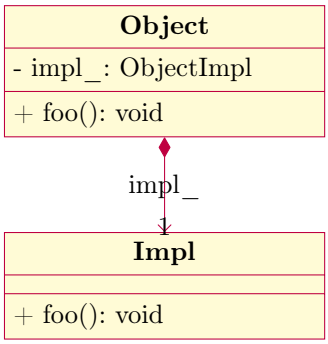


Рис. 7.3: Указатель на реализацию

В данном примере объект типа Object реализован посредством внутреннего класса Impl. Для этого он хранит единственное поле impl_, которое является указателем на реализацию.

Код на C++ может выглядеть так:

```
1 #ifndef OBJECT_HPP
2 #define OBJECT_HPP
3
4 class Object
5 {
6 public:
7     Object();
```

```
 8    Object(const Object &);
 9
10    ~Object();
11
12    Object & operator =(const Object &);
13
14    void foo();
15
16  private:
17    class Impl;
18
19    std::auto_ptr< Impl > impl_;
20  };
21
22  #endif OBJECT_HPP
```

```
 1  #include "Object.hpp"
 2
 3  class Object::Impl
 4  {
 5  public:
 6    void foo();
 7  };
 8
 9  Object::Object():
10    impl_(new Impl)
11  {}
12
13  Object::Object(const Object & rhs):
14    impl_(new Impl(*rhs.impl_))
15  {}
16
17  Object::~Object()
18  {}
19
20  Object & Object::operator =(const Object & rhs)
21  {
22    if (this == &rhs) {
23      return *this;
24    }
25
26    impl_ = std::auto_ptr< Impl >(new ObjectImpl(*rhs.impl_));
27
28    return *this;
29  }
30
31  void Object::foo()
32  {
33    impl_->foo();
34  }
```

В данном простом примере тоже есть тонкие моменты:

- Класс-реализация представляет собой предварительно задекларированный в строке 17 вложенный класс. Это позволяет сделать недоступным для клиентов имя реализации. Данная техника неприменима, если имеется более одной реализации и выбор осуществляется во время исполнения.

- В строке 10 декларируется и на строке 17 реализуется тривиальный деструктор для класса Object. Обычно не требуется реализовывать тривиальный деструктор, так как компилятор генерирует его автоматически. В данном случае это не работает, так как в точке его определения компилятором (а это будет точка разрушения объекта типа Object) отсутствует определение вложенного класса

Object::Impl. В то же время, определив этот деструктор отдельно после определения вложенного класса, мы тем самым сделали в точке удаления impl_ доступным описание вложенного класса.

- Сделать объект, реализованный через PIMPL, копируемым невозможно автоматически. Поэтому в строке 13 можно видеть простой конструктор копирования, который копирует объект реализацию.
- Стандартный оператор присваивания тоже не подходит, поэтому в строке 20 присутствует своя реализация. Важным моментом является проверка на присваивание самому себе в строке 22. В данном случае это просто оптимизация, в других же случаях возможен вариант разрушения оригинального объекта в процессе присваивания самому себе.
- Содержимое функции Object::foo() предельно просто — это просто перенаправление вызова в объект-реализацию.

7.4.2 Невиртуальный интерфейс

В процессе разработки встречаются случаи, когда какой-либо алгоритм достаточно универсален для применения к различным типам объектов, но при этом в нем должны присутствовать части, зависящие от конкретного типа объектов. Например, в графической программе вывод изображения в общем и целом не зависит от конкретного устройства вывода. Но каждому типу устройства вывода требуется своя инициализация и очистка. Это может быть решено 2-я способами: через обобщенное программирование (и параметрический полиморфизм) и через ООП (с применением полиморфизма подтипов). Вариант через ООП представлен на рис. 7.4 и рис. 7.5.

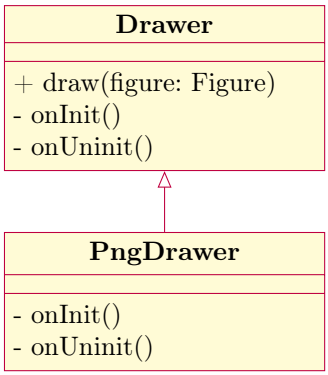


Рис. 7.4: Шаблонный метод: классы

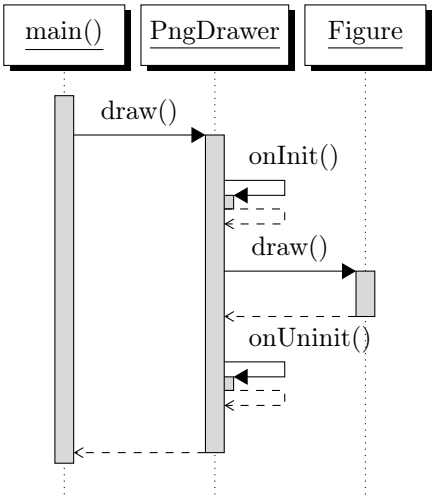


Рис. 7.5: Шаблонный метод: вызовы

Данный случай описывается паттерном проектирования «Шаблонный метод» [Гам+13]. Классической реализацией этого паттерна в C++ является идиома «Невиртуальный интерфейс»: метод реализуется как невиртуальный. В точках адаптации поведения он вызывает виртуальные функции, тем самым предоставляя наследникам возможность модифицировать поведение.

Для приведенного выше примера, реализация может выглядеть так (для простоты, реализация метода Drawer::draw() помещена в определение класса):

```
1  class Drawer
2  {
3  public:
4    void draw(const Shape & shape)
5    {
6      onInit();
7
8      shape.draw();
9
10     onUninit();
11   }
12
13 private:
14   virtual void onInit() = 0;
15   virtual void onUninit() = 0;
16 };
```

Этот код содержит только один тонкий момент: виртуальные функции Drawer::onInit() и Drawer::onUninit находятся в закрытой части определения класса. Тем не менее, в соответствии со сноской 97 к параграфу 10.3.2 [C++ 03], это не является препятствием для их переопределения в наследнике. В новом стандарте это сноска 111 к тому же параграфу [C++ 11].

Глава 8

Копирование и перемещение

8.1 Обшие понятия

Компилятор C++ часто вынужден создавать копии различный значений в процессе вызова функций. Например:

```
1  void baz(std::string);
2
3  void foo()
4  {
5    std::string bar();
6
7    std::string s = bar();
8
9    baz(s);
10   baz("Literal String");
11 }
```

копии создаются в следующих случаях:

- передача параметра по значению (строка 9);
- возврат результата по значению (строка 7).

В этих случаях для объектов вызывается конструктор копирования или оператор присваивания. Необходимо отметить, что в строке 10 нет копирования, так как там будет конструироваться экземпляр std :: string из строкового литерала.

8.2 Копирование

Копирование — это единственный способ создать новый экземпляр значения на основе другого в стандарте C++ 03.

8.2.1 Конструктор копирования

Конструктор копирования —это третий специальный метод класса после конструктора по умолчанию и деструктора. В соответствии с параграфом 2 раздела 12.8 [C++ 03; C++ 11], конструктором копирования является конструктор, принимающий первым аргументом ссылку на тот же тип. Этот аргумент должен быть единственным обязательным аргументом контруктора, другие аргументы, если они присутствуют, должны иметь значения по умолчанию. Таким образом, следующие варианты являются конутрукторами копирования для класса X:

- X(**const** X &)
- X(X &)
- X(**volatile** X &)
- X(**const volatile** X &)

- X(**const** X &, **int** depth = −1)

Распространенным заблуждением является утверждение, что конструктором копирования будет только первый вариант. Компилятор может использовать и все остальные версии в качестве конструктора копирования, если они подходят в точке вызова. Допустимо определить более одного варианта конструктора копирования, компилятор выполнит выбор подходящего в соответствии с правилами перегрузки.

Конструктор копирования, если он не объявлен в классе, может быть создан компилятором. Созданный компилятором конструктор копирования для класса X будет обладать следующими свойствами:

- его прототип X(**const** X &) или X(X &) (параграф 5 [C++ 03] или параграф 9 [C++ 11] раздела 12.8);
- он располагается в открытой части класса;
- имеет неявный спецификатор **inline**;
- вызывает конструкторы копирования всех базовых классов;
- выполняет побитовое копирование всех полей встроенных типов;
- вызывает конструкторы копирования для каждого поля пользовательского типа.

В случае, если конструкторы копирования для каких-либо полей или базовых классов недоступны, конструктор копирования автоматически создаваться не будет. Также автоматическое создание конструктора копирования не происходит, если в классе не объявлено конструктора перемещения и оператора перемещающего присваивания.

8.2.2 Оператор копирующего присваивания

Оператор копирующего присваивания (copy assignment) вызывается каждый раз, когда выполняется присваивание значений одного типа, например, при возврате значений из функций. В соответствии с параграфом 9 [C++ 03] или параграфом 18 [C++ 11] раздела 12.9, оператор копирующего присваивания должен иметь следующий вид (как обычно, возможно определение более чем одной перегрузки одновременно):

- **operator** =(**const** X &)
- **operator** =(X &)
- **operator** =(**volatile** X &)
- **operator** =(**const volatile** X &)

Как и для конструктора копирования, возможна реализация более одной перегруженной версии.

Аналогично конструктору копирования, оператор копирующего присваивания будет создан компилятором автоматически, если в классе не объявлено конструктора перемещения или оператора перемещающего присваивания. Его поведение аналогично конструктору копирования, а сигнатурой будет X & X::**operator**=(**const** X
или X& X::**operator**=(X &).

8.3 Перемещение

Множественные копирования данных приводят к снижению производительности. В связи с этим стандарт C++ 11 вводит понятие перемещения. Так как безопасным является только перемещение данных из временных объектов, для реализации семантики перемещения используются ссылки на r-value.

8.3.1 Перемещающий конструктор

Перемещающий конструктор выполняет конструирование экземпляра класса путем перемещения данных из временного объекта того же типа. Поэтому он должен принимать ссылку на временное значение того же типа [C++ 11, параграф 3 раздела 12.8]. Другие аргументы, если они присутствуют, должны иметь значения по умолчанию. Таким образом, следующие конструкторы могут быть использованы для перемещения объектов типа X:

- X(X &&)
- X(**const** X &&)

- X(**volatile** X &&)
- X(**const volatile** X &&)
- X(X &&, **int** depth = −1)

Как и для копирования, компилятор создаст конструктор перемещения, если он потребуется и в классе не объявлены:

- конструктор копирования;
- оператор копирующего присваивания;
- оператор перемещающего присваивания;
- деструктор.

Его сигнатура будет соответствовать X::X(X&&) [C++ 11, параграф 11 раздела 12.8], а тело будет:

- вызывать конструкторы перемещения всех базовых классов;
- выполнять побитовое перемещение всех полей встроенных типов;
- вызывать конструкторы перемещения для каждого поля пользовательского типа.

8.3.2 Оператор перемещающего присваивания

Аналогично копированию, для перемещения используется оператор перемещающего присваивания. Так же как и конструктор перемещения, он принимает ссылку на временный объект того же типа [C++ 11, параграф 20 раздела 12.8]:

- **operator** =(X &&)
- **operator** =(**const** X &&)
- **operator** =(**volatile** X &&)
- **operator** =(**const volatile** X &&)

Компилятор также создаст версию оператора перемещающего присваивания, если он потребуется и в классе не объявлены:

- конструктор копирования;
- конструктор перемещения;
- оператор копирующего присваивания;
- деструктор.

8.4 Пример реализации

Для примера, рассмотрим обертку над файловым POSIX API. Для работы с файлами стандарт POSIX предлагает следующие функции:

- **int** open(**const char** ∗ name, **int** flags, **int** access)

  Открывает файл по его имени с требуемым доступом на чтение и/или запись, переданным в параметре flags. При необходимости, файл может быть создан, в этом случае права доступа к файлу будут установлены в соответствии с параметром access.

  Функция возвращает файловый дескриптор, используемый для работы с файлом до его закрытия.

- **int** close(**int** fd)

  Закрывает файловый дескриптор, открытый ранее при помощи open().

- **int** read(**int** fd, **void** ∗ buffer, size_t size)

  Читает до size байтов из файла с дескриптором fd в буфер buffer и возвращает количество прочитанных байт.

- **int** write(**int** fd, **const void** * buffer, size_t size)

  Записывает в файл с декстриптором fd size байтов из буфера buffer. Возращает количество успешно записанных байт.

- **int** dup(**int** fd)

  Дублирует файловый дескриптор.

Все эти функции возвращают отрицательное значение в случае ошибки. Код ошибки находится в глобальной переменной errno.

Интерфейс обертки над таким API может выглядеть так:

```
 1 #ifndef POSIX_FILE_HPP
 2 #define POSIX_FILE_HPP
 3
 4 #include <string>
 5
 6 class PosixFile
 7 {
 8 public:
 9   enum access_t {
10     Read,
11     Write,
12     ReadWrite
13   };
14   PosixFile(const std::string & name, access_t access);
15   ~PosixFile();
16
17   int read(void * buffer, size_t size);
18   int write(const void * buffer, size_t size);
19
20 private:
21   int fd_;
22 };
23
24 #endif
```

А реализация, соответственно, так:

```
 1 #include "posix_file.hpp"
 2
 3 #include <sys/types.h>
 4 #include <sys/stat.h>
 5 #include <fcntl.h>
 6 #include <unistd.h>
 7
 8 namespace {
 9   inline int mapAccess(PosixFile::access_t access)
10   {
11     switch (access) {
12     case PosixFile::Read:
13       return O_RDONLY;
14
15     case PosixFile::Write:
16       return O_WRONLY;
17
18     case PosixFile::ReadWrite:
19       return O_RDWR;
20     }
21   }
22 }
23
24 PosixFile::PosixFile(const std::string & name, PosixFile::access_t access):
```

```
25    fd_(open(name.c_str(), mapAccess(access), S_IRUSR | S_IWUSR))
26  {
27    if (fd_ < 0) {
28      // TODO: Report an error
29    }
30  }
31
32  PosixFile::~PosixFile()
33  {
34    if (fd_ >= 0) {
35      close(fd_);
36    }
37  }
38
39  int PosixFile::read(void * buffer, size_t size)
40  {
41    return ::read(fd_, buffer, size);
42  }
43
44  int PosixFile::write(const void * buffer, size_t size)
45  {
46    return ::write(fd_, buffer, size);
47  }
```

Очевидно, что для класса PosixFile компилятор автоматически создаст конструктор копирования при следующем сценарии использования (C++ 03):

```
1  #include "posix_file.hpp"
2
3  PosixFile openLog()
4  {
5    return PosixFile("program.log", PosixFile::Write);
6  }
7
8  void foo()
9  {
10   PosixFile log = openLog();
11
12   const char MESSAGE[] = "Log␣Message\n";
13
14   log.write(MESSAGE, sizeof(MESSAGE) - 1);
15 }
```

В этом случае возможен один из двух вариантов поведения в строке 14:

1. попытка записи в закрытый файл;
2. запись в другой файл, открытый для других целей.

Это связано с тем, что в строке 10 будет вызван конструктор копирования для переменной log и деструктор для временного объекта, созданного внутри функции openLog(). Если какой-либо код, выполненный между строками 10 и 14 откроет новый файл и он получит тот же файловый дескриптор, что файл протокола, то реализуется второй сценарий. В противном случае будет реализован первый сценарий.

Очевидно, что автоматически создаваемые конструктор копирования и оператор копирующего присваивания не могут работать правильно. Для исправления необходимо написать их самостоятельно:

```
1  #ifndef POSIX_FILE_HPP
2  #define POSIX_FILE_HPP
3
4  #include <string>
5
6  class PosixFile
7  {
```

```
 8 public:
 9   enum access_t {
10     Read,
11     Write,
12     ReadWrite
13   };
14   PosixFile(const std::string & name, access_t access);
15   PosixFile(const PosixFile & rhs);
16   ~PosixFile();
17
18   PosixFile & operator =(const PosixFile & rhs);
19
20   int read(void * buffer, size_t size);
21   int write(const void * buffer, size_t size);
22
23 private:
24   int fd_;
25 };
26
27 #endif
```

```
 1 PosixFile::PosixFile(const PosixFile & rhs):
 2   fd_(dup(rhs.fd_))
 3 {
 4   if (fd_ < 0) {
 5     // TODO: Report an error
 6   }
 7 }
 8
 9 PosixFile & PosixFile::operator =(const PosixFile & rhs)
10 {
11   int new_fd = dup(rhs.fd_);
12
13   if (new_fd >= 0) {
14     if (fd_ >= 0) {
15       close(fd_);
16     }
17     fd_ = new_fd;
18   } else {
19     // TODO: Report an error
20   }
21
22   return *this;
23 }
```

Таким образом, копирование объектов будет выполняться правильно. Но с точки зрения производительности это неэффективно, так как dup() представляет собой системный вызов, который дорого обходится по времени исполнения. Поэтому реализация семантики перемещения может выглядеть так:

```
 1 #ifndef POSIX_FILE_HPP
 2 #define POSIX_FILE_HPP
 3
 4 #include <string>
 5
 6 class PosixFile
 7 {
 8 public:
 9   enum access_t {
10     Read,
11     Write,
12     ReadWrite
```

```
13    };
14    PosixFile(const std::string & name, access_t access);
15    PosixFile(const PosixFile & rhs);
16    PosixFile(PosixFile && rhs);
17    ~PosixFile();
18
19    PosixFile & operator =(const PosixFile & rhs);
20    PosixFile & operator =(PosixFile && rhs);
21
22    int read(void * buffer, size_t size);
23    int write(const void * buffer, size_t size);
24
25  private:
26    int fd_;
27  };
28
29  #endif
```

```
1  PosixFile::PosixFile(PosixFile && rhs):
2    fd_(rhs.fd_)
3  {
4    rhs.fd_ = -1;
5  }
6
7  PosixFile & PosixFile::operator =(PosixFile && rhs)
8  {
9    if (fd_ >= 0) {
10     close(fd_);
11   }
12
13   fd_ = rhs.fd_;
14   rhs.fd_ = -1;
15
16   return *this;
17 }
```

Автоматически созданные конструктор перемещения и оператор перемещающего присваивания для PosixFile не работают, так как необходимо предотвратить закрытие файла временным объектом при его разрушении. В примере за это отвечает присваивание -1 полю дескриптора в классе-источнике.

Тонкости, связанные с перемещением объектов, на этом не заканчиваются. Если вместо обертки над POSIX API попытаться сделать объект, осуществляющий протоколирование, то класс PosixFile будет представлять собой деталь реализации:

```
1  #include <iostream>
2
3  class PosixFile
4  {
5  public:
6    PosixFile();
7    PosixFile(const PosixFile &);
8    PosixFile(PosixFile &&);
9    ~PosixFile();
10 };
11 class PosixLog
12 {
13 public:
14   PosixLog();
15   PosixLog(const PosixLog & rhs):
16     file_(rhs.file_)
17   {}
18   PosixLog(PosixLog && rhs):
```

```
19      file_(rhs.file_)
20    {}
21    ~PosixLog();
22
23  private:
24    PosixFile file_;
25  };
26  PosixLog openLog()
27  {
28    return PosixLog();
29  }
30  int main(int, char *[])
31  {
32    PosixLog log = openLog();
33
34    return 0;
35  }
```

Предполагая, что каждый служебный метод выводит свое название при исполнении, вывод программы будет следующим:

```
$ ./posix_log
PosixFile::PosixFile()
PosixLog::PosixLog()
PosixFile::PosixFile(const PosixFile &)
PosixLog::PosixLog(PosixLog &&)
PosixLog::~PosixLog()
PosixFile::~PosixFile()
PosixFile::PosixFile(const PosixFile &)
PosixLog::PosixLog(PosixLog &&)
PosixLog::~PosixLog()
PosixFile::~PosixFile()
PosixLog::~PosixLog()
PosixFile::~PosixFile()
```

На первый взгляд может показаться удивительным, но конструктор перемещения для PosixFile *не* вызывается из контруктора перемещения PosixLog. Причина этого в том, что ссылка на PosixLog внутри конструктора перемещения является ссылкой на временный объект, в то время как поля внутри этого временного объекта не являются временными объектами сами по себе, поэтому ссылки на них являются ссылками на l-value. Для того, чтобы перемещение полей заработало, необходимо превратить их в ссылки на r-value. Для этого можно применить **static_cast**, но гораздо короче и понятнее будет применить станлартную функцию move():

```
1    PosixLog(PosixLog && rhs):
2      file_(std::move(rhs.file_))
3    {
4    }
```

8.5 Правило 3-х

Правило 3-х гласит:

> Если класс определяет хотя бы один из следующих методов, он должен определить их все:
>
> - деструктор
> - конструктор копирования
> - оператор копирующего присваивания

Появление этого правила связано с тем, что по стандарту C++ 03 компилятор услужливо создает недостающие служебные методы. Но логика программы обычно требует написания, например, деструктора в

тех случаях, когда автоматически созданный не подходит, например, потому, что в классе есть указатели на объекты, требующие удаления. В этом случае конструктор копирования и оператор копирующего присваивания также будут работать неверно и требуют самостоятельного написания.

В мире C++ 11 ситуация частично исправлена, так как перемещающие методы не создаются автоматически в подобных ситуациях, но в части копирования ничего не изменилось, за исключением того, что автоматически созданные конструктор копирования и оператор копирующего присваивания объявлены устаревшими, если в класс объявлены деструктор, конструктор копирования (для оператора) или оператор копирующего присваивания (для конструктора).

Тем не менее, бывают ситуации, когда компилятор не будет создавать служебные функции, но они нужны и автоматически созданная версия работает корректно. В этой ситуации в C++ 11 можно заставить компилятор создать эти методы, воспользовавшись ключевым словом **default**:

```
1  class PosixFile
2  {
3  public:
4          enum access_t {
5                  Read,
6                  Write,
7                  ReadWrite
8          };
9          PosixFile(const std::string & name, access_t access);
10         PosixFile(PosixFile &&) = default;
11         PosixFile & operator =(PosixFile &&) = default;
12         ~PosixFile();
13
14         int read(void * buffer, size_t size);
15         int write(const void * buffer, size_t size);
16
17 private:
18         int fd_;
19
20         PosixFile(const PosixFile &) = delete;
21         PosixFile & operator =(const PosixFile &) = delete;
22 };
```

8.6 Предотвращение копирования и перемещения

Не всякий объект допускает копирование и перемещение. Например, объект, владеющий примитивами синхронизации потоков исполнения, не может быть скопирован или перемещен, потому что это нарушает синхронизацию потоков. Для таких объектов эти операции должны быть запрещены.

Существует 2 способа выполнения этой задачи. В C++ 03 необходимо воспользоваться тем, что конструктор копирования и оператор присваивания не создаются автоматически, если они объявлены в классе. Стандартная идиома для этого: объявить их в закрытой части класса и не определять:

```
1  #ifndef POSIX_FILE_HPP
2  #define POSIX_FILE_HPP
3
4  #include <string>
5
6  class PosixFile
7  {
8  public:
9    enum access_t {
10     Read,
11     Write,
12     ReadWrite
13   };
14   PosixFile(const std::string & name, access_t access);
15   ~PosixFile();
16
```

```
17    int read(void * buffer, size_t size);
18    int write(const void * buffer, size_t size);
19
20  private:
21    int fd_;
22
23    PosixFile(const PosixFile &);
24    PosixFile & operator =(const PosixFile &);
25  };
26
27  #endif
```

Внешние клиенты класса не смогут копировать его, так как копирующие операции недоступны. Сам класс не сможет создавать своих копий, так как компоновщик сообщит о неопределенных символах.

В C++ 11 имеет смысл использовать ключевое слово **delete**:

```
1   #ifndef POSIX_FILE_HPP
2   #define POSIX_FILE_HPP
3
4   #include <string>
5
6   class PosixFile
7   {
8   public:
9     enum access_t {
10      Read,
11      Write,
12      ReadWrite
13    };
14    PosixFile(const std::string & name, access_t access);
15    ~PosixFile();
16
17    int read(void * buffer, size_t size);
18    int write(const void * buffer, size_t size);
19
20  private:
21    int fd_;
22
23    PosixFile(const PosixFile &) = delete;
24    PosixFile(PosixFile &&) = delete;
25    PosixFile & operator =(const PosixFile &) = delete;
26    PosixFile & operator =(PosixFile &&) = delete;
27  };
28
29  #endif
```

Если проект использует библиотеку [BOOST], то лучше воспользоваться переносимым способом: закрыто унаследоваться от класса boost::noncopyable. Это также приведет к тому, что операции копирования и перемещения не будут созданы, так как они недоступны для одного из базовых классов.

8.7 Допустимые оптимизации

Стандарт языка C++ не описывает набор допустимых оптимизаций. Любые оптимизации, выполняемые компилятором, не должны менять ожидаемое поведение программы. Но в одном случае стандарт делает неочевидное допущение. Компилятор имеет право устранять вызовы конструкторов копирования и перемещения вместе с соответствующими им вызовами деструкторов в ограниченном наборе случаев, когда источником является временный объект:

- при возврате из функции (оператор **return**);
- при генерации исключения (**throw**);

- при обработке исключения (**catch**).

Эта операция называется «copy/move elision» и описывается в разделе 12.8 стандарта.
Например:

```
1  #include <iostream>
2
3  class PosixFile
4  {
5  public:
6    PosixFile();
7    PosixFile(const PosixFile &);
8    PosixFile(PosixFile &&);
9    ~PosixFile();
10 };
11 class PosixLog
12 {
13 public:
14   PosixLog();
15   PosixLog(const PosixLog & rhs):
16     file_(rhs.file_)
17   {}
18   PosixLog(PosixLog && rhs):
19     file_(rhs.file_)
20   {}
21   ~PosixLog();
22
23 private:
24   PosixFile file_;
25 };
26 PosixLog openLog()
27 {
28   return PosixLog();
29 }
30 int main(int, char *[])
31 {
32   PosixLog log = openLog();
33
34   return 0;
35 }
```

В строке 28 срабатывает оптимизация, обычно называемая «оптимизация возвращаемого значения» (Return Value Optimization, RVO), что приводит к отсутствию вызова конструктора перемещения и деструктора.

Вывод программы будет выглядеть так:

```
$ ./posix_log
PosixFile::PosixFile()
PosixLog::PosixLog()
PosixLog::~PosixLog()
PosixFile::~PosixFile()
```

Второй вариант оптимизации называется «оптимизация именованного возвращаемого значения» (Named Return Value Optimization, NRVO), и она срабатывает в случаях, когда возвращается значение локальной автоматической переменной.

Глава 9

Динамическая память

9.1 Динамическая память

Создание объектов в виде автоматических переменных крайне удобно ввиду детерминированного вызова деструктора. Но написать программу, работающую с произвольным количеством объектов так невозможно. Для снятия такого ограничения существуют механизмы динамического распределения памяти, взаимодействующие с операционной системой.

9.1.1 Куча

Первым таким механизмом, унаследованным из стандартной библиотеки C является куча. Работа с кучей осуществляется при помощи следующих функций, объявленных в `cstdio` (в C необходимо использовать заголовочный файл `stdio.h`):

- **void** $*$ calloc(size_t count, size_t size);

  Распределяет блок памяти, достаточный для размещения count объектов размером size байтов каждый и возвращает указатель на начало блока. Блок будет заполнен байтами с нулевым значением.

- **void** $*$ malloc(size_t size);

  Распределяет блок памяти в size байтов размером и возвращает указатель на него. Содержимое блока никак не инициализируется.

- **void** free(**void** $*$ ptr);

  Освобождает ранее распределенный блок памяти с адресом ptr.

- **void** $*$ realloc(**void** $*$ ptr, size_t size);

  Универсальная функция распределения памяти. В зависимости от параметров она выполняет следующую работу:

  1. распределяет новый блок памяти размером в size байтов, если параметр ptr равен пустому указателю;
  2. удаляет ранее распределенный блок памяти, если size равен 0;
  3. изменяет размер блока ptr так, чтобы он вмещал size байт данных, старые данные будут перенесены в начало нового блока, если новый блок больше, то конец его никак не инициализируется, возвращенный указатель может быть как новым, так и совпадать с ptr.

Все распределяющие функции возвращают пустой указатель (в C для сравнения следует использовать константу NULL), а глобальная переменная errno устанавливается в значение ENOMEM.

В программах на C++ этот интерфейс считается устаревшим, так как он не взаимодействует с объектной системой C++ и не обладает должной гибкостью. Обычно он применяется для интеграции с библиотеками, написанными на C.

9.1.2 Free Store

Для динамического распределения памяти в C++ используется отдельный механизм, называемый «Free Store». Для взаимодействия с Free Store в языке существуют встроенные операторы для работы с динамической памятью.

За размещение памяти отвечает оператор **new**. Оператор **new** выполняет 2 важных шага:

1. распределяет необходимый объем памяти;
2. вызывает конструктор для создаваемого объекта.

Существует несколько способов применения оператора:

```
1  int * a = new int;
2  int * b = new int(5);
3  int * c = new int();
4  int * d = new int[5];
5  Person * e = new  Person("First Name", "Last Name");
6
7  void * ptr = ...;
8
9  Person * e = new (ptr) Person("First Name", "Last Name");
```

Листинг 9.1: Оператор new

На строке 1 динамически размещается целочисленное значение, которое никак не инициализируется. По указателю, возвращенному оператором, будут находится произвольные данные, оставшиеся после предыдущего использования этого места в памяти. Так как работа с неинициализированной памятью неудобна, то существуют способы принудительно проинициализовать размещенное значение. На строке 2 размещенное целое число инициализируется значением 5, а не строке 3 — значением по умолчанию, что для встроенных типов эквиваленто 0. Размещение отдельных значений встроенных типов применяется крайне редко, гораздо чаще требуется создание массивов. Для этого используется вариант оператора с квадратными скобками (строка 4).

Для создания объекта и передачи конструктору параметров используется обычный синтаксис вызова конструктора, следующие за оператором **new** (строка 5).

В некоторых случаях память для объекта уже распределена каким-либо образом и требуется только вызвать конструктор. Для этого случая существует особый вариант оператора, называемый «размещающим» или «placement». Пример его использования приведен на строке 9. Адрес блока памяти указывается в круглых скобках после ключевого слова **new** и перед типом значения.

В случае, если в системе недостаточно памяти для сигнализирования об ошибке оператор **new** генерирует исключение std::bad_alloc. Подробнее об исключениях в главе «Исключения».

Распределенная память освобождается при помощи оператора **delete**.

```
1  delete a;
2  delete e;
3  delete [] d;
```

Листинг 9.2: Оператор delete

Существует 2 основных формы оператора **delete**:

1. для одиночного объекта (строка 1);
2. для массивов (строка 3).

Эти две формы *не* являются взаимозаменяемыми, так как версия для массивов выполняет вызов нескольких деструкторов для объектов в зависимости от размерности созданного массива. Тем не менее, достаточно распространненой ошибкой является вызов неправильной формы.

Оператор **delete** можно вызывать безопасно для пустых указателей. В этом случае он не выполняет никакой работы.

Несмотря на то, что Free Store по поведению очень похож на кучу, они не являются взаимозаменяемыми. Ни при каких условиях недопустимо освобождать память, полученную через malloc() при помощи оператора **delete**. Хотя многие реализации используют malloc() и free() внутри операторов **new** и **delete**, стандарт этого не гарантирует, так же как он не гарантирует, что Free Store, реализованный посредством кучи, не хранит дополнительную информацию в каждом блоке.

9.2 Полиморфное удаление

Полиморфизм подтипов является очень мощным механизмом, позволяющим создавать обощенные версии алгоритмов, работающие с объектами, реализующими некий фиксированный интерфейс. Например, программа, работающая с геометрическими фигурами может реализовывать классы с рис. 9.1. Любая функция, работающая с интерфейсом Shape может работать с любой его реализацией, не только с Rectangle и Circle, но и любым другим наследником, реализованным позднее и неизвестным на момент компиляции этой функции.

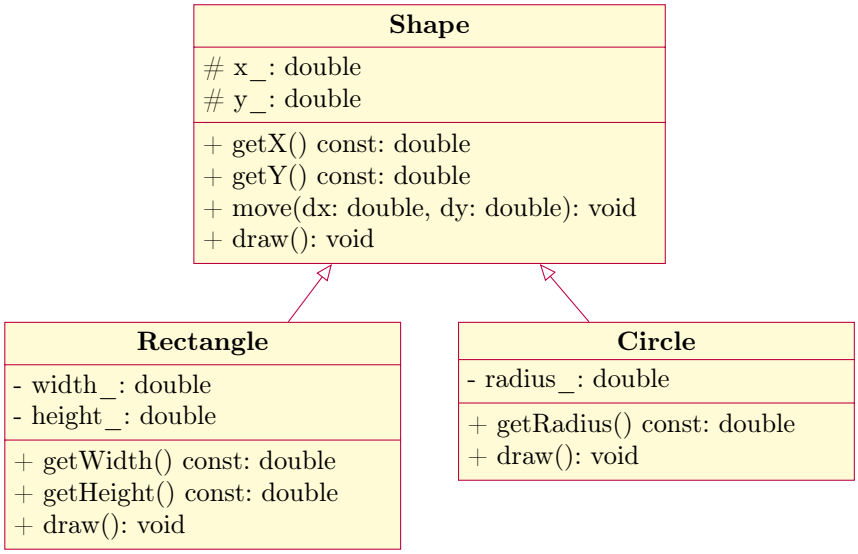


Рис. 9.1: Диаграмма наследования геометрических фигур

Реализация класса Shape приведена на листинге 9.3.

```
1  class Shape {
2  public:
3    double getX() const;
4    double getY() const;
5
6    void move(double dx, double dy);
7    virtual void draw() const;
8
9  protected:
10   double x_;
11   double y_;
12
13   Shape(double x, double y):
14     x_(x),
15     y_(y)
16   {}
17 };
```

Листинг 9.3: Реализация базового класса Shape

Использоваться эти классы могут, например, так:

```
1  Shape * shape = new Rectangle{ 0, 0, 15, 5 };
2
3  ...
4
5  delete shape;
```

Листинг 9.4: Создание и удаление полиморфного класса

На строке 5 объект Rectangle удаляется по указателю на базовый класс Shape. В процессе разрушения оббъекта вызывается его деструктор. Оператор **delete** определяет вызываемый деструктор по статическому типу указателя, в данном случае это деструктор класса Shape. Деструктор же Rectangle никогда не

будет вызван и, если он содержит какую-либо логику по удалению других объектов, он не будет выполнена. С другой стороны, в любой объектно-ориентированной программе чаще с объектами работают по их базовым интерфейсам нежели по действительным типам, поэтому операция удаления по указателю на базовый класс является обычным явлением и не должна приводитm к каким-либо проблемам. Стандартный способ подсказывает сама задача по динамическому определению вызываемого метода: деструктор в классе Shape необходимо сделать виртуальным:

```
1  class Shape {
2  public:
3    virtual ~Shape()
4    {}
5
6    double getX() const;
7    double getY() const;
8
9    void move(double dx, double dy);
10   virtual void draw() const;
11
12 protected:
13   double x_;
14   double y_;
15
16   Shape(double x, double y):
17     x_(x),
18     y_(y)
19   {}
20 };
```

Листинг 9.5: Класс с виртуальным деструктором

На строке 3 определяется виртуальный деструктор с тривиальной реализацией.
Также возможен вариант с массивом:

```
1 Shape * shape = new Rectangle[5];
2
3 ...
4
5 delete [] shape;
```

Листинг 9.6: Полиморфный массив

На строке 5 массив из 5 прямоугольников удаляется по указателю на базовый класс. Этот код не может работать корректно практически ни при каких условиях, так как он должен делать следующее:

1. определить количество элементов в массиве;
2. вызывать деструктор для первого элемента;
3. прибавить размер одного элемента массива к указателю;
4. повторять 2 предыдщих шага для каждого элемента массива;
5. освободить память, занимаемую массивом.

Главная проблема в размере элемента массива. Он определяется по статическому типу переменной, который в этом примере Shape *. Размер Rectangle больше, поэтому при попытке вызвать деструктор для второго элемента массива будет использован некорректный адрес, что ведет к неопределенному поведению.

Таким образом, можно сформулировать следующие правила:

- Деструктор базового класса должен быть виртуальным или защищенным невиртуальным.
- Наследоваться от классов с открытыми невиртуальными деструкторами опасно.
- Не существует полиморфных массивов.

9.3 Управление размещением объектов

Неавтоматические объекты традиционно размещаются во Free Store. Но бывают ситуации, когда желателен больший контроль за размещением, например, какая-либо функция создает много временных объектов. В этой ситуации невыгодно размещать их во Free Store, так как это средство в силу своей универсальности может замедлять работу программы в целом. Гораздо удобнее все временные объекты разместить в отдельной области памяти, а потом эту область вернуть системе обратно целиком. Для решения этой задачи применяется перегрузка операторов **new** и **delete** [C++ 03, раздел 18.4] [C++ 11, раздел 18.6] вместе с дополнительными механизмами.

9.3.1 Перегрузка new и delete

Строго говоря, стандарт не допускает перегрузки *операторов* **new** и **delete**. Вместо этого происходит перегрузка *функций*, имеющих специальные названия **operator new**() и **operator delete**(). Это связано с тем, что настоящие операторы выполняют дополнительную работу по вызову конструкторов и деструкторов у объектов. Эта работа крайне важна и поэтому разработчику не предоставляется возможностей по нарушению этого контракта. Это еще один из примеров применения паттерна «Шаблонный метод» [Гам+13].

Таким образом, задачей функций **operator new**() и **operator delete**() остается только управления памятью - размещение и освобождение. Стандарт предлагает некоторое количество таких функций «из коробки»:

```
 1 namespace std {
 2   class bad_alloc;
 3   struct nothrow_t {};
 4   extern const nothrow_t nothrow;
 5 }
 6 void * operator new(std::size_t size)
 7   throw(std::bad_alloc);
 8 void * operator new(std::size_t size,
 9     const std::nothrow_t &) throw();
10 void operator delete(void * ptr) throw();
11 void operator delete(void * ptr,
12     const std::nothrow_t &) throw();
13 void * operator new[](std::size_t size)
14   throw(std::bad_alloc);
15 void * operator new[](std::size_t size,
16     const std::nothrow_t &) throw();
17 void operator delete[](void * ptr) throw();
18 void operator delete[](void * ptr,
19     const std::nothrow_t&) throw();
20
21 void * operator new (std::size_t size, void * ptr) throw();
22 void * operator new[](std::size_t size, void * ptr)
23   throw();
24 void operator delete (void * ptr, void *) throw();
25 void operator delete[](void * ptr, void *) throw();
```

Листинг 9.7: Стандартные функции управления памятью

Класс std::bad_alloc представляет собой тип исключения, генерируемого при невозможности распределения памяти. Класс std::nothrow_t вместе с глобальной переменной std::nothrow представляет собой флаг, указывающий что оператор **new** при невозможности размещения памяти должен вернуть 0 (или **nullptr**) вместо генерации исключения std::bad_alloc.

Первые 2 версии функций **operator new**() (строки 6 и 8) отвечают за стандартные размещающие функции. Различие между этими 2-мя версиями исключительно в поведении: версия с std::nothrow не генерирует исключений в случае ошибок размещения. К каждой размещающей функции обезательно существует пара в виде освобождающей функции. Освобождающие функции для этих вариантов **operator new**() приведены на строках 10 и 11.

Приведенные выше версии функций отвечают за размещение и освобождение одного объекта. За массивы отвечают функции **operator new**[]() и **operator delete**[]() (строки 13, 15, 17 и 18).

Еще один комплект стандартных функций отвечает за ситуацию, когда память под объекты или массивы уже размещена каким-либо образом. Эти функции называются «размещающие» или «placement». Они приведены на строках с 21 по 25.

Глобальная перегрузка

Для реализации собственного поведения при распределении и освобождении памяти необходимо перегрузить функции размещения и освобождения памяти. Но при этом не допускается перегружать размещающие версии этих функций (параграф 1 разделов 18.4.1.3 [C++ 03] и 18.6.1.3 [C++ 11]).

Самый простой вариант изменить способ управления памятью во всей программе это перегрузить глобальные функции управления памятью. Например, для реализации подсчета объема размещенной памяти можно осуществить следующую перегрузку[1]:

```
 1 #include <iostream>
 2 #include <cstdlib>
 3
 4 static std::size_t TOTAL_ALLOCATION = 0u;
 5
 6 void * operator new(std::size_t size) throw(std::bad_alloc)
 7 {
 8   if (size == 0u) {
 9     size = 1u;
10   }
11   TOTAL_ALLOCATION += size;
12
13   return std::malloc(size);
14 }
15
16 void operator delete(void * ptr) throw()
17 {
18   if (ptr != nullptr) {
19     std::free(ptr);
20   }
21 }
22
23 int main(int, char *[])
24 {
25   int * value = new (std::nothrow) int(5);
26   int * array = new int[10];
27
28   delete value;
29   delete [] array;
30
31   std::cout << "Total memory used: " << TOTAL_ALLOCATION << " bytes\n";
32
33   return 0;
34 }
```

Листинг 9.8: Глобальная перегрузка функций управления памятью

Реализация подсчета проста: перегруженная функция размещения увеличивает счетчик размещенной памяти на размер объекта. Так как передача нулевого размера в размещающую функцию допустима и при этом необходимо возвращать уникальный и корректный адрес, в этом случае размер исправляется на значение в 1 байт. Также для простоты примера отсутствует генерация исключения при ошибке.

Функция освобождения должна корректно работать при переданном пустом указателе, поэтому перед вызовом std::free() выполняется проверка.

В данном примере перегружены версия для одного объекта. Так как стандарт утверждает, что версии для массивов по умолчанию вызывают версию для одиночных объектов, в данном примере нет необходимости перегружать версии для массивов. С другой стороны, версии, не генерирующие исключения, не

[1]Пример перегрузки упрощен, так как не учитывает вопросы параллельного исполнения кода, а также не выполняет перегрузку всех необходимых версий и не полностью повторяет семантику стандартных функций.

обязаны работать путем вызова обычных функций размещения, поэтому полный пример должен перегружать и их.

Крайне важно для любой перегруженной функции **operator new**() перегружать соответствующую версию **operator delete**(), так как освобождение памяти всегда должно выполняться парным вызовом аналогичного оператора **delete**.

Перегрузка для класса

Глобальная перегрузка влияет на на все программу в целом. Для более тонкого управления памятью допускается перегружать функции управления памятью для классов:

```
1  class Shape {
2  public:
3    virtual ~Shape()
4    {}
5
6    double getX() const;
7    double getY() const;
8
9    void move(double dx, double dy);
10   virtual void draw() const;
11
12   void * operator new(size_t);
13   void operator delete(void *);
14
15 protected:
16   double x_;
17   double y_;
18
19   Shape(double x, double y):
20     x_(x),
21     y_(y)
22   {}
23 };
```

Листинг 9.9: Перегрузка функций управления памятью для классов

В целом, все работает аналогично глобальной перегрузке, но при этом необходимо помнить о следующих тонких моментах:

- перегруженные для класса [C++ 03; C++ 11, раздел 12.5] функции размещения являются статическими даже если они не объявлены такими явно;
- количество размещаемой памяти необязательно совпадает с размером класса, так как эти функции будут вызываться также и для наследников этого класса;
- функция освбождения памяти будет корректно найдена и вызвана при полиморфном удалении независимо от того, что она статическая, в случае, если деструктор класса виртуальный.

9.3.2 Обработчик new

Дополнительным механизмом управления размещением памяти является так называемый «new handler»:

```
1  typedef void (*new_handler)();
2
3  new_handler set_new_handler(new_handler new_p) throw();
```

Листинг 9.10: Обработчик new

Обработчик new_handler устанавливается при помощи стандартной функции std::set_new_handler(), которая принимает указатель на новый обработчик и возвращает старый. Обработчик вызывается в случае, если размещающая функция не может разместить новый блок памяти.

Задачей обработчика является выполнение одного из следующих вариантов:

- увеличить объем доступной памяти;

- сгенерировать исключение std::bad_alloc;
- вызывать abort() или exit().

Воспользовавшись этими знаниями и предположив, что std::malloc() и «new handler» связаны (что, впрочем, совершенно необязательно), полная реализация функции размещения с подсчетом памяти будет выглядеть так:

```
1  #include <new>
2  #include <cstdlib>
3
4  static std::size_t TOTAL_ALLOCATION = 0u;
5
6  void * operator new(std::size_t size) throw(std::bad_alloc)
7  {
8    if (size == 0u) {
9      size = 1u;
10   }
11   TOTAL_ALLOCATION += size;
12
13   void * ptr = nullptr;
14
15   do {
16     ptr = std::malloc(size);
17
18     if (ptr == nullptr) {
19       std::new_handler handler = std::set_new_handler(nullptr);
20
21       std::set_new_handler(handler);
22       if (handler != nullptr) {
23         (*handler)();
24       } else {
25         throw std::bad_alloc();
26       }
27     }
28   } while (ptr == nullptr);
29
30   return ptr;
31 }
32
33 void operator delete(void * ptr) throw()
34 {
35   if (ptr != nullptr) {
36     std::free(ptr);
37   }
38 }
```

Листинг 9.11: Полная реализация глобальных функций распределения памяти

Эта реализация вызывает в цикле стандартную функцию рапределения памяти. В случае неуспеха извлекается указатель на «new handler». Если текущий «new handler» не пуст, то он вызывается в расчете, что объем доступной памяти увеличится. Если его нет, но на строке 25 генерируется исключение.

Двойной вызов std::set_new_handler() на строках 19 и 21 связан с тем, что прямого доступа у казателю на обработчик нет, поэтому для получения указателя его сначала меняют на пустой указатель, а затем возвращают обратно. Эти строки вместе с увеличением счетчика предотвращают корректную работу в многопоточном окружении.

9.3.3 Дополнительные параметры

В некоторых случаях описанных средств недостаточно. Например, некий класс требует размещения при помощи разных механизмов в различных ситуациях. Например, обычно для класса подходит стандартный способ размещения, но при работе некоторых алгоритмов его удобнее разместить в специальном объекте-аллокаторе. Для этого требуется либо воспользоваться размещающими версиями операторов **new**

и **delete**, что неудобно, так как всегда необходимо помнить о 2-х этапах создания объекта, либо найти способ передать дополнительные параметры в функции управления памятью. Стандарт дает нам такую возможность: если размещающие функции имеют дополнительные аргументы, указанные после стандартных, то соответствующие параметры передаются между именем оператора и типом объекта.

Предположим, что существует некий класс Allocator, реализующий некую логику упраления памятью. Тогда для его использования в качестве дополнительного параметра необходимо перегрузить операторы следующим образом:

```
 2  class Allocator
 3  {
 4    ...
 5  };
 6
 7  class Object {
 8  public:
 9    Object(double x, double y):
10      x_(x),
11      y_(y)
12    {}
13    virtual ~Object()
14    {}
15
16    void * operator new(size_t, Allocator &);
17    void operator delete(void *, size_t);
18
19  protected:
20    double x_;
21    double y_;
22  };
23
24  int main(int, char *[])
25  {
26    Allocator alloc;
27    Object * obj = new (alloc) Object(1.5, 1.25);
28
29    delete obj;
30
31    return 0;
32  }
```

Листинг 9.12: Перегрузка распределения памяти с дополнительным параметром

Важным моментом является то, что хоть функция **operator new**() и определена с дополнительным параметром, аналогичной функции **operator delete** существовать не может. Поэтому реализация **operator delete**() должна правильно обрабатывать результаты передачи дополнительных параметров в **operator new**().

Еще одним артефактом такой перегрузки **operator new**() является сокрытие стандартных версий. Например:

```
25  int main(int, char *[])
26  {
27    Object * obj = new Object(1.5, 1.25);
28
29    return 0;
30  }
```

Листинг 9.13: Сокрытие стандартных распределителей памяти

Результат компиляции этого примера может быть таким (Clang):

```
$ c++ -Wall -Wextra -std=c++11 sample.cpp
sample.cpp:27:17: error: no matching function for call to
                        'operator new'
```

```
        Object * obj = new Object(1.5, 1.25);
                       ^
sample.cpp:17:9: note: candidate function not viable:
                       requires 2 arguments, but 1
      was provided
        void * operator new(size_t, Allocator &);
               ^
1 error generated.
```

9.4 Проблемы с динамической памятью

Применение динамической памяти сопряжено с различными проблемами. Среди них наиболее часто встречается утечка памяти, которая возникает каждый раз, когда распределенная память не освобождается. Косвенным признаком утечки памяти является постоянный рост используемой программой памяти.

Реже встречаются проблемы с двойным освобождением памяти и «висящими» (dangling) указателями. Всиящие указатели появляются тогда, когда память по указателю освобождается, но сам указатель не обнуляется. Доступ по таком указателю ведет к неопределенному поведению.

Эти проблемы диганостируются довольно сложно в большинстве случаев. Применение специальных средств динамического анализа программ, таких как Valgrind, позволяет обнаруживать множество проблем. Хороший процесс разработки программного обеспечения должен включать в себя такое тестирование.

Еще одной проблемой использования динамической памяти является фрагментация памяти. Характерным признаком этой проблемы будет невозможность размещения небольшого блока при большом объеме свободной памяти. Обычный способ борьбы с фрагментацией памяти это применение специализированных средств распределения памяти. Они не только могут снижать фрагментацию, но сделать программу существенно быстрее, так как стандартные средства управления памятью универсальны, в то время как специализированные могут лучше соответствовать конкретной задаче.

Глава 10

Шаблоны и обобщенное программирование

10.1 Понятие обобщенного программирования

В процессе разработки программных систем часто применяются одинаковые алгоритмы для разных типов данных, в качестве примера можно рассмотреть поиск максимального значения в массиве. Алгоритм поиска подходит для любого типа, но написать его придется 2 для **int** и **double**:

```
1  int max(const int * data, size_t n)
2  {
3    assert(n != 0);
4
5    int result = *(data++);
6
7    while (--n > 0) {
8      if (result < *data) {
9        result = *data;
10     }
11     ++data;
12   }
13
14   return result;
15 }
16
17 double max(const double * data, size_t n)
18 {
19   assert(n != 0);
20
21   double result = *(data++);
22
23   while (--n > 0) {
24     if (result < *data) {
25       result = *data;
26     }
27     ++data;
28   }
29
30   return result;
31 }
```

Очевидно, что эти 2 функции отличаются только в 2 местах:

1. типами данных в сигнатуре,
2. типом данных в переменной result.

Функция max() в примере использует перегрузку. Для пользовательских типов возможно написание аналога за счет полиморфизма включения, но это неэффективно, так как функция будет работать только для типов, реализующих некий интерфейс.

Возможен вариант реализации этой функции при помощи препроцессора. Но полученный результат будет небезопасен и несопровождаем для чего-либо более сложного.

Таким образом, в языке требуется новый механизм, обеспечивающий написание обобщенных функций. В C++ эту функциональность предоставляют шаблоны.

10.2 Шаблоны функций

Шаблон функции начинается с ключевого слова **template**, после которого в угловых скобках идут аргументы шаблона. Каждый аргумент шаблона начинается с ключевого слова **typename** или **class**, за которым следует имя аргумента. В процессе инстанциирования шаблона компилятор заменит имя аргумента на действительный тип, переданный параметром, в теле шаблона:

```
 1 template < typename T >
 2 T max(const T * data, size_t n)
 3 {
 4   assert(n != 0);
 5
 6   T result = *(data++);
 7
 8   while (--n > 0) {
 9     if (result < *data) {
10       result = *data;
11     }
12     ++data;
13   }
14
15   return result;
16 }
```

Для использования шаблонной функции в точке вызова нужно указать параметры шаблона в дополнение к параметрам функции:

```
 1 int main(int, char *[])
 2 {
 3   int int_values[] = { 1, 2, 5, 6, 7 };
 4   double int_values[] = { 2.25, 12.4, 456.3 };
 5
 6   std::cout << max< int >(int_values, 5) << "\n"
 7             << max< double >(double_values, 3) << "\n";
 8
 9   return 0;
10 }
```

В случаях, когда параметры шаблона используются в качестве типов аргументов функции, указание этих же типов в параметрах шаблона приводит к лишнему повторению информации, что может приводить к ошибкам. Поэтому компилятор содержит логику по автоматическому вычислению значений параметров шаблонов функций (type deduction). В этом случае указание параметров шаблона можно пропустить:

```
1   std::cout << max(int_values, 5) << "\n"
2             << max(double_values, 3) << "\n";
```

Шаблоны функций можно перегружать, например:

```
1 //   I
2 template < typename T >
3 void foo(T t)
4 { ... }
5
6 //  II
```

```
 7  template < typename T >
 8  void foo(const T * p)
 9  { ... }
10
11  // III
12  void foo(double d)
13  { ... }
```

Компилятор выберет наиболее подходящий вариант в точке вызова, при этом нешаблонные функции будут иметь приоритет над шаблонными:

```
1  int int_val;
2  double double_val;
3
4  foo(int_val);       //    I
5  foo(&int_val);      //   II
6  foo(double_val);    //  III
7  foo(&double_val);   //   II
```

Полные правила выбора перегрузки приведены в разделе 14.8.3 стандартов [C++ 03; C++ 11].

10.3 Шаблоны классов

Коллекции объектов это еще одна часто использующаяся сущность. Применение шаблонов классов позволяет избавляется от дублирования кода для поддержки разных типов.

Например, динамически распределенный массив объектов будет представлять собой шаблон класса, параметром которого является тип элемента:

```
 0  template < typename T >
 1  class Array
 2  {
 3  public:
 4    using value_type = T;
 5    using this_type = Array< T >;
 6
 7    Array(size_t size):
 8      data_(new T[size]),
 9      size_(size)
10    {}
11
12    Array(const this_type &) = delete;
13    Array(this_type &&) = delete;
14
15    ~Array()
16    {
17      delete [] data_;
18    }
19
20    this_type & operator =(const this_type &) = delete;
21    this_type & operator =(this_type &&) = delete;
22
23    size_t size() const
24    {
25      return size_;
26    }
27
28    value_type & operator [](size_t i) const
29    {
30      if (i >= size_) {
31        throw std::out_of_range("Index out of range");
32      }
```

```
33
34      return data_[i];
35    }
36
37  private:
38    value_type * data_;
39    size_t size_;
40  };
```

Этот шаблон предоставляет минимальный безопасный вариант встроенного массива.

Используются шаблоны классов аналогично шаблонам функций путем указания параметров шаблона в угловых скобках после имени:

```
1  void foo()
2  {
3    Array< int > int_arr(25);
4
5    int_arr[5] = 23456;
6
7    std::cout << int_arr.size() << "\n";
8
9    int_arr[25] = 16; // Exception std::out_of_range
10 }
```

10.4 Псевдонимы шаблонов

Указание всех параметров шаблона каждый раз, когда он упоминается крайне неудобно и длинно. Поэтому часто применяются псевдонимы шаблонов. В стандарте C++ 03 есть только один механизм создания псевдонима типа: ключевое слово **typedef**. Например, тип строки std :: string является псевдонимом для шаблона std :: basic_string:

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

При помощи **typedef** невозможно создать псевдоним, у которого часть параметров шаблона может варьироваться. Поскольку такая задача возникает часто, в стандарте C++ 11 было предложено расширения синтаксиса ключевого слова **using**:

```
class MyAllocator {
  ...
};

template < typename T >
using my_allocated_vector = std:vector< T, MyAllocator >;
```

10.5 Зависимые имена

Шаблоны классов часто содержат различные вложенные типы или псевдонимы типов, составляющие их интерфейсы. Например, внутри класса std :: vector определены вложенные типы iterator и const_iterator, отвечающие за обход вектора. Внутри других контейнеров определены типы с такими же именами, но другой реализацией, абстрагирующие обход соответствующих контейнеров. Таким образом, становится возможным написать шаблонную функцию, проверяющую наличие значения в контейнере:

```
1  template < typename Container, typename T >
2  bool contains(const Container & data, const T & value)
3  {
4    Container::const_iterator item = data.begin();
5
6    while ((item != data.end()) && (*item != value)) {
7      ++item;
8    }
```

```
 9
10    return item != data.end();
11  }
```

Но результатом компиляции этой функции будет:

```
$ c++ -std=c++11 -Wall -Wextra -o /dev/null contains.cpp
contains.cpp:4:2: error: missing 'typename' prior to
                  dependent type name 'Container::const_iterator'
        Container::const_iterator item = data.begin();
        ^~~~~~~~~~~~~~~~~~~~~~~~
        typename
1 error generated.
```

Причина этого в том, что при компиляции шаблона компилятору неизвестны параметры. Поэтому он не знает, является ли const_iterator типом, полем или методом. Для разрешения этой неоднозначности компилятору требуется подсказка, более того, Clang услужливо подсказывает, как это сделать: необходимо добавить ключевое слово **typename**:

```
 1  template < typename Container, typename T >
 2  bool contains(const Container & data, const T & value)
 3  {
 4    typename Container::const_iterator item = data.begin();
 5
 6    while ((item != data.end()) && (*item != value)) {
 7      ++item;
 8    }
 9
10    return item != data.end();
11  }
```

10.6 Инстанцирование шаблонов

10.6.1 Неявное

Процесс генерации кода для шаблона с параметрами называется инстанцированием. Неявное инстанцирование происходит в точке использования шаблона. При этом единица трансляции, в которой используется шаблон получает не более одной реализации каждой используемой шаблонной функции и не более одного каждого используемого метода шаблонного класса. При компоновке программы компоновщик исключает дубликаты символов, предотвращая излишнее разрастание кода программы.

Из этого процесса следует, что в точке инстанцирования шаблона необходимо иметь доступ к его определению. Это ведет к тому, что шаблоны функций и классов полностью описываются в заголовочных файлах. Для небольших функций и классов это прекрасно работает, но по мере роста класса становится разумным разделять объявления методов и их определения. Достигается это путем указания параметров шаблона в определении метода:

```
 1  template < typename T >
 2  class Array
 3  {
 4  public:
 5    using value_type = T;
 6    using this_type = Array< T >;
 7
 8    Array(size_t size);
 9    Array(const this_type &) = delete;
10    Array(this_type &&) = delete;
11    ~Array();
12
13    this_type & operator =(const this_type &) = delete;
14    this_type & operator =(this_type &&) = delete;
15
```

```
16    size_t size() const;
17    value_type & operator [](size_t i) const;
18
19  private:
20    value_type * data_;
21    size_t size_;
22  };
23
24  template < typename T >
25  Array< T >::Array(size_t size):
26    data_(new T[size]),
27    size_(size)
28  {}
29
30  template < typename T >
31  Array< T >::~Array()
32  {
33    delete [] data_;
34  }
35
36  template < typename T >
37  size_t Array< T >::size() const
38  {
39    return size_;
40  }
41
42  template < typename T >
43  T & Array< T >::operator [](size_t i) const
44  {
45    if (i >= size_) {
46      throw std::out_of_range("Index out of range");
47    }
48
49    return data_[i];
50  }
```

10.6.2 Явное

В некоторых случаях неявное инстанцирование неэффективно. Например, стандартный тип строки в C++ std::string применяется настолько часто, что иметь в каждой единице трансляции результаты его инстанцирования невыгодно: это требует места и дополнительного времени на обработку. Поэтому для таких шаблонов применяется техника явного (epxlicit) инстацирования. Внутри стандартной библиотеки в заголовке `string` объявлен экземпляр шаблона:

```
1  namespace std {
2    typedef basic_string< char > string;
3    extern template class basic_string< char >;
4  }
```

Ключевое слово **extern** является полезным расширением, поддерживаемым большинством компиляторов: оно говорит о том, полный экземпляр шаблона с такими параметрами где-то уже определен. Внутри единицы трансляции, входящей в стандартную библиотеку происходит само инстанцирование:

```
1  namespace std {
2    template class basic_string< char >;
3  }
```

10.7 Специализация шаблонов

Иногда возникает ситуация, когда общая версия шаблона не подходит или работает неэффективно для каких-либо параметров. Вместо создания шаблонов и классов с новыми именами C++ предлагает мощный механизм создания особых версий шаблонов, называемый «специализация шаблонов» (template specialization).

10.7.1 Полная специализация

Простейшим вариантом специализации является полная специализация, когда создается версия шаблона для полного набора параметров.

В стандарте языка описана функция std::swap(), выполняющая обмен значениями двух аргументов. Упрощенно, в стандарте C++ 03 она выглядит так:

```
1  template < typename T >
2  void swap(T & a, T & b)
3  {
4    T temp = a;
5
6    a = b;
7    b = temp;
8  }
```

Этот вариант подходит для любых типов, объекты которых можно копировать. Но если класс предоставляет более эффективный метод swap(), то разумно сделать так, чтобы std::swap() вызывала именно его. Для этого идеально подходит полная специализация:

```
1  namespace std {
2    template <>
3    inline void swap< Object >(Object & a, Object & b)
4    {
5      a.swap(b);
6    }
7  }
```

Полная специализация классов выполняется аналогично.

TODO: Пример полной специализации классов

10.7.2 Частичная специализация

Частичная специализация подразумевает, что специальная версия шаблона создается только для части параметров. Например, класс, хранящий пару «ключ-значение» достаточно просто реализуется так:

```
1  template < typename Key,
2             typename Value >
3  struct KeyValuePair {
4    using this_type = KeyValuePair< Key, Value >;
5    const Key key;
6    Value value;
7
8    bool operator <(const this_type & rhs) const
9    {
10     return key < rhs.key;
11   }
12 };
```

operator <() служит для сравнения ключей и может использоваться для хранения и поиска в контейнерах. Если все строковые ключи в программе должны сравниваться нечувствительно к регистру букв, то требуется вызов специальной функции, например, strcasecmp(). Для этого идеально подходит частичная специализация:

```
1  template < typename Value >
2  struct KeyValuePair< std::string, Value > {
```

```
 3    using this_type = KeyValuePair< std::string, Value >;
 4    const std::string key;
 5    Value value;
 6
 7    bool operator <(const this_type & rhs) const
 8    {
 9      return strcasecmp(key.c_str(), rhs.key.c_str()) < 0;
10    }
11  };
```

Стандарт C++ допускает частичную специализацию только для шаблонов классов.

10.8 Особые аргументы шаблонов

10.8.1 Аргументы по умолчанию

Аргументы шаблонов, также как и аргументы функций, часто имеют часто используемое значение. Например, класс std::vector предоставляет возможность управлять распределением памяти при помощи аллокаторов. Тип аллокатора передается вторым параметром шаблона. Но использование нестандартного аллокатора является оптимизацией, потому обычно этот параметр добавляется в процессе оптимизации программы, а большинство случаев использует стандартный аллокатор. Для упрощения, этот стандартный аллокатор является аргументом со значением по умолчанию:

```
1  template < typename T,
2             typename Allocator = std::allocator< T > >
3  class vector {
4    ...
5  };
```

Как видно из примера, в значении по умолчанию допустимо ссылаться на объявленные ранее аргументы шаблона.

10.8.2 Аргументы-«нетипы»

Следующим вариантом аргумента шаблонов являются значения. В качестве аргумента могут использоваться:

- целочисленная константа, в том числе вычисляемая на этапе компиляции;
- адрес функции или объекта с внешним или внутренним (C++ 11) связыванием;
- указатель на член класса.

Для таких аргументов в определении шаблона вместо **typename** и **class** используется имя типа. Например, шаблонная функция, определяющая размер массива, может быть описана следующим образом:

```
1  #include <iostream>
2
3  template < typename T, size_t N >
4  size_t count_of(const T (&)[N])
5  {
6    return N;
7  }
8
9  int main(int, char *[])
10 {
11   struct {
12     int i;
13   } bar[10];
14
15   std::cout << count_of(bar) << "\n";
16
17   return 0;
```

```
18 }
```

Этот пример работает для C++ 11. В C++ 03 параметром шаблона не может быть локальный тип, поэтому определение структуры придется переместить из функции:

```
1  #include <iostream>
2
3  template < typename T, size_t N >
4  size_t count_of(const T (&)[N])
5  {
6    return N;
7  }
8
9  namespace {
10   struct foo_t {
11     int i;
12   };
13 }
14
15 int main(int, char *[])
16 {
17   foo_t bar[10];
18
19   std::cout << count_of(bar) << "\n";
20
21   return 0;
22 }
```

Необходимо отметить, что массивы и адреса элементов массива на могут быть параметрами шаблонов. Так как строковые литералы фактически являются массивами, они также не могут быть параметрами шаблонов [C++ 03; C++ 11, параграфы 2 и 3 раздела 14.3.2].

10.8.3 Шаблонные аргументы

Также существует возможность аргументом шаблона другой шаблон [C++ 03; C++ 11, раздел 14.3.3]:

```
1  template < typename T,
2             template < typename U,
3                        typename Allocator >
4             class Container >
5  class Queue {
6    ...
7  };
```

В качестве второго параметра шаблона Queue допустимо передавать имена шаблонов, принимающих 2 параметра, например, std::vector. Конкретные инстансы шаблонов, например, std::vector< **int** >, не подходят.

10.9 Параметрический полиморфизм

Как уже отмечалось, в C++ присутствуют 3 типа полиморфизма:

1. ситуативный (ad-hoc),
2. подтипов (включения),
3. параметрический.

Первые два были рассмотрены ранее, теперь очередь параметрического.

Упрощенно, параметрический полиморфизм возникает в языках программирования тогда, когда данные различных типов обрабатываются одинаково. В C++ за этот вид полиморфизма отвечают шаблоны.

Например, функция вычисления максимального значения внутри контейнера может выглядеть так:

```
1  template < typename InputIterator >
2  typename std::iterator_traits< InputIterator >::value_type
3  max(InputIterator begin, const InputIterator & end)
4  {
5      assert(begin != end);
6
7      typename
8      std::iterator_traits< InputIterator >::value_type
9          result = *begin;
10
11     while (++begin != end) {
12         if (res < *begin) {
13             res = *begin;
14         }
15     }
16
17     return result;
18 }
```

Данная функция полиморфна как по типу контейнера (std::vector, std:: list и т.п.), так и по типу значения в контейнере. Единственное требование — объект должен предоставлять реализацию **operator** <(). Таким образом, она также демонстрирует и ситуативный полиморфизм.

Глава 11

Управление ресурсами

11.1 Описание проблемы

По мере роста разрабатываемых систем, все сложнее и сложнее становится оценивать их требования к ресурсам. Это связано как с внешними факторами, например, необходимостью обрабатывать на сервере данные заранее неизвестного размера, так и внутренними, такими как рост сложности архитектуры программного обеспечения. Желание контролировать этот рост обычно приводит к выделению компонентов с четко определенными интерфейсами. При этом система во всех деталях становится еще более трудной для понимания, но взамен появляется возможность рассматривать ее покомпонентно, изолируя отдельные, несложные сами по себе, части функциональности друг от друга. В этой ситуации управление ресурсами усложняется, так как различные ресурсы начинают передаваться между компонентами и модулями.

Различные языки программирования и фреймворки предлагают различные способы решения данной проблемы. Но и простой C++ предлагает пути решения.

11.2 Динамическая память

Абсолютное большинство задач не может быть решено в заранее определенном и ограниченном объёме памяти, так как определение этого объёма затруднено или невозможно. Поэтому в языках программирования присутствует концепция динамической памяти. Для C++ пример работы с динамической памятью может выглядеть так:

```
1 void foo()
2 {
3   Interface * obj = new Object;
4
5   obj->execute();
6 }
```

В данном, довольно абстрактном, примере в строке 3 динамически создается реализация интерфейса Interface, её адрес сохраняется в переменной obj, и память принимает вид с рис. 11.1.

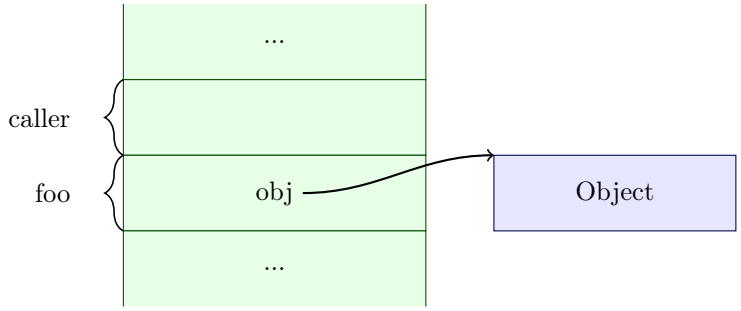


Рис. 11.1: Результат распределения памяти

Далее, в строке 5, созданный объект используется путём вызова функции Interface :: execute(). Наш пример заканчивается в строке 6. В этом месте происходит разрушение автоматических переменных блока,

с данном случае это obj.

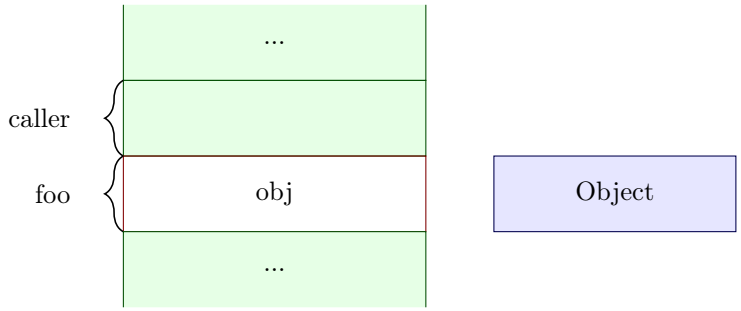


Рис. 11.2: Память после завершения функции

Таким образом, ссылка на размещенный объект утеряна, и объект не может быть удален. Простейшее решение — добавить пропущенный вызов **delete**. Например, вот так:

```
1  void foo()
2  {
3    Interface * obj = new Object;
4
5    obj->execute();
6    delete obj;
7  }
```

К сожалению, данное решение крайне ограничено, так как не охватывает следующие случаи:

- Исключения — в случае генерации исключения функцией Object::execute() у оператора **delete** не будет шанса на исполнение.
- При более сложной логике, внутри функции могут возникнуть и другие точки выхода помимо простого завершения. В этом случае добавленный оператор также не выполнится.

Трудности с исключениями могут быть решены относительно просто путем перехвата исключений:

```
1  void foo()
2  {
3    Interface * obj = new Object;
4
5    try {
6      obj->execute();
7      delete obj;
8    } catch (...) {
9      delete obj;
10     throw;
11   }
12   // delete obj;
13 }
```

Несколько важных моментов, на которые необходимо обратить внимание:

- Объект необходимо удалять в двух местах — в строках 7 и 9, потому что выполняться будет только одна из них.
- Первое удаление можно перенести из строки 7 на строку 12. Правильными будут оба варианта, выбор зависит от вкуса разработчика.
- Поскольку исключение *не обрабатывается*, оно не должно быть потеряно или искажено. Поэтому необходимо использовать специальный синтаксис для перегенерации текущего исключения.

К сожалению, данный путь является экстенсивным — по мере увеличения количества объектов, создаваемых внутри функции, растет количество операторов **delete**. Например, для 3-х объектов код приобретает вид:

```
1  void foo()
2  {
3    Interface * obj = new Object;
4    Interface2 * obj2 = nullptr;
5    Interface3 * obj3 = nullptr;
6
7    try {
8      obj->execute();
9
10     obj2 = new Object2;
11     obj2->execute2();
12
13     obj3 = new Object3;
14     obj3->execute3();
15
16     delete obj3;
17     delete obj2;
18     delete obj;
19   } catch (...) {
20     delete obj3;
21     delete obj2;
22     delete obj;
23     throw;
24   }
25 }
```

Ситуация несколько упрощается тем, что стандарт языка требует, чтобы операция удаления нулевого указателя корректно исполнялась. Поэтому проверок на ненулевой указатель не требуется.

Тем не менее, проблема с множественными выходами из функции не решена. Можно двигаться тем же экстенсивным путем, копируя пакет операторов **delete** в каждую точку выхода. Но, как и со всеми подобными решениями, есть серьезная проблема сопровождения — при каждом добавлении нового объекта необходимо не забыть добавить его удаление во все точки выхода. Второй, не менее важный, момент связан с тем, что определение переменной отдаляется от ее фактического применения, и, более того, переменная определяется в том месте, где корректно проинициализировать ее просто невозможно. В этом случае неизбежно возникает необходимость четкого понимания, в каких местах переменная неинициализирована, в каком месте она получает корректное значение и может быть использована. Так как такие знания можно получить только после тщательного изучения кода, все рекомендации по написанию кода содержат указания на инициализацию переменной осмысленным значением в точке определения.

11.3 Концепция владения

Прежде чем рассматривать другие пути решения проблемы, необходимо разобраться с концепцией владения.

В программах на C++ любым объектом кто-либо владеет. Владелец объекта — это тот, кто непосредственно влияет на время жизни объекта. Например, для автоматического объекта (локальная переменная в функции) владельцем является блок, в котором она определена. Второй пример — поле класса: для объекта поля класса владельцем является объемлющий класс.

Важно не путать достижимость объекта и владение. Если объект достижим, то это значит, что имеется указатель или ссылка на него. Но при этом управление временем жизни отсутствует. В общем случае, ссылки никогда не владеют объектами (мы не рассматриваем крайне неудачный вариант с вызовом **delete** &reference;). Указатели предоставляют полную свободу, так как дают возможность удалить объект после использования и, соответственно, позволяют обеспечивать владение объектом.

Владение бывает 2-х типов: разделяемое и неразделяемое. Простейший тип — без разделения владения. Это значит, что один и только один указатель, способный достичь некоего объекта target, отвечает за его время жизни (см. рис. 11.3), в примере это переменная owner. Я бы предпочел избегать применения выражения «эксклюзивное владение», так как интуитивно может казаться, что на этот объект других ссылок и указателей быть не может.

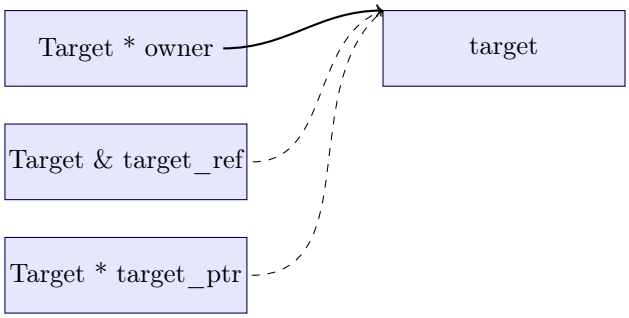


Рис. 11.3: Неразделяющее владение

Необходимо обратить внимание, что если ссылка никогда не владеет объектом и не предоставляет возможность его удаления, то для обычного (сырого или raw) указателя всегда существует неоднозначность: невозможно понять, владеет указатель объектом, на который указывает, или нет, исходя из кода использования — только из комментария в точке определения указателя.

Второй вариант владения — с разделением (shared ownership). Это значит, что несколько указателей управляют временем жизни объекта. Например, на рис. 11.4 владение разделяют указатели owner1 и owner2. В случае разделяемого владения требуется некий метод согласования между владельцами для определения момента, когда объект можно удалить. В данном примере используется счетчик ссылок refs внутри объекта для подсчета количества владеющих указателей.

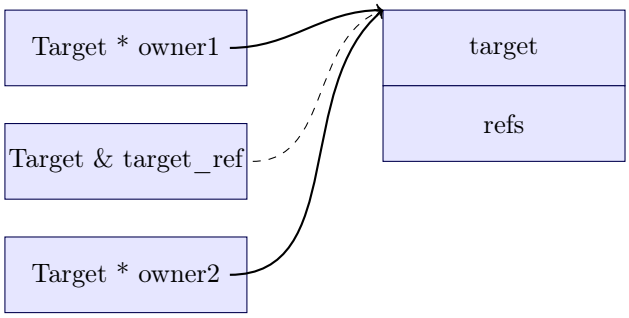


Рис. 11.4: Разделение владения

11.4 Умные указатели

Как известно, большое количество ручной работы ведет к большому количеству ошибок, причем значительная часть их появляется в процессе сопровождения: где-то забыли добавить удаление объекта, где-то удалили объект слишком рано, в каком-то месте удалили лишнее. А где-то объект удалили, а указатель не обнулили, как в примерах выше. Но если в этих примерах момент удаления объекта совпадает с моментом удаления переменной-указателя, то в других случаях возможно появление «висящего» указателя со всеми неприятными последствиями. Таким образом, было бы неплохо избавить разработчика от этой рутинной работы.

Стандарт C++ 2003 года содержит следующую информацию:

- Параграф 10 раздела 12.4 [C++ 03; C++ 11] говорит о том, когда вызываются деструкторы объектов. В данном случае наиболее интересна часть, посвященная временным и автоматическим объектам. Для них деструктор автоматически вызывается при окончании времени жизни объекта — в конце выражения или блока, в зависимости от ситуации.

- Параграф 2 раздела 13.5 [C++ 03; C++ 11] разрешает перегрузить унарный оператор $*$, имеющий семантику разыменования указателя, раздел 13.5.6 [C++ 03; C++ 11] описывает перегрузку оператора $->$, реализующего доступ к членам класса по указателю на класс.

Воспользовавшись этим можно создать класс, экземпляр которого будет похож на обычный указатель. Такой класс будет называться «умный указатель».

Идиома — устоявшаяся практика применения каких-либо средств языка или подходов к дизайну. Мир C++ полон различных идиом, например, RAII или PImpl.

Умные указатели это удачный пример одной из идиом языка C++. Эта идиома обычно встречается под названием RAII — Resource Acquisition Is Initialization. При этом сама идиома шире, так как охватывает не только указатели, но и другие ресурсы, например, открытые файлы. Характерными признаками RAII является захват какого-либо ресурса в конструкторе объекта и освобождение в деструкторе.

11.4.1 Умные указатели как пример применения идиомы RAII

Какие требования должны быть предъявлены к классу умного указателя? Логичным было бы потребовать:

- Экземпляр должен конструироваться из «сырого» указателя (raw pointer).
- Деструктор должен освобождать объект, на который указывает данный умный указатель.
- Оператор * должен возвращать ссылку на объект, хранящийся внутри умного указателя.
- Оператор -> должен возвращать сырой указатель на хранящийся объект.
- Копирование, присваивание и перемещение умного указателя можно запретить. Это несколько упростит реализацию.

Вооружившись этими требованиями, можно написать следующий класс:

```
1  class Interface;
2
3  class InterfacePtr
4  {
5  public:
6    explicit InterfacePtr(Interface * object);
7    ~InterfacePtr();
8
9    Interface & operator *() const;
10   Interface * operator ->() const;
11
12 private:
13   Interface * ptr_;
14
15   InterfacePtr(const InterfacePtr &) = delete;
16   InterfacePtr(InterfacePtr &&) = delete;
17   InterfacePtr & operator =(const InterfacePtr &) = delete;
18   InterfacePtr & operator =(InterfacePtr &&) = delete;
19 };
```

На строке 1 осуществляется декларирование класса интерфейса. В данном примере предполагается, что сам интерфейс определен в другом заголовочном файле, поэтому для уменьшения зависимостей и ускорения сборки заголовочный файл не используется, а вместо этого интерфейс предварительно декларируется.

TODO: Describe «explicit»

На строках 15 и 17 декларируются, соответственно, конструктор копирования и оператор присваивания как удаленные. Это предотавращает их автоматическую генерацию. Премещающие версии также удалены.

```
1  #include <Interface.hpp>
2
3  #include <cassert>
4
5  InterfacePtr::InterfacePtr(Interface * ptr):
6    ptr_(ptr)
7  {
8    assert(ptr != 0);
9  }
10
```

```
11 InterfacePtr::~InterfacePtr()
12 {
13   delete ptr_;
14 }
15
16 Interface & InterfacePtr::operator *() const
17 {
18   return *ptr_;
19 }
20
21 Interface * InterfacePtr::operator ->() const
22 {
23   return ptr_;
24 }
```

Реализация методов умного указателя прямолинейна и проста. Важными моментами являются:

- Поскольку мы удаляем объект, нам требуется доступ к его деструктору. Следовательно, нам требуется определение класса Interface, поэтому мы используем дополнительный заголовочный файл на строке 1.

- Стандарт требует (параграф 5.3.5.2), чтобы операция **delete**, примененная к нулевому указателю, была корректной. Это делает проверку на нулевой указатель перед строкой 13 излишней.

Последний пример может быть переделан с использованием умного указателя так (предполагается, что умные указатели для других интерфейсов написаны подобным образом):

```
1  void foo()
2  {
3    InterfacePtr obj(new Object);
4
5    obj->execute();
6
7    Interface2Ptr obj2(new Object2);
8
9    obj2->execute2();
10
11   Interface3Ptr obj3(new Object3);
12
13   obj3->execute3();
14 }
```

Как видно из примера, код стал значительно короче и проще.

Очевидно, что приведенный пример умного указателя не способен правильно работать с указателями на константный объект. Эта проблема элементарно решается путем превращения его в шаблон.

11.4.2 Типы умных указателей

Каждый разработчик, прежде чем предалагать новую реализацию умных указателей, должен рассмотреть возможность применения уже готовых. Их доступно довольно много.

1. В стандартной библиотеке C++ 2003 [C++ 03] года:

   std::auto_ptr Можно спорить, насколько этот класс умный, или он слишком умный, но он удобен в применении, если больше ничего нет. Необходимо помнить, что std::auto_ptr *передает* владение объектом при копировании или присваивании. Поэтому надо быть крайне осторожными, передавая такие умные указатели аргументами функций или возвращая их из функций. Также не стоит помещать их в стандартные контейнеры: в лучшем случае код не скомпилируется, в худшем — проблемы проявятся в самый неподходящий момент.

2. В стандартной библиотеке C++ Technical Report 1 (TR1) помимо std::auto_ptr:

std::tr1::shared_ptr Умный указатель со счетчиком ссылок. Обладает довольно изощренной функциональностью, позволяющей, например, решить проблему с недоступностью деструктора объекта в точке определения умного указателя или применить нестандартный способ разрушения объекта.

std::tr1::weak_ptr Используется в паре с std::tr1::shared_ptr для реализации семантики «слабой» или невладеющей ссылки. Объект будет удален, когда он хранится только внутри std::tr1::weak_ptr'ов Для работы с объектом, std::tr1::weak_ptr необходимо конвертировать в std::tr1::shared_ptr.

3. В стандартной библиотеке C++ 2011 [C++ 11] года отсутствует std::auto_ptr. Вместо этого добавлены более полезные указатели:

   std::unique_ptr Аналог std::auto_ptr, но без опасной семантики передачи владения и с поддержкой массивов. Также поддерживает пользовательский способ разрушения объекта.

   std::shared_ptr Полный аналог такового из C++ TR1.

   std::weak_ptr Полный аналог такового из C++ TR1.

4. Библиотека Boost [BOOST] предлагает следующие типы умных указателей для компиляторов стандартов 2003 и 2011 годов:

   boost::shared_ptr Обычный указатель с разделяемым владением. Именно данный класс использовался в качестве прототипа для включения в C++ TR1 и новый стандарт.

   boost::scoped_ptr Этот указатель был прототипом std::unique_ptr, хотя и обладает несколько меньшей функциональностью, например, не содержит поддержки пользовательского способа удаления объекта.

   boost::intrusive_ptr Указатель с разделяемым владением, для которого счетчик хранится «внутри объекта». На самом деле, счетчик может быть реализован любым способом. Слово «intrusive» в названии указателя означает, что указатель использует детали реализиции объектов, на которые он указывает.

   boost::shared_array Умный указатель для массива с разделяемым владением.

   boost::scoped_array Умный указатель для массива с неразделяемым владением.

Библиотека Boost также предлагает реализацию C++ TR1 для тех компиляторов, которые ее не содержат.

11.4.3 Использование умных указателей для улучшения дизайна интерфейсов

Пусть имеется функция:

```
1  class Result;
2  class Parameter;
3
4  Result * perform_task(const Parameter * param,
5      int flags);
```

Как видно, в интерфейсе задействовано 2 указателя: на параметр и на результат. Рассмотрим варианты использования этой функции:

```
1  perform_task(0, 0);
2
3  Parameter params;
4  perform_task(&params, 0);
5
6  Parameter * params2 = new Parameter;
7  perform_task(params2, 0);
8
9  Result * res = perform_task(&params, 0);
10
11 delete res;
```

Первый вариант использования (строка 1) не передает первый параметр и не использует результат. К сожалению, невозможно определить, не заглядывая в документацию (которой может и не быть), не вызывает ли это каких-либо проблем. Самая очевидная — неизвестно, кто владеет объектом, возвращаемым из функции. Если владение передается клиенту, то возникает утечка памяти. Второй момент — передача пустого указателя может вызывать ошибки внутри функции, но нет информации о том, требуется ли передавать указатель на корректный объект при вызове функции или допустимо передать пустой указатель.

Второй вариант (строка 4) передает указатель на стековый объект. Все выглядит хорошо, но что будет, если функция сохраняет у себя полученный указатель и попытается его использовать позже? Стековый объект будет разрушен по окончании блока, и функция, как минимум, будет использовать некорректные данные.

Третий вариант (строка 7) передает параметр, размещенный в динамической памяти и не удаляет его в расчете на то, что функция примет владение им. Если же владение не переходит, возникает утечка памяти.

В последнем варианте (строка 9) результат выполнения сохраняется в переменной и удаляется потом при помощи оператора **delete**. Если это указатель на статическую переменную внутри функции, данный код некорректен, и его поведение не определено.

Необходимо отметить, что независимо от того, что написано в документации о функции, все варианты будут успешно скомпилированы. Поэтому возникает вопрос: существует ли способ защититься от этих ошибок?

Ответ очень прост: используя семантику умных указателей и другие элементы C++, легко донести требуемую информацию до клиента и предотвратить неправильное использование. Если предположить, что объекты слишком дороги для копирования и, как следствие, передачи по значению, возможны следующие варианты изменений интерфейса функции:

```
// Обязательный параметр без передачи владения,возвращаемое значение без передачи владения
Result & perform_task(const Parameter & param, int flags);
// Необязательный аргумент без передачи владения,возвращаемое значение без передачи владения
Result & perform_task(int flags);
Result & perform_task(const Parameter & param, int flags);
// Аргумент с передачей владения,возвращаемое значение с передачей владения
std::auto_ptr< Result > perform_task(
    std::auto_ptr< const Parameter > & param, int flags);
// или, так как параметр выбывает из владения вызывающего,
std::auto_ptr< Result > perform_task(
    std::auto_ptr< Parameter > param, int flags);
// или, если C++ 2011
std::unique_ptr< Result > perform_task(
    std::unique_ptr< Parameter > param, int flags);
// Аргумент с разделением владения,возвращаемое значение с разделением владения
std::tr1::shared_ptr< Result > perform_task(
    const std::tr1::shared_ptr< const Parameter > & param,
    int flags);
```

11.5 Scope Guards и управление другими типами ресурсов

Данная идиома применима не только к динамической памяти. Она прекрасно подходит ко всем ресурсам. Необходимо только помнить, что освобождение ресурса не должно приводить к ошибкам.

Вот далеко не полный перечень того, чем можно управлять через идиому RAII:

1. файлы, открытые через платформенный API (open(), CreateFile());
2. сокеты;
3. примитивы синхронизации (мьютексы, условные переменные, события).

Но среди всего множества ресурсов есть такие, которые должны иметь четкую область использования, обычно совпадающую с неким блоком в программе. Например, правила хорошего дизайна требуют, чтобы мьютекс захватывался и освобождался внутри одной функции. Таким образом, область, в которой мьютекс захвачен, легко идентифицировать.

Для решения этой задачи также подходит идиома RAII, причем данное ее применение имеет специфическое название — «Scope Guard».

В качестве примера можно рассмотреть следующий вариант реализации:

```
1  class MutexLock
2  {
3  public:
4    MutexLock(pthread_mutex_t & mutex):
5      mutex_(mutex)
6    {
7      pthread_mutex_lock(&mutex_);
8    }
9
10   ~MutexLock()
11   {
12     pthread_mutex_unlock(&mutex_);
13   }
14
15 private:
16   pthread_mutex_t & mutex_;
17
18   MutexLock(const MutexLock &);
19   MutexLock & operator =(const MutexLock &);
20 };
```

Использовать его можно так:

```
1  pthread_mutex_t resource_mutex;
2
3  ...
4
5  void foo()
6  {
7    ...
8    {
9      MutexLock lock(resource_mutex);
10
11     // Работа с ресурсом
12     ...
13   }
14   ...
15 }
```

Таким образом гарантируется, что мьютекс будет освобожден при выходе из блока, независимо от того, каким образом этот выход произойдет.

Глава 12

Исключения

12.1 Общие принципы

Исключения языка C++ представляют собой мощный инструмент для сигнализирования и обработки ошибок. Как и любой мощный механизм, при неправильном применении он может принести больше проблем, чем решить задач.

Исключениям в стандарте C++ 03 [C++ 03] посвящен раздел 15.

12.2 Генерация и обработка исключений

Для генерации исключения в языке предназначен оператор **throw**. Его параметром является объект исключения, идентифицирующий произошедшее событие.

Обработчики исключений формируются при помощи блоков **try**/**catch**. Генерация исключения внутри блока **try** приводит к поиску подходящего по типу обработчика. Временный объект исключения, переданный оператору **throw** существует все время до завершения обработчика данного исключения [C++ 03, параграф 15.1.4]. Если обработчик исключения выполняет только частичную работу по обработке он может повторно генерировать то же исключение путем использования оператора **throw** без параметров. В этой ситуации временный объект используется повторно [C++ 03, параграф 15.1.6].

В процессе поиска подходящего обработчика исключения вызывваются деструкторы для всех автоматичексих объектов, созданных с момента входа в соответствующий блок **try** [C++ 03, параграф 15.2.1]. Объекты при этом разрушаются в порядке, обратном их созданию. Для частично созданных объектов деструкторы будут вызваны для всех полностью сконструированных и еще не разрушенных подобъектов [C++ 03, параграф 15.2.2].

Данных процесс называется «раскрутка стека» (stack unwinding). Если какой-либо деструктор в процессе раскрутки стека завершается с исключением, то происходит вызов функции terminate() [C++ 03, параграф 15.2.2].

12.2.1 Иерархии исключений

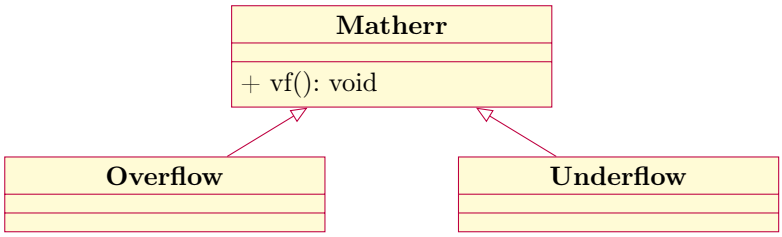
В качестве объекта исключения может фигурировать практически любой тип языка C++, но обычно используются экземпляры классов. Это позволяет передавать дополнительную информацию о событии, вызвавшем исключение, а также построить иерархию исключений, обобщая информацию о событии.

Например:

```
 1  class Matherr
 2  {
 3  public:
 4    virtual void vf();
 5  };
 6
 7  class Overflow:
 8    public Matherr
 9  {
10    ...
11  };
12
13  class Underflow:
```

```
14    public Matherr
15  {
16    ...
17  };
18
19  void f()
20  {
21    try {
22      g();
23    } catch (const Overflow &) {
24      std::cout << "Got an overflow" << std::endl;
25    } catch (const Matherr &) {
26      throw;
27    }
28  }
```

В данном примере имеется следующая иерархия исключений:



В строке 21 начинается блок **try** и все исключения, сгенерированные внутри него будут обрабатываться при помощи ассоциированных обработчиков. Строка 23 содерджит описание для обработчика исключений типа Overflow, а в строке 25 — для исключений типа Matherr. Таким образом, если функция g() выбросит исключение Overflow, то оно будет обработано первым обработчиком. Исключение Underflow обрабатывается вторым обработчиком, так как оно унаследовано от Matherr. Все остальные исключения не будут обработаны здесь и поиск обработчиков продолжится во внешних блоках по отношению к вызову f().

Подобное приведение к базовому классу объекта исключения при выборе обработчика выполняется автоматически. Но стандарт четко оговаривает, что наследование должно быть открытым, а приведение к базовому классу — непротиворечивым [C++ 03, параграф 15.3.3].

Выбор обработчика выполняется последовательно. Это значит, что если поменять в примере обработчики местами, то обработчик Overflow никогда не будет вызван, так как Matherr является открытой и единственной (а значит и непритиворечивой) базой для Overflow.

Построение иерархии классов исключений обеспечивает корректную детализацию информации об ошибках. В примере выше имеется обобщенный класс Matherr, он будет использоваться теми клиентами, которых не интересуют подробности происходящего. Те же клиенты, кто хочет обрабатывать различные исключительные ситуации по-разному, смогут воспользоваться его наследниками Overflow и Underflow.

12.3 Безопасность исключений

Фактически, исключения представляют собой механизм нелокального перехода (вне текущего блока и/или функции). Зачастую это вдеет к различным проблемам.

Рассмотрим простой пример:

```
1  string EvaluateSalaryAndReturnName(Employee e)
2  {
3    if ((e.Title() == "CEO") || (e.Salary() > 100000))
4    {
5      cout << e.First() << " " << e.Last()
6           << " is overpaid" << endl;
7    }
8
9    return e.First() + " " + e.Last();
10 }
```

В данном примере есть 3 основных пути исполнения:

1. e.Title() равен «CEO».
2. e.Title() не равен «CEO» и e.Salary() больше 100 000.
3. Ни то, ни другое не верно.

В случае же генерации исключений, количество путей исполнения вырастает:

1. Employee передается по значению, поэтому будет вызван конструктор копирования, который может выбросить исключение.
2. e.Title() возвращает скорее всего std::string, так как возврат простого указателя не дает возможности корректно сравнивать строки. Возврат объекта по значению может генерировать исключение в конструкторе копирования, например, для std::string это может быть std::bad_alloc. Если же возвращается ссылка на константный объект, исключения про возврате значения не будет. В любом случае, функция e.Title() может сгенерировать исключение в процессе работы.
3. Аналогично могут вести себя функции e.Salary(),
4. e.First() и
5. e.Last().
6. Оператор вывода в поток также может генерировать исключение при ошибке вывода.
7. Оператор объединения строк также может генерировать исключение, так как в общем случае он размещает память динамически.
8. Операторы ==, > и || могут быть перегруженными и, как следствие, тоже бросать исключения.
9. Литеральные значения могут требовать преобразований типов в пользовательские и тоже генерировать исключения.

Итого, учитывая множественные вызовы, путей исполнения может быть 23.

В общем случае, код, работающий с исключениями должен быть готов к их генерации и корректно обрабатывать все ресурсы и инварианты. При этом присутствует 2 важных момента:

1. Если в процессе раскрутки стека какой-либо деструктор завершится генерированием исключения, и то программ будет завершена вызовом terminate() [C++ 03, параграф 15.2.3].
2. Если подходящего обработчика исключения найдено не будет, программа будет завершена вызовом terminate(). Будет ли при этом раскручиваться стек определяется реализацией [C++ 03, параграф 15.3.9].

12.3.1 Гарантии безопасности

Для обеспечения создания корректных программ Дэйв Абрахамс сформулировал гарантии безопасности исключений:

Базовая В случае генерации исключения все ресурсы корректно освобождаются, все объекты могут быть использованы или корректно разрушены, но их состояние непредсказуемо.

Строгая В случае генерации исключения состояние программы не меняется. В общем случае это предполагает, что реализована семантика «commit or rollback».

Отсутствия Функция ни при каких условиях не генерирует исключений.

Очевидно, что базовая гарантия является минимально применимой. Ее достаточно в тех случаях, когда вызывающий код в состоянии справится со всеми возникающими проблемами. Все функции должны быть написаны так, чтобы она выполнялась.

Строгая же гарантия может выдвигать дополнительные требования к дизайну классов. Например, для контейнеров строгая гарантия ведет к тому, что при генерации исключения не только данные контейнера не изменятся, но то, что все ссылки на его содержимое, включая итераторы, остаются корректными.

Существенным моментом является то, что никакая безопасность исключений невозможна без некоторых функций, которые гарантируют отсутствие исключений. Например, деструкторы не должны генерировать исключения.

При разработке библиотек присутствует требование «нейтральности к исключениям». Если библиотека не обрабатывает исключение, полученное с нижнего уровня или от клиентского кода (например, для контейнера это может быть исключение из конструктора копирования объекта, хранящегося в контейнере), то это исключение должно быть передано без изменений. В этом случае клиентский код может решить сам, как его обрабатывать.

12.3.2 Принципы безопасного дизайна

Безопасность с точки зрения исключений есть неотъемлемая часть дизайна программы. Ее нельзя откладывать «на потом», так как обеспечение этой безопасности зачастую приводит к масштабным изменениям.

В качестве базового принципа дизайна обычно используется строгое разделение кода, способного генерировать исключения, и кода, изменяющего состояние программы.

Например, для оператора присваивания:

```
1  class Widget
2  {
3  public:
4    ...
5
6    Widget & operator =(const Widget &);
7
8    ...
9  private:
10   T1 t1_;
11   T2 t2_;
12 };
13
14 Widget & Widget::operator =(const Widget & rhs)
15 {
16   t1_ = rhs.t1_;
17   t2_ = rhs.t2_;
18
19   return *this;
20 }
```

Простая реализация удовлетворяет в лучшем случае базовой гарантии: если при присваивании t2_ генерируется исключение, то t1_ уже изменен, а значит строгой гарантии код не удовлетворяет. Попытка вернуть его старое значение также может провалиться.

Более того, данный код потенциально обладает еще одной проблемой: он неправильно обрабатывает присваивание самому себе.

Разделив операции в соответствии с принципом безопасного дизайна можно предложить следующую реализацию:

```
1  class Widget
2  {
3  public:
4    ...
5
6    Widget & operator =(const Widget &);
7
8    void swap(Widget & rhs) throw();
9    ...
10 private:
11   T1 t1_;
12   T2 t2_;
13 };
14
15 namespace std
16 {
17   template <>
18   inline void swap(Widget & lhs, Widget & rhs)
19   {
20     lhs.swap(rhs);
21   }
22 }
23
24 void Widget::swap(Widget & rhs)
```

```
25 {
26   std::swap(t1_, rhs.t1_);
27   std::swap(t2_, rhs.t2_);
28 }
29
30 Widget & Widget::operator =(const Widget & rhs)
31 {
32   Widget temp(rhs);
33
34   swap(temp);
35
36   return *this;
37 }
```

В данном варианте все операции внутри оператора присваивания разделены на 2 части:

- создание копии объекта, которое может сгенерировать исключение и
- обмен копии с самим собой, которое не бросает искючений.

Таким образом оператор присваивания предоставляет строгую гарантию безопасности исключений.

Данный вариант плохо работает в случае, если T1 или T2 не предоставляют безопасных операций обмена. В этой ситуации возможно применение умных указателей:

```
1  class Widget
2  {
3  public:
4    ...
5
6    Widget & operator =(const Widget &);
7
8    void swap(Widget & rhs) throw();
9    ...
10 private:
11   std::auto_ptr< T1 > t1_;
12   std::auto_ptr< T2 > t2_;
13 };
```

Возможен также более радикальный вариант:

```
1  class Widget
2  {
3  public:
4    ...
5
6  private:
7    class impl;
8    std::auto_ptr< impl > impl_;
9  };
10
11 class Widget::impl
12 {
13 private:
14   T1 t1_;
15   T2 t2_;
16 };
```

Вызов функций

Вызов функций является еще одним местом, где возможны проблемы. Например:

```
1  class A;
2  class B;
3
4  void foo(const A * a, const B * b);
5
6  void bar()
7  {
8    foo(new A, new B);
9  }
```

В точке вызова функции foo() динамически создаются 2 объекта. Предположим, что первый аргумент вычисляется в первую очередь. В этом случае, если при создании объекта для второго аргумента произойдет исключение, то объект для первого аргумента утечет.

Умные указатели должны решить эту проблему:

```
1  class A;
2  class B;
3
4  void foo(std::auto_ptr< const A > a,
5           std::auto_ptr< const B > b);
6
7  void bar()
8  {
9    foo(std::auto_ptr< const A >(new A),
10       std::auto_ptr< const B >(new B));
11 }
```

К сожалению, данная попытка не решает проблемы. Как было отмечено, порядок вычисления аргументов функции не определен [C++ 03, параграф 5.2.2.8]. Поэтому возможен такой порядок вычислений при вызове функции:

1. Создается объект типа A.
2. Создается объект типа B.
3. Конструируется std::auto_ptr для первого аргумента.
4. Конструируется std::auto_ptr для второго аргумента.
5. Вызывается функция foo().

При таком порядке вычислений очевидно, что предложенное исправление неверно. Корректным будет создать и проинициализировать умные указатели до вызова функции:

```
1  class A;
2  class B;
3
4  void foo(std::auto_ptr< const A > a,
5           std::auto_ptr< const B > b);
6
7  void bar()
8  {
9    std::auto_ptr< const A >(new A);
10   std::auto_ptr< const B >(new B);
11
12   foo(a, b);
13 }
```

Для boost::shared_ptr есть дополнительный способ безопасно и эффективно создать умный указатель. Для этого служит функция boost::make_shared< T >(). Она принимает аргументы конструктора объекта типа T и создает его экземпляр динамически. Полученный указатель безопасно помещается в boost::shared_ptr и возвращается из функции. Например, для данного примера, где объекты конструируются конструкторами по умолчанию:

```
 1 class A;
 2 class B;
 3
 4 void foo(boost::shared_ptr< A > a,
 5           boost::shared_ptr< B > b);
 6
 7 void bar()
 8 {
 9   foo(boost::make_shared< A >(),
10       boost::make_shared< B >());
11 }
```

Глава 13

Физический дизайн

13.1 О логическом и физическом дизайне

Рассмотренные ранее вопросы касались исключительно логического дизайна системы: классы, функции, управление ресурсами и обработка ошибок. Это важные элементы проектирования системы, так как при правильном применении вместе с другими техниками они обеспечат архитектуру, соответствующую росту проекта. С другой стороны, по мере роста проекта все тонкие моменты начинают проявляться острее. Начиная с определенного момента становится значимым физический дизайн системы. Важное замечание: под «физическим дизайном системы» в данном случае не пытаются скрыться проблемы аппаратных средств, на которых будет работать код, или тонкости развертывания. Строго говоря, компоненты, классы и прочее это элементы логического дизайна системы, поэтому под физическим дизайном подразумевается все связанное с его кодовой базой.

Недостатки и ошибки в физическом дизайне системы проявляются в виде растущих паразитных зависимостей и роста времени построения проекта.

К сожалению, тема физического дизайна крупных систем на C++ очень плохо освещена. Информация разрознена и единственная серьезная работа на эту тему [Lak96] выпущена в 1996 году.

13.2 Составные элементы проекта

Стандарты C++ 1998, 2003 и 2011 годов рассматривают программу как единый монолитный процесс. В то же время значительная часть кода любой системы может быть разделена с другими системами, показательными примерами являются стандартные библиотеки C и C++, которые используются практически во всех программах.

Достаточно широко известно, что библиотеки бывают статические и динамические. Простым способом различить их будет способ использования: содержимое статической библиотеки присоединяется к программе редактором связей (линкером), код и данные динамической библиотеки отсутствуют внутри программы и загружаются в процесс системными компонентами. Сами стандарты ничего не говорят об устройстве библиотек и о процессе их связывания. Влияние динамических библиотек будет рассматриваться дальше, в разделе о системных аспектах.

На рис. 13.1 показан процесс построения простой программы состоящей из двух файлов реализаций.

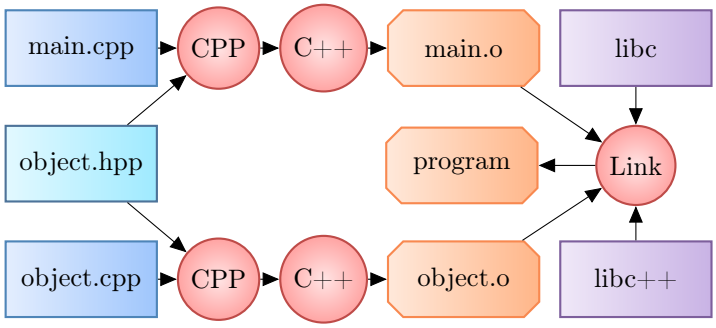


Рис. 13.1: Построение простой программы

В данном примере файл `main.cpp` содержит функцию main(), а файлы `object.hpp` и `object.cpp`

определение и реализацию некоторого объекта.

Процесс построения программы проще всего описывается с конца:

- Программа с именем файла `program` формируется редактором связей (Link).
- Для построения программы требуются 2 объектных файла `main.o` и `object.o`. Эта информация непосредственно известна из проекта.
- Дополнительно на этапе построения программы требуются стандартные библиотеки C (libc) и C++ (libc++). Конкретные имена файлов этих библиотек зависят от компилятора и платформы. Две эти зависимости носят неявный характер и формируются компоновщиком на основании того, что программа написана на языке C++.
- Каждый из объектных файлов является независимой единицей трансляции и формируется компилятором C++.
- На вход компилятора C++ подается текст, сформированный из соответствующего исходного файла с суффиксом «cpp» при помощи препроцессора.
- Препроцессор C++ читает исходный файл и вместо каждой директивы **#include** помещает содержимое соответствующего файла заголовка, в данном случае `object.hpp`.

В случае применения каких-либо дополнительных инструментов, генерирующих код на C или C++, возможен один из двух вариантов с рис. 13.2 или рис. 13.3.

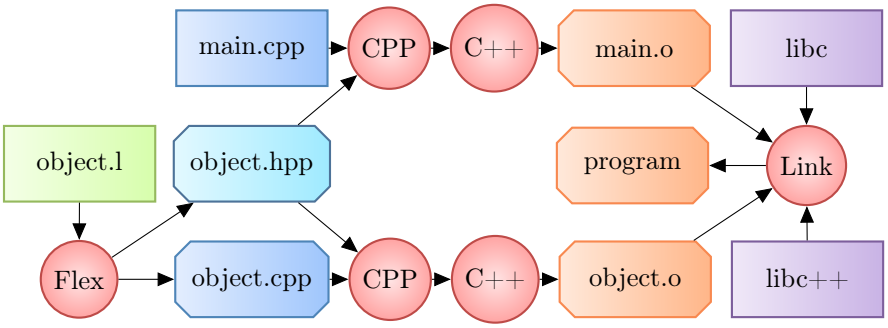


Рис. 13.2: Построение программы с использованием Flex

В данном случае генератор кода (Flex) генерирует реализацию лексического анализатора. Исходный код этого анализатора непосредственно передается компилятору.

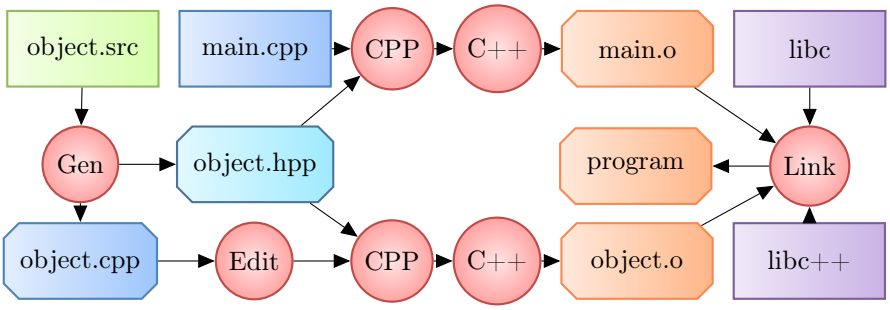


Рис. 13.3: Построение программы с использованием генератора кода

Второй вариант более изощренный: сгенерированный код подвергается ручному редактированию, например, для внесения реализаций функций.

Предпочтительным способом реализации генераторов кода является, очевидно, первый. Это связано, в первую очередь, с тем, что если исходный файл для генератора меняется, код должен быть перегенерирован. Соответственно, меняются и файлы, которые подверглись правке после генерации, поэтому требуется аккуратное слияние свежесгенерированного текста и внесенных изменений. В отличие от этого, исходный файл для Flex содержит полное описание того, что должно быть сгенерировано, и выход генератора последующей правки не требует.

13.3 Зависимости

Очевидно, что если несколько доработать рис. 13.1, то можно построить граф зависимостей для программы. Полученный вариант изображен на рис. 13.4, зависимости изображены пунктирными линиями.

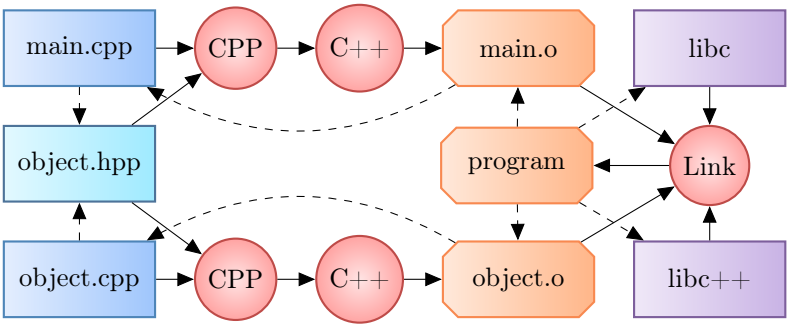


Рис. 13.4: Зависимости простой программы

Для простоты, зависимости от заголовочных файлов, включенных в `object.hpp`, не показаны.

Данный граф зависимостей показывает не только что должно быть перекомпилировано, если что-либо меняется. Из него также видно, каков объем работы для компилятора. Для проектов на C это обычно не является проблемой, в то время как на C++ очень легко создать такой класс, время компиляции которого будет значительным. Как следствие, все, что зависит прямо или опосредованно от заголовочного файла этот класса, также будут компилироваться медленно.

В связи с этим требуется уменьшать зависимости настолько, насколько можно. В этой ситуации можно воспользоваться крайне эффективной техникой предварительного объявления.

13.3.1 Нюансы заголовочных файлов

Заголовочные файлы это неотъемлемая часть любого проекта на C и C++, а равно и Objective C. Они заменяют собой интерфейсную часть модулей, которые отсутствуют в языке. Содержимое заголовочных файлов помещается в единицу трансляции препроцессором.

С заголовочными файлами связан важный момент. Язык C++ не допускает повторное определение любой сущности внутри единицы трансляции. В то же время, невозможно предотвратить многократное включение файла, если он включается опосредованно внутри других заголовочных файлов. Для предотвращения повторных определений существует 2 техники:

1. нестандартный способ при помощи директивы #**pragma** once. Этот вариант поддерживается Microsoft Visual C++ и GCC как минимум, но может отсутствовать в других компиляторах.

2. стандартный способ при помощи «header guards».

TODO: Вступление про header guards

Различают внутренние и внешние header guards. Внутренние используются чаще всего: заголовочный файл определяет и проверяет некий символ препроцессора для определения повторного включения.

Внешний guard проверяет этот символ в точке использования директивы #**include**. Для языка C это может иметь смысл, так как в заголовочных файлах на C крайне редко встречаются конструкции, время компиляции которых заметно. Поэтому время, потраченное на открытие файла и обработку внутреннего guard'а может быть заметно. С другой стороны, в C++ заголовочные файлы могут быть крайне сложны и на этом фоне время обработки внутреннего guard'а незаметно, в то время как проверка внешнего guard'а нарушает инкапсуляцию.

13.3.2 Минимизация зависимостей

Очевидно, что в некоторых случаях невозможно избежать включения полного определения какого-либо класса. Рассмотрим пример:

```
1 class Derived:
2   public Base1,
3   private Base2
4 {
5 public:
```

```
 6    void foo(B a);
 7    B bar(const C & c, const D * d);
 8
 9  private:
10    std::ifstream & input_;
11    std::ostream & output_;
12
13    const A a_;
14
15    std::vector< std::string > & mapping_;
16  };
```

Какие заголовочные файлы необходимо использовать в данном файле?

Для наследования требуется знать полное определение класса, поэтому `Base1.hpp` и `Base2.hpp` необходимы. Причина этого довольно проста: компилятору как минимум требуется знать размер базового класса и доступные в нем имена.

Класс A используется в качестве типа поля класса Derived. Для этого требуется знать его размер, поэтому для класса A требуется включение `A.hpp`.

Поля input_ и output_ являются ссылками. Для создания ссылки, а равно и указателя, знать детали не требуется, так как размер указателя не зависит от типа. Поэтому для std::ifstream и std::ostream достаточно предварительного объявления. И здесь есть тонкий момент: std::ostream и std::ifstream являются экземплярами шаблона и реализованы внутри стандартной библиотеки C++. Поэтому необходимо использовать файл `iosfwd` для получения предварительных объявлений от стандартной библиотеки потоков.

Действуя по аналогии, очевидно, что классы C и D также не требуют включения соответствующих заголовков — эти классы используются в виде ссылки и указателя соответственно, поэтому предварительного объявления достаточно.

std::vector<> является шаблоном, поэтому его определение требуется для создания экземпляра std::vector< std::string >, равно как требуется и определение std::string. Поэтому должны быть включены файлы `vector` и string.

Что касается класса B, то он используется как параметр и как возвращаемое значение. Поэтому не требуется его полное определение и достаточно предварительного объявления.

Таким образом, нам потребуются следующие заголовочные файлы:

- `iosfwd`
- `vector`
- `string`
- `Base1.hpp`
- `Base2.hpp`
- `A.hpp`

Остальные заголовочные файлы будут использованы в файле реализации класса.

Заметим, что обычно крайне непрактично иметь все поля в виде указателей, поэтому для «тяжелых» классов идиома «Указатель на реализацию» является мощной техникой сокращения внешних зависимостей.

Особо осторожным необходимо быть со всем, что приходит из других компонентов. В общем случае неизвестно, чем является имя типа — классом или псевдонимом экземпляра шаблона. Поэтому нельзя предварительно объявлять «чужие» имена, а для клиентов своего компонента разумно предлагать заголовочный файл с предварительными объявлениями.

13.3.3 Паразитные зависимости

Рассмотрим некий компонент «Component». У него присутствует внешний интерфейс в виде классов Object1 и Object2 (рис. 13.5). Для реализации компонента внутри используются классы Impl1 и Impl2. Для информирования об ошибках используется класс Error. Дополнительно имеется реализация итератора по элементам внешнего интерфейса в файле `Iterator.hpp`.

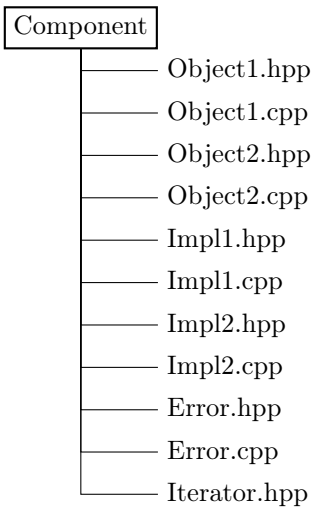


Рис. 13.5: Пример компонента

Очевидно, что при текущем размещении файлов клиент имеет полный доступ ко всем деталям реализации. Например, он может воспользоваться заголовочными файлами `Impl1.hpp` или `Impl2.hpp`, создав таким образом зависимость, который быть не должно. Отсутствие разделения на внешний и внутренний интерфейсы зачастую провоцирует клиентов компонента на подобные паразитные зависимости.

Избежать этого достаточно просто, если следовать простым правилам:

1. Четко разделять внутренний и внешний интерфейсы. Внешний интерфейс должен находиться в отдельном каталоге файловой системы так, чтобы все внутренние подробности компонента не были доступны при добавлении этого каталога в список путей поиска заголовков.

2. Внешний интерфейс должен находиться в таком каталоге, что имена включаемых файлов на стороне клиента содержали имя компонента. Это предотвращает коллизии имен файлов, например, имя `Error.hpp` достаточно общее, чтобы пересечься с другим компонентом. Включение имени компонента в имя (#**include** <Component/Error.hpp>) снижает вероятность этого.

3. Внутренний интерфейс делится на 2 части: детали реализации, необходимые заголовочным файлам внешнего интерфейса, и все остальные. Первые *должны* быть доступны клиентам, в то же время необходимо четко отделить их от непосредственно интерфейса. Поэтому имеет смысл разместить их в подкаталоге, например, `detail`.

4. Внутри компонента все заголовочные файлы внешнего интерфейса должны включаться так же, как это будет делать клиент — с использование угловых скобок и имени компонента.

5. Все заголовочные файлы внешнего интерфейса должны быть самодостаточными — клиент не должен думать о том, что еще надо включить перед заголовочным файлом. Внутренние заголовочные файлы также желательно иметь самодостаточными. Просто способ обеспечить это — файл реализации должен включать первым свой заголовочный файл.

6. Необходимо избегать внесения чего бы то ни было в глобальную область видимости. Поэтому в заголовочных файлах недопустимо применять конструкцию **using namespace** в глобальной области видимости.

Руководствуясь этими правилами, компонент может быть реструктурирован следующим образом:

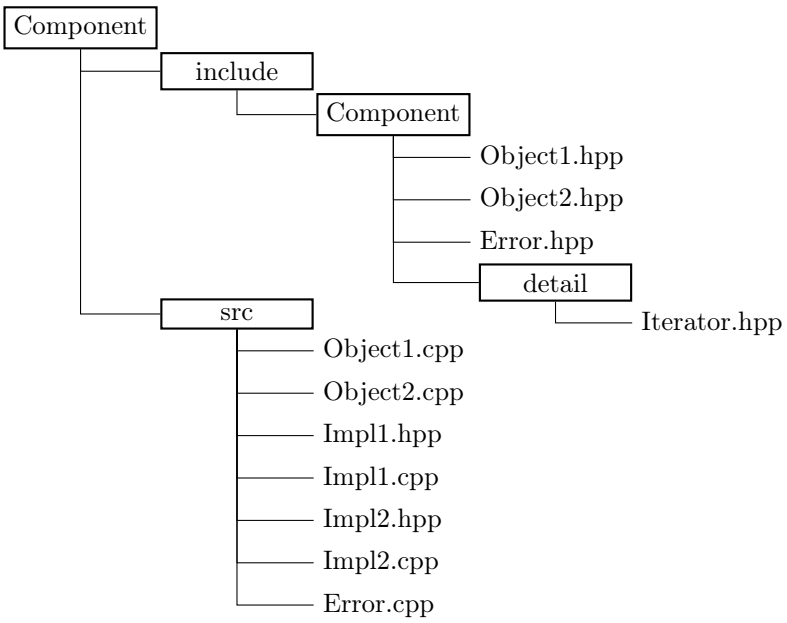


Рис. 13.6: Переработанный компонент

13.4 Влияние платформы

С точки зрения стандарта C++ модулей не существует. С другой стороны, практически все платформы предлагают ту или иную реализацию модулей. Для Microsoft Windows® это динамически загружаемые библиотеки (Dynamic Load Libraries, DLL). В POSIX системах используется понятие shared object. В целом, они реализуют сходные концепции разделяемого кода, но при этом отличаются в деталях. Далее они будут просто называться модулями.

Без дополнительных ухищрений, shared objects ведут себя так же, как и обычные объектные файлы на этапе связывания. Все символы разрешаются в порядке загрузки модулей, все символы являются видимыми. Из этого следует:

- Видны все символы, за исключением глобальных символов с внутренним связыванием (с модификатором **static**). Таким образом, все внутренние детали реализации видны и могут конфликтовать с символами в других модулях. Как следствие, применение неименованных пространств имен необходимо для предотвращения таких конфликтов.
- Символы могут дублироваться в разных модулях. Например, экземпляры шаблонов и встраиваемых функций. Это не составляет проблемы во время исполнения.

С другой стороны, DLL ведут себя совершенно по-другому:

- По умолчанию, символы являются внутренними и вне модуля не видны. Для придания символу видимости извне требуется специальный квалификатор (__declspec(dllexport)).
- По умолчанию, символы являются локальными, а не импортируемыми из других модулей. Для импорта необходим специальный квалификатор (__declspec(dllimport)).
- Символы связываются во время исполнения по имени символа *и имени модуля*. Это приводит к тому, что символы могут иметь различное значение в разных модулях, и проблемы не возникает до тех пор, пока один модуль не попытается использовать оба варианта символа одновременно.
- Разные DLL могут использовать различные экземпляры кучи и Free Space, поэтому требуется особое внимание к управлению памятью.

Исходя из всего вышесказанного, разумно потребовать следующее:

- Имена символов должны быть уникальны в системе. Это соответствует видению стандарта и повышает переносимость системы на разные платформы.
- Необходимо скрывать детали реализации настолько, насколько это возможно. Реализация этого на POSIX платформах требует применения дополнительного знания об управлении видимостью.

- Управление памятью должно учитывать возможную несовместимость реализаций кучи и Free Space между разными модулями. Как начальный вариант, использование shared_ptr и boost::intrusive_ptr должно решить проблему.

Все детали реализации сокрытия символов на платформе POSIX выходят за рамки данной статьи, поэтому приведена идея:

- Все символы по-умолчанию декларируются с visiblity=hidden. Это эффективно скрывает все детали реализации.
- Символам внешнего интерфейса, включая классы исключений, назначается visibility=default при помощи специфичного для компилятора ключевого слова.

Вследствие таких шагов на POSIX платформах таблицы символов модулей становятся меньше и, как следствие, поиск символов происходит быстрее.

Глава 14

Потоки ввода-вывода и локализация

14.1 Потоки ввода-вывода

14.1.1 Мир C

Практически перед любой программой стоит проблема ввода-вывода, так как входные данные необходимо откуда-то получить, а вычисленные результаты вывести. Каждый язык программирования решает эту проблему по-своему.

В мире C язык программирования сам по себе не содержит никаких средств ввода-вывода. Вместо этого стандартная библиотека C предоставляет набор средств для решения этой задачи.

Основным средством вывода в мире C является семейство функций printf(). Библиотека предоставляет варианты этой функции на все случаи жизни:

- вывод на терминал,
- вывод в строку,
- вывод в произвольный файл.

В общем случае, функция семейства выглядит и используется следующим образом:

```
1  /* In the stdio.h */
2  int printf(const char * format, ...);
3  int fprintf(FILE * stream, const char * format, ...);
4  int sprintf(char * str, const char * format, ...);
5  int snprintf(char * str, size_t size,
6               const char * format, ...);
7
8  ...
9  char * filename = ...
10 long int filesize = ...
11
12 printf("The file \"%s\" has size %5ld bytes\n",
13        filename, filesize);
```

В данном примере в стандартный канал вывода выводится строка с именем файла и его размером. Размер файла занимает поле минимум 5 символов, значение выровнено по правому краю и слева дополнено пробелами. Значения для вывода передаются после форматной строки. Точный тип параметра определяется в спецификаторе формата. В данном случае, символ «s» означает указатель на строку символов, а «ld» означает вывод десятичного числа размера long.

Для ввода данных служит семейство функций scanf. Они работают в обратную сторону и, как следствие, в качестве параметров после форматной строки принимают указатели на соответствующие переменные:

```
1  /* In the stdio.h */
2  int scanf(const char * format, ...);
3  int fscanf(FILE * stream, const char * format, ...);
4  int sscanf(const char * str, const char * format, ...);
5
```

```
6  ...
7  char filename[256];
8  long int filesize = 0;
9
10 scanf("%s\n%ld\n", filename, &filesize);
```

В данном примере на входе ожидается 2 строки. На первой строке указано имя файла, а на второй — десятичное число.

Достоинствами данного способа ввода-вывода являются присутствие общего вида строки в тексте программы и контроль за форматом данных. На этом очевидные достоинства заканчиваются и начинаются существенные недостатки:

- количество ожидаемых аргументов определяется форматной строкой, их количество должно быть не меньше, чем указано в форматной строке;
- тип каждого аргумента привязан к соответствующему элементу форматной строки, несоответствие типов спецификатора формата и фактического параметра ведет в лучшем случае к неправильной обработке данных, а в худшем — к неопределенному поведению;
- не существует способа научить эти функции работать с объектами C++.

14.1.2 Ввод-вывод в мире C++

Такое положение не могло существовать при развитии языка. Как следствие, язык получил библиотеку потоков ввода-вывода, которые призваны решить эти проблемы.

Аналогичные примеры в мире C++ будут выглядеть так:

```
1  #include <iostream>
2
3  ...
4
5  std::string filename;
6  long int filesize;
7
8  std::cin  >> filename >> filesize;
9  std::cout << "The file \"" << filename
10           << "\" has size " << filesize
11           << " bytes\n";
```

std::cin, std::cout, std::cerr и std::clog являются стандартными и определенными в библиотеке потоками ввода, стандартного вывода, стандартного вывода ошибок и буферизованного вывода ошибок.

Для обеспечения гибкого взаимодействия с внешней средой библиотека реализована в виде набора шаблонных классов. Все шаблоны принимают 2 параметра: тип символа и характеристики (traits) символа, которые по умолчанию используются стандартные (std::char_traits< Char >).

В библиотеке присутствуют следующие основные типы потоков:

basic_istream<> поток ввода, std::istream представляет собой псевдоним с фиксированным типом символа **char**.

basic_ostream<> поток вывода, std::ostream также является псевдонимом с типом символа **char**.

basic_iostream<> поток ввода-вывода, унаследованный от потоков ввода и вывода, аналогично предыдущим классам, псевдоним для **char** также присутствует под именем std::iostream.

Обычно создавать экземпляры приведенных выше классов не приходится: экземпляры стандартных каналов создает библиотека при инициализации, а использование пользовательских средств ввода-вывода, таких как сеть, встречается существенно реже.

Наиболее часто применяются классы, работающие со строками и файлами в качестве средств ввода-вывода. Библиотека содержит 2 набора шаблонных классов для этого, с базовыми именами fstream для файлов и stringstream для строк. Как и в базовых классов потоков, полные имена и псевдонимы можно вычислить следующим образом:

1. Для шаблонных классов префиксом имени будет basic_.

2. После префикса у шаблонного класса или в начале имени у псевдонима указывается направление данных: i для ввода, o для вывода и никакого для двунаправленного потока (исключение — для базового двунаправленного потока используется io).

3. Далее используется базовое имя потока, определяющее его тип.

4. Для шаблонного класса в конце указываются 2 параметра шаблона: тип символа и характеристики символа.

Например, для шаблона потока вывода в файл имя будет basic_ofstream<>, а псевдоним для потока ввода из строки символов — istringstream.

Потоковые классы обладают некоторыми важными свойствами:

- Потоки обладают состоянием. Состояние потока определяется установленными флагами goodbit, eofbit, failbit и badbit.

- Потоки не генерируют исключений по умолчанию. Генерацию исключений можно включить, но стандарт не требует, чтобы объект исключения содержал информацию об ошибке.

- Операции ввода-вывода попадают либо в категорию форматированного ввода-вывода (при помощи операторов << и >>), либо неформатированного ввода-вывода (при помощи функций потоков).

- Для управления форматированием потоки содержат форматные флаги. Менять их состояние можно при помощи функций потоков или манипуляторов.

Флаги состояния отражают то, как прошла последняя операция с потоком. Если в процессе операции возникла ошибка, то будет установлен failbit или badbit. Если установлен badbit, то поток находится в неопределенном состоянии и восстановление его корректной работы может быть невозможным. В противоположность этому, failbit подразумевает такую ошибку, после которой возможно восстановление. При попытке чтения за концом потока выставляется eofbit. Важным моментом является то, что он выставляется не тогда,когда прочитан последний символ потока, а при следующей попытке чтения.

Быстрым способом проверить состояние потока является применение перегруженных операторов **operator** !() и **operator void** *(). Если состояние потока корректно (установлен goodbit), то оператор отрицания возвращает **false**, а оператор приведения не 0.

Пример:

```
1  #include <ifstream>
2
3  int main(int, char *[])
4  {
5    std::ifstream source("data.txt");
6
7    if (!source) {
8      std::cerr << "The file \"data.txt\" is not "
9                   "accessible\n";
10     return 1;
11   }
12
13   while (!source.eof()) {
14     int value;
15
16     source >> value;
17   }
18
19   return 0;
20 }
```

Данная программа читает из файла с именем «data.txt» целые числа, разделенные пробельными символами. Если файл не был открыт успешно, в стандартный поток ошибок выводится сообщение и программа завершается на кодом возврата 1.

14.1.3 Потоковые манипуляторы

Для управления форматированным вводом-выводом используются манипуляторы. Манипуляторы предоставляют удобный доступ к функциональности потокового класса, например, следующие примеры выводят одно и то же, но понятность кода совершенно разная:

```
1 std::cout.setf(std::ios_base::left,
2                std::ios_base::adjustfield);
3 std::cout.setw(6);
4 std::cout << "Name:" << "Anthony" << "\n";
5 std::cout.setw(6);
6 std::cout << "Age:" << 25 << "\n";
```

```
1 std::cout << std::left << std::setw(6)
2           << "Name:" << "Anthony" << "\n"
3           << std::setw(6) << "Age:" << 25 << "\n";
```

Основные манипуляторы приведены в таблице(CharT представляет собой тип символа потока):

Название	Тип потока	Назначение
`endl`	Output	Выводит в поток перевод строки и выгружает (flush) буфер потока.
`ends`	Output	Выводит в поток символ с кодом 0.
`flush`	Output	Выгружает буфер потока.
`ws`	Input	Пропускает пробельные символы в потоке до первого непробельного или до конца потока.
`setw(w)`	Input/Output	Устанавливает ширину поля следующего значения.
`setfill(c)`	Input/Output	Устанавливает заполнитель для поля вывода.
`left`, `right`, `internal`	Input/Output	Выравнивание вывода в поле: по левому краю, по правому краю и знак слева, значение справа соответственно.
`boolaplha`	Input/Output	Устанавливает вывод булевских значений в виде текста.
`noboolalpha`	Input/Output	Устанавливает вывод булевских значений в виде чисел 0 и 1.
`skipws`	Input	Включает пропуск пробельных символов при чтении.
`noskipws`	Input	Выключает пропуск пробельный символов при чтении.
`dec`, `oct`, `hex`	Input/Output	Устанавливает систему счисления для целых чисел.
`showbase`	Input/Output	Включает вывод информации о системе счисления.
`noshowbase`	Input/Output	Выключает вывод информации о системе счисления.
`showpos`	Input/Output	Включает вывод знака для положительных чисел.
`noshowpos`	Input/Output	Выключает вывод знака для положительных чисел.
`uppercase`	Input/Output	Обеспечивает вывод чисел в верхнем регистре.
`nouppercase`	Input/Output	Обеспечивает вывод числе в нижнем регистре.
`scientific`	Input/Output	Вывод чисел с плавающей запятой в научном формате.
`fixed`	Input/Output	Вывод чисел с плавающей запятой в фиксированном формате.
`setprecision(s)`	Input/Output	Устанавливает точность вывода чисел с плавающей запятой.

14.1.4 Управление выводом объектов

Для встроенных и множества библиотечных типов операторы ввода и вывода определены,для пользовательских типов их необходимо определять самостоятельно.

В качестве примера рассмотрим следующий класс:

```
1  namespace sample
2  {
3    class Employee
4    {
5    public:
6      Employee(const std::string & first_name,
7        const std::string & last_name,
8        unsigned int salary);
9
10   private:
```

```
11      std::string first_name_;
12      std::string last_name_;
13      unsigned int salary_;
14    };
15  }
```

Задача будет выглядеть следующим образом: необходимо обеспечить вывод содержимого данного объекта в поток вывода в двух форматах — для чтения человеком и для сохранения в файл с последующим чтением при помощи какой-либо программы.

Данную задачу следует решать последовательно. Для начала необходимо определиться с тем, где и как должен быть определен оператор вывода в поток. Так как первым аргументом у него является поток, а не сам выводимый объект, этот оператор не может быть методом класса Employee. Простая логика подсказывает, что при стандартной записи вызова оператора не существует способа указать пространство имен, в котором должен вестись поиск. На первый взгляд, это ведет к тому, что оператор должен быть определен в глобальном пространстве имен. В то же время, засорение глобального пространства имен представляет проблему в любой большой программе. Поэтому стандарт содержит специальное указание на то, как должен производится поиск неквалифицированного имени. В кратком виде, компилятор обязан провести поиск вызываемой функции в пространствах имен аргументов этой функции [C++ 03, раздел 3.4.2]. Таким образом, наиболее правильным место будет определить оператор вывода в том же пространстве имен, что и класс Employee.

Второй момент определения оператора связан с тем, что ему требуется доступ к данным класса, которые обычно находятся в закрытой части. Существует 2 решения этой проблемы: назначить оператор вывода другом класса или добавить в класс метод вывода и вызвать его из оператора. Второй вариант потенциально дает возможность полиморфного вывода для наследников класса. В данном примере будет использован вариант с другом класса.

> Важное замечание: при объявлении кого-либо другом класса необходимо помещать его объявление и определение как можно ближе к классу, чтобы предотвратить непреднамеренный доступ к закрытым данным.

Простейший вариант оператора вывода в поток может выглядеть так:

```
1  std::ostream & sample::operator <<(std::ostream & out,
2      const sample::Employee & e)
3  {
4    using namespace std;
5
6    out << left << std::setw(14) << "First␣name:␣"
7        << e.first_name_ << "\n"
8        << setw(14) << "Last␣name:␣"
9        << e.last_name_ << "\n"
10       << setw(14) << "Salary:␣" << setw(20)
11       << right << dec << (e.salary_ / 100) << "."
12       << setw(2) << setfill('0') << (e.salary_ % 100)
13       << "\n";
14
15   return out;
16 }
```

Данный вариант оператора реализует только текстовый вывод объекта. Помимо этого, данный оператор имеет следующие проблемы:

- некорректно работает в ситуации, когда возникают ошибки ввода-вывода различного вида.
- влияет на состояние потока путем изменения символа заполнения и выравнивания.

Первая проблема решается созданием и проверкой состояния объекта sentry перед началом работы [C++ 03, раздел 27.6.2.5.1]. Вторая проблема требует реализации scope guard для состояния потока. Например:

```
1  namespace
2  {
3    class io_state_guard
```

```
4   {
5   public:
6     io_state_guard(std::ostream & s):
7       s(s),
8       fill_(s.fill()),
9       fmt_(s.flags())
10    {}
11
12    ~io_state_guard()
13    {
14      s.fill(fill_);
15      s.flags(fmt_);
16    }
17
18  private:
19    std::ostream & s;
20    char fill_;
21    ostream::fmtflags fmt_;
22  };
23 }
```

```
1  std::ostream & sample::operator <<(std::ostream & out,
2      const sample::Employee & e)
3  {
4    using namespace std;
5    ostream::sentry sentry(out);
6
7    if (sentry) {
8      io_state_guard state(out);
9
10     out << left << std::setw(14) << "First␣name:␣"
11         << e.first_name_ << "\n"
12         << setw(14) << "Last␣name:␣"
13         << e.last_name_ << "\n"
14         << setw(14) << "Salary:␣" << setw(20)
15         << right << dec << (e.salary_ / 100) << "."
16         << setw(2) << setfill('0') << (e.salary_ % 100)
17         << "\n";
18   }
19
20   return out;
21 }
```

Поддержка второго формата вывода требует дополнительной работы. Примерный вывод тех же данных в формате S-expression может выглядеть так:

```
1  (:first-name "John" :last-name "Doe" :salary 5500000)
```

Для реализации второго формата необходимо найти возможность передать дополнительные пользовательские флаги в поток. Класс ios_base предоставляет такой сервис. При помощи 2 методов:

```
long int & iword(int index);
void * & pword(int index);
```

есть возможность сохранять произвольное число или указатель внутри потока. Для предотвращения возможных коллизий индексы должны быть получены при помощи функции **int** xalloc(). Таким образом, пользовательский манипулятор для управления выводом может быть реализован так:

```
1  static const int employee_format_index = std::ios_base::xalloc();
2  static const long int EMPLOYEE_FORMAT_HUMAN_READABLE = 0;
3  static const long int EMPLOYEE_FORMAT_SEXPRESSION = 1;
4
```

```
5  template < typename Char, typename CharTraits >
6  std::basic_ostream< Char, CharTraits > &
7  sample::sexpression(std::basic_ostream< Char, CharTraits > & out)
8  {
9    out.iword(employee_format_index) = EMPLOYEE_FORMAT_SEXPRESSION;
10
11   return out;
12 }
13
14 template < typename Char, typename CharTraits >
15 std::basic_ostream< Char, CharTraits > &
16 sample::human_readable(std::basic_ostream< Char,
17                                             CharTraits > & out)
18 {
19   out.iword(employee_format_index) =
20     EMPLOYEE_FORMAT_HUMAN_READABLE;
21
22   return out;
23 }
```

Остается только проверить флаг и выбрать вариант вывода:

```
1  std::ostream & sample::operator <<(std::ostream & out,
2      const Employee & e)
3  {
4    using namespace std;
5    ostream::sentry sentry(out);
6
7    if (sentry) {
8      io_state_guard state(out);
9
10     if (out.iword(employee_format_index)
11         == EMPLOYEE_FORMAT_HUMAN_READABLE) {
12       out << left << std::setw(14) << "First name: "
13           << e.first_name_ << "\n"
14           << setw(14) << "Last name: "
15           << e.last_name_ << "\n"
16           << setw(14) << "Salary: " << setw(20)
17           << right << dec << (e.salary_ / 100) << "."
18           << setw(2) << setfill('0') << (e.salary_ % 100) << "\n";
19         } else {
20           // Assume S-Expression format
21           out << "(:first-name \"" << e.first_name_
22               << "\" :last-name \"" << e.last_name_
23               << "\" :salary " << e.salary_ << ")";
24         }
25   }
26
27   return out;
28 }
```

Но, как обычно, первый вариант страдает от проблем: если вызывающий код установил ширину поля, это повлияет только на вывод открывающей скобки и тега. Для избежания этой проблемы необходимо воспользоваться промежуточным потоком:

```
1  std::ostream & sample::operator <<(std::ostream & out,
2      const Employee & e)
3  {
4    ...
5      } else {
6        // Assume S-Expression format
7        // Conform to the outer field width by
```

```
 8         // using a temporary stream
 9         std::ostringstream str;
10
11         str << "(:first-name␣\"" << e.first_name_
12             << "\"␣:last-name␣\"" << e.last_name_
13             << "\"␣:salary␣" << e.salary_ << ")";
14
15         out << str.str();
16       }
17     ...
18   }
```

14.2 Интернационализация

В XXI веке современное приложение должно выглядеть естественно для пользователя из любой страны мира и на любом языке. Процесс подобной адаптации называется интернационализацией или i18n. Должны образом интернационализированная программа может быть подвергнута процессу локализации (l10n) для какого-либо конкретного языка в конкретной стране.

В процессе подобной адаптации ожидается, что числа вводятся и выводятся в том виде, как это принято в стране, названия месяцев соответствуют национальным и так далее. Например, в России в качестве десятичного разделителя используется запятая, а в длинных числах группы цифр по 3 разделяются пробелами. В США приняты другие правила: десятичный разделитель точка, а группы цифр разделяются запятыми.

Для поддержки подобной функциональности стандартная библиотека предоставляет объект локального контекста, называемый locale.

14.2.1 Аспекты локального контекста

Объект locale в теории содержит всю необходимую информацию об используемых сиволах, правилах форматирования и т.п. Вся эта информация разделена на различные аспекты (в стандарте используется слово «facet»).

Категория	Тип	Использование
numeric	`num_get<>`	Ввод числовых данных
	`num_put<>`	Вывод числовых данных
	`numpunct<>`	Символы, используемые для ввода-ввода чисел
time	`time_get<>`	Ввод даты и времени
	`time_put<>`	Вывод даты и времени
monetary	`money_get<>`	Ввод денежных данных
	`money_put<>`	Вывод денежных данных
	`moneypunct<>`	Символы, используемые для ввода-вывода денежных данных
ctype	`ctype<>`	Информация о символах языка
	`codecvt<>`	Преобразование между кодировками
collate	`collate<>`	Контекстное сравнение строк
messages	`messages<>`	Локализованные строковые сообщения

14.2.2 Использование контекста

Для использования контекстной информации необходимо получить соответствующий аспект. Потоки ввода-вывода хранят внутри себя свой активный объект контекста и позволяют получить к нему доступ с помощью функций std::locale getloc() **const** и std::locale imbue(**const** locale &). Активный объект контекста автоматически используется потоком при вводе-выводе.

Для получения нужного аспекта служит шаблонная функция use_facet:

```
1 template < typename Facet>
2 const Facet & use_facet(const locale & loc);
```

Наш пример с выводом информации об объекте не интернационализирован, так как всегда выводит точку в качестве десятичного разделителя. Доработка проста:

```
1  std::ostream & sample::operator <<(std::ostream & out,
2      const Employee & e)
3  {
4    ...
5
6      if (out.iword(employee_format_index)
7          == EMPLOYEE_FORMAT_HUMAN_READABLE) {
8        const numpunct< char > & punct =
9            std::use_facet< numpunct< char > >(out.getloc());
10
11       out << left << std::setw(14) << "First name: "
12           << e.first_name_ << "\n"
13           << setw(14) << "Last name: "
14           << e.last_name_ << "\n"
15           << setw(14) << "Salary: " << setw(20)
16           << right << dec << (e.salary_ / 100)
17           << punct.decimal_point()
18           << setw(2) << setfill('0') << (e.salary_ % 100) << "\n";
19         } else {
20
21   ...
22   return out;
23 }
```

Заметим, что вывод в формате S-expression в последней версии оператора не страдает от проблем интернационализации, так как временный объект потока по умолчанию использует классический контекст, не зависящий от пользовательских установок.

14.3 Вспомогательные утилиты

Стандартная библиотека также предлагает некоторое количество полезных утилит, которые хотелось бы упомянуть перед рассмотрением контейнеров.

14.3.1 swap

Шаблонная функция swap позволяет обменять 2 значения одного типа местами.

```
1  template < typename T >
2  void swap(T & a, T & b);
```

Наличие этого шаблона позволяет предоставлять эффективные реализации путем специализации этого шаблона.

14.3.2 pair

Периодически возникает необходимость вернуть из функции 2 значения, например, вычисленное значение и его погрешность. Для решения данной задачи стандартная библиотека предлагает шаблонный класс pair:

```
1  template < typename T1, typename T2 >
2  struct pair {
3    typedef T1 first_type;
4    typedef T2 second_type;
5
6    T1 first;
7    T2 second;
8
9    pair();
10   pair(const T1& x, const T2& y);
11   template < class U, class V >
12   pair(const pair< U, V > & p);
```

```
13 };
14
15 template < class T1, class T2 >
16 pair< T1, T2 > make_pair(T1 x, T2 y);
```

Функция make_pair предоставляет простой способ создать экземпляр класса по двум значениям не указывая конкретные типы.

14.3.3 numeric_limits

При проведении различных вычислений зачастую требуется определить характеристики числового типа, такие как максимальное и минимальное значение, количество десятичных разрядов, которые он может представить, погрешность представления чисел с плавающей запятой и так далее. Для решения этой задачи в C присутствует набор констант. Но эти константы малоприменимы при реализации шаблонов, так как они не дают возможности обобщенно и полиморфно обрабатывать различные численные типы.

Для решения этой задачи в библиотеке присутствует шаблон numeric_limits. Наиболее часто применяющиеся элементы его приведены ниже.

```
1  template < typename T >
2  class numeric_limits {
3  public:
4    static T min() throw();
5    static T max() throw();
6    static const int digits;
7    static const int digits10;
8    static const bool is_signed;
9    static const int radix;
10
11   ...
12 };
```

min минимальное значение для целочисленных типов и минимальное положительное нормализованное значение для типов с плавающей запятой;

max максимальное значение;

radix основание для преставления численного типа;

digits число знаков основания, представляемых типом без потери точности;

digits10 число десятичных знаков, представляемых типов без потери точности;

is_signed **true** в случае, если тип представляет собой числа со знаком.

Глава 15

Итераторы

15.1 Адресная арифметика

C++ как язык высокого уровня, поддерживающий эффективные операции содержит в себе различные низкоуровневые концепции. Одной из самых мощных и опасных являются указатели. Часть проблем с указателями и пути их решения были рассмотрены в разделе, посвященном управлению ресурсами. Но существенная часть силы указателей живет в таком понятии как адресная арифметика.

Адресная арифметика дает возможность выполнять различные арифметические операции над указателями для получения доступа к различным объектам. Главной опасностью адресной арифметики является возможность получения некорректных указателей.

Для указателей A * p, *q и целого числа неотрицательного n определены следующие операции:

p + n	Указатель на n-й элемент массива с началом p
p[n] n[p]	Ссылка на n-й элемент массива p
p - n	Указатель на n-й элемент массива перед p
p++ ++p	Переход к следующему элементу массива p
p– –p	Переход к предыдущему элементу массива p
p += n	Переход к n-му элементу массива p
p -= n	Переход к $-n$-му элементу массива p
p == q p != q	Проверка указателей на равенство
p < q p > q p <= q p >= q	Определение порядка указателей, если p и q принадлежат к разным массивам, результат не специфицирован

Определение порядка указателей определено в стандарте только для указателей, принадлежащих одному массиву, независимо от способа его создания (путем определения переменной типа «массив» или вызова оператора **new** []). Сравнение же на равенство и неравенство определено для всех указателей. Второй тонкий момент связан с пустым указателем: стандарт не оговаривает его точное значение и результат сравнения, в зависимости от реализации, пустой указатель может предшествовать всем указателям, а может и указывать в произвольное место.

Второй момент связан с работой некорректных указателей. Стандарт никоим образом не описывает работу с ними, за исключением слов «неопределенное поведение». Существуют аппаратные платформы, на которых любая попытка даже сравнения некорректного указателя приводит к аппаратной ошибке.

Функцию, осуществляющую последовательный поиск элемента в массиве можно определить следующими способами:

```
1  template < typename T >
2  const T * find1(const T * list , size_t n, const T & value)
3  {
4    size_t i = 0;
5    while ((i < n) && (list[i] != value)) {
6      ++i;
```

```
 7    }
 8
 9    return (i < n) ? (list + i) : 0;
10  }
11
12  template < typename T >
13  const T * find2(const T * begin, const T * end,
14                  const T & value)
15  {
16    while ((begin != end) && (*begin != value)) {
17      ++begin;
18    }
19
20    return begin;
21  }
```

Устройство первой функции очевидно: осуществляется проход по массиву до тех пор, пока индекс не станет равен размеру массива или текщий элемент не будет равен эталонному. Если после завершения цикла индекс равен размеру, то эталон не найден, так как индексация массивов начинается с 0, и возвращается пустой указатель, в противном случае элемент найден и возвращается указатель на него.

Вторая функция устроена на базе понятия «полуоткрытый интервал»: на вход ей передаются 2 указателя, определяющих массив как $[begin, end)$. Другими словами, begin указывает на первый элемент массива, а end — сразу за последним элементом массива. Несмотря на то, что этот указатель может быть некорректным на какой-либо аппаратной платформе, стандарт требует, что чтобы его загрузка в регистры и операции сравнения не вызывали никаких ошибок. Операции разыменования, тем не менее, по прежнему являются неопределенными.

Вторая функция поиска возвращает указатель на найденный элемент или конец интервала. Она может быть вызвана следующим образом:

```
1  A objects[10] = { ... };
2
3  const A * found = find2(objects, objects + 10, value);
4
5  if (found != objects + 10) {
6    ...
7  }
```

В промышленном коде константа 10 должна иметь имя, а в строках 3 и 5 значение должно вычисляться.

15.2 Итераторы как абстракция

На первый взгляд, вторая версия функции find2() ничем не лучше первой, более того, ее вызов сопровождается более сложной проверкой результата. Но если сделать еще один шаг:

```
1  template < typename PointerType, typename T >
2  PointerType find2(PointerType begin,
3                    const PointerType & end,
4                    const T & value)
5  {
6    while ((begin != end) && (*begin != value)) {
7      ++begin;
8    }
9
10   return begin;
11 }
```

В этом случае функция принимает в качестве диапазона 2 объекта, указывающие на начало и сразу за концом диапазона. Все что требуется от этих объектов, это:

1. поддерживать операцию сравнения на неравенство;
2. операция инкремента должна осуществлять переход к следующему элементу;

3. разыменование объекта должно возвращать ссылку на какой-либо объект такого типа, что для него определена операция сравнения на неравенство с типом T.

Очевидно, что указатели полностью удовлетворяют этим требованиям. С другой стороны, механизм ситуативного полиморфизма повзволяет перегрузить все требуемые операторы для произвольного пользовательского типа. В этом случае мы получаем некий «обобщенный указатель». Такой обощенный указатель не обязан указывать внутрь какого-либо массива, а, например, реализовывать последовательный доступ элементам односвязного списка. Такие обобщенные указатели в мире C++ называются итераторами, так как позволяют последовательно просматривать (или итерировать) коллекции объектов.

Прежде чем продемонстрировать создание итератора, необходимо остановиться на нескольких дополнительных аспектах итераторов.

15.2.1 Терминология

Экземпляр итератора называется «разыменовываемым» (dereferenceable) если операция разыменования не приводит к неопределенному поведению. Например, указатель на первый элемент массива является разыменовываемым, в то время как указатель на элемент сразу за последним — нет [C++ 03, параграф 24.1.5].

Итератор j называется достижимым (reachable) из итератора i если существует конечная последовательность операторов ++i, которая приводит к истинности выражения i == j. Если итератор j достижим из i, то они относятся к одной коллекции [C++ 03, параграф 24.1.6].

Корректным диапазоном итератороа нызвается такая пара итераторав i и j, что j достижим из i. Вместе они формируют полуоткрытый диапазон $[i, j)$ [C++ 03, параграф 24.1.7]. Применение каких-либо стандартных функций к некорректным диапазонам является неопределнным поведением.

15.2.2 Категории итераторов

Очевидно, что если итератор предоставляет только возможность перехода к следующему элементу, некоторые операции невозможно написать эффективно, даже если реализация коллекции это позволяет. Например, двоичный (бинарный) поиск подразумевает доступ к произвольному элементу коллекции за $O(1)$. Массивы это позволяют, в то время как односвязные списки — нет. Для массивов двоичный поиск имеет сложность $O(\log_2^n)$, измеренную в операциях сравнения и в операциях перехода к нужному элементу, в то время как для односвязного списка сложность, измеренная в количестве операциях сравнения попрежнему $O(\log_2^n)$, но количество переходов к следующему элементу будет $O(n \log_2^n)$.

В связи с этим все итераторы делятся на несколько категорий, в основу разбиения на которые положено требование алгоритмической сложности реализации: все требуемые операторы и функции для каждой категории должны быть реализованы с амортизированной сложностью $O(1)$. Полный список всех требований к объекту-итератору может быть найден в стандарте [C++ 03, раздел 24.1], здесь же будет приведено только общее описание.

Итераторы ввода (Input)

Итераторы ввода (input iterator) имитируют коллекцию, получаемую с какого-либо устройства ввода. Естественным поведением такого устройства является невозможность возврата назад.

Из этого следует, что итераторы данной категории могут быть использованы для однократного прохода «вперед» по коллекции. Более того, стандарт не требует, чтобы тип, по которому осуществляется итерация, имел возможность присваивания.

Еще одним немаловажным моментом является то, что итераторы ввода не поддерживают ссылочной прозрачности. Другими словами, для любого итератора ввода iter:

```
1 InputIterator a = iter;
2 InputIterator b = iter;
3
4 std::cout << std::boolaplha
5           << (a == b) << "␣"
6           << (++a == ++b) << std::endl;
```

может быть выведено

true false.

Третий тонкий момент заключается в том, что после инкремента итератора все его копии могут перестать быть разыменовываемыми.

Итераторы вывода (Output)

Итераторы вывода имитируют вывод на некое устройство. Они также предоставляют возможность однократного прохода, как и итераторы ввода, но при этом есть некоторые дполнительные особенности:

- операция разыменования допустима только слева от оператора присваивания;
- присваивание по одному и тому же значению итератора должно случаться только один раз;
- операции сравнения итераторов на равенство и неравенство могут быть неопределены.

Прямые итераторы (Forward)

Прямые итераторы предназначены для многократного прохода по коллекции в прямом направлении. Типичный пример — проход по односвязному списку. Свойство ссылочной прозрачности для прямых итераторов должно соблюдаться, а также если 2 итератора равны и разыменовываемы, то и объекты, на которые они ссылаются также равны.

Двунаправленные итераторы (Bidirectional)

Следующими по возможностям являются двунаправленные итераторы. По своим свойствам они полностью соответствуют прямым итераторам с добавлением возможности перемещения назад при помощи **operator** $--()$.

Итераторы произвольного доступа (Random Access)

Итераторы произвольного доступа являются наиболее мощными итераторами. Например, указатели являются итераторами произвольного доступа.

Дополнительными требованиями для таких итераторов являются возможность перехода к несоседним элементам коллекции, получение объекта по индексу относительно итератора и определение порядка между итераторами.

Обобщенные операции над итераторами

При написании шаблонного кода точная категория итератора неизвестна. В то же время существует необходимость перемещения наиболее эффективным методом. Второй встречающейся задачей является вычисление расстояния между двумя итераторами.

Для эффекктивного решения данных задач библиотека содержит дополнительные шаблонные функции:

```
1 template < typename InputIterator , typename Distance >
2 void advance(InputItertaor & i, Distance n);
3
4 template < typename InputIterator >
5 typename iterator_traits< InputIterator >::difference_type
6 distance(InputIterator first , InputIterator second);
```

Эти функции используют операторы $+$ и $-$ для итераторов произвольного типа и оператор $++$ для всех остальных.

Также при написании обобщенного кода бывает необходимо знать некоторые особенности реализации используемого итератора, например, в примере выше требовалось знать какой тип используется для представления расстояния между итераторами. Так как обычные указатели являются итераторами произвольного доступа, но не содержат внутри себя никаких определений типов, прямой доступ к вложенным типам в обобщенном коде невозможен. Для решения этой задачи стандартная библиотека предлагает класс характеристик итератора, который специализирован для указателей. В общем случае, определение шаблона iterator_traits<> выглядит следующим образом:

```
1  template < typename Iterator >
2  struct iterator_traits
3  {
4    typedef typename Iterator::difference_type
5                     difference_type;
6    typedef typename Iterator::value_type value_type;
7    typedef typename Iterator::pointer pointer;
8    typedef typename Iterator::reference reference;
9    typedef typename Iterator::iterator_category
10                    iterator_category;
11 };
```

15.3 Реализация итератора

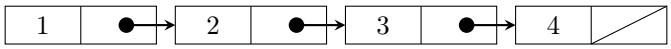
15.3.1 Итератор по односвязному списку

В качестве примера реализуем итератор по односвязному списку.

Односвязный список будет построен на базе шаблонного узла:

```
1  template < typename T >
2  struct node_t
3  {
4    T value;
5    node_t< T > * next;
6  };
```

Таким образом, в памяти односвязный список целых числе будет иметь вид:



Очевидно, что самым мощным итератором для односвязного списка является прямой итератор, так как на пути даже двунаправленного итератора стоит проблема реализации перехода к предыдущему элементу за $O(1)$.

Класс итератора может быть определен следующим образом:

```
1  template < typename T >
2  class slist_iterator:
3    public std::iterator< std::forward_iterator_tag, T >
4  {
5  public:
6    typedef node_t< T > node_type;
7    typedef slist_iterator< T > this_type;
8
9    slist_iterator();
10   slist_iterator(const node_type * slist);
11
12   bool operator ==(const this_type & rhs) const;
13   bool operator !=(const this_type & rhs) const;
14
15   T & operator *() const;
16   T * operator ->() const;
17
18   this_type & operator ++();
19   this_type operator ++(int);
20
21 private:
22   node_type current_;
23 };
```

Для данного итератора нет необходимости в ручной реализации конструктора копирования и оператора присваивания, так как он не владеет узлом списка, на который указывает.

Итератор, указывающий на конец списка будет содержать пустой указатель на узел. Таким образом, итератор автоматически станет неразыменовываемым после перемещения за последний элемент списка.

Функции могут быть реализованы следующим образом:

```
1  template < typename T >
2  slist_iterator< T >::slist_iterator():
3    current_(0)
4  {}
5
6  template < typename T >
7  slist_iterator< T >::slist_iterator(const slist_iterator< T >::node_type * slist)
8    current_(node)
9  {}
10
11 template < typename T >
12 bool slist_iterator< T >::operator ==(const slist_iterator< T > & rhs) const
13 {
14   return current_ == rhs.current_;
15 }
16
17 template < typename T >
18 bool slist_iterator< T >::operator !=(const slist_iterator< T > & rhs) const
19 {
20   return current_ != rhs.current_;
21 }
22
23 template < typename T >
24 T & slist_iterator< T >::operator *() const
25 {
26   assert(current_ != 0);
27
28   return current->value;
29 }
30
31 template < typename T >
32 T * slist_iterator< T >::operator ->() const
33 {
34   assert(current_ != 0);
35
36   return &current_->value;
37 }
38
39 template < typename T >
40 slist_iterator< T > & slist_iterator< T >::operator ++()
41 {
42   assert(current_ != 0);
43
44   current_ = current_->next;
45
46   return *this;
47 }
48
49 template < typename T >
50 slist_iterator< T > slist_iterator< T >::operator ++(int)
51 {
52   this_type temp(*this);
53
54   this->operator ++();
```

```
55
56    return temp;
57  }
```

15.3.2 Итераторы вставки

В стандартной библиотеке присутствует некоторое количество стандартных итераторов. Самыми простыми из них являются итераторы вставки. Сами по себе они являются итераторами вывода. Библиотека предлагает 3 итератора вставки:

- back_insert_iterator
- front_insert_iterator
- insert_iterator

Как следует из названия, back_insert_iterator добавляет элементы в конец коллекции, front_insert_iterator — в начало, а insert_iterator — в позицию переданного итератора.

Все эти итераторы являются шаблонами, параметризованными типом контейнера. Для удобства использования библиотека также предлагает 3 шаблонных функции для создания соответствующих итераторов:

```
1  template < typename Container >
2  back_insert_iterator< Container > back_inserter(Container &);
3
4  template < typename Container >
5  front_insert_iterator< Container > front_inserter(Container &);
6
7  template < typename Container, typename Iterator >
8  insert_iterator< Container > inserter(Container &, Iterator i);
```

15.3.3 Потоковые итераторы

Второй группой итераторов, предлагаемых стандартной библиотекой, являются потоковые итераторы. Их 2: один для чтения данных и второй для записи.

Итератором для чтения данных является шаблон istream_iterator<>:

```
1  template < typename T,
2             typename Char = char,
3             typename Traits = char_traits < Char >,
4             typename Distance = ptrdiff_t >
5  class istream_iterator;
```

Параметры шаблона имеют следующий смысл:

T тип читаемых данных, например, std::string или **double**

Char тип символа потока

Traits характеристики символа потока

Distance тип данных для вычисления расстояния между итераторами

При последовательном применении оператора ++, итератор читает из потока следующий элемент и операция разыменования возвращает ссылку на него. Например, чтение и заполнение вектора целыми числами может быть реализовано следующим образом:

```
1  std::istream_iterator< int > in(std::cin);
2  std::istream_iterator< int > end;
3  typedef std::vector< int > int_vector;
4  int_vector data;
5  std::back_insert_iterator< int_vector > target(data);
```

```
 6
 7  while (in != end) {
 8    *target = *in;
 9    ++in;
10    ++target;
11  }
```

Итератор вывода ostream_iterator<> принимает те же аргументы шаблона и работает аналогично итератору ввода, но в дургом направлении — на вывод и, соответственно, используется слева в операторе присваивания.

15.3.4 Инвалидация итераторов

Некоторые алгоритмы подразумевают изменение последовательности в процессе их работы. Например, некий алгоритм может удалять из списка банковских счетов уже закрытые. В зависимости от реализации данного списка возможны различные дополнительные эффекты.

Для итератора по односвязному списку, реализованному ранее, удаление текущего элемента списка приводит к тому, что хранимый внутри итератора указатель на узел «повисает». Это делает итератор неразыменовываемым.

Такой процесс существует для любых итераторов и называется «инвалидация итератора». Условия, при которых происходит инвалидация итераторов различны для разных контейнеров. Подробнее эти условия будут рассмотрены в соответствующих разделах.

Глава 16

Последовательности

16.1 Общие понятия

Различные контейнеры являются неотъемлемой частью любой программы. Поэтому абсолютное большинство языков и систем программирования предлагают набор контейнеров в комплекте.

Самыми базовыми из них являются последовательности. Последовательности хранят свои объекты без соблюдения какого-либо порядка.

16.2 Требования

Прежде чем рассматривать предлагаемые STL контейнеры, необходимо рассмотреть некоторые базовые моменты и требования.

Все измерения сложности операций для контейнеров измеряются в количестве операций над объектами, хранящимися в контейнерах.

Самым базовыми требованиями к объектам, которые будут помещены в контейнера является их способность быть скопированными конструктором копирования (CopyConstructible) и оператором присваивания (Assignable).

Стандарт 2011 года изменил набор требований. Объекты теперь могут не удовлетворять возможностям CopyConstructible и Assignable, но взамен они могут быть перемещены (MoveInsertable) или сконструированы в контейнере «по-месту» (EmplaceConstructible).

Набор требований к самому контейнеру довольно велик. В дальнейшем используются следующие обозначения:

T тип объекта, хранящегося в контейнере

X класс контейнера объектов T

a и b экземпляры контейнера X

Контейнер должен содержать следующие определения вложенных типов ([C++ 03, раздел 23.1], [C++ 11, раздел 23.2.1]):

X::value_type Тип объекта, хранящегося в контейнере.

X::reference Тип ссылки на объект, хранящийся в контейнере.

X::const_reference Тип ссылки на константный объект.

X::iterator Тип итератора по элементам контейнера; может принадлежать любой категории, кроме итераторов вывода; должен преобразовываться в const_iterator.

X::const_iterator Тип итератора, не допускающий изменения контейнера; может быть любой категории, кроме итератора вывода.

X::difference_type Знаковый целый тип, представляющий собой разницу между итераторами.

X::size_type Беззнаковый целый тип, вмещающий разницу между итераторами.

Помимо вложенных типов, к контейнеру предъявлены требования к операциям:

X() Конструктор по умолчанию должен конструировать за постоянное время пустой контейнер.

X(const X &) Конструктор копирования должен создавать копию контейнера за линейное ($O(1)$) время.

X() Деструктор должен обеспечивать вызов деструкторов каждого элемента контейнера за линейное время.

a.begin() Итератор, указывающий на первый элемент контейнера.

a.end() Итератор, указывающий за последний элемент контейнера.

a == b Выражение, приводящееся к **bool**. Должно быть **true** если размеры контейнеров и каждый элемент одного соответственно равен элементу другого.

a != b Выражение, приводящееся к **bool** и эквивалентное !(a == b).

a.swap(b) Обмен содержимого контейнеров, должно выполняться за постоянное время.

a = b Присваивание контейнеров должно выполняться за линейное время.

a.size() Количество элементов контейнера, эквивалентно std::distance(a.begin(), a.end()).

a.max_size() Максимально допустимый размер контейнера.

a.empty() Возвращает значение, приводящееся к **true** когда контейнер пуст, выполняется за постоянное время.

a < b Лексикографическое сравнение контейнеров, выполняющееся за линейное время; также определены операции a > b, a <= b, a >= b; требуется наличие операции T::**operator** <().

Все приведенные выше требования не зависят от контейнера и применимы как к последовательностям, так и к ассоциативным контейнерам.

К последовательностям предъявлено еще несколько требований ([C++ 03, раздел 23.1.1], [C++ 11, раздел 23.2.3]):

X(n, t) Для целочисленного n помещает в контейнер n копий t.

X(i, j) Для итераторов, конструирует контейнер копиями данных в диапазоне $[i, j)$.

a.insert(p, t) Вставляет t в контейнер перед элементом в позиции итератора p и возвращает итератор, указывающий на только что вставленный элемент.

a.insert(p, n, t) Вставляет n копий t перед элементом в позиции p.

a.insert(p, i, j) Для итераторов i и j], не принадлежащих a, вставляет копию диапазона $[i, j)$ перед элементом в позиции итератора p.

a.erase(p) Удаляет элемент в позиции итератора p.

a.erase(p, q) Удаляет элементы в диапазоне $[q, q)$

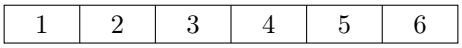
a.clear() Удаляет всес одержимое контейнера.

Все функции удаления возвращают итератор на элемент, которые был после последнего удаляемого перед началом удаления. Если такого элемента нет, то возвращается итератор, указывающий за последний элемент.

16.3 Типы последовательностей в STL

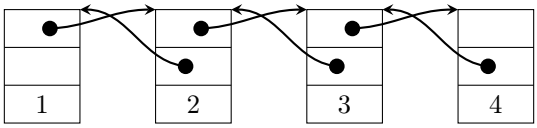
На основе приведенных требований можно создать много различных контейнеров, но STL содержит только 3: vector, list и deque. Эти три вида контейнера отличаются внутренней организацией данных.

Vector хранит свои данные последовательно в одном блоке данных:



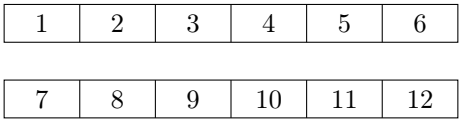
Такая организация данных обеспечивает доступ к любому элементу за постоянное время, но при этом вставка любого элемента в середину или начало контейнера ведет к множественному копированию данных.

List представляет собой двусвязный список.



Список допускает вставку элементов в любую позицию за постоянное время, но при этом может обеспечить только двунаправленные итераторы.

Deque представляет собой компромисс между списком и вектором. Данные хранятся с списке блоков фиксированного размера. Это позволяет получить амортизированное постоянное время доступа к элементу.



Для удобства, последовательности предлагают некоторое количество операций. Но эти операции доступны не во всех последовательных контейнерах:

a.front() Ссылка на первый элемент контейнера.

a.back() Ссылка на последний элемент контейнера.

a.push_front() Вставка элемента в начало контейнера.

a.push_back() Вставка элемента в конец контейнера.

a.pop_front() Удаление элемента из начала контейнера.

a.pop_back() Удаление элемента из конца контейнера.

a.at(n) Получение элемента по индексу с проверкой границ.

Функция at() проверяет индекс и генерирует исключение std::out_of_range, если он выходит за пределы. Существует быстрый вариант без проверки: оператор индексации a[n].

С учетом того, как устроены контейнеры, доступность операций можно свести в таблицу:

Операция	vector	list	deque
front()	•	•	•
back()	•	•	•
push_front()		•	•
push_back()	•	•	•
pop_front()		•	•
pop_back()	•	•	•
a[n]	•		•
at()	•		•

16.4 Тонкие моменты

16.4.1 Прямые и обратные итераторы

Если итераторы контейнера являются двунаправленными или с прямым доступом, то контейнер должен также содержать:

X::reverse_iterator Тип итератора, обеспечивающего проход в обратном порядке.

X::const_reverse_iterator Тип итератора, обеспечивающего проход в обратном порядке и не допускающего изменения контейнера.

a.rbegin() Возвращает итератор на последний элемент контейнера.

a.rend() Возвращает итератор, указывающий перед первым элементом контейнера.

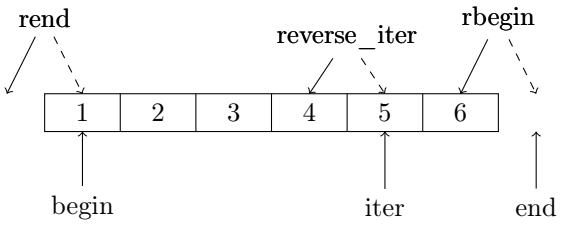
С обратными итераторами есть сложный момент:

```
1  const int arr[ ]= { 1, 2, 3, 4, 5, 6 };
2
3  std::vector< int > v(arr,
4                       arr + sizeof(arr) / sizeof(arr[0]));
5
6  std::vector< int >::iterator iter = v.begin() + 4;
7  std::vector< int >::reverse_iterator rev_iter(iter);
8  std::vector< int >::iterator rst_iter(rev_iter.base());
9
10 std::cout << "Iterator: " << *iter
11           << "\nReverse:  " << *rev_iter
12           << "\nRestored: " << *rst_iter
13           << std::endl;
```

Данный пример выводит:

```
Iterator: 5
Reverse:  4
Restored: 5
```

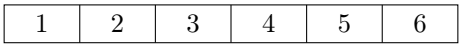
При всей неожиданности такого поведения в нем есть логика.



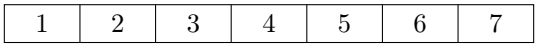
Стандарт требует, чтобы указатель за последним элементом массива (по сути, конец диапазона итераторов) можно было использовать для сравнений, но не разыменований. В то же время, такого требования для указателя перед первым элементом массива нет. Поэтому обратный итератор сохраняет текущую позицию как есть. Но при разыменовании итератора rbegin с одной стороны необходимо вернуть значение, с другой стороны внутренний итератор указывает за конец контейнера. Поэтому обратный итератор сдвигается на одно значение к началу контейнера при разыменовании.

16.4.2 std::vector: размер и емкость

Внутреннее устройство вектора обладает некоторыми ограничениями. Предположим, что есть вектор числе от 1 до 6.

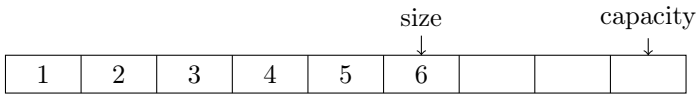


При добавлении нового элемента необходимо где-то его разместить, но места в буфере нет. Поэтому требуется разместить новый буфер большего размера и скопировать в него имеющиеся данные, а потом добавить новый элемент.



При большом количестве операций вставки и удаления это неэффективно. Поэтому буфер внутри вектора обычно больше, чем количество элементов внутри. Это позволяет добавлять

новые элементы без обращения к Free Store. Если же буфер исчерпан, при размещении нового элемента создается новый буфер, который больше предыдущего в определенное количество раз. Обычно этот коэффициент равен 1.6 или 1.8



В целом это повышает эффективность удаления и вставки. Более того, вектор дает возможность ограниченного контроля за размером буфера при помощи функции reserve(size_type). Если аргумент больше, чем текущий размер буфера, то буфер увеличивается так, чтобы вмещать указанное количество элементов.

Поскольку буфер никогда не уменьшается, то даже функция clear() не уменьшает использование памяти. Для решения этой проблемы существует идиома с обменом:

```
1  std::vector< Employee > employees;
2
3  ...
4  std::vector< Employee >(employees).swap(employees);
```

16.4.3 Инвалидация итераторов

Как было отмечено, итераторы могут переходить в неразыменовываемое состояние. Понимание устройства контейнеров позволяет правильно предотвращать ошибки при работе с итераторами.

Устройство вектора как динамического массива позволяет реализовывать для не эффективные итераторы в виде указателей. С другой стороны это приводит к тому, что итераторы могут перестать быть корректными после практически любой модификации контейнера. В действительности только итераторы, указывающие на элементы перед удаляемым не инвалидируются. Во всех остальных случаях они могут стать некорректными, включая операцию вставки элемента в контейнер, так как она может привести к переразмещению буфера в памяти.

С другой стороны, для списка некорректными становятся только итераторы, указывающие на непосредственно удаляемый элемент.

Для предотвращения работы с некорректными итераторами функции insert() и erase() возвращают новый итератор, который должен использоваться вместо старого.

16.4.4 Безопасность исключений

Стандарт требует, чтобы функции erase(), pop_back() и pop_front() не генерировали исключений [C++ 03, параграф 23.1.10]. С другой стороны, если при использовании insert() для вставки одного элемента, а также push_back() и push_front(), генерируется исключение, то эти функции не модифицируют контейнер.

Также предъявлено требование, что копирование итераторов не может генерировать исключения.

16.4.5 std:vector< bool >

Класс vector< **bool** > является примером того, как можно специализировать шаблон. Но при этом он не является контейнером STL. Это связано с тем, что в действительности он не хранит **bool**. Каждое отдельное значение хранится в виде отдельного бита.

Поскольку указатели на отдельные биты невозможны, то оператор индексации возвращает некий proxy-объект. Но его адрес не является адресом значения типа **bool**:

```
1  std::vector< bool > v;
2
3  ...
4
5  bool * p = &v[0];
```

Данный пример не компилируется. Как результат, он не удовлетворяет требованиям, предъявленным к вектору.

В свете всего выше сказанного рекомендуется избегать его использования.

Глава 17

Ассоциативные контейнеры

Последовательности представляют собой самый общий вид контейнера, с их помощью можно решить любую задачу. Но в некоторых случаях их применение неэффективно. Например, поиск информации о пользователе по его идентификатору выполняется намного чаще, чем ее изменение, поэтому необходимо оптимизировать поиск, а не изменение списка пользователей. Эта задача может быть решена различными способами, например, можно поддерживать список в отсортированном по идентификатору пользователя порядке. Таким образом, сложность поиска снижается с $O(n)$ до $O(\log_2^n)$. Но это решение обладает некоторыми менее очевидными недостатками:

- Если в качестве контейнера использован std :: vector<>, то переход к следующему элементу при поиске занимает $O(1)$, но зато вставка и удаление элементов имеет сложность $O(n)$, а главная проблема — инвалидация итераторов при изменениях контейнера.
- Если в качестве контейнера используется std :: list <>, то переход к следующему элементу при поиске имеет сложность $O(n)$, изменение контейнера $O(1)$ и проблемы с инвалидацией итераторов гораздо меньше, так как инвалидируются только итераторы, указывающие на удаленный элемент.
- Использование std :: deque<> представляет собой компромисс по всем фронтам: амортизированные сложности удаления, вставки и перехода к следующему элементу при поиске компенсируются сложностями в определении корректности итераторов после изменения контейнера.

Также решение на базе последовательностей обладает более высокой сложностью, так как необходимо поддерживать контейнер в отсортированном состоянии. Эту задачу можно упростить путем инкапсуляции контейнера в классе Users.

Для решения подобных проблем созданы ассоциативные контейнеры, которые в общем случае позволяют по ключу быстро находить ассоциированное значение. В других языках программирования они часто называются словарями.

17.1 Упорядоченные контейнеры

Стандарт C++ 03 описывает ассоциативные контейнеры в разделе 23.3 [C++ 03]. Но прежде чем приступать к их рассмотрению необходимо прояснить вопрос минимально полного функционального набора.

17.1.1 Минимальный полный функциональный набор

Минимально полным функциональным набором называется такой набор операций или функций, при помощи которого можно выразить все операции. Наиболее часто этот термин применяется в цифровой схемотехнике, например, логическая операция `И-НЕ` является таким набором, так как с ее помощью можно выразить любое логическое выражение, как то `И`, `ИЛИ`, `НЕ` и т.д.

Для поддержания контейнера в отсортированном состоянии и исключения дубликатов необходимы различные операции сравнения. Просто решение состоит в том, чтобы потребовать наличия 2-х операций: сравнение на равенство и установка отношения, например, **operator** ==() и **operator** <().

В то же время, эта пара операторов не является минимальным полным функциональным набором. Как следствие, они могут быть реализованы противоречиво, например, возможно, что следующий код выводит 1:

```
Object a, b;

std::cout << ((a < b) == (a == b)) << "\n";
```

Для исключения таких ситуаций правильно требовать реализацию минимального набора. Для упорядочивания объектов такими наборами из одной операции являются операции сравнения **operator** <() и **operator** >(). Например, если выбрать в качестве базисной операции сравнение «меньше», то остальные сравнения могут быть реализованы следующим образом:

$$\begin{array}{ll} a < b & a < b \\ a > b & b < a \\ a = b & (a \bar{<} b) \wedge (b \bar{<} a) \\ a \neq b & (a < b) \vee (b \bar{<} a) \\ a \leq b & (a < b) \vee (b \bar{<} a) \\ a \geq b & (b < a) \vee (a \bar{<} b) \end{array}$$

Ассоциативные контейнеры требуют реализации операции сравнения, семантика которой будет «меньше» или «больше», по умолчанию контейнеры будут использовать **operator** <(). В случае, если для не выполняется условие $(a = b) \equiv (a \bar{<} b) \wedge (b \bar{<} a)$, порядок в контейнере не определен и поиск по ключу будет работать неправильно.

17.1.2 Требования

Все ассоциативные контейнеры, описанные в стандартах 2003 и 2011 годов, должны удовлетворять общим требованиям для всех STL-совместимых контейнеров ([C++ 03, раздел 23.1], [C++ 11, раздел 23.2.1]).

Ассоциативные контейнеры представляют собой сложные структуры в памяти, поэтому они реализуют только двунаправленные итераторы. Реализация итераторов прямого доступа не будет обеспечивать переходы к элементам за постоянное время.

Специфические требования к ассоциативным контейнерам описаны в разделах 23.1.2 [C++ 03] и 23.2.4 [C++ 11].

Любой ассоциативный контейнер также должен содержать следующие вложенные типы:

X::key_type Тип ключа.

X::key_compare Тип компаратора ключей.

X::value_compare Тип компаратора, сравнивающего объекты в контейнере.

X::mapped_type Тип значения в контейнере, C++ 11, только для std :: map и std :: multimap.

Так как ассоциативные контейнеры принимают новый параметр шаблона — компаратор — требования к некоторым методам класса меняются. Помимо описанных стандратных вариантов конструкторов также добавляются варианты, работающие с компараторами:

X(c) Конструктор создает за постоянное время пустой контейнер, который будет использовать в качестве компаратора копию переданного в аргументе объекта.

X(i, j, c) Для итераторов i и j, а также компаратора c, создает контейнер, содержащий копии данных в диапазоне $[i, j)$, упорядоченные при помощи компаратора c.

X(i,j) Аналогично предыдущему варианту, но в качестве компаратора будет использован объект, сконструированный по умолчанию.

a.key_comp() Возвращает экземпляр компаратора, использующегося для сравнения ключей.

a.value_comp() Возвращает экземпляр компаратора, использующегося для сранения значений в контейнере. О разнице между этим и предыдущим методом далее.

a.insert(t) Вставляет значение t в соответствующую позицию. Возвращаемое значение зависит от типа контейнера.

a.insert(p, t) Вставляет значение t перед элементом, на который указывает итератор p. Если эта позицию соответствует добаляемому элементу, то операция выполняется за $O(1)$, в противном случае элемент вставляется в правилнуб позицию за $O(\log_2^n)$. В любом случае возвращается итератор, указывющий на позицию, соответствующую ключу t.

a.insert(j, j) Для итераторов i и j вставляет значения из даипазона $[i, j)$ в контейнер на соответствующие ключам позиции. Алгоритмическая сложность $O(m \log_2^n)$.

a.erase(k) Удаляет из контейнера элементы с ключом k и возвращает количество удаленных элементов.

a.erase(q) Удаляет элемент, на который указывает итератор q.

a.erase(q1, q2) Удаляет элементы в дапазоне итераторов $[q1, q2)$.

a.clear() Очищает контейнер за линейное время.

a.count(k) Возвращает количество элементов с ключом k.

17.1.3 Типы контейнеров

Стандарт C++ 03 описывает несколько различных типов контейнеров. Одним оз различий между этими типами является хранимые данные. Один из типов предоставляет возможность отображения ключей на значения. Все контейнеры, выполняющие отображения, содержат в имени слово «map». Дргие контейнеры просто хранят объекты внутри. Такие контейнеры содержат в имени слово «set».

По умолчанию, все контейнеры хранят элементы в порядке возрастания ключей.

С уникальными ключами

Наиболее часто используемыми ассоциативными контейнерами являются контейнеры с уникальными ключами. Эти контейнеры игнорируют попытку вставки дубликатов. Таких контейнеров в стандарте по одному каждого типа:

```
template < typename Key,
  typename Compare = less< Key >,
  typename Allocator = allocator< Key > >
class set
{ ... };

template < typename Key, typename T,
  typename Compare = less< Key >,
  typename Allocator = allocator<pair<const Key, T> > >
class map
{ ... };
```

Для контейнеров, поддерживающих уникальность, метода вставки одного элемента возвращает 2 значения:

```
std::pair< iterator, bool > insert(const value_type &);
```

Первый элемент пары является итератором, указывающим на элемент с эквивалентым ключом, второй элемент содержит **true** если элемент был добавлен контейнер и **false** в случае, если такой ключ в контейнере уже присутствовал.

Также std::map содержит перегруженный оператор индексации:

```
const value_type & operator [](const key_type &) const;
value_type & operator(const key_type &);
```

Этот оператор возвращает ссылку на значение, соответствующее ключу. Если такого ключа к контейнере нет, то контейнер получит новый элемент со значением, сконструированным по умолчанию. Очевидно, что если тип значения не имеет конструктор по умолчанию, то вызов оператора индексации не скомпилируется.

С точки зрения эффективности, этот оператор делает двойную работу при вставке элемента, так как сначала вызывается конструктор по умолчанию для значения, а потом — оператор присваивания.

Для std::set и std::map тип value_type означает совершенно разное. В std::set он эквивалентен параметру шаблона Key. Например:

```
1  std::set< int > ints;
2
3  ints.insert(5);
4
5  const int values[] { 1, 2, 3, 4, 5 };
6
7  ints.insert(values, values + sizeof(values) / sizeof(values[0]);
```

И, соответственно, методы key_comp() и value_comp() возвращают одно и то же — компаратор контейнера.

В std::map value_type представляет собой пару из ключа и значения:

```
    typedef std::pair< const Key, T > value_type;
```

Поэтому практическая жизнь несколько усложняется:

```
1  std::map< int, std::string > strings;
2
3  strings[0] = "Test String";
4
5  strings.insert(std::make_pair(1, std::string("String 2")));
```

Метод key_comp() по-прежнему возвращает компаратор контейнера, но метод value_comp() в действительности возвращает объект, способный сравнивать по ключу value_type контейнера.

С повторяющимися ключами

Также стандарт C++ 03 описывает по одному контейнеру, ключи которого могут повторяться. Такие контейнеры содержат слово «multi» в названии.

```
template < typename Key,
  typename Compare = less< Key >,
  typename Allocator = allocator< Key > >
class multiset
{ ... };

template < typename Key, typename T,
  typename Compare = less< Key >,
  typename Allocator = allocator<pair<const Key, T> > >
class multimap
{ ... };
```

Для таких контейнеров метод вставки значения возвращает итератор, указывающий на новый элемент:

```
    iterator insert(const value_type &);
```

17.2 Неупорядоченные контейнеры

Ассоциативные контейнеры стандарта C++ 03 представляют собой деревья, поэтому поиск в них имеет алгоритмическую сложность $O(\log_2^n)$. Иногда этого недостаточно. Поэтому C++ TR1 и стандарт C++ 11 содержат неупорядоченные контейнеры, построенные на базе хэш-таблиц [Кор+13].

Для реализации хэш-таблицы, в отличие от дерева, требуется 2 дополнительных функции:

- сравнение на равенство,
- вычисление хэш-суммы.

В качестве первой обычно используется оператор сравнения на равенство, что касается второго, то вычислитель хэш-сумм для ключей должен быть предоставлен пользователем контейнера. Для встроенных типов такой вычислитель по умолчанию предоставляется библиотекой.

Контейнеры, построенные на базе хэш-таблиц, принимают вместо типа компаратора типы функций, ответственных за сравнение на равенство и вычисления хэш-суммы. Соответственно, конструкторы имеют версии, принимающие экземпляры соответствующих функций.

Все неупорядоченные ассоциативные контейнеры имеют в своем названии слово «unordered» и бывают как с уникальными ключами, так и с повторяющимися.

Контейнеры с уникальными ключами:

```
template < typename Key,
  typename Hash = hash< Key >,
  typename Pred = equal_to< Key >,
  typename Allocator = allocator< Key > >
class unordered_set
{ ... };

template < typename Key, typename T,
  typename Hash = hash< Key >,
  typename Pred = equal_to< Key >,
  typename Allocator = allocator<pair<const Key, T> > >
class unordered_map
{ ... };
```

Контейнеры с повторяющимися ключами:

```
template < typename Key,
  typename Hash = hash< Key >,
  typename Pred = equal_to< Key >,
  typename Allocator = allocator< Key > >
class unordered_multiset
{ ... };

template < typename Key, typename T,
  typename Hash = hash< Key >,
  typename Pred = equal_to< Key >,
  typename Allocator = allocator<pair<const Key, T> > >
class unordered_multimap
{ ... };
```

Дополнительные методы неупорядоченных контейнеров полностью описаны в разделе 23.2.5 C++ 11 [C++ 11].

17.3 Тонкие моменты

Выбор между ассоциативными контейнерами на базе деревьев и неупорядоченными контейнерами должен происходить не только с учетом алгоритмической сложности, но и учитывать следующие факторы:

- Деревья хранят объекты упорядоченными.
- Хэш-таблицы обычно требуют больше памяти.
- Эффективность хэш-таблиц сильно зависит от функции хэш-суммы. Если возникает много коллизий, алгоритмическая сложность поиска в контейнере начинает стремиться к $O(n)$.

Ключи ассоциативных контейнеров не должны модифицироваться. Например, std :: map ключ отмечает словом **const**. С другой стороны, ключевое слово **mutable** позволяет обойти ограничение физической константности. Таким образом, если ключ модифицирован так, что он находится в неправильной позиции в дереве, все операции в контейнере перестают работать корректно. С другой стороны, если ключ требует изменения, проще элемент удалить и вставить заново или использовать другой контейнер.

Если порядок элементов в контейнере важен, то нет выбора и приходится использовать деревья. Но если операции изменения содержимого крайне редки, а поиск — част, то оптимальным решением можно оказаться использование отсортированного std :: vector. Правильное решение должно подразумевать инкапсуляцию логики так, чтобы порядок в контейнере невозможно было нарушить извне. Возможно, что бинарный поиск в векторе будет много эффективнее перехода по узлам дерева.

17.4 Корректность итераторов

Правила корректности итераторов отличаются для упорядоченных и неупорядоченных контейнеров.

Для упорядоченных контейнеров работает принцип, аналогичный std :: list :

- вставка элементов не приводит к инвалидации итераторов;
- удаление элементов инвалидирует итераторы, указывающие на удаляемые элементы.

Что касается неупорядоченных контейнеров, то:

- вставка элементов не влияет на корректность ссылок на элементы контейнера, но может инвалидировать все итераторы [C++ 11, параграф 23.5.2.13];
- удаление элементов инвалидирует только итераторы и ссылки на удаляемые элементы.

Параграф 23.5.2.14 содержит условие инвалидации итераторов при вставке, но его проверка никоим образом не упрощает построение систем и алгоритмов.

Глава 18

Функторы

18.1 Типы функций

В C++ действия и вычисления содержатся внутри функций. Язык не выдвигает никаких особенных требований к тому как написаны функции. Рассмотрим несколько примеров:

```
 1  unsigned int factorial(unsigned int n)
 2  {
 3    return (n == 1) ? 1 : n * factorial(n - 1);
 4  }
 5
 6  int log_file = -1;
 7
 8  void log_message(const std::string & message)
 9  {
10    if (log_file >= 0) {
11      static const char NEWLINE[] = "\n";
12
13      write(log_file, message.c_str(), message.size());
14      write(log_file, NEWLINE, sizeof(NEWLINE) - 1);
15    }
16  }
```

В данном примере функция factorial вычисляет факториал. Эта функция является примером сразу 2-х видов: это рекурсивная чистая функция. «Рекурсивная» означает, что она вызывает саму себя, либо непосредственно, либо посредством вызова другой функции, которая в конце концов вызовет эту. Чистой называют функции, результат которых зависит исключительно от их аргументов.

Функция log_message демонстрирует другой тип функций: с побочным эффектом. Результат выполнения таких функций либо проявляется в виде дополнительных эффектов (в других языках таких функции называются процедурами), либо зависит от внешних условий. Например, все функции, выполняющие обмен в внешним миром, являются функциями с побочными эффектами.

18.2 Функции высшего порядка

Некоторые функции естественно реализуются в терминах других функций. Классическим примером является функции стандартной библиотеки C qsort():

```
1  void qsort(void *base, size_t nel, size_t width,
2             int (*compare)(const void *, const void *));
```

Функция qsort принимает следующие аргументы:

base адрес начала массива для сортировки;

nel количество элементов в массиве;

width размер одного элемента массива;

compare адрес функции, выполняющей сравнение элементов для сортировки.

Передавая размер одного элемента и функцию сравнения, становится возможным написать алгоритм сортировки один раз и использовать его для различных типов. Пример в стиле C может выглядеть так:

```
1  struct user_t {
2    const char * first_name;
3    const char * last_name;
4    ...
5  };
6
7  int compare_users_by_name(const void * lhs,
8                            const void * rhs)
9  {
10   const struct user_t * lhs_user = (const struct user_t *)lhs;
11   const struct user_t * rhs_user = (const struct user_t *)rhs;
12
13   int result = strcmp(lhs_user->last_name, rhs_user->last_name);
14
15   if (result == 0) {
16     result = strcmp(lhs_user->first_name, rhs_user->first_name);
17   }
18
19   return result;
20 }
21
22 void foo()
23 {
24   struct user_t * users = ...;
25   size_t user_count = ...;
26   ...
27   qsort(users, user_count, sizeof(struct user_t), &compare_users_by_name);
28 }
```

Мир C ограничен использованием адресов функций. Мир C++ предлагает несколько больше возможностей. Следующим шагом является использованием указателей на метода классов (pointer to member). Например, если требуется написать функцию, которая будет выводить результат вызова любого метода класса, не принимающего аргументов и возвращающего std::string, то решение может выглядеть так:

```
1  class Object
2  {
3  public:
4    ...
5    std::string getId() const;
6    std::string getName() const;
7  };
8
9  typedef std;:string (Object::*string_getter_t)() const;
10
11 template < typename InputIterator >
12 void print(InputIterator begin, const InputIterator & end,
13            string_getter_t get)
14 {
15   while (begin != end) {
16     std::cout << (begin->*get)() << "\n";
17     ++begin;
18   }
```

```
19 }
20
21 void foo()
22 {
23   std::vector< Object > objects;
24
25   print(objects.begin(), objects.end(), &Object::getName);
26 }
```

18.3 Классы как функции

Указатели на функции и указатели на методы это самые простые способы написания функций высшего порядка. Следующим по сложности является перегрузка функций, а именно **operator** (). Использоваться это может так:

```
1  class Object
2  {
3  public:
4    ...
5    std::string getId() const;
6    std::string getName() const;
7  };
8
9  template < typename InputIterator, typename Getter >
10 void print(InputIterator begin, const InputIterator & end,
11            Getter get)
12 {
13   while (begin != end) {
14     std::cout << get(*begin) << "\n";
15     ++begin;
16   }
17 }
18
19 struct get_name
20 {
21   std::string operator ()(const Object & obj) const
22   {
23     return obj.getName();
24   }
25 };
26
27 void foo()
28 {
29   std::vector< Object > objects;
30
31   ...
32   print(objects.begin(), objects.end(), get_name());
33 }
```

Такие объекты, у которых перегружен оператор () называются функторами. Обычно они используются вместе с шаблонными функциями. При этом код зачастую оказывается более эффективным, чем с указателями на функции, потому что компилятор способен подставить тело функтора в точку вызова.

18.3.1 Стандартные функторы

Стандартная библиотека C++ содержит множество готовых функторов на разные случаи жизни. Все они находятся в заголовке `functional`. Список включает в себя арифметические опе-

рации, сравнения и логические операции. Также присутствуют функторы для вызова методов классов.

Все эти функторы унаследованы от одного из двух базовых классов: unary_function или binary_function. Эти шаблонные классы довольно примитивны и, на первый взгляд, особой ценности не имеют:

```
template < typename Arg,
           typename Result >
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};

template < typename Arg1, typename Arg2,
           typename Result>
struct binary_function {
  typedef Arg1 first_argument_type;
  typedef Arg2 second_argument_type;
  typedef Result result_type;
};
```

Определения типов помогают при использовании внутри шаблонных функций для определения типов параметров и возвращаемых значений.

Таким образом, правильная реализация get_name должна выглядеть так:

```
struct get_name:
  std::unary_function< Object, std::string >
{
  std::string operator ()(const Object & obj) const
  {
    return obj.getName();
  }
};
```

18.3.2 Частичное применение

Несмотря на то, что в стандарте C++ 11 эти базовые шаблоны названы устаревшими, в коде, соответствующем C++ 03 они необходимы для эффективной реализации такой концепции как частичное применение функций.

Частичное применение функций (partial application) предполагает, что вызывается функция многих аргументов, но при этом передаются только некоторые из них. В функциональных языках это действие выглядит совершенно естественно для первых аргументов, так как вызов (или применение) функции в них не требует применения скобок. Результатом частичного применения является функция оставшихся аргументов. Обычно эта техника используется при передаче функции в функцию высшего порядка.

В C++ эта операция называется «связывание». В рамках стандарта C++ 03 связывание возможно только для функторов с 2 аргументами. Для этого существуют 2 шаблона: binder1st и binder2nd. Их определения выглядят так (на примере binder1st):

```
template < typename Operation >
class binder1st:
    public std::unary_function<
        typename Operation::second_argument_type,
        typename Operation::result_type >
{
public:
  binder1st(const Operation & op,
            const typename
                Operation::first_argument_type & value);

```

```
12    typename Operation::result_type operator ()(
13        const typename
14            Operation::second_argument_type & x) const
15    {
16      return op(value, x);
17    }
18  };
```

Поскольку указывать параметры шаблона каждый раз не хочется, в комплекте идут вспомогательные шаблонные функции, для которых работает вывод типов аргументов:

```
1  template < typename Operation, typename T >
2  binder1st< Operation > bind1st(const Operation & op,
3                                 const T & value)
4  {
5    return binder1st< Operation >(op, value);
6  }
7
8  template < typename Operation, typename T >
9  binder2nd< Operation > bind2nd(const Operation & op,
10                                const T & value)
11 {
12   return binder2nd< Operation >(op, value);
13 }
```

Эти шаблоны позволяют зафиксировать первый или второй аргумент функтора, принимающего 2 аргумента. Таким образом получается функтор, принимающий один аргумент. Например, если есть шаблонная функция, осуществляющая поиск элемента, для которого некий функтор возвращает **true**, то поиск совпадающего по значению объекта можно реализовать так:

```
1  template < typename InputIterator, typename Operation >
2  InputIterator find(InputIterator begin, const InputIterator & end,
3                     Operation op)
4  {
5    while (begin != end) {
6      if (op(*begin)) {
7        return begin;
8      }
9      ++begin;
10   }
11
12   return end;
13 }
14
15 void bar()
16 {
17   std::vector< Object > objects;
18
19   Object key;
20   ...
21   if (find(objects.begin(), objects.end(),
22       std::bind1st(std::equal_to< Object >(), key)) != objects.end()) {
23     ...
24   }
25 }
```

Стандартные способы связывания позволяют сделать многое, но далеко не все. Поэтому библиотека Boost и стандарт C++ 11 предлагают мощный шаблон bind. Данный шаблон позволяет легко конструировать функторы, использующие многие связанные аргументы, а в качестве базового функтора могут быть использованы указатели на функции, указатели на члены класса (методы *и поля*) и функторы.

Примеры выше могут быть переписаны так:

```
#include <boost/bind.hpp> // C++ 03
using boost::bind;
/* Or
#include <functional>      // C++ 11
using std::bind;
using namespace std::placeholders;
*/

void foo()
{
  std::vector< Object > objects;

  ...
  print(objects.begin(), objects.end(), boost::bind(&Object::getName, _1));
}

void bar()
{
  std::vector< Object > objects;

  Object key;
  ...
  if (find(objects.begin(), objects.end(),
           bind(std::equal_to< Object >(), key, _1)) != objects.end()) {
    ...
  }
}
```

Идентификаторы _1, _2 и так далее обозначают порядковый номер аргумента, передающегося в результирующий функтор.

Применение bind в коде, требующего высокой производительности обычно не оправданно, так как его конструирование являются дорогостоящей операцией. В то же время, его вызов не так дорог и вполне допустим до тех пор, пока профилирование не покажет, что именно это место требует оптимизации. В этот момент можно написать требуемый функтор самостоятельно.

18.3.3 Замыкания

Следующей мощной техникой, крайне широко использующейся в функциональных языках, являются замыкания.

Упрощенно, замыканием является локальная функция, использующая свое окружение, в частности, локальные переменные окружающей функции. В C++ локальных функций не существует. Однако они могут быть имитированы при помощи локальных классов, которые фактически будут функторами. Но на пути применения этой техники стоит ограничение стандарта C++ 03: локальные классы не имеют внешнего связывания и не могут быть параметром шаблона. С другой стороны, они могут реализованы внутри неименованного пространства имен.

Как следствие, замыкание само по себе отсутствующее в C++, может быть сформировано при помощи функтора, имеющего внутреннее состояние. Например, поиск объекта, имя которого совпадает с некоторой строкой может быть реализовано в виде следующего функтора:

```
class compare_name:
    public std::unary_function< Object, bool >
{
public:
  compare_name(const std::string & name):
    name(name)
  {}

  bool operator ()(const Object & obj) const
```

```
10    {
11      return name == object.getName();
12    }
13
14  private:
15    std::string name;
16  };
```

По сути, поле name и будет частью замыкания.

Глава 19

Стандартные алгоритмы

Контейнеры и функторы представляют собой полезные вещи. Но их истинная сила начинает проявляться только тогда, когда они начинают объединятся вместе для решения различных задач. Стандартные алгоритмы обеспечивают такое объединение.

В стандартной библиотеке C++ много алгоритмов на самые разные случаи. Далее рассмотрены самые часто использующиеся, информацию об остальных можно найти в разделе 25 стандарта [C++ 03; C++ 11].

Все алгоритмы, описанные в стандарте не работают с контейнерами. Вместо этого они взаимодействуют с итераторами. Поскольку итератор представляет собой обобщенный указатель, диапазон итераторов прекрасно абстрагирует различные типы контейнеров, массивы и другие сущности, представляющие собой набор объектов, например, потоковый итератор ввода позволяет лениво читать объекты из потока ввода. При этом они могут и не присутствовать в памяти одновременно, но это не мешает использовать стандартные алгоритмы с ним, так как интерфейс итератора скрывает детали реализации и для алгоритма он неотличим от итератора по контейнеру.

Все стандартные алгоритмы находятся в заголовке `algorithm`.

19.1 Немодифицирующие алгоритмы

Немодифицирующие алгоритмы подразумевают сохранение набора объектов неизменным.

for_each

Самым базовым алгоритмом является алгоритм std::for_each().

```
template < typename InputIterator ,
           typename Function >
Function for_each(InputIterator first ,
                  InputIterator last ,
                  Function f);
```

Он предоставляет возможность вызова функтора f для каждого элемента последовательности в диапазоне $[first, last)$. Например:

```
1  typedef std::map< unsigned int ,
2                    boost::shared_ptr< User > >
3          user_map;
4
5  user_map users;
6
7  ...
8
9  std::for_each(users.begin(), users.end(),
10   boost::bind(&User::print ,
```

```
11      boost::bind(&user_map::value_type::second, _1)));
```

В примере предполагается, что класс User содержит метод print(), не принимающий никаких аргументов и выводящий объект на печать. Второй boost::bind в строке 11 требуется, так как std::map хранит пару «идентификатор —пользователь», поэтому объект пользователя необходимо извлечь из user_map::value_type.

У алгоритма std::for_each() есть 2 важные особенности. Во-первых, он позволяет модифицировать объекты диапазона. Для этого функтор должен принимать значение по ссылке на неконстанту. Во-вторых, стандарт ничего не говорит о том, как будет передаваться функтор внутр алгоритма и сколько раз он будет копироваться. Поэтому для получения данных из функтора с состоянием следует использовать возвращаемое значение алгоритма. std::for_each() возвращает экземпляр переданного функтора, которые был вызван для каждого объекта один раз. Например:

```
1  struct print_and_count:
2    public std::unary_function< user_map::value_type,
3                                void >
4  {
5    int admin_count;
6
7    print_and_count():
8      admin_count(0)
9    {}
10
11   void operator ()(const User & user_info) const
12   {
13     user_info.second->print();
14
15     if (user_info.second->isAdmin()) {
16       ++admin_count;
17     }
18   }
19 };
20
21 user_map users;
22
23 ...
24
25 int admins = std::for_each(users.begin(), users.end(),
26   print_and_count()).admin_count;
```

В строке 1 определяется функтор, выполняющий несколько задач:

- вывод на экран объектов;
- подсчет количества пользователей с административными привилегиями.

Для того, чтобы получить количество пользователей с административными привилегиями, на строке 26 используется возвращаемый функтор, содержащий результаты подсчета.

Строго говоря, с точки зрения дизайна, такой функтор неудобен и неправилен, так как он выполняет более одной задачи (печать *и* подсчет). Таким образом, его становится трудно использовать повторно и комбинировать с другими функторами. Но такое решение имеет право на существование в случае оптимизации, так как позволяет простейшим способом избегнуть второго прохода по диапазону.

find

Алгоритм std::find() и его аналог std::find_if осуществляют последовательный поиск в диапазоне:

```
template < typename InputIterator, typename T >
InputIterator find(InputIterator first, InputIterator last,
                   const T & value);
template < typename InputIterator, typename Predicate >
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

std::find() отыскивает указанный элемент в диапазоне и возвращает итератор, указывающий не него. Если такой элемент отсутствует будет возвращен last. Его аналог std::find_if() отыскивает прервый элемент, для которого предикат pred вернет **true**. Например:

```
1  struct is_even:
2    public std::unary_function< int, bool >
3  {
4    bool operator ()(int value) const
5    {
6      return (value % 2) == 0;
7    }
8  };
9
10 std::vector< int > values;
11
12 std::vector< int >::iterator even =
13     std::find_if(values.begin(), values.end(), is_even());
```

В строке 1 определяется функтор, проверяющий целочисленное значение на четность. Таким образом, вызов std::find_if() в строке 13 найдет первое четное число в векторе или вернет values.end().

Как и в случае с std::for_each(), количество копирований функтора не определено, поэтому использование функтора, меняющего свое состояние приводит к некорректным результатам.

accumulate

Алгоритм std::accumulate стоит особняком среди всего списка, так как он вместе с 3-мя реже используемыми алгоритмами находится в заголовке `numeric`. Также, его название несколько необычно и привязано к расчетам, в то время как в других языках он называется «fold» или «reduce».

```
template < typename InputIterator, typename T >
T accumulate(InputIterator first, InputIterator last,
             T init);
template < typename InputIterator, typename T,
           typename BinaryOperation >
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryOperation op);
```

Как обычно, стандарт предоставляет 2 версии алгоритма: для простейшего случая и обобщенный вариант. В простейшем случае алгоритм вычисляет сумму $init + \sum_{i=first}^{last} i$. Для произвольных объектов будет использован **operator** +().

Обобщенный вариант выполняет свертку при помощи функтора op. В это случае вычисления будут эквивалентны следующему коду (для 3 элементов в диапазоне):

```
op(
   op(
      op(init, *first),
      *(first + 1)
     ),
   *(first + 2)
   )
```

Например, фонд заработной платы в американском стиле, когда сотрудники платят все налоги самостоятельно, может быть рассчитан следующим образом:

```
1  std::list< Employee > employees;
2
3  int total_salary = std::accumulate(employees.begin(),
4      employees.end(), 0,
5      boost::bind(std::plus< int >(), _1,
6          boost::bind(&Employee::getSalary, _2)));
```

Простейший вариант с автоматическим использованием сложения в данном примере не подходит, так как заработная плата для каждого сотрудника получается дополнительным вызовом метода (геттера) **int** Employee::getSalary() **const**.

Применение свертки не ограничивается получением атомарных значений. Строго говоря, свертка это один самых фундаментальных алгоритмов для множеств и через нее могут быть выражены практически все алгоритмы. Например, список всех подразделений (Organisation Unit, OU) в организации по списку сотрудников можно получить следующим образом:

```
1  std::list< Employee > employees;
2
3  typedef std::set< Unit > ou_set;
4
5  struct add_org_unit:
6      public std::binary_function< ou_set, Employee, ou_set >
7  {
8    ou_set operator ()(ou_set units,
9                       const Employee & emp) const
10   {
11     units.insert(emp.getOU());
12
13     return units;
14   }
15 };
16
17 out_set units = std::accumulate(employees.begin(),
18     employees.end(), ou_set(),
19     add_org_unit());
```

Так как в качестве контейнера для подразделений используется std::set, который не содержит дубликатов, допустимо просто вставлять каждое подразделение в набор. Если такое подразделение там уже присутствует, вставка ничего не делает.

Это решение не самое быстрое, так как при использовании стандарта C++ 03 будет произведено много операций копирования данных. С другой стороны, компилятор, реализующий C++ 11, в состоянии оптимизировать множественные копирования данных в ou_set, использовав имеющиеся возможности по перемещению данных.

19.2 Модифицирующие алгоритмы

copy

Простейшим модифицирующим алгоритмом является std::copy() вместе его вариантом std::copy_backward():

```
template < typename InputIterator,
           typename OutputIterator >
OutputIterator copy(InputIterator first,
                    InputIterator last,
                    OutputIterator result);
template < typename InputIterator,
                   typename OutputIterator >
OutputIterator copy_backward(InputIterator first,
```

```
                                InputIterator last,
                                OutputIterator result);
```

std::copy копирует объекты в диапазоне $[first, last)$ в диапазон $[result, result + (last - first))$. Алгоритм подразумевает, что целевой диапазон существует. Если целевой диапазон принадлежит, например, какому-либо пустому контейнеру, в качестве result следует использовать итератор вставки.

Например, вывод всех объектов контейнера в поток стандартного вывода может быть осуществлен следующим образом:

```
std::list< Employee > employees;

std::copy(employees.begin(), employees.end(),
  std::ostream_iterator< Employee >(std::cout));
```

std::copy_backward() выполняет аналогичное копирование, но производит его а обратном порядке от last − 1 к first.

transform

Преобразование объектов в диапазоне осуществляет алгоритм std::tranform():

```
template < typename InputIterator,
           typename OutputIterator,
           typename UnaryOperation >
OutputIterator transform(InputIterator first,
                         InputIterator last,
                         OutputIterator result,
                         UnaryOperation op);
template < typename InputIterator1,
           typename InputIterator2,
           typename OutputIterator,
           typename BinaryOperation >
OutputIterator transform(InputIterator1 first1,
                         InputIterator1 last1,
                         InputIterator2 first2,
                         OutputIterator result,
                         BinaryOperation op);
```

Первая версия алгоритма обрабатывает один диапазон, обновляя результирующий диапазон. Вторая версия параллельно обрабатывает 2 диапазона $[first1, last1)$ и $[first2, first2 + (last1 - first1))$. Как обычно, подразумевается, что result указывает на начало диапазона, способного вместить $last1 - first1$ объектов.

Например, список геометрических фигур может быть преобразован в список из площадей:

```
1 std::vector< std::shared_ptr< Figure > > geometry;
2
3 std::vector< double > areas;
4
5 std::transform(geomery.begin(), geometry.end(),
6     std::back_inserter(areas),
7     boost::bind(&Figure::getArea, _1));
```

Результирующий итератор не обязан указывать в другой контейнер. Для изменения объектов «на месте» (in-place) в качестве result можно передать first (или, для второй версии, first1 или first2).

remove

Для удаления элементов из последовательности служит алгоритм std::remove:

```
template < typename ForwardIterator, typename T >
ForwardIterator remove(ForwardIterator first,
                       ForwardIterator last,
                       const T & value);
template < typename ForwardIterator,
           typename Predicate >
ForwardIterator remove_if(ForwardIterator first,
                          ForwardIterator last,
                          Predicate pred);
```

std::remove удаляет элементы, совпадающие (в терминах **operator** ==()) с переданным значением, std::remove_if() удаляет все элементы, для которых предикат возвращает **true**. Оба алгоритма возвращают итератор, указывающий за последний элемент модифицированного диапазона. Так как алгоритмы не принимают контейнер в качестве аргумента, они не могут физически удалить данные из контейнера. В действительности, они всего лишь перемещают данные внутри диапазона. Например, для списка целых чисел:

1	2	5	4	5	6

Удаление числе 5 из контейнера приведет к тому, что контейнер будет иметь вид:

1	2	4	6	5	6

Алгоритм std::remove() вернет итератор, указывающий на число 5 в результирующем контейнере. Фактически, std::remove() переместит все данные, подлежащие сохранению, на место удаляемых элементов, поэтому конец диапазона остается неизменным. Для физического удаления элементов из контейнера необходимо вызвать remove():

```
1 std::vector< int > values;
2
3 values.erase(std::remove(values.begin(), values.end(), 5),
4     values.end());
```

Совместный вызов remove() и erase() представляет собой стандартную идиому языка.

19.3 Сортировка

Стандарт определяет различные виды сортировок, начиная от простейших и заканчивая частичной сортировкой диапазона. Наиболее употребимыми являются простая сортировка диапазона и стабильная сортировка.

```
template < typename RandomAccessIterator >
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
template < typename RandomAccessIterator,
           typename Compare >
void sort(RandomAccessIterator first,
          RandomAccessIterator last,
          Compare comp);
```

Первая версия сортировки сортирует объекты в диапазоне $[first, last)$, используя для сравнения элементов **operator** <(). Эта версия хорошо работает, когда объекты необходимо отсортировать в «естественном» порядке, например, фамилии по алфавиту. Вторая версия позволяет задать произвольный критерий сортировки. К этому критерию предъявляются те же требования, что и к компаратору, использующемуся в ассоциативных контейнерах (см. 17.1.1).

Оба этих алгоритма реализуют сортировку со сложностью $n \log_2^n$ сравнений объектов в среднем. При этом относительный порядок объектов, для которых выполняется условие $(a < b) \wedge (b < a)$ может не сохраняться. В некоторых случаях это недопустимо, поэтому в стандартной библиотеке также присутствуют алгоритмы стабильной сортировки:

```
template < typename RandomAccessIterator >
void stable_sort(RandomAccessIterator first,
                 RandomAccessIterator last);
template < typename RandomAccessIterator,
           typename Compare >
void stable_sort(RandomAccessIterator first,
                 RandomAccessIterator last,
                 Compare comp);
```

Эти алгоритмы функционируют аналогично простейшим, но гарантируют сохранение порядка эквивалентных в терминах comp элементов. Алгоритмическая сложность std::stable_sort() выше: гарантируется, что она не хуже чем $n(\log_2^n)^2$ сравнений.

Все алгоритмы сортировки описаны в разделах 25.3.1 и 25.3.2 [C++ 03].

19.4 Поиск

С алгоритмами сортировки тесно связаны алгоритмы поиска. Простейший последовательный поиск был рассмотрен ранее. Все последующие алгоритмы опираются на тот факт, что диапазон поиска отсортирован в соответствии с тем же критерием, что и использующийся при поиске.

Простейшим вариантом поиска в отсортированном диапазоне является бинарный поиск:

```
template < typename ForwardIterator,
           typename T >
ForwardIterator binary_search(ForwardIterator first,
                              ForwardIterator last,
                              const T & value);
template < typename ForwardIterator,
           typename T,
           typename Compare >
ForwardIterator binary_search(ForwardIterator first,
                              ForwardIterator last,
                              const T & value,
                              Compare comp);
```

Как и все прочие алгоритмы, первый вариант использует **operator** <(), а второй — явно переданный компаратор comp. Возвращаемое значение также аналогично std::find — итератор, указывающие на найденный элемент и last, если такого элемента нет.

Следующий набор алгоритмов позволяет ответить на один из двух дополнительных вопросов (или даже на оба) :

1. Если такого элемента нет, то где бы находился первый эквивалентный?
2. Если такого элемента нет, то где бы находился последний эквивалентный?

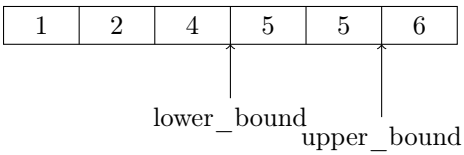
```
template < typename ForwardIterator, typename T >
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T & value);
template < typename ForwardIterator, typename T, typename Compare >
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T & value,
                            Compare comp);

template < typename ForwardIterator, typename T >
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
```

```
                                const T & value);
template < typename ForwardIterator, typename T, typename Compare >
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                                const T & value,
                                Compare comp);

template < typename ForwardIterator, typename T >
pair< ForwardIterator, ForwardIterator >
    equal_range(ForwardIterator first, ForwardIterator last,
                const T & value);
template < typename ForwardIterator, typename T, typename Compare >
pair< ForwardIterator, ForwardIterator >
    equal_range(ForwardIterator first, ForwardIterator last,
                const T & value,
                Compare comp);
```

Алгоритм std::lower_bound() возвращает итератор, указывающий на первый эквивалентный элемент или, если такого элемента нет, на то место, куда его можно вставить не нарушая требования отсортированности. std::upper_bound() возвращает итератор, указывающий на элемент, находящийся после последнего эквивалентного. Алгоритм std::equal_range() представляет собой объединение первых двух, он возвращает пару итераторов, первый из которых аналогичен вызову std::lower_bound(), а второй — std::upper_bound(). Проще всего это проиллюстрировать на примере. Для вектора целых чисел, отсортированных по возврастанию, эти алгоритмы для числа 5 вернут следующие позиции:



Чаще всего эти алгоритмы используются для вставки или обновления содержимого контейнера. Например, для списка сотрудников вставка нового или обновление существующего может быть выполнено следующим образом:

```
 1  class Employee
 2  {
 3  public:
 4    bool operator <(const Employee & rhs) const
 5    {
 6      ...
 7    }
 8  };
 9  typedef std::list< Employee > employee_list;
10  typedef employee_list::iterator employee_iterator;
11  ...
12
13  void insert_new_employee(employee_list & employees,
14                           const Employee & emp)
15  {
16    std::pair< employee_iterator, employee_iterator > range =
17        std::equal_range(employees.begin(), employees.end(), emp);
18
19   if (range.first == range.second) {
20     employees.insert(range.first, emp);
21   } else {
22     *range.first = emp;
23   }
24  }
```

В строке 19 проверяется равенство итераторов, полученных из вызова std::equal_range(). Если они равны, то такого сотрудника нет в списке и его надо добавить в это место. Если же они неравны, то необходимо просто обновить данные. Эта методика часто применяется для std::map.

19.5 Алгоритмы и встроенные функции

Некоторые алгоритмы также представлены в виде методов контейнеров. Например, std::list<>::sort(), std::map<>::equal_range() и т.д. Это связано с одной из 2-х причин:

1. Невозможно реализовать внешний алгоритм, это справедливо для std::list<>::sort(), так как std::sort() требует итераторов прямого доступа, в то время как у std::list<> только двунаправленные.
2. Встроенный метод эффективнее. Например, std::map<>::equal_range() использует внутреннее знание об устройстве дерева, а std::list<>::sort() вместо обмена значений просто меняет указатели в узлах, что более эффективно для больших объектов, хранящихся в контейнере.

Например, для std::set<> вторая реализация эффективнее:

```
1  typedef std::set< Employee > employee_set;
2
3  employee_set::iterator algo(employee_set & employees,
4                              const Employee & emp)
5  {
6    return std::lower_bound(employees.begin(),
7                            employees.end(),
8                            emp);
9  }
10
11 employee_set::iterator method(employee_set & employees,
12                               const Employee & emp)
13 {
14   return employees.lower_bound(emp);
15 }
```

Приложение А

Применение Test Driven Development в лабораторных работах

При разработке любого программного продукта, включая лабораторные работы и курсовые проекты, написание кода обычно занимает только малую часть. Остальное время занимают:

- проверка результата на соответствие требованиям или заданию;
- тестирование на отсутствие ошибок, не охватываемых требованиями либо являющихся неявными, например, отсутствие утечек ресурсов, выходов за границу массива и т.п.;
- написание сопровдительной документации.

Основное время жизни любого продукта проходит в стадии сопровождения, когда в него добавляются новые возможности и справляются найденные ошибки. В процессе исправления ошибок могут вноситься новые. Для того, чтобы убедится в их отсутствии, процесс тестирования необходимо повторить. Более того, только очень хорошее знание всех взаимосвязей в системе позволяет сказать, на какие части влияет одно изменение. Зачастую такое предсказание невозможно и процесс тестирования проходится проводить для всей системы. Поскольку тестирование по неизменившимся требованиям сильно напоминает рутинный процесс, его стараются автоматизировать.

Отдельная методология, называемая Test-Driven Developement (TDD), рассматривает весь процесс разработки с позиций тестирования: сначала разработываются автоматические тесты, которые, естественно, правильно не работают, и только потом разрабатывается код основной функциональности так, чтобы тесты показывали его корректную работу.

А.1 Задача

Разработать функцию, вычисляющую площадь треугольника по трем сторонам.

$$S = \sqrt{p(p-a)(p-b)(p-c)} \tag{А.1}$$

$$p = \frac{a+b+c}{2} \tag{А.2}$$

А.2 Этапы решения

А.2.1 Техническое задание

Первым этапом является разработка технического задания, которое дополнительно включает в себя описание интерфейса компонента и его контракт, а также некоторые детали реализации, существенные для исполнителя.

Для задачи с площадью треугольника можно сформировать следующий интерфейс:

1. Функция принимает 3 значения типа **double** и возвращает значение площади типа **double**.
2. В случае если одна из сторон меньше либо равна 0, функция возвращает 0 и устанавливает глобальную переменную errno в значение EINVAL.
3. Если 3 стороны не формируют треугольник, функция возвращает 0 и устанавливает глобальную переменную errno в значение EINVAL.

Соответственно, прототип функции будет выглядеть следующим образом:

```
#ifndef LAB_TRIANGLE_AREA
#define LAB_TRIANGLE_AREA

double triangle_area(double a, double b, double c);

#endif
```

А.2.2 Разработка тестовых сценариев

Перед реализацией тестов необходимо разработать тестовые сценарии (test cases), в которых описывается при каких входных данных какие результаты и какое поведение ожидается. Хорошо разработанные сценарии должны покрывать все возможные (дрпустимые и недопустимые) входные значения.

Например, для данной задачи тестовые сценарии должны покрывать следующие случаи:

1. Проверка на корректных данных, например, $a = 3, b = 3, c = 2, S = 8$;
2. Проверки на некорректных данных:
   (a) Отрицательные стороны (3 проверки);
   (b) Нулевые стороны (3 проверки);
   (c) Стороны не формируют треугольник (3 проверки)

Итого, все должно быть не менее 10 тестовых сценариев.

А.2.3 Разработка тестов

В рамках TDD следующим этапом идет не разработка функции расчета площади, а разработка тестов. Каждый сценарий должен представлять собой функцию, не принимающую никаких аргументов и возвращающую **true**, если тест получил ожидаемое значение и **false** в противном случае. Например, для проверки на корректных данных можно написать следующий код:

```
 1 #include <iostream>
 2 #include <cmath>
 3
 4 #include <errno.h>
 5
 6 #include "triangle_area.h"
 7
 8 const double EPSILON = 0.0001;
 9 const double a = 3.0;
10 const double b = 3.0;
11 const double c = 2.0;
12 const double S = 8;
13
14 bool correct_data()
15 {
16   const double s = triangle_area(a, b, c);
17
18   if (std::abs(S - s) > EPSILON) {
19     std::cerr << "Failure: a = " << a << ", b = "
20               << b << ", c = " << c
```

```
21                << ", S = " << s << ", expected to be "
22                << S << std::endl;
23
24      return false;
25    }
26
27    return true;
28  }
```

Очевидно, что сравнивать значения с плавающей запятой необходимо с учетом погрешности, поэтому за точность сравнения отвечает константа EPLSION.

Тесты на некорректных данных также достаточно просты. Например, по одному тесту каждого вида:

```
1   bool negative_data_a()
2   {
3     if (double s = triangle_area(-1, b, c)) {
4       std::cerr << "Failure: a < 0, S = " << s
5                 << ", expected to be 0" << std::endl;
6       return false;
7     } else if (errno != EINVAL) {
8       std::cerr << "Failure: a < 0, errno is not EINVAL"
9                 << std::endl;
10      return false;
11    }
12
13    return true;
14  }
15  bool zero_data_a()
16  {
17    if (double s = triangle_area(0, b, c)) {
18      std::cerr << "Failure: a = 0, S = " << s
19                << ", expected to be 0" << std::endl;
20      return false;
21    } else if (errno != EINVAL) {
22      std::cerr << "Failure: a = 0, errno is not EINVAL"
23                << std::endl;
24      return false;
25    }
26
27    return true;
28  }
29  bool non_triangle_data1()
30  {
31    if (double s = triangle_area(6, 1, 1)) {
32      std::cerr << "Failure: a = 6, b = 1, c = 1, S = "
33                << s << ", expected to be 0" << std::endl;
34      return false;
35    } else if (errno != EINVAL) {
36      std::cerr << "Failure: a = 6, b = 1, c = 1, errno"
37          ", is not EINVAL" << std::endl;
38      return false;
39    }
40
41    return true;
42  }
```

Для того, чтобы выполнять все эти тесты необходимо написать небольшой фреймворк. Устройство фреймворка просто:

- адреса всех тестовых функций помещаются в один массив;

- в функции main() все функции запускаются одна за другой;
- подсчитывается количество успешных и неуспешных тестов;
- в конце выводится статистика;
- результат выполнения программы (exit code) будет 0, если все тесты успешны, и 1 в противном случае.

```
1  using test_function_t = bool (*)();
2
3  const test_function_t TESTS[] = {
4    &correct_data,
5    &negative_data_a,
6    &negative_data_b,
7    &negative_data_c,
8    &zero_data_a,
9    &zero_data_b,
10   &zero_data_c,
11   &non_triangle_data1,
12   &non_triangle_data3,
13   &non_triangle_data3
14 };
15
16 int main(int, char *[])
17 {
18   int failures = 0;
19
20   std::cout << "Running␣"
21             << sizeof(TESTS) / sizeof(TESTS[0])
22             << "␣tests" << std::endl;
23
24   for (size_t i = 0;
25        i < sizeof(TESTS) / sizeof(TESTS[0]);
26        ++i) {
27     if (!TESTS[i]()) {
28       ++failures;
29     }
30   }
31   std::cout << failures << "␣of␣"
32             << sizeof(TESTS) / sizeof(TESTS[0])
33             << "␣are␣failed" << std::endl;
34
35   return (failures == 0) ? 0 : 1;
36 }
```

После разработки тестов можно переходить к реализации функции triangle_area() так, чтобы все тесты проходили успешно.

А.3 Последующие шаги

Такой набор тестов, которые не требуют никакого специального окружения и выполняются быстро, обычно включают в процесс сборки компонента, чтобы сразу была доступна информация о его работоспособности. В дальнейшем, в процессе сопровождения и исправления ошибок набор тестов расширяется. Например, исправление ошибки в компоненте начинается с добавления теста, который ее демонстрирует своей неуспешностью. После этого компонент исправляется так, чтобы все тесты опять стали успешными.

Опыт показывает, что несмотря на больший объем работы при начальной разработке, наличие тестов сильно ускоряет сопровождение: каждый раз, когда ломается поведение компонента в покрытой тестами части, набор тестов становится неуспешен. Таким образом, ошибки исправляются на самом раннем этапе.

Более детально весь процесс разработки через тестирование и его влияния на полный цикл разработки программного обеспечения можно найти в книге Кента Бека «Экстремальное программирование: разработка через тестирование» [Бек03].

Приложение B

Задания к лабораторным работам

Курс сопровождается заданиями для самостоятельной работы студентов. Эти задания с минимальными изменениями были взяты с предыдущего года. Вследствие их ориентированности на академический вариант использования C++ предложенные варианты интерфейсов и подсказки слабоприменимы при промышленной разработке. Это открывает возможность студентам проявить себя, предложив более удачные варианты интерфейсов и решений.

B.1 Векторы

Необходимо выполнить *все* задания.

1. Напишите алгоритм сортировки (любой простейший) содержимого вектора целых чисел, используя оператор **operator** [].
2. Напишите алгоритм сортировки (любой простейший) содержимого вектора целых чисел, используя метод at().
3. Напишите алгоритм сортировки (любой простейший) содержимого вектора целых чисел, используя для доступа к содержимому вектора только итераторы. Для работы с итераторами допустимо использовать только операторы получения текущего элемента и перехода в следующему (подсказка: можно сохранять копию итератора указывающего на некоторый элемент).
4. Прочитайте во встроенный массив C содержимое текстового файла, скопируйте данные в вектор одной строкой кода (без циклов и алгоритмов STL).
5. Напишите программу, сохраняющую в векторе числа, полученные из стандартного ввода (окончанием ввода является число 0). Удалите все элементы, которые делятся на 2 (не используя стандартные алгоритмы STL), если последнее число 1. Если последнее число 2, добавьте после каждого числа, которое делится на 3, три единицы.
6. Напишите функцию **void** fillRandom(**double** $*$ array, **int** size), заполняющую массив случайными значениями в интервале от -1.0 до $+1.0$. Заполните с помощью заданной функции вектора размером 5, 10, 25, 50, 100 и отсортируйте его содержимое (с помощью любого разработанного ранее алгоритма, модифицированного для сортировки действительных чисел).

B.2 Строки

Разработать программу, которая должна сделать следующее:

1. Прочитать содержимое текстового файла. Файл может содержать:
   (a) Слова — состоят из латинских строчных и заглавных букв, а также цифр, длина слова должна быть не более 20 символов.
   (b) Знаки препинания — «.», «,», «!», «?», «:», «;».
   (c) Пробельные символы — пробел, табуляция, символ новой строки.
2. Отформатировать текст следующим образом:

(a) Не должно быть пробельных символов отличных от пробела.
(b) Не должно идти подряд более одного пробела.
(c) Между словом и знаком препинания не должно быть пробела.
(d) После знака препинания всегда должен идти пробел.
(e) Слова длиной более 10 символов заменяются на слово «Vau!!!».

3. Преобразовать полученный текст в набор строк, каждая из которых содержит целое количество слов (слово должно целиком находиться в строке) и ее длина не превышает 40 символов.

Подсказки:

- Для хранения всего входного текста можно использовать одну строку.
- Можно создать строки, содержащие символы принадлежащие какой-либо категории, например знаки препинания.
- Для хранения результирующих строк можно использовать std :: vector< std:: string >.

B.3 Последовательности

1

Ниже приведен интерфейс класса очереди с приоритетами, который функционирует следующим образом:

В очередь могут быть добавлены элементы, каждому элементу при добавлении присваивается один из трех уровней приоритета (low, normal, high).

Элементы из очереди извлекаются в соответствии с их приоритетами (сначала извлекаются элементы с приоритетом high, потом normal, потом low), элементы с одинаковыми приоритетами извлекаются из очереди в порядки их поступления.

В очереди также может происходить операция акселерации — все элементы с приоритетом low находящиеся в момент акселерации в очереди увеличивают свой приоритет до high и «обгоняют» элементы с приоритетом normal.

Ниже приведен интерфейс этого класса:

```
 1  typedef enum
 2  {
 3    LOW,
 4    NORMAL,
 5    HIGH
 6  } ElementPriority;
 7
 8  typedef struct
 9  {
10    std::string name;
11  } QueueElement;
12
13  class QueueWithPriority
14  {
15    QueueWithPriority();
16
17    ~QueueWithPriority();
18
19    void PutElementToQueue(const QueueElement & element,
20        ElementPriority priority);
21
22    QueueElement GetElementFromQueue();
23
24    void Accelerate();
25  };
```

Реализовать этот класс, используя std :: list <> или std :: deque<>. Объяснить свой выбор. Протестируйте программу, добавьте отладочный вывод, показывающий, что класс работает правильно.

2

Разработайте программу, которая:

1. Заполняет std :: list < **int** > 15 случайными значениями от 1 до 20, список может содержать от 0 до 20 значений (обязательно проверить на длине списка 0, 1, 2, 3, 4, 5, 7, 14).
2. Выводит содержимое списка в следующем порядке: первый элемент, последний элемент, второй элемент, предпоследний элемент, тритий элемент и т.д.

Например если список содержит:

```
1 2 3 4 5 6 7 8
```

то вывод будет иметь вид

```
1 8 2 7 3 6 4 5
```

Подсказка: можно использовать рекурсию и двунаправленные итераторы.

B.4 Итераторы

Выполните *одно из* заданий:

1

Напишите программу-«телефонную книжку».

Записи (имя и телефон) должны хранится в каком-либо STL-контейнере (std :: vector<> или std :: list <>), причем крайне желательно, чтобы от типа контейнера не зависело ничего, кроме одной строки в программе — объявления этого контейнера (указание: используйте **typedef**).

Программа должна поддерживать следующие операции:

- Просмотр текущей записи.
- Переход к следующей записи.
- Переход к предыдущей записи.
- Вставка записи перед/после просматриваемой.
- Замена просматриваемой записи.
- Вставка записи в конец базы данных.
- Переход вперед/назад через n записей.

Помните, что после вставки элемента итераторы становятся недействительными, поэтому после вставки целесообразно переставлять итератор на начало базы данных.

Постарайтесь реализовать операции вставки и замены с помощью одной и той же функции, которой в зависимости от требуемого действия передается либо обычный итератор, либо адаптер — один из итераторов вставки: **void** modifyRecord(iterator pCurrentRecord, CRecord newRecord). Программа может сразу после запуска сама (без команд пользователя) заполнить записную книжку некоторым количеством записей.

2

Реализуйте следующие классы:

- Контейнер, который содержит значения факториала от 1! до 10!.
  Интерфейс класса должен включать в себя как минимум:
  - Конструктор по умолчанию.
  - Функцию получения итератора указывающего на первый элемент контейнера — begin().
  - Функцию получения итератора указывающего на элемент, следующий за последним — end().

  Доступ к элементам этого контейнера возможен только с помощью итераторов, возвращаемых функциями begin() и end().

  Контейнер не должен содержать в памяти свои элементы, они должны вычисляться при обращении к ним через итератор.
- Класс итератора для перечисления элементов этого контейнера, объекты этого класса возвращаются функциями begin() и end(). Итератор должен быть двунаправленным. Итератор должен быть совместимым с STL (проверить это можно используя алгоритм std :: copy для копирования содержимого разработанного контейнера в std :: vector< **int** >).

B.5 Алгоритмы I

Написать программу, которая выполняет следующие действия:

1. Заполняет std :: vector< DataStruct > структурами DataStruct, при этом key1 и key2 генерируются случайным образом в диапазоне от −5 до +5, str заполняется из таблицы (таблица содержит 10 произвольных строк, индекс строки генерируется случайным образом).
2. Выводит полученный вектор на печать.
3. Сортирует вектор следующим образом:
   (a) По возрастанию key1.
   (b) Если key1 одинаковые, то по возрастанию key2.
   (c) Если key1 и key2 одинаковые, то по возрастанию длины строки str.
4. Выводит полученный вектор на печать

DataStruct определена следующим образом:

```
1 typedef struct
2 {
3   int         key1;
4   int         key2;
5   std::string str;
6 } DataStruct;
```

B.6 Алгоритмы II

1

Написать программу, которая выполняет следующие действия:

1. Читает содержимое текстового файла.
2. Выделяет слова, словом считается последовательность символов, разделенных пробелами и/или знаками табуляции и/или символами новой строки.
3. Вывести список слов, присутствующий в тексте без повторений (имеется в виду, что одно и то же слово может присутствовать в списке только один раз).

2

Написать программу, которая выполняет следующие действия (все операции должны выполняться с помощью стандартных алгоритмов):

1. Заполняет вектор геометрическими фигурами. Геометрическая фигура может быть треугольником, квадратом, прямоугольником или пятиугольником. Структура описывающая геометрическую фигуру определена ниже.
2. Подсчитывает общее количество вершин всех фигур, содержащихся в векторе (так треугольник добавляет к общему числу 3, квадрат 4 и т.д.).
3. Подсчитывает количество треугольников, квадратов и прямоугольников в векторе.
4. Удаляет все пятиугольники.
5. На основании этого вектора создает std::vector< Point >, который содержит координаты одной из вершин (любой) каждой фигуры, т.е. первый элемент этого вектора содержит координаты одной из вершин первой фигуры, второй элемент этого вектора содержит координаты одной из вершин второй фигуры и т.д.
6. Изменяет вектор так, чтобы он содержал в начале все треугольники, потом все квадраты, а потом прямоугольники.
7. Распечатывает вектор после каждого этапа работы.

Геометрическая фигура задается следующей структурой:

```
typedef struct
{
  int                    vertex_num;
  std::vector< Point > vertexes;
} Shape;

typedef struct
{
  int x,y;
} Point;
```

Подсказка: кроме алгоритмов рассмотренных в этой работе можно применять все средства описанные в предыдущих работах, включая алгоритмы сортировки.

B.7 Функторы I

Разработать функтор, позволяющий собирать статистику о последовательности целых чисел (последовательность может содержать числа от -500 до 500). Функтор после обработки последовательности алгоритмом std::for_each должен предоставлять следующую статистику:

1. Максимальное число в последовательности.
2. Минимальное число в последовательности.
3. Среднее чисел в последовательности.
4. Количество положительных чисел.
5. Количество отрицательных чисел.
6. Сумму нечетных элементов последовательности.
7. Сумму четных элементов последовательности.
8. Совпадают ли первый и последний элементы последовательности.

Проверить работу программы на случайно сгенерированных последовательностях.

В.8 Функторы II

1

Разработать программу, которая, используя только стандартные алгоритмы и функторы, умножает каждый элемент списка чисел с плавающей точкой на число π.

2

Разработать программу, которая:

1. Реализует иерархию геометрических фигур состоящую из:
   (a) Класс Shape, содержащий:
   - информацию о положении центра фигуры (координаты x и y);
   - метод IsMoreLeft, позволяющий определить расположена ли данная фигура левее (определяется по положению центра) чем фигура, переданная в качестве аргумента;
   - метод IsUpper, позволяющий определить расположена ли данная фигура выше (определяется по положению центра) чем фигура, переданная в качестве аргумента;
   - чисто виртуальную функцию рисования Draw (каждая фигура в реализации этой функции должна выводить на стандартный вывод свое название и положение центра).

   (b) Класс Circle, производный от класса Shape.
   (c) Класс Triangle, производный от класса Shape.
   (d) Класс Square, производный от класса Shape.
2. Содержит список std :: list < Shape ∗ >, заполненный указателями на различные фигуры.
3. С помощью стандартных алгоритмов и адаптеров выводит все фигуры.
4. С помощью стандартных алгоритмов и адаптеров сортирует список по положению центра слева-направо (имется в виду, что в начале списка должны идти фигуры находящиеся левее) и выводит фигуры.
5. С помощью стандартных алгоритмов и адаптеров сортирует список по положению центра справа-налево и выводит фигуры.
6. С помощью стандартных алгоритмов и адаптеров сортирует список по положению центра сверху-вниз и выводит фигуры.
7. С помощью стандартных алгоритмов и адаптеров сортирует список по положению центра снизу-вверх и выводит фигуры.

Приложение C

Замечания по реализации лабораторных работ

C.1 Векторы

Это задание является прекрасной возможностью воспользоваться принципами Test Driven Development (TDD). Программа может быть реализована в виде комплекта тестов, каждый из которых содержит несколько тест-кейсов, проверяющих отдельные аспекты каждого пункта.

Для выполнения всех заданий было разрешено использовать возможности библиотеки C++ TR1. Также можно было использовать библиотеку Boost [BOOST] при наличии обоснования, умные указатели и Boost.Test допустимо использовать без обоснования.

Эта работа естественным образом разбивается на 3 самостоятельные части.

Сортировки

Задания 1, 2, 3 и 6 требуют реализации произвольного алгоритма сортировки с незначительными отличиями в виде способа доступа к элементам и реализации функции fillRandom().

Прямое решение приводит к появлению набора функций, отличающихся только в деталях. Для предотвращения излишнего копирования кода могут быть предприняты следующие шаги:

1. Написание одной из функций сортировки в виде шаблона вида

   ```
   template < typename T, typename Allocator >
   void sort(std::vector< T, Allocator > & data);
   ```

   Второй аргумент Allocator необходим для обработки любого вектора, в том числе и с пользовательским аллокатором.

   Это решение позволяет сортировать одной функцией целые числа и числа с плавающей запятой.

2. Использование шаблона проектирования «Стратегия». Так как 2 первых функции сортировки отличаются только способом доступа по индексу (оператор индексации и функция at()), возможно написать 2 класса, которыми можно параметризовать функцию сортировки:

   ```
   1 struct index_access
   2 {
   3   template < typename T, typename Allocator >
   4   static T & element(std::vector< T, Allocator > & data,
   5       typename std::vector< T, Allocator >::size_type index)
   6   {
   7     return data[index];
   8   }
   9 };
   ```

```
10
11 struct at_access
12 {
13   template < typename T, typename Allocator >
14   static T & element(std::vector< T, Allocator > & data,
15       typename std::vector< T, Allocator >::size_type index)
16   {
17     return data.at(index);
18   }
19 };
20
21 template < typename Access, typename T, typename Allocator >
22 void sort(std::vector< T, Allocator > & data)
23 {
24   typedef std::vector< T, Allocator > vector_type;
25   typedef typename vector_type::size_type size_type;
26
27   for (size_type i = 0u; i != data.size(); ++i)
28   {
29     for (size_type j = i; j != data.size(); ++j)
30     {
31       if (Access::element(data, j) < Access::element(data, i))
32       {
33         std::swap(Access::element(data, j),
34                   Access::element(data, i));
35       }
36     }
37   }
38 }
39
40 ...
41
42 std::vector< int > v;
43
44 sort< index_access >(v);
45 sort< at_access >(v);
```

Классы index_access и at_access определяют стратегии доступа к элементам: оператором индексации и при помощи метода at(). Каждая стратегия реализует статическую шаблонную функцию element(), которая возвращает ссылку на элемент с соответствующим индексом.

Функция sort() реализует простейшую сортировку, используя для доступа к элементам статическую функцию element() первого параметра шаблона.

Этот шаг позволяет сократить число функций сортировки в программе до 2-х: универсальная, способная сортировать любые данные в векторе через индексацию и метод at(), и сортировка через итераторы.

3. Следующим шагом по сокращению количества сортирующих функций может быть расширении стратегии доступа так, чтобы она могла работать с итераторами. Функция sort() требует индекса первого элемента и индекса элемента сразу за последним. Оператор инкремента обеспечивает переход к следующему элементу и для индексов и для итераторов. Таким образом, добавив методы begin() и end(), а также определение типа в стратегию, можно получить:

```
1 template < typename Vector >
2 struct index_access
3 {
4   typedef typename Vector::size_type iterator;
5   typedef typename Vector::value_type value_type;
6
7   static iterator begin(Vector & /*data*/)
8   {
```

```
 9     return 0u;
10   }
11
12   static iterator end(Vector & data)
13   {
14     return data.size();
15   }
16
17   static value_type & element(Vector & data,
18                               iterator index)
19   {
20     return data[index];
21   }
22 };
23
24 template < typename Vector >
25 struct at_access
26 {
27   typedef typename Vector::size_type iterator;
28   typedef typename Vector::value_type value_type;
29
30   static iterator begin(Vector & /*data*/)
31   {
32     return 0u;
33   }
34
35   static iterator end(Vector & data)
36   {
37     return data.size();
38   }
39
40   static value_type & element(Vector & data,
41                               iterator index)
42   {
43     return data.at(index);
44   }
45 };
46
47 template < typename Vector >
48 struct iterator_access
49 {
50   typedef typename Vector::iterator iterator;
51   typedef typename Vector::value_type value_type;
52
53   static iterator begin(Vector & data)
54   {
55     return data.begin();
56   }
57
58   static iterator end(Vector & data)
59   {
60     return data.end();
61   }
62
63   static value_type & element(Vector & /*data*/,
64                               iterator index)
65   {
66     return *index;
67   }
68 };
69
```

```
70  template < template < typename > class Access,
71             typename T,
72             typename Allocator >
73  void sort(std::vector< T, Allocator > & data)
74  {
75    typedef Access< std::vector< T, Allocator > > access;
76    typedef typename access::iterator iterator;
77    for (iterator i = access::begin(data);
78         i != access::end(data);
79         ++i)
80    {
81      for (iterator j = i; j != access::end(data); ++j)
82      {
83        if (access::element(data, j) < access::element(data, i))
84        {
85          std::swap(access::element(data, j),
86                    access::element(data, i));
87        }
88      }
89    }
90  }
91
92  ...
93
94  std::vector< int > ints;
95
96  sort< index_access >(ints);
97  sort< at_access >(ints);
98  sort< iterator_access >(ints);
99
100 std::vector< double > doubles;
101
102 sort< index_access >(doubles);
103 sort< at_access >(doubles);
104 sort< iterator_access >(doubles);
```

Это решение длиннее, но практически не содержит повторяющегося кода. Это позволяет уменьшить количество тестов, так как необходимо протестировать 3 стратегии, 1 функцию сортировки, и 2 варианта типов данных в векторе.

Что же касается функции fillRandom(), ее реализация обчно не доставляет трудностей, за исключением вызова функции std::srand(). Этот вызов должен происходить внутри функции main(), а не внутри fillRandom().

Вызов же функции fillRandom() должен осуществляться путем передачи адреса первого элемента непустого вектора.

Чтение данных

Задание 4 сформулировано не очень четко, но раз упоминаются встроенный массив и текстовый файл, то можно предположить, что необходимо заполнить std::vector< **char** > данными из файла.

Главные трудности в этом задании связаны с неизвестным размером файла. Размер файла можно определить при помощи методов seekg() и tellg() потоков ввода вывода. Но это не гарантирует правильного размера данных после чтения, так как файл может быть изменен в процессе чтения. Метод gcount() позволяет определить количество прочитанных символов в последней операции неформатированного ввода. При помощи этих функций можно написать цикл, который определяет размер файла и читает данные из него до тех пор, пока не определенный размер не совпадет с размером реально прочитанных данных.

Вторая трудность вытекает из того, что размер файла неизвестен на этапе компиляции и, как следствие, массив требует динамического размещения при помощи оператора **new**. Во

избежание утечки памяти, этот массив должен быть немедленно помещен в умный указатель, например, boost::shared_array<>.

Модификация вектора

Задание 5 содержит 2 коварных момента: инвалидация итераторов и проблемы с чтением чисел.

При модификации вектора необходимо помнить, что итераторы могут стать невалидными при любой модификации контейнера, поэтому необходимо использовать возвращаемое значение методов insert() и erase().

Состояние End-Of-File (EOF), а также различные ошибки ввода-вывода, возможны во время чтения и при работе с std::cin. В результате чтения данных ввод может оканчиваться состоянием EOF, а не 0 в качестве прочитанного числа. Так же возможны варианты, когда:

1. последнее число не 1 и не 2;
2. вектор пуст, так как ввели только 0.

Все эти варианты должны учитываться при написании цикла чтения данных и при обработке вектора во избежание бесконечных циклов и неопределенного поведения.

При написании кода, модифицирующего вектор, целесообразно поместить 2 случая (когда вектор заканчивается на 1 и когда на 2) в 2 разных функции.

Оптимальный алгоритм удаления четных чисел выглядит так: все нечетные числа смещаются в начало, замещая собой четные. После этой операции в векторе образуется «хвост» из оставшихся чисел, которые были перемещенны. Размер этого «хвоста» равен количеству четных чисел. Они должны быть удалены вызовом erase().

Наиболее неэффективной частью вставки единиц после чисел, кратных 3 являются множественные переразмещения памяти внутри вектора. Проведя предварительный подсчет количества вставляемых единиц можно улучшить производительность, использовав вызов reserve().

C.2 Строки

Задание на обработку строк изначально подразумевало применение методов класса std::string. С академической точки зрения этот подход правилен, но в результате получается крайне запутанный и трудно понимаемый код.

В промышленной системе решение подобных задач обычно разбивается на 2 этапа:

1. Чтение исходных данных во внутреннее представление.
2. Формирование результатов из внутреннего представления.

Все искусство заключается в том, чтобы выбрать внутреннее представление.

В этой задаче процесс чтения данных имеет конкретное название: лексический разбор. Входной поток (текст) разбивается на элементы (токены) 2 типов:

Слова последовательности алфавитно-цифровых символов.

Пунктуация последовательности пунктуационных знаков.

Пробельные символы могут быть успешно удалены в процессе разбора.

В результате лексического разбора будет получен список токенов с указанием их типов. Из этого списка несложно сформировать любой вывод.

Простейшей реализацией лексического разбора является функция, заполняющая или, для C++ 11, возвращающая последовательность объектов-токенов. Но этот вариант неэффективен в использовании памяти. Идеальный вариант — реализация итератора, возвращающего токены из произвольного потока.

В случае, если основная логика этого задания выполнена в виде функции, принимающей 2 потока ввода-вывода, задание также может быть выполнено в виде набора тестов.

С.3 Последовательности

Очередь

В задании предлагается выбрать между std :: list и std :: deque. Выбор между этими двумя контейнерами представляет собой классический компромисс между эффективностью на добавлении и удалении одного элемента и эффективностью при операции Accelerate(). В зависимости от способа использования конетйнера, может оказаться удобной реализация Accelerate() за $O(1)$, которая возможна при применении std :: list :: splice (). Если же опрерация Accelerate() выполняется редко и только для небольшого количества элементов, то создавать дополнительные расходы на динамическое создание каждого элемента std :: list может быть и неэффективным.

Простейший способ реализовать очередь с 3 приоритетами быдет создать по контейнеру для каждого проиритета внутри очереди. Это делает практически тривиальной добавление элемента, получение самого приоритетного и повышение приоритета.

К сожалению, приведенный интерфейс очереди обладает существенными недостатками:

1. такая очередь обрабатывает элементы только одного типа и должна быть скопирована для элементов другого типа;
2. интерфейс совершенно не следует принципам инкапсуляции;
3. интерфейс предоставляет базовую гарантию безопасности исключений.

Проблема типа элементов элементарно решается путем преобразования класса QueueWithPriority в шаблон:

```
template < typename QueueElement >
class QueueWithPriority
{
  ...
};
```

Инкапсуляция также элементарно достигается путем перемещения определения приоритетов внутрь определения шаблона. В простом задании трудно столкнуться с нарушением ODR, в то время как в реальном приложении имена HIGH, NORMAL и LOW не представлют из себя ничего магического и могут встречаться часто. Помещение их внутрь класса создает им уникальные имена и предотвращает нарушение ODR.

Приведенный интерфейс очереди обеспечивает базовую гарантию безопасности исключений: в случае генерации исключения никакие структуры не оказываются в состоянии, что невозможно корректное их разрушение. В то же время, строгая гарантия не предоставляется, так как возвращение объекта по значению может генерировать исключения. Для примера из задания таким исключением может оказаться std :: bad_alloc при копировании std :: string.

Решение этой проблемы требует выбора из различных вариантов:

- Разделить GetElementFromQueue() на 2 функции:

  ```
  QueueElement & FirstElement() const;
  void RemoveFirstElement();
  ```

  Данное решение применяется в STL, но оно требует внешней синхронизации доступа при параллельном программировании, что ухудшает инкапсуляцию класса.
- Использовать выходной параметр: **void** GetElementFromQueue(QueueElement & out). Это требует создания экземпляра объекта-данных очереди, что не всегда возможно, так как конструктор по-умолчанию может отсутствовать, а вызвать другой конструктор с конкретными параметрами может быть невозможно. Также, применение выходных параметров обычно является неудачной идеей, так как усложняет клиентский код.
- Вернуть указатель на извлеченный элемент: std :: auto_ptr< QueueElement > GetElementFromQueue() или std :: unique_ptr< QueueElement > GetElementFromQueue(). Но работать с указателями клиентскому коду можент быть неудобно и, как минимум, это нарушает симметричность интерфейса, так как объект был помещен в очередь по значению, а возвращается по указателю.

- Принять функцию, которая обработает очередной элемент:

```
template < typename F >
void ProcessElement(F f);
```

  Функция должна принимать единственный аргумент: QueueElement &.

Все эти решения представляют собой различные компромиссы. Наиболее остро эти проблемы возникают в многопоточном окружении. Применительно к стандарту C++ 2003 эта тема рассмотрена в статье «Multithreaded queue in C++».

Вывод списка

Вывод списка из второго задания обычно не доставляет существенных трудностей и может быть реализован различными способами:

- С помощью прямых и обратных итераторов, при таком решении требуется аккуратность при сравнении прямых и обратных итераторов.
- С помощью только обычных итераторов так как они двунаправленные и, соответственно, имеют операцию декремента.

Также решение может быть итеративным (при помощи цикла) или рекурсивным. В случае рекурсивного решения необходимо использовать хвостовую рекурсию, так как оптимизирующий компилятор может выполнять ее при постоянном размере использованного стека вызовов.

C.4 Итераторы

Записная книжка

Реализация записной книжки должна быть разделена на 2 части:

- Модель данных, по сути, сама записная книжка.
- Пользовательский интерфейс.

Для удобства, пользовательским интерфейсом может быть набор тестов.

Стандартной ошибкой, допускаемой при разработке модели данных для записной книжки, является попытка хранить итератор, указывающий не текущий элемент, внутри класса записной книжки. С одной стороны, это допустимый вариант и он неявно подразумевается в подсказке про недействительные итераторы. С другой стороны, такая реализация ограничивает возможные сценарии использования. Более правильным вариантом будет экспортировать итератор по данным наружу, предоставив возможность клиентскому коду отслеживать текущую позицию. Это позволит, в частности, просматривать данные в разных позициях одновременно.

При необходимости, экспортированный итератор можно заставить проверять свою корректность.

Контейнер факториалов

Эта задача имеет 3 реализации:

1. Вычисление значения факториала при разыменовании итератора.
2. Вычисление значения факториала при переходе.
3. Промежуточный вариант: вычисление значения при первом разыменовании и использование закэшированного значения при повторных разыменованиях в той же позиции.

Очевидно, что решение 2 в целом более эффективно с точки зрения объема вычислений, так как обеспечивает переход к соседним элементам и разыменование за $O(1)$, в то время как другие решения обеспечивают сложность разыменования $O(n)$. Как результат, исходя из требования постоянности времени разыменования, единственным правильным решением является вариант 2.

Также очевидно, что итератор контейнера попадает в категорию двунаправленных итераторов, так как переход к произвольному элементу невозможно выполнить за постоянное время.

С.5 Алгоритмы I

Это задание подразумевает правильное написание компаратора, требущегося для алгоритма сортировки. Компаратор может быть реализован как **operator** <() и как внешний функтор. Вариант с оператором может быть предпочтительным только в том случае, если такой способ упорядочивания является естественным для данных. Поскольку в задании ничего не сказано, правильнее будет написать внешний функтор, тем более что ничто не мешает доступу к данным.

При написании компаратора важно учесть, что если выполняется условие lhs . key1 > rhs.key1, то не нужно сравнивать другие поля, так как выполняется условие lhs > rhs.

С.6 Алгоритмы II

Чтение файла

Эта задача имеет короткое решение при помощи ассоциативного контейнера std :: set<>. Заполнение контейнера и вывод его содержимого выполняется при помощи потоковых итераторов и алгоритма std :: copy(). Для правильной обработки слов в разном регистре контейнер должен использовать написанный вручную компаратор, сравнивающий строки нечувствительным к регистру букв способом (strcasecmp() для GNU LIBC).

Геометрические фигуры

Это задание практически не вызывает трудностей в реализации. Необходимо отметить только следующие моменты:

- Необходимости в **typedef** здесь нет, так как в C++ теги структур являются именами типов.
- Поле количества вершин в Shape имеет неправильный тип (должен быть size_t) и вообще лишнее, так как std :: vector :: size () отвечает на этот вопрос не менее эффективно.
- Структуры должны быть снабжены конструкторами, так как это не только упрощает код, но и делает его более надежным.
- Изначальное заполнение списка фигурами следует выполнять алгоритмом std :: generate().
- Подсчет количества различных типов фигур и вершин можно выполнить за один проход при помощи агоритмов std :: for_each() или std :: accumulate().

С.7 Функторы I

Сложным моментами этого задания являются вычисление минимального, максимального значений и определение равенства первого и последнего чисел. Сложность связана с попытками написать макисмально обобщенный код так, чтобы игнорировать особый случай первого элемента. Исходя из задания, для минимального и максимального значений можно использовать начальные значения, выходящие за границы диапазона $[-500, +500]$. Для определения первого числа такой ход работает плохо. Оптимальным решением будет завести внутри функтора поле **bool** is_first_, которое конструктор установит в **true**. После обработки первого элемента оно сбрасывается в **false**. Это также даст возможность уведомить клиента, что последовательность была пуста.

При выводе результатов работы алгоритма std :: for_each() необходимо помнить, что количество копирований функтора неопределено, поэтому алогритм возвращает функтор, который был применен к каждому элементу диапазона. Результат работы находится именно в возвращаемомо значении, а не в переданном объекте.

Несмотря на то, что в задании сказано использовать std :: for_each(), сбор статистики можно рассматривать как свертку множества в статистику. В этом случае применение алгоритма std :: accumulate() выглядит более оправданным. В функциональных языках этот вариант выглядит более естественным, а сам алгоритм будет называться `reduce` или `fold`.

C.8 Функторы II

Умножение

Задача умножения чисел может решена в условиях только стандратных функторов при помощи алгоритма std :: transform(). В качестве последовательности, принимающей результат умножения необходимо передать начало той же последовательности.

Задача умножения выполняется при помощи стандартного функтора std :: multiplies <>, один из аргументов которого фиксируется при помощи std :: bind1st() или std :: bind2nd().

Геометрические фигуры

Очевидным исправлением в данной работе является применение умных указателей для хранения объектов в списке. С учетом содержания курса, решение с простыми указателями и ручным удалением объектов в конце не может считаться выполненным выше чем на «удовлетворительно». В случае, если используются простые указатели и объекты «утекают» задание следует считать невыполненным.

Наличие виртуального деструктора в базовом классе Shape определяется здравым смыслом и используемым умным указателем. При использовании boost::shared_ptr<> допустимо деструктор Shape::~Shape сделать невиртуальным и поместить его в защищенную секцию. Во всех остальных случаях он должен быть виртуальным.

Так как функция Shape::IsMoreLeft является методом класса, она принимает единственный аргумент — другую фигуру. Таким образом, ее прототип должен выглядеть как:

```
bool IsMoreLeft(const Shape & rhs) const;
```

С точки зрения данной задачи более удобным может оказаться вариант, принимающий указатель, например, для boost::shared_ptr<>:

```
bool IsMoreLeft(const boost::shared_ptr<const Shape> & rhs) const;
```

В прототипе первый **const** отвечает за неизменность умного указателя, второй — за неизменность объекта, на который указывает умный указатель, третий — за неизменность объекта, у которого вызывается метод.

Сортировки «слева-направо» и «сверху-вниз» при таком прототипе не вызывают проблем, так как последовательность аргументов при вызове компаратора имеет естественный порядок. Для реализации обратной сортировки необходимо воспользоваться boost::bind или написать функтор flip <>, меняющий при вызове вложенного функтора порядок аргументов.

Список литературы

[BOOST] Boost. *boost.org: Boost C++ Libraries*. boost.org. URL: http://www.boost.org/.

[C++ 03] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2003, с. 757. ISBN: ???? URL: http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110.

[C++ 11] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 28 февр. 2012, 1338 (est.) ISBN: ???? URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[C++ 98] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. Available in electronic form for online purchase at http://webstore.ansi.org/ and http://www.cssinfo.com/. Geneva, Switzerland: International Organization for Standardization, 1 сент. 1998, с. 732. ISBN: ???? URL: http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO/IEC+14882-1998;%20http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO/IEC+14882:1998;%20http://www.iso.ch/cate/d25845.html;%20https://webstore.ansi.org/.

[ES90] Margaret A. Ellis и Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-51459-1.

[ISO11] ISO. *ISO/IEC/IEEE 60559:2011 Information technology — Microprocessor Systems — Floating-Point arithmetic*. Geneva, Switzerland: International Organization for Standardization, 2011, с. 58. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469.

[Lak96] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley Professional, 1996, с. 896. ISBN: 978-0201633627.

[Mey96] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 1996. ISBN: 978-0201633719.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-12078-X.

[Бек03] Кент Бек. *Экстремальное программирование: разработка через тестирование*. Питер, 2003. ISBN: 5-8046-0051-6.

[Гам+13] Эрих Гамма и др. *Приемы объектно-ориентированного проектирования*. Питер, 2013, с. 366. ISBN: 978-5-459-01720-5.

[Кор+13] Томас Х. Кормен и др. *Алгоритмы. Построение и анализ*. Вильямс, 2013. ISBN: 978-5-8459-1794-2.