Алгоритмы и структуры данных 1305, 1306, 1307, 1308, 1310

Алгоритмы и структуры данных - часть 1-ВТ

Объявления

Пользовательские структуры данных: мет одические указания

✔ Пройдено

Экзамен

Открывается: Пятница, 10 февраля 2023, 13:30
Закрывается: Пятница, 10 февраля 2023, 17:00

Экзамен проводится в форме теста из 20 случайных вопросов.

Варианты ответов:

- выбор из 3-6 предложенных альтернатив;

- короткий ответ (ввести термин);

- числовой ответ.

Максимальный балл - 20. Оценки за 18-20 - отлично, 16-17 - хорошо, 14-15 удовлетворительно.

Итоговая оценка выводится с учётом результатов сдачи зачётных работ

Объявления

Уважаемые студенты!
Данную дисциплину ведет:
старший преподаватель кафедры ВТ Колинько Павел Георгиевич.

Для связи с преподавателями необходимо использовать личный кабинет (lk.etu.ru)
Электронная почта для решения срочных вопросов: clgn@mail.ru

Отметить как пройденное

1. Цели и задачи курса

Колинько П. Г. Алгоритмы и структуры данных. Часть 1. Пользовательские структуры данных
Курс «Алгоритмы и структуры данных» задуман как естественное продолжение курса «Основы программирования», переход от изучения возможностей и конструкций алгоритмического языка к проектированию программ, реализующих сложные алгоритмы. В курсе изучается связь свойств программного изделия с рациональностью организации данных.
Структура — это пользовательский тип данных, с которым связана некоторая область памяти и набор допустимых операций с ней, т. е. объект. Работа со структурами данных — это объектно-ориентированное программирование.
В качестве базового языка программирования выбран язык C++ в варианте последних стандартов C++11, C++14, C++17,

C++20. Курс начинается с изучения основ C++ и заканчивается (во второй части курса) знакомством со способами взаимодействия в программе нескольких классов (наследованием и т. п.), обработкой исключительных ситуаций и использованием Стандартной библиотеки шаблонов (STL).

В качестве источника алгоритмов используется параллельно изучаемый курс «Дискретная математика и теория алгоритмов». Из этого курса заимствуются постановки задач и доказательства работоспособности алгоритмов, а в настоящем курсе делается упор на их реализацию в виде программы на ЭВМ, тестирование и отладку.

Поэтому в качестве литературы по курсу «Алгоритмы и структуры данных» студентам предлагаются — в качестве источника сведений об алгоритмах — книги по дискретной математике. Это, в первую очередь:

— Поздняков С. Н., Рыбин С. В. Дискретная математика: учебник для вузов. М.: Академия, 2008. – 448 с.

— Хагерти Р. Дискретная математика для программистов. Изд. 2-е, испр. — М.: Техносфера, 2012. — 400 с.;

— Новиков Ф. А. Дискретная математика для программистов. — СПб.: Питер, 2000. — 304 с., ил.

Многие идеи курса взяты из классических источников:

— Ахо Дж., Хопкрофт А., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979;

— Ахо Дж., Хопкрофт А., Ульман Дж. Структуры данных и алгоритмы. — СПб.: И. Д. Вильямс, 2001. — 382 с.;

— Липский В. Комбинаторика для программистов. — М.: Мир, 1978. — 213 с.;

— Макконелл Дж. Основы современных алгоритмов. 2-е. изд. — М.: Техносфера, 2004. — 368 с.

— Кнут Д. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. — М.: Мир, 2013.

В качестве дополнительных источников информации об алгоритмах могут быть полезны книги:

— Гэри М., Джонсон Д. Вычислительные машины и трудно решаемые задачи. — М.: Мир, 1982. — 419 с.;

— Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 2-е изд. / пер. с англ. — М.: Мир, 2005. — 1296 с.: ил.;

— Седжвик Р. Фундаментальные алгоритмы C++. — К.: Диасофт, 2001. — 484 с.;

— Седжвик Р. Алгоритмы на C++ / пер. с англ. — М.: И. Д. Вильямс, 2011. — 1156 с.: ил.

Другой очень важный источник информации по курсу — книги по программированию. И здесь, при кажущемся изобилии, действительно полезных книг очень немного.

Есть книги по программированию на «суржике», т. е. на Си с элементами C++. Это в принципе неправильно, C++ задуман не как дополнение, а как замена языка Си с его особенностью слишком многое оставлять «на усмотрение программиста» и провоцировать его на трюки, создавать проблему неинициализированных и повторно используемых переменных и злоупотребления указателями.

Есть много книг с добросовестным описанием языка (копия стандарта), описывающие «как писать», но не говорящие «зачем».

Есть книги, переучивающие готовых программистов с других языков. Так, классическая книга по языку Си Кернигана и Ритчи переучивает программистов с фортрана. Есть книги, натаскивающие на конкретные системы программирования, сейчас уже устаревшие — Borland C++, Visual C++ 2005, 2008.

Так, великолепный учебник Лафоре (Лафоре Р. Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. — СПб.: Питер, 2015. — 928 с.: ил.) ориентирован на уже не актуальный Borland C++.

По C++ наиболее полезны книги автора языка C++ Б. Страуструпа, в которых говорится не только «как писать», но и «зачем», откуда появилось то или иное средство языка и какие задачи он призвано решать:

— Страуструп Б. Язык программирования C++. 2-е доп. изд. — М.: Бином-пресс, 2001. – 1098 с.;

— Страуструп Б. Язык программирования C++. Специальное издание. Пер, с англ. — М.: Изд-во Бином, 2015. — 1136 с.: ил.

— Страуструп Б. Программирование: принципы и практика с использованием C++: Второе издание. — СПб.: ООО «Диалектика», 2019. — 1328 с.

Общий недостаток всех перечисленных книг: в них ничего нет о новом стандарте языка — C++11, снявшем многие из существовавших в языке проблем и добавившем многие возможности. Зачем это понадобилось, можно узнать из книги Дьюхэрста:

— Стефан К. Дьюхэрст. Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ. — М., ДМК Пресс, 2012. — 264 с., ил.

В этой книге — много о том, как обычно пишут программы на C++ и как это делать правильно, и говорится о проблемах, которые решены новым стандартом языка. Книга написана до появления этого стандарта. К сожалению, более позднее издание этой книги (2017 г.) оказалось стереотипным.

О проблемах при работе с языком C++ и рекомендациях по решению сложных задач, тонкостям применения языка можно прочесть в книгах серии «C++ in Depth», написанных Скоттом Мейерсом, Гербом Саттером, Андреем Александреску и др.:

— Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. — М.: ДМК Пресс, 2012. — 294 с.: ил.;

— Саттер Г. Решение сложных задач на C++. Пер. с англ. — М.: И. Д. Вильямс, 2015. — 400 с.: ил.

Лучшая из существующих на сегодня книг, учитывающих новый стандарт языка — это:

— Прата С. Язык программирования C++. 6-е изд. — М.: И. Д. Вильямс, 2011. — 1244 с.

Она интересна тем, что в ней параллельно описываются языки Си, C++ (в стандарте 97/03) и C++11, показаны все различия в коде.
На новый стандарт ориентирован популярный учебник C++:
— Липпман С. Б., Лажойе Ж., Му Б. Э. Язык программирования C++. Базовый курс. 5-е изд. Пер. с англ. — М.: И.Д.Вильямс, 2014. — 1120 с.: ил.
Новая книга Страуструпа учитывает стандарты C++11 и
C++14:
— Страуструп Б. Программирование: принципы и практика с использованием C++: Второе издание. — СПб.: ООО «Диалектика», 2019. — 1328 с.
Язык C++ продолжает развиваться, уже доступны компиляторы для стандарта C++14, учтённого в книгах Питера Готтлинга и Скотта Мейерса:
— Готтшлинг П. Современный C++ для программистов, инженеров и учёных. — М.: И. Д. Вильямс, 2016. — 512 с.: ил.;
— Мейерс С. Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14: пер. с англ. — М.: И. Д. Вильямс, 2018. — 304 с.: ил.
В качестве справочника по применению Стандартной библиотеки шаблонов (STL) рекомендуется книга:
— Джоссатис Н. М. Стандартная библиотека C++: справочное руководство. 2-е изд. Пер. с англ. — М.: И. Д. Вильямс, 2014. — 1136 с.: ил.
Большинство упомянутых книг доступно на машинных носителях. Их можно скачать из интернета, в частности, с популярного сайта http://proklondike.com.
Актуальный справочник по функциям и другим средствам языка C++, включая последние стандарты: http://ru.cppreference.com
Поскольку русификация этого справочника сделана с помощью машинного перевода, иногда может быть полезен и оригинал: http://en.cppreference.com

2. Оценка временной сложности алгоритмов

При проектировании алгоритмов, как правило, не представляет интереса точное число шагов, необходимых для решения задачи на конкретном наборе данных. Гораздо важнее знать, как будет изменяться время решения задачи T, если размер входа n растёт.
Класс алгоритмов, время работы которых растёт, по крайней мере, так же быстро, как некоторая функция $f(n)$, обозначается как $\Omega(f(n))$. Это означает, что при всех n, превышающих порог n_0, $T(n) \geq C \cdot f(n)$ для некоторого положительного числа C.
Оценка времени работы снизу может представлять интерес только как теоретическая нижняя граница эффективности любого алгоритма для некоторой задачи, которую преодолеть невозможно.
Класс алгоритмов, время работы которых растёт не быстрее функции $f(n)$, обозначается $\mathbf{O(f(n))}$, что означает существование положительных чисел n_0 и C таких, что при $n > n_0$ $T(n) \leq C \cdot f(n)$. Этот класс — важнейшая характеристика алгоритма, его *временная сложность*. По скорости роста этого времени в зависимости от размера входа алгоритмы делятся на следующие классы временной сложности:
— алгоритмы константной сложности — $T(n) \in O(1)$;
— логарифмической сложности — $T(n) \in O(\log n)$;
— линейной сложности — $T(n) \in O(n)$;
— квадратичной сложности — $T(n) \in O(n^2)$;
— кубической сложности — $T(n) \in O(n^3)$;
— полиномиальной сложности — $T(n) \in O(n^k)$, где k = const; k = 0, 1, 2 или 3 — это частные случаи классов полиномиальной сложности;
— экспоненциальной сложности — $T(n) \in O(a^n)$.
Очевидно, что классы в этом перечне упорядочены по возрастанию мощности. Так, класс O(1) является подмножеством любого из остальных классов. Задача программиста — найти или разработать алгоритм класса минимально возможной мощности и реализовать его так, чтобы оценка временной сложности не ухудшилась.
Алгоритм, для которого оценки $\Omega(f(n))$ и $O(f(n))$ совпадают, называется *оптимальным*. Так, очевидно, что алгоритм, имеющий на входе некоторое множество, будет оптимальным, если его временная сложность O(1). Такой алгоритм можно попытаться найти, если задача не требует рассмотреть множество целиком. Если же требуется что-то сделать с каждым элементом множества мощностью n, оптимальный алгоритм будет иметь сложность $O(n)$. Если имеется два множества, и нужно обработать все возможные пары их элементов, можно ожидать сложности $O(n^2)$, для трёх множеств, если обрабатываются все тройки, — $O(n^3)$, и т. д.

Если же для получения результата необходимо рассмотреть все подмножества исходного множества или все перестановки его элементов — это задача на полный перебор, принадлежащая классу экспоненциальной сложности. В таких случаях говорят об отсутствии *эффективного* алгоритма.

Время работы программы часто зависит не только от мощности входных данных, но и от того, какие именно данные поступили на вход. В таких случаях делаются две оценки временной сложности:

— для самого неудобного набора данных — сложность «в худшем случае»;

— для типового набора данных — сложность «в среднем».

Тривиальные входные данные («лучший случай») обычно интереса не представляют.

Для оценки временной сложности по реализации алгоритма (**по тексту программы**) можно руководствоваться следующими соображениями:

— операции присваивания (копирования) и проверки условия для базовых типов данных выполняются за константное время. Если при этом вызывается функция, сложность шага алгоритма определяется сложностью функции (в операциях с объектами функции могут вызываться неявно);

— сложность алгоритма, состоящего из последовательности шагов, определяется по самому сложному шагу;

— сложность выбора по условию определяется по самой сложной из альтернатив. В порядке исключения можно не принимать во внимание альтернативы, выбираемые очень редко. Можно учесть такие альтернативы, как «худший случай»;

— если какие-то шаги являются внутренней частью цикла с количеством повторений, зависящим от размера входа n, в оценку сложности циклического шага добавляется множитель n. Если же количество повторений не зависит от n, цикл игнорируется, поскольку его можно рассматривать просто как повторение некоторого количества одинаковых шагов алгоритма;

— рекурсия рассматривается как тот же цикл. Её сложность определяется как произведение сложности одного вызова функции на количество вызовов.

Пример 1. Вычислить $b = (a \in A)$, где множество A мощностью nA представлено массивом целых чисел.

Решение:

```
b = false; for (i = 0; !b && (i < nA); i++) b |= (a == A[ i ]);
```

Временная сложность алгоритма — O(nA). Если элемент a найден, алгоритм прекращает работу, выполнив от 1 до nA шагов. В среднем количество шагов будет $nA / 2$, в худшем случае ($a \notin A$) — nA.

Пример 2. Вычислить $C = A \cap B$ для множеств, представленных неупорядоченными массивами.

Решение: проверяем все возможные пары элементов двух множеств и отбираем совпадения.

```
for (i = k = 0; i < nA; i++)
for (j = 0; j < nB; j++) if (A[ i ] == B[ j ]) C[ k++ ] = A[ i ];
```

Проверка на совпадение и присваивание выполняются за константное время, поэтому сложность алгоритма — $O(nA \times nB)$, или $O(n^2)$, где n — средняя мощность множеств.

Пример 3. Вычислить $D = A \cap B \cap C$.

Очевидное решение

```
for (int i = 0, k = 0; A[ i ]; ++i)
    for (int  j = 0; B[ j ]; ++j)
        for (int r = 0;  C[ r ]; ++r)
          if ((A[ i ] == B[ j ]) && (A[ i ] == C[ r ])) D[ k++ ] = A[ i ];
```

имеет временную сложность $O(n^3)$, поскольку перебираются все возможные тройки. Однако перебирать все тройки никакой необходимости нет.

Модифицируем алгоритм:

```
for (int i = 0, k = 0; A [ i ]; ++i)
    for (int j = 0; B[ j ]; ++j)
        if (A[ i ] == B[ j ]) for (int r = 0; C[ r ]; ++r)
            if (A[ i ] == C[ r ]) D[ k++ ] = A[ i ];
```

В алгоритме по-прежнему три вложенных цикла, но внутренний цикл теперь зависит от условия $A[i] == B[j]$, которое проверяется n^2 раз, но удовлетворяется не более чем n раз, т. е. рассматриваются только такие тройки, у которых первые два элемента совпадают. Проверка $A[i] == C[r]$ выполняется, таким образом, не более n^2 раз, и общая сложность алгоритма — $O(n^2)$.

Пример 4. Вычислить $C = A \cap B$ для множеств, представленных строками символов.

Решение:

```
for (int i = k = 0; i < strlen(A); ++i)
  for (int j = 0; j < strlen(B); ++j) if (A[ i ] == B[ j ]) C[ k++ ] = A[ i ];
```

По аналогии с примером 2 можно оценить сложность как $O(n^2)$. Однако это неверно, так как в обоих циклах по n раз вычисляется функция определения длины строки *strlen*( ), которая подсчитывает в строке количество символов до ближайшего нуля перебором, т. е. имеет линейную сложность. Вычисление этой функции — один из шагов внутренней части обоих циклов.

Таким образом, внутренняя часть вложенного цикла состоит из трёх шагов, двух константных (проверка и присваивание) и линейного (вычисление функции). С учётом n повторений сложность всего цикла — $O(n^2)$. Внешний цикл добавляет сюда ещё шаг вычисления функции сложностью $O(n)$. Сложность его внутренней части — $O(n^2)$, а всего алгоритма — $O(n^3)$! Это цена экономии двух целых переменных. На самом деле нужно вычислить пределы заранее *nA* = *strlen*(*A*), *nB* = *strlen*(*B*), а затем использовать алгоритм из примера 2. Альтернатива: если известно, что массивы символов ограничены нулём, это можно использовать, как показано в примере 3. Подумайте, как ограничить нулём результат!

Контрольные вопросы.

1. Какой класс временной сложности алгоритма является самым широким?
2. Алгоритм какой временной сложности является самым предпочтительным в общем случае?
3. Алгоритм какой временной сложности является оптимальным для поэлементной обработки множества мощностью n?

3. Понятие данных. Примитивы и структуры. Стандартные структуры данных

Данные — это то, чем манипулирует программа. Цель работы любой программы состоит в изменении данных. На языке высокого уровня программа манипулирует переменными — именами, каждое из которых связано с некоторой областью памяти, содержимое которой можно изменять, и константами, тоже обозначающими память, но с указанным неизменяемым содержимым. С данными связано понятие типа, который задаёт размер области памяти для переменной (константы) и набор возможных операций с ней.

Мы будем различать данные-примитивы, для обработки которых достаточно одного шага алгоритма, и структуры, требующие нескольких шагов.

Эта классификация относительна. Примитив с точки зрения ЭВМ — это минимальная адресуемая область памяти — байт или машинное слово. С другой стороны, машинное слово состоит из отдельных битов, и существуют средства для работы с битами, т. е. слово — это структура. Мы с вами познакомимся также с ситуациями, когда область данных, состоящая из нескольких частей, копируется в другое место как единое целое, т. е. выступает как примитив. В конце концов, цель создания программы может быть сформулирована как сведение обработки большого объёма сложных данных к одному шагу — запуску программы.

Типы данных, встроенные в язык C++ — это примитивы, в частности: целые (*int*, *long*, *long long*), вещественные (*float*, *double*, *long double*), символьные (*char*), булевские (*bool*), указатели. Для целых и вещественных предусмотрены арифметические операции, для символов — ввод и вывод, для указателей — сложение с целой константой и разыменование для доступа к памяти, адрес которой хранит указатель. С указателем всегда связан тип области памяти, на которую тот указывает. Исключение — указатель на *void* (указатель вообще), который при работе с данными подменяется указателем конкретного типа.

Структуры данных в программе на C++ — это данные, добавляемые в язык пользователем-программистом. Некоторые возможности предусмотрены в самом языке.

В первую очередь это **массивы** — указатели на области памяти, состоящие из заданного количества одинаковых частей — примитивов или структур обязательно одного и того же типа. Например, *int A*[10] обозначает память для 10 целых переменных.

Массив является памятью прямого доступа: любой элемент массива доступен непосредственно, если известен его порядковый номер. Для получения указателя на элемент массива используется индексная арифметика (обычно неявно, с помощью операции «квадратные скобки» [ ]): *A*[5] — то же, что и *(*A*+5).

Единственная операция с массивом целиком — это определение его размера: *sizeof*(*A*) вернёт то же, что и 10**sizeof*(*int*) для указанного выше массива A.

При попытке копирования массива скопируется только указатель на занимаемую им область памяти: в результате *int *B = A* указатель *B* можно будет индексировать для доступа к данным массива *A*, но операция *sizeof*(*B*) вернёт не размер массива, а размер указателя. Но нет никакого способа определить, указывает ли указатель *B* на массив или на простую переменную, и нет способа определить размер такого массива. Корректность манипуляций с указателем *B* возлагается на программиста.

Объединение данных нескольких, в общем случае разных, типов в единое целое возможно в языке C++ объявлением **структуры**, например:

struct *Type* { int p; char s; int A[10]; };

Объявление структуры — это объявление пользовательского типа данных *Type*. Можно объявлять переменные, массивы и указатели такого типа: *Type P*, **Q*, *R*[15].

Для доступа к полям структуры используется операция «прямое членство», обозначаемая точкой между именем структуры и именем поля, или «косвенное членство — стрелочка между указателем и именем поля:

P.p, *Q->s*, *R*[0].*A*[5].
В отличие от массивов, структуры можно копировать. Если нужно копировать массив, его можно сделать полем, возможно, единственным, структуры, играющей роль его оболочки. Так, корректно утверждение *R*[0] = *P*, копирующее три поля структуры *R* типа *Type* в нулевой элемент массива структур *R* того же типа.
Важным частным случаем являются структуры *struct*, имеющие в своём составе поля — указатели на структуры того же типа. Из таких структур формируются списки — наборы данных из разных областей памяти, связанных с помощью указателей в единое целое. Элемент списка здесь выступает как примитив, а список в целом — как структура данных.
Объединения (*union*) в общем случае не могут рассматриваться как структуры данных. Они похожи на структуры (*struct*), но каждое поле в них — это отдельный тип для общей для всех полей области данных. Назначение объединений — программистский трюк, предназначенный для обхода некоторых ограничений языка программирования.

Кроме пользовательских, существуют ***абстрактные***, или стандартные структуры данных, с помощью которых описываются алгоритмы на языке математики. Это *множество*, *отображение*, *последовательность*, *очередь* и *стек*.
Множество – это совокупность объектов, называемых элементами множеств. Множество может задано перечислением своих элементов:
S = { 2, 5, 7, 12, 24 }; *T* = { понедельник, вторник, среда }.
Чтобы показать, что речь идёт о множестве, набор элементов заключается в фигурные скобки. Порядок элементов в перечислении не имеет значения.
Очень важной характеристикой множества является количество элементов в нём — *мощность*. Она может быть бесконечной. В этом случае множество можно задать с помощью предиката — функции на элементах, возвращающей булевское значение (*true* или *false*): *S* = { *x* : *P*(*x*) } — множество всех *x*, для которых *P*(*x*) = *true*. Например, *S* = { *x* : *x* — число месяца, приходящееся на понедельник }.
Некоторые часто используемые множества имеют стандартные названия и обозначения:
∅ — пустое множество;
U — полное множество для некоторой задачи, универсум;
ℕ = { 1, 2, 3, … } — множество *натуральных* чисел;
Z = { 0, ±1, ±2, ±3, … } — множество *целых* чисел;
R — множество *вещественных* чисел;
R^+ — множество *неотрицательных вещественных* чисел.
Отображение — это некоторая функция *f* на элементах множества *A* со значениями из множества *B*:

$$f : A \to B.$$

Отображение — это частный случай *бинарного отношения* $R \subseteq A \times B$, удовлетворяющий ограничению: каждый элемент множества *A* связан с единственным элементом множества *B*.
Задать отображение можно, например, перечислением пар:

$$f = \{ <a, b>: a \in A, b \in B \}.$$

Если функция *f* — инъективна (взаимно однозначна), то множества *A* и *B* — изоморфны, и мы можем вместо элементов множества *A* работать с соответствующими элементами множества *B*. Наличие такой функции и определяет изоморфизм
Важный частный случай — использование в качестве множества *B* натуральных чисел: $f : A \to \mathbb{N}$. Такое отображение мы будем называть *разметкой* множества A, т. е. присвоением каждому элементу из A порядкового номера для дальнейшей работы с этими номерами вместо самих элементов множества A.
Отображение множества натуральных чисел на произвольное множество *A* образует **последовательность** из элементов этого множества. Задать последовательность можно тоже перечислением элементов множества, но в этом перечислении порядок следования элементов будет важен. Этот факт мы будем обозначать угловыми скобками:

$$S = < 2, 5, 7, 12, 5, 24, 2 >.$$

В этом примере 5 стоит на 2-й позиции — после 2 и перед 7. В отличие от множества, элементы последовательности в общем случае могут повторяться.
Очередь и стек суть последовательности, в которые можно помещать и затем забирать из них элементы множества. В случае очереди элементы множества извлекаются из начала последовательности, а добавляются в её конец. В результате элементы множества будут извлекаться из очереди в том же порядке, в котором в неё помещались. В стеке и добавление, и извлечение элементов выполняются в конце последовательности. Поэтому из стека при извлечении элементов из стека их порядок будет обратный порядку их размещения.
Оценку временной сложности алгоритма, использующего абстрактные структуры данных, мы будем называть теоретической. При реализации алгоритма в виде программы абстрактные структуры данных заменяются данными пользователя-программиста исходя из возможностей, предоставляемых языком C++. В общем случае возможно несколько вариантов такой замены, поэтому ставится задача выбора оптимального варианта и оценка результата такого выбора — оценка временной сложности алгоритма по тексту программы.

Структуры, предоставляемые языком — массив и список — пригодны как основа для реализации любой из перечисленных выше абстрактных структур данных. В конкретную структуру массив превращается, если обрабатывать его по соответствующим правилам. Так, например, если массив должен быть структурой данных для множества, требуется, чтобы тип элемента массива совпадал с типом элемента множества, размер массива был достаточен для хранения множества, были бы определены операции над множествами — объединение, пересечение и т. п. Если в массиве реализован стек, должны быть определены операции «опустить в стек», «поднять из стека».
Программист может сам придерживаться правил, установленных им для конкретной структуры данных и написать необходимые функции обработки в рамках парадигмы процедурного программирования.
Но язык C++ предлагает более удобный механизм — класс, в котором одновременно с набором данных можно определить и набор функций-членов для исключительного доступа к этим данным, получив тем самым полноценный пользовательский тип данных. Конкретные экземпляры данных некоторого класса называются объектами, а программирование с использованием классов — объектно-ориентированным программированием.
Такое программирование является естественной средой для получения навыков работы с пользовательскими структурами данных.

4. Множества в памяти ЭВМ

Для представления множества в памяти ЭВМ можно использовать один из двух способов: последовательность из элементов множества или отображение на универсум. Последовательность элементов может храниться в форме массива или списка. Отображение может быть в развёрнутой форме — массив битов (булевских значений) или в компактной — машинное слово.

4.1. Представление множества набором элементов

Одним из способов задания множества является простое перечисление входящих в него элементов. В памяти ЭВМ элементы такого множества могут быть расположены в последовательности ячеек, т. е. образуют массив. Это самый экономный способ хранения множества в тех случаях, когда мощность множества известна, а его элементы — данные одного типа (во всех случаях, когда это не так, элементы множества можно расположить в памяти произвольным образом и работать с ними через указатели). Память под массив удобно выделить статически. В этом случае её можно сразу инициализировать.
Пример. Объявление и инициализация массива для множества десятичных цифр.

```
const int Nmax = 10;
char A[ Nmax+1 ] = { '1', '3', '4', '7' };
```

В память, выделяемую под универсум, помещено множество-константа. Множество символов предполагается обрабатывать как строку, поэтому нужно позаботиться о месте под ограничивающий нуль. Однако в примере выше в этом нет необходимости: поскольку инициализаторов всего 4, остальные элементы массива будут по умолчанию заполнены нулями.
Если множество расширяться не будет, размер памяти можно не указывать:

```
char A[ ] = {"1347"};
```

Выделяется память под множество из четырёх элементов и ограничивающий нуль.
Мощность множества, заданного строкой-константой, можно вычислять:

```
int nA = strlen(A);
```

или просто обрабатывать строку до появления нуля.
Если множество создаётся в результате вычислений, его мощность может быть неизвестна. Поскольку память под массив должна быть выделена до начала работы с ним, приходится делать это с запасом. Проще всего выделить память сразу под универсум, а если универсум слишком велик, то под множество максимальной мощности, которое может быть реально получено. Если же память оказалась исчерпана до окончания вычислений, можно попытаться заказать область памяти большего размера и перенести в неё накопленные элементы множества. Это дорогая операция, поскольку приходится полностью копировать множество. Она может быть невыполнима, если в памяти нет непрерывного участка достаточного размера. Освобождаемый участок памяти тоже не всегда можно использовать. Если же оценка мощности результата слишком пессимистична, память под множество используется нерационально. Так, её приходится выделять даже для результата — пустого множества.
Таких проблем нет, если для представления множества в памяти используется односвязный список. Память в этом случае выделяется под каждый элемент множества отдельно, и её ровно столько, сколько необходимо. Так, под пустое множество-список память вообще не выделяется. Память от удаляемых элементов списка легко использовать под вновь создаваемые элементы. Перенос элемента из одного множества в другое вообще не требует дополнительной памяти.
Главный недостаток способа — необходимость хранить с каждым элементом множества указатель на следующий элемент и тратить время на работу с ним. Так, создание копии списка в другом месте памяти требует не только отдельной обработки

каждого элемента множества, но и создания вновь всех указателей. Имеются и скрытые потери: минимальная область, выделяемая в динамической памяти, часто больше, чем требуется для хранения одного элемента множества вместе с указателем.

Реализация алгоритмов для работы с множествами, представленным массивами или списками, отличается только способом перебора этих структур данных. Подтвердим это примером реализации операции принадлежности элемента множеству десятичных цифр.

Для массива используем объявление из предыдущего примера. Множество-список объявим так:

```
struct Set { char el;
Set * next;
}
Set *LA;     // Указатель на начало списка для множества A.
```

Вариант 1. Поиск элемента множества в строке символов.

```
bool exist(char s, char A [ ])
{ bool b{false};
  for(int i = 0; A[ i ] && !b; ++i ) b |= (s == A[ i ]);
  return b;
}
```

Вариант 2. Поиск элемента множества в списке.

```
bool exist(char s, Set * LA)
{ bool b(false);
  for(Set * p = LA; p && !b; p = p->next ) b |= (s == p->el);
  return b;
}
```

Очевидно, что оба варианта реализации имеют линейную временную сложность по мощности множества: O(*nA*). Оценка не меняется и в том случае, если при обнаружении элемента в множестве цикл прерывается.

Для ввода и вывода множества символов удобным способом представления является строка — массив, ограниченный нулём. Такой массив можно ввести и вывести в один приём, без применения цикла и легко инициализировать константой:

```
cout ≪ "\n A="; cin ≫ A;
cout ≪ " Введено A=" ≪ A ≪ '\n';
```

Наличие ограничителя позволяет отказаться от использования переменной для хранения мощности множества.

Преобразование массива A в список LA выполняется по тривиальному алгоритму:

```
Set *LA = nullptr;
for( int i = 0; A[ i ]; ++i)
  { Set *temp = new Set;
    temp->el =  A[ i ]; temp->next = LA; LA = temp; }
```

Если немножко дополнить объявление структуры Set:

```
struct Set { char el; Set * next;
  Set(char e, Set * n) : el(e), next(n) { }
  ~Set( ) { delete next; }
};
```

то преобразование массива в список станет ещё проще:

```
for( int i = 0; A[ i ]; ++i) LA = new Set(A[ i ], LA);
```

а для удаления списка из памяти не понадобится цикл, достаточно написать delete LA.

То, что при преобразовании массива в список порядок элементов меняется на противоположный, не имеет значения, поскольку речь идёт о множестве, где порядок элементов безразличен.

4.2. Представление множества отображением на универсум

Если элементы универсума упорядочить, т. е. представить в виде последовательности $U = <u_0, u_1, u_2 \dots u_{m-1}>$, то любое его подмножество $A \subseteq U$ может быть задано вектором логических значений $C = <c_0, c_1, c_2 \dots c_{m-1}>$, где $c_i = \{u_i \in A\}$, или вектором битов $c_i = 1$, если $u_i \in A$, иначе — 0.

Такой способ представления множеств в памяти имеет практическое значение, если мощность универсума $m = |U|$ не очень велика и существует простая функция $f : U \rightarrow [0 \dots m - 1]$ отображения элемента множества в соответствующий ему порядковый номер бита.

Так, например, если U — множество десятичных цифр, подходящей функцией будет $f(a) = a - '0'$, где a — символьная переменная с кодом цифры, поскольку известно, что коды цифр образуют монотонную последовательность. Аналогично, для множества прописных латинских букв можно взять $f(a) = a - 'A'$. Для шестнадцатеричных цифр, коды которых образуют два интервала, функция будет сложнее: $f(a) = a \le '9' \ ? \ a - '0' \ : \ a - 'A' + 10$. В общем случае можно использовать словарь — массив,

содержащий универсум, например, для шестнадцатеричных цифр — U[ ] = "0123456789ABCDEF".
Операции над множествами в форме вектора битов сводятся к логическим операциям над соответствующими битами множеств. Для вычисления объединения $A \cup B$ следует выполнить *a* [*i*] || *b* [*i*], для пересечения $A \cap B$ — *a* [*i*] && *b* [*i*], для разности $A \setminus B$ — *a* [*i*] && !*b* [*i*] для всех битов от $i = 0$ до $i = m - 1$. Следовательно, временная сложность двуместной операции с множествами *A* и *B* в форме вектора битов будет $O(m)$, что при фиксированном *m* соответствует O(1), т. е. не зависит от мощности этих множеств.
Для получения вектора битов *bA* из строки символов *A* следует заполнить вектор *bA* нулями, а затем установить в 1 биты, соответствующие каждому символу из *A*:
for (int i = 0; A[i]; i++) bA [f (A[i])] = 1.
Можно использовать автоматическое заполнение нулями, происходящее при объявлении массива с инициализацией:
int bA[m]{ };

Обратное преобразование очевидно:
for (int i = 0, k = 0; i < m; ++i) if (bA[i]) A[k++] = f^{-1}(i),
где $f^{-1}(i)$ — функция, обратная для $f(a)$. Так, если $f(a) = a - '0'$, то $f^{-1}(i) = i + '0'$.
Можно, хотя и не обязательно, использовать для массива битов тип *bool*.
bool bA[m]{ };
for (int i = 0; A[i]; ++i) bA [f (A[i])] = true.
Использование массива битов в качестве промежуточной памяти при работе с множеством, представленным набором элементов — самый простой способ устранить дубликаты. Достаточно преобразовать массив (список) элементов в массив битов, а затем обратно. Затраты на такое преобразование будут иметь порядок $O(n)$. Устранение же дубликатов «в лоб», т. е. проверка на уникальность каждого добавляемого в множество элемента, потребует $O(n^2)$ шагов.
Вектор битов может быть представлен в памяти в компактной форме — форме машинного слова, в качестве которого на языке C++ могут использоваться переменные целого типа *int, long, long long*. Для таких переменных в языке предусмотрены поразрядные логические операции:
— логическое сложение (поразрядное «ИЛИ») *A* | *B*, реализующее объединение множеств $A \cup B$;
— логическое умножение (поразрядное «И») *A* & *B* — пересечение $A \cap B$;
— поразрядное сложение по модулю 2 (исключающее «ИЛИ», сравнение кодов) *A* ^ *B* — симметрическая разность $A \oplus B = (A \cup B) \setminus (A \cap B)$;
— инвертирование ~*A*, соответствующее $\bar{A}$ — дополнению до универсума.
Операции над множествами в форме машинного слова выполняются за один шаг алгоритма независимо от мощности множеств, т. е. имеют временную сложность O(1). Например, вычисление $E = (A \cup B \cap C) \setminus D$ реализуется оператором *wE* = (*wA* | *wB* & *wC*) & ~*wD*, где *wA*, *wB*, *wC*, *wD* — машинные слова, хранящие соответствующие множества.
Способ применим, если размер универсума *m* не превосходит разрядности переменной (8 для *char*, 16 для *short*, 16 или 32 для *int*, 32 для *long,* 64 для *long long*). Если $m > 64$, можно использовать несколько слов. Если *m* не равно размеру слова, часть битов слова не используется. Обычно это не вызывает проблем. Исключение: если переменная сравнивается с нулём для выявления пустого множества, нужно, чтобы неиспользуемые биты содержали 0.
Недостаток способа — в отсутствии удобного доступа к каждому биту машинного слова, как к элементу массива. Вместо этого приходится генерировать множество из одного элемента {*a*} сдвигом 1 на *f* (*a*) битов влево. Далее с помощью поразрядного «ИЛИ» можно добавить элемент в множество, а с помощью поразрядного «И» — проверить его наличие в нём. Так, для преобразования множества из строки символов в машинное слово можно использовать алгоритм
wA = 0;
for (int i = 0; A[i]; ++i) wA = wA | (1 ≪ f(A));
Примечание. Если программа пишется для компилятора, поддерживающего разрядность данных *int* — 16, а мощность универсума больше (переменная *wA* имеет тип *long*), вместо константы 1 следует использовать 1*L*.
Для обратного преобразования (из машинного слова в строку символов) удобнее использовать сдвиг слова вправо и умножение на 1:
for (int i = 0, k = 0; i < m; ++i) if ((wA ≫ i) & 1) A[k++] = f^{-1}(i).
Отметим, что элементы массива битов нумеруются справа налево, а биты машинного слова — слева направо. Поэтому, если для массива битов и для машинного слова используется одна и та же функция отображения *f* (*a*), порядок битов в этих структурах данных будет противоположный.
Вместо операции сдвига иногда удобно применить словарь — массив из машинных слов с единицей в соответствующем разряде. Для универсума мощностью 10 массив будет выглядеть так:
W[ ] = { 1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80, 0x100, 0x200 }.
При желании константы в словаре можно расположить в противоположном или в произвольном порядке.
Машинное слово отличается от других способов хранения множеств ещё и тем, что в нём одновременно можно устанавливать

или проверять несколько битов. Так, поразрядное «или» с константой 0x7FF превращает множество десятичных цифр в полное, а поразрядное «и» позволяет убедиться, что оно пустое, независимо от состояния неиспользуемых битов. Константа 0x155 позволит проделать это с подмножеством нечётных цифр {1, 3, 5, 7, 9}, а 0x2AA — чётных {0, 2, 4, 6, 8}.

4.3. Замечание о функциях — операциях над множествами

Двуместные операции над множествами можно (хотя и необязательно) реализовать в виде функций.
Так, функция для вычисления пересечения двух множеств, заданных массивами, ограниченными нулём, может выглядеть так:

```
void AND(char *E, char A[ ], char B[ ]) //Вариант 1
{ int k = 0; for(int i = 0; A[ i ]; ++i) if(exist(A[ i ], B) E[k++] = A[ i ]; }
```

или так:

```
char * AND(char A[ ], char B[ ]) //Вариант 2
{ int k = 0; char * E = new char[N + 1];
  for(int i = 0; A[ i ]; ++i) if(exist(A[ i ], B) E[k++] = A[ i ];
  E[ k ] = 0;  return E;
}
```

Сложность обоих вариантов — O(n^2): функция *exist*( ) линейной сложности находится внутри цикла просмотра множества *A* из *n* элементов (*n* — средняя мощность множеств *A*, *B*, *E*). Какой вариант лучше? Я полагаю, первый. Множество *E* предполагается объявленным размером под универсум + 1 и инициализированным нулями, так что об ограничивающем нуле можно не беспокоиться.
Во втором варианте массив *E* создаётся в свободной памяти без возможности инициализации, поэтому приходится заботиться об ограничивающем нуле, и, что гораздо важнее, об освобождении этой памяти, когда работа с массивом *E* будет закончена. Это должна делать вызывающая программа.
Но последнее не всегда возможно. Так, при вычислении выражения $E = A \cap B \cap C \cap D$ «естественным» способом

```
E = AND(A, AND(B, AND(C, D)));
```

промежуточные результаты $C \cap D$ и $B \cap (C \cap D)$ будут недоступны для удаления и останутся в памяти в качестве мусора.
Примечание. В последнем примере проблему утечки памяти можно решить с помощью следующего трюка:

```
E = AND(A, ABC=AND(B, CD=AND(C, D))); delete [ ]ABC; delete [ ]CD;
```

Здесь с помощью присваиваний промежуточным множествам даются имена, что делает возможным их удаление.
Точно такая же ситуация возникает при работе с массивами битов. Предпочтительная реализация выглядит так:

```
void AND(bool E[ ], bool A[ ], bool B[ ])
{ for(int i = 0; i < Nmax; ++i) E[ i ] = A[ i ] && B[ i ]; }
```

Для варианта «машинные слова» функция тривиальна.
В варианте «списки» оба подхода оказываются функционально эквивалентны, потому что списки всегда создаются в свободной памяти:

```
void AND(Set *&E, Set *A, Set *B)   //Вариант 1
{ for(Set * p = A; p; p = p->next)
    if(exist(p->el, B) E = new Set(p->el, E); }
```

или так:

```
Set * AND(char A[ ], char B[ ]) //Вариант 2
{ Set * E = nullptr;
  for(Set * p = A; p; p = p->next)
    if(exist(p->el, B) E = new Set(p->el, E);
 return E;
}
```

Здесь проблемным может оказаться первый вариант, потому что предполагается, что фактический указатель на результат *E* обнулён. Если это не так, то функция будет добавлять результат вычисления к имеющемуся содержимому списка *E*. Если же в намерения программиста это не входит, рекомендуется добавить в первый вариант функции *AND* первую строку:

```
delete E; E = nullptr;
```

Для передачи вызывающей программе нового значения указателя *E* соответствующий аргумент объявлен ссылкой.
Контрольные вопросы

1. Какой способ размещения множества в памяти — самый экономный?
2. Какой способ размещения множества в памяти — самый универсальный?
3. Какой способ размещения множества в памяти — самый удобный для проверки принадлежности элемента множеству?
4. Какой способ размещения множеств в памяти обеспечивает самый быстрый алгоритм выполнения двуместной операции над ними?

5. При какой форме представления множеств в памяти можно задать множество-константу?
6. Какой способ представления в памяти множества символов самый удобный для ввода с клавиатуры?
7. Сколько различных подмножеств можно получить для множества мощностью *n*?

5. Генерация тестов

Программу, проверенную на тестах, введённых вручную или из специально подготовленного файла, можно затем дополнительно проверить подачей на вход некоторого количества случайных тестов, генерацию которых разумно поручить машине.

5.1. Генерация случайного подмножества

Достаточно просто получить случайное множество в форме машинного слова. Для этого можно использовать функцию *rand*( ) из стандартной библиотеки *stdlib*. Функция возвращает псевдослучайное целое в интервале 0…*MAXINT*. Случайное слово из *m* битов (*m* — мощность универсума) можно получить, выделив его из возвращаемого значения. Так, для *m* = 10 это будет:
w = rand( ) %0x3FF.
А можно просто положить *w* = *rand*( ) и игнорировать лишние биты.
Если разрядность *int* равна 16, для получения слова типа *long* можно использовать функцию дважды и объединить возвращаемые значения:
w = static_cast<long> ((rand( ) ≪ 16) | rand( )).
Если мощность универсума превышает разрядность машинного слова, генерируется несколько слов.
Можно вместо нескольких слов сгенерировать массив из *n* случайных битов:
for (int i = 0; i < m; ++i) X[i] = rand( ) % 2.
Замечание. Последний способ в общем случае следует применять с осторожностью, так как младший бит генерируемого машинного слова при плохом генераторе может оказаться не вполне случайным, например, чередованием 0 и 1.
Следует отметить, что датчик *rand* ( ) даёт не случайные, а псевдослучайные числа. При каждом новом запуске программы последовательность этих чисел будет повторяться. Это очень удобно для отладки. В начало функции *main*( ) нужно вставить строку *srand*(*start*), которая обеспечит запуск датчика с указанного значения *start* (целого числа). Подбором значения start можно добиться удовлетворительного теста, необходимого для полноценной отладки. Когда отладка закончена, константу *start* заменяют вызовом функции из стандартной библиотеки *time.h*, возвращающей текущее время: *time*(*nullptr*).
Случайное множество в форме массива или списка проще всего получается генерированием последовательности битов:
int k = 0; Set *LA = nullptr;
for (int i = 0; i < m; i++) if (rand( )%2)
{ S[k++] = f^{-1}(i); Set *LA = new Set(f^{-1}(i), LA); }
S[k] = 0;
Здесь, как и ранее, $f^{-1}(i)$ — функция, преобразующая позицию *i* массива битов в элемент универсума.
Результат — строка символов *S* и список *LA* — множество со случайной мощностью $k \in [0 \ldots m-1]$.
Все рассмотренные генераторы создают множества, в которых каждый элемент универсума появляется с вероятностью 0,5. Может получиться и пустое, и полное множество, но в среднем мощность получается близкой к *m* / 2. Если требуется получать множества почти пустые или почти полные, нужно сделать так, чтобы вероятности появления 0 или 1 различались. Например, генератор массива битов (булевских значений) может выглядеть так:
for (int i = 0; i < m; ++i) X[i] = static_cast<bool>(rand( ) % p > q).
В этом генераторе вероятность появления 1 зависит от соотношения значений констант *p* и *q*. Так, например, при *p* = 5 датчик будет давать с равной вероятностью элементы множества {0, 1, 2, 3, 4}; если выбрать *q* = 3, вероятность генерации *true* будет 0,2, а при *q* = 0 — 0,8.

5.2. Генерация последовательности всех подмножеств заданного множества

Подача на вход алгоритма последовательности всех подмножеств некоторого множества *X* может потребоваться для полного тестирования алгоритма или для решения задачи полным перебором в случаях, когда эффективного алгоритма не существует. Если мощность множества $|X| = m$, мощность множества всех его подмножеств — булеана (общее количество тестов) $|2^X| = 2^m$.
Если $m \le 32$, последовательность подмножеств проще всего получить в форме машинных слов по очевидному алгоритму:
for (w = 0; w < 2^m; ++w) **yield**(w).
Здесь и далее ***yield***(*w*) — некоторая функция, использующая множество *w*.
Для практических целей часто бывает удобнее, чтобы каждое подмножество в последовательности отличалось от

предыдущего появлением или исчезновением ровно одного элемента (*m*-битный код Грея).

а) вариант для массива битов

```
void GGen (int m, int A[ ])
{
  for (int i = 0; i < m; ++i) A[ i ] = 0; // Исходный код (из нулей)
  int i = 0; // Счётчик кодов
  do {
    yield(A); //Использовать очередной код
    ++i; int p( 0 ), j( i );
    while ( (j & 1) == 0) { j = j/2; ++p; } // Искать младший бит != 0
    if ( p < n ) A[ p ] = !A[ p ];              //    и инвертировать его
   } while ( p < m);
}
```

б) вариант для машинного слова

```
for ( int i = 0; i < 2^m; i++) { w = i ^ (i ≫ 1); yield(w); }.
```

5.3. Множества с повторениями. Тест — аналог кода Грея

Для некоторых задач необходимо обеспечить возможность нескольких вхождений элемента множества в тестовый набор. В этом случае можно считать, что источником тестовых наборов является *множество с повторениями*, или *мультимножество*, в котором каждый элемент может появляться несколько раз. Число возможных вхождений является существенным и носит название *кратности* элемента в множестве. Множество с повторениями, содержащее, например, элемент *a* кратности 2, элемент *b* кратности 3 и элемент *c* кратности 1, можно обозначить как { *a*, *a*, *b*, *b*, *b*, *c* } или { 2∘*a*, 3∘*b*, 1∘*c* }.

Порядок элементов не существенен, существенна только кратность: { *a*, *a*, *b*, *b*, *b*, *c* } = { *a*, *b*, *a*, *b*, *c*, *b* } ≠ { *a*, *b*, *c* }.

Для обычных множеств { *a*, *a*, *b*, *b*, *b*, *c* } = { *a*, *b*, *c* }.

Мощность множества с повторениями равна сумме кратностей его элементов.

Для представления множества с повторениями в памяти, кроме набора элементов, можно использовать массив кратностей, являющийся обобщением массива битов. Универсум M такого множества — это массив максимальных кратностей. Располагая таким массивом, можно построить алгоритм генерации последовательности множеств с повторениями, отличающимися друг от друга добавлением или удалением одного элемента — аналог кода Грея для мультимножеств.

```
void MGen(int m, int M[ ]) // Все подмножества мультимножества
{  // M в виде последовательности
   // с минимальным кодовым расстоянием
  int *D = new int[ m ], *S = new int[ m ];
  for (int j = 0; j < m; ++j) { S[ j ] = 0; D[ j ] = 1; } //инициализация
  do {
    yield(S); //Использовать S
    int j = 1, i = 0;
    while ( j ) {
      if (D[ i ])
 {  if (S[ i ] == M[ i ]) { D[ i ] = 0; --i; } // счёт назад
    else j=0;
 }
      else {
       if (S[ i ] == 0) { D[ i ] = 1; ++i; } // счёт вперёд
     else j = 0;
     }
   }
   if (i<n) if (D[ i ]) ++S[ i ]; else --S[ i ]; //счёт
} while (i < n);
delete [ ] D; delete [ ] S;
}
```

5.4. Случайное подмножество заданной мощности

Все рассмотренные ранее датчики генерировали множество случайной мощности. Если же требуется случайное подмножество заданной мощности *k*, например, в форме массива, его иногда пытаются получить следующим алгоритмом:

```
for ( int i  = 0; i < k; ++i )  X[ i ] = rand( ) % m.
```

Этот способ не годится, потому что он даёт не множество, а последовательность, в которой возможны повторы, и их будет много, если k близко к m, т. е. фактическая мощность множества будет меньше заданного k.
Можно усовершенствовать этот алгоритм: повторять генерацию очередного элемента множества до тех пор, пока не кончатся совпадения с уже имеющимися. Способ рекомендуется при больших m ($k \ll m$). Если же m не намного больше k, то с ростом k вероятность получить новый элемент множества очень быстро уменьшается, а при $k = m$ алгоритм может вообще никогда не остановиться.
Способ, рекомендуемый для небольших m: сформировать в памяти для результата массив — универсум, на каждом шаге убирать сгенерированный элемент множества в его начало и разыгрывать оставшиеся. Результат будет получен за время $O(k)$.

```
for ( int i = 0; i < m; ++i ) X[ i ] = i + 1;
                                  //Формирование универсума {1 … m}
for ( int i = 0; i < k; ++i) // Генерация подмножества мощностью k
{ int p = rand( ) % (m – i); // Случайный выбор среди оставшихся
if (p) Swap ( X[ i + p ], X [ p ] ); // Если p ≠ 0, обменять.
 }.
```

Результат — первые k элементов массива X. Способ легко приспособить для генерации последовательности подмножеств нарастающей мощности: использовать массив X в качестве теста после каждого добавления в него очередного элемента.
Если взять $k = m – 1$, получается *алгоритм генерации случайной перестановки*. Без ограничения общности он может использовать очередную перестановку для генерации следующей, не требуя для новой перестановки обязательного перезапуска датчика случайных чисел.

5.5. Последовательность всех подмножеств заданной мощности

Подавать на вход алгоритма последовательность всех подмножеств некоторого множества можно разными способами. Иногда может быть удобен лексикографический порядок последовательности подмножеств. Если разметить универсум и работать с номерами, то лексикографический порядок — это последовательность, упорядоченная по возрастанию образуемых этими номерами чисел. Так, для $m = 8$ и $k = 4$ это будет:

1 2 3 4 1 2 3 5 1 2 3 6 1 2 3 7 1 2 3 8 1 2 4 5 1 2 4 6
1 2 4 7 1 2 4 8 1 2 5 6 … 5 6 7 8.

K-элементные подмножества — лексикографический порядок

```
void GenK (int n, int k, int S[ ])
                          // K-элементные подмножества
{ int i, p;                 // в лексикографическом порядке
  for (i=0; i < k; ++i) S[ i ] = i + 1;
  do {
    yield(S); // Использовать S
    if (S[ k-1 ] == n) --p;
    else p = k - 1;
    if (p >= 0)
      for (i = k - 1; i >= p; --i)
S[ i ] = S[ p ] + i - p + 1;
  } while (p >= 0);
}
```

Для генерации последовательности подмножеств с минимальным кодовым расстоянием, т. е. отличающихся на каждом шаге заменой ровно одного элемента, необходим более сложный алгоритм. Он получает полный набор таких последовательностей и выводит каждую на экран. Алгоритм сводит задачу для последовательности из m по k к комбинации последовательностей меньшей мощности, вычисляемых рекурсивно. Результат выводится на экран.

K-элементные подмножества — последовательность с минимальным кодовым расстоянием

```
void GenPk (int n, int k, int S[ ], int &p) // K-элементные подмножества
{ int *B, i, j, p1, p2; // в виде последовательности
                        // с минимальным кодовым расстоянием
  if ((k == 1) || (k == n))
  { for (i = 0; i < n; ++i) S[ i ] = i + 1; // Тривиальный случай
    if (k == 1)
```

```
    { p = n; for(i = 0; i < n; ++i) cout << A[i];}
   else {p = 1; for(i = 0; i < n; ++i) cout << A[i];}
   cout << "!";
  }
  else {
   GenPk (n-1, k-1, A, p2); //Получить и запомнить окончание
   B = new int[ p2 * (k - 1) ];
   for (i = 0; i < p2 * (k - 1); ++i) B[ i ] = S[ i ];
   GenPk(n-1, k, A, p1);     // Получить начало
   for( i = 0; i < p2; ++i)      //   и присоединить окончание
   { for ( j = 0; j < k - 1; ++j)
         S[ p1 * k + i * k + j ] = B[ (p2 - (i + 1)) * (k - 1) + j ];
    S[ p1 * k + i * k + k – 1 ] = n;
   }
   delete [ ] B;
  p = p1 + p2; //Мощность результата
  for ( i = 0; i < p; ++i) { //Результат
   for (j = 0; j < k; ++j) cout << A[ i * k + j ];
   cout << "; ";
   }
  }
  cout << endl;
}
```

5.6. Генерация перестановок

Некоторые алгоритмы требуют подачи на вход полного множества *X* в виде последовательности, отличающейся порядком расположения элементов. Пример такого алгоритма — проверка двух графов одинаковой мощности на изоморфизм, заключающаяся в подборе такой нумерации вершин второго графа, чтобы его рёбра совпали с рёбрами первого графа. Функция *Neith*( ) генерирует все перестановки множества чисел от 1 до m в виде последовательности, в которой на каждом шаге меняются местами два смежных элемента.

```
inline void Swap( int &p, int &q) { int r (p); p = q; q = r; }
void Neith( int n )
{ int *X = new int[ m ], *C = new int[ m ], *D = new int[ m ], i, j, k, x;
  for (i = 0; i < m; ++i) // Инициализация
{ X[ i ] = i + 1; C[ i ] = 0; D[ i ] = 1; }
yield (X); // Использование первой перестановки
C[ m – 1 ] = –1; i = 0;
while ( i < m – 1 ) // Цикл перестановок
{     i = 0; x = 0;
while (C[ i ] == (m – i – 1))
{ D[ i ] = !D[ i ]; C[ i ] = 0; if (D[ i ]) ++x; ++i; }
if (i < m – 1) // Вычисление позиции k и перестановка смежных
{ k = D[ i ] ? C[ i ] + m : m – i – C[ i ] + x – 2;
        Swap( X[ k ], X[k + 1] ); }
yield (X); // Использование очередной перестановки
++C[ i ];
}
}.
```

Контрольные вопросы.

1. Какой способ размещения множеств в памяти является оптимальным для генерации кода Грея?
2. Получена последовательность кодов Грея: 0000 0001 0011 0010

Укажите следующий по порядку код.

3. Какова временная сложность оптимального алгоритма генерации произвольного случайного подмножества?
4. Какова временная сложность оптимального алгоритма генерации случайного подмножества заданной мощности?
5. Какова временная сложность оптимального алгоритма генерации всех перестановок элементов множества?

6. Множество — пользовательский тип данных

Если некоторая структура данных, например, массив, используется как реализация множества, это означает, что программист просто устанавливает для себя некоторые правила для работы с этим массивом и последовательно их придерживается. Часто большего и не требуется. Однако можно рассматривать множество как абстрактную структуру данных — область памяти, доступ к которой возможен только через некоторый интерфейс, т. е. набор функций, специально созданных для работы с этой памятью. Язык C++ поддерживает работу с абстрактными данными через механизм классов: абстрактная структура данных определяются как класс, в котором задаются как данные, так и связанные с ними операции. Определение класса позволяет расширить язык C++, включив в него множество как пользовательский тип данных.

Рассмотрим пример — класс для работы с множеством, представленным массивом символов (строкой). Кроме массива символов, в котором будет храниться множество, в класс будет естественно включить переменную *n* — мощность множества и *S* — символ-тег, с помощью которого множество можно идентифицировать.

```
class Set {
private:
  const int Nmax;
  int n; char S;
  char A[Nmax+1];
public:
  void AND(const Set &A, const Set &B);                    // (1)
  Set AND(const Set &B) const;                             // (2)
  friend void AND(Set &E, const Set &A, const Set &B); // (3)
  bool exist(char s) const { bool b(false);
      for(int i = 0; A[ i ] && !b; ++i) b |= A[ i ] == s;
      return b; }
  int power( ) const { return n; }
//…
} X1, X2, X3;
```

Класс *Set* является пользовательской структурой данных. Объявляя класс, можно (но не обязательно) объявить и объекты соответствующего типа: *X1*, *X2*, *X3*.

Синтаксис объявления класса — такой же, как и для структуры struct. Более того, ключевые слова *class* и *struct* в этом контексте полностью взаимозаменяемы. Отличие только в том, что всё содержимое класса class по умолчанию закрыто (*private*), а структуры struct — открыто (*public*). Тем самым обеспечивается обратная совместимость с программами на Си. На практике ключевое слово *struct* применяется для объявления простейших классов, в которых не предполагается ничего закрывать.

Назначение закрытой части — запретить произвольное использование объявленных в ней идентификаторов. Защита обеспечивается компилятором. Так, при попытке вычислить среднюю мощность объявленных множеств: (*X1.n* + *X2.n* + *X3.n*)/3 компилятор укажет на ошибку и сообщит «недоступно».

Для доступа к закрытым полям нужно предусмотреть соответствующие функции-члены. Такую роль выполняет функция power(). Следующее утверждение будет компилироваться:

```
int middle_power = (X1.power( ) + X2.power( ) + X3.power( ))/3;
```

Функция *power*( ) позволяет узнать значение переменной n, но не позволяет изменять его. Если нужна возможность изменения, функция должна возвращать ссылку:

```
  int & power( ) { return n; }
```

Но это считается плохой практикой. Возможность изменения проще обеспечить размещением переменной в открытой части класса.

В объявлении класса *Set* в качестве примера объявлены три варианта функции, вычисляющей пересечение двух множеств. Определить эти функции можно так:

```
void Set :: AND(const Set & A, const Set & B)                    // (1)
{ int k = 0;
  for(int i = 0; A.A[ i ]; ++i) if(B.exist(A.A[ i ]) A[k++] = A.A[ i ]; }
Set Set :: AND(const Set & B)                                    // (2)
{  Set E;
```

```
  for(int i = 0; A[ i ]; ++i) if(B.exist(A[ i ]) E.A[E.n++] = A[ i ];
  E.A[ n ] = 0;  return E;
}
void AND(Set & E, const Set & A, const Set B)                    // (3)
{  E.n = 0; for(int i = 0; A.A[ i ]; ++i)
   if(B.exist(A.A[ i ], B) E.A[E.n++] = A.A[ i ];
   E.A[E.n] = 0; }
```

Если сравнить два варианта функции того же назначения, обсуждённые ранее, можно видеть, что изменилось в объявлении и определении этих функций при работе с классом *Set*.

1. У функций в вариантах (1) и (2) исчез первый аргумент. Теперь его роль исполняет объект, для которого функция вызывается. Вызов для вычисления *X1 = X2 ∩ X3* может выглядеть так:

```
X1.AND(X2, X3);          // (1)
X1 = X2.AND(X3);         // (2)
AND(X1, X2, X3);         // (3)
```

2. Аргументами функций теперь являются объекты. Объявление таких аргументов обычным образом (просто заменой *char* [ ] на *Set*) приведёт к тому, фактические аргументы будут копироваться в формальные. Это возможно, но нежелательно. Для исключения копирования аргументы передаются как ссылки, а для того, чтобы защитить их от изменения в функции, применяются модификаторы *const*. Модификатор *const* в заголовке функции после скобок означает, что функция не меняет и объект, для которого вызывается. ВАЖНО: фактическим параметром функции, принимающей константную ссылку, может быть временный объект, образующийся в результате вычислений. Например, возможен следующий вызов:

```
X1.AND( X2.AND( X3.AND( X4 ) ) );
```

3. Функция-член имеет доступ ко всем полям своего объекта (*n*, *S*, *A*), как глобальным переменным, без всяких дополнительных указаний. Если объект — единственный аргумент, другие аргументы не нужны.
4. Функция (3) с модификатором *friend* — это функция-друг. Она вызывается как обычная функция, без неявного аргумента-объекта, но хотя бы один её аргумент должен быть объектом соответствующего типа. Функция-друг имеет те же права на доступ к внутренней части класса, что и функция-член.

Все три варианта функции *AND* могут быть определены одновременно (т. е. перегружены) и затем использоваться по усмотрению программиста.

Определить функцию можно как внутри класса (функции *exist*, *power*), так и вне его. ВАЖНО: функции, определённые внутри объявления класса, по умолчанию считаются встроенными, или макросами (*inline*). Так, вместо вызова функции *power*( ) будет просто подставлено поле *n* соответствующего объекта.

Хорошая практика: определять простые функции внутри определения класса, а более сложные — отложить, чтобы определение класса не было слишком громоздким и легко читалось.

Модификатор *inline* при желании можно указывать явно. Встраивание функции в любом случае не гарантируется. Более того, современные компиляторы часто игнорируют указание *inline* и могут самостоятельно сделать одни вызовы функции встроенными, а другие — нет, в зависимости от контекста вызова.

7. Типы классов и служебные функции-члены

Каждое объявление класса означает и объявление служебных функций-членов, предоставляемых системой по умолчанию. Для каждой из этих функций программист может, а часто и обязан предоставлять свою версию.

В первую очередь это функция-конструктор, автоматически вызываемая при создании объекта. Имя этой функции совпадает с именем класса, тип возвращаемого значения никогда не указывается. Назначение конструктора — инициализировать все поля объекта как подготовка к их использованию. Так, конструктор для приведённого выше класса Set может выглядеть так:

```
Set( ) : Nmax(26), n(0), S('R') { A[0] = 0; }
```

Важная особенность конструктора, отличающая его от других функций — наличие **списка инициализации**, состоящего из последовательности полей и присваиваемых им значений. Присваивание значений возможно и в теле конструктора, но этот способ рекомендуется применять, только если непосредственная инициализация невозможна или неудобна. ВАЖНО: порядок инициализации полей всегда совпадает с порядком их объявления, поэтому менять их порядок в списке инициализации не рекомендуется. Если в классе есть поля с модификатором const, список инициализации — единственный способ присвоить им значения, поскольку присваивание для таких полей запрещено.

В классе может быть определено любое количество конструкторов, различающихся типом и/или количеством аргументов.

Например, кроме приведённого выше конструктора пустого множества, можно иметь конструктор множества случайной мощности:

```
Set(char s ) : Nmax(26), n(0), S(s) {
  for(auto i = 0; i < Nmax; ++i) if(rand( )%2) A[n++] = U[ i ];
}
```

Этот конструктор отличается от предыдущего наличием аргумента — тега создаваемого множества. Он использует датчик случайных чисел, делая *Nmax* попыток. Если выпадает 1, к множеству добавляется *i*-ый символ из словаря *U*[ ], содержащего универсум.

Конструктор без аргументов — это важный частный случай, называемый конструктором **по умолчанию**. Конструктор, предоставляемый системой по умолчанию, ничего с объектом не делает, поэтому его обязательно нужно определять. Иногда это бывает невозможно, поэтому действует правило: любой конструктор с аргументом делает конструктор по умолчанию недоступным. С другой стороны, наличие конструктора по умолчанию — обязательное условие для создания из объектов этого класса массивов. Проблема может быть снята явным объявлением конструктора по умолчанию:

```
Set( ) = default;
```

Этот конструктор сам по себе ничего не делает, но он запускает конструкторы для отдельных полей, если они являются объектами некоторого класса.

Современные компиляторы (C++17 и выше) разрешают инициализацию полей непосредственно в объявлении класса:

```
class Set {
private:
  const int Nmax = 26;
  int n = 0; char S = 'R';
  char A[Nmax+1];
public: …
```

Это означает, что соответствующие инициализаторы будут включены во все конструкторы, с правом программиста при необходимости изменить инициализацию по умолчанию.

Проблему конструктора по умолчанию можно решить также указанием значений аргументов по умолчанию, например, так:

```
Set(int n = 0; char s = 'R' ) : Nmax(26), n(n), S(s) {  }
```

Такой конструктор может быть вызван с двумя аргументами Set(10, 'A'), с одним аргументом Set(10) или вообще без аргументов, как конструктор по умолчанию. Заметим, что это не три разных функции, а одна функция с тремя способами вызова.

Кроме конструктора по умолчанию, система предоставляет каждому классу конструктор копии, перегрузку присваивания и деструктор.

Конструктор копии используется для создания нового объекта копированием существующего. Это конструктор с аргументом — константной ссылкой на объект того же типа.

```
Set(const Set &);
```

Такой же аргумент имеет и перегрузка присваивания:

```
Set & operator = (const Set &);
```

Перегрузка присваивания отличается тем, что вызывается для уже существующего объекта. Обе функции по умолчанию копируют свой аргумент в создаваемый или существующий объект по принципу байт в байт.

Деструктор — это функция с именем, совпадающим с именем класса, которому предшествует знак '~':

```
~Set( );
```

Для этой функции также никогда не указывается тип возвращаемого значения. Список аргументов у неё всегда пуст, поэтому функция существует в единственном варианте, и перекрытие её невозможно. Назначение функции — действия при уничтожении объекта. Если такие действия нужны, функцию следует определить, потому что по умолчанию она ничего не делает.

Существуют **три типа классов**, различающихся необходимым набором служебных функций:

— класс-понятие;

— класс-значение;

— класс-контроллер.

Класс-понятие — это класс без данных (обычно *struct*, т. к. в закрытую часть такого класса нечего помещать). Примеры:

```
struct input_iterator_tag{ };
struct Error{ };
struct less { bool operator( )(int a, int b){ return a < b; }
```

С такими классами мы встретимся во второй части курса. Первый пример — это тег итератора, с помощью которого задаются его свойства. Второй пример — это объявление класса ошибки. Такой класс используется как аргумент оператора возбуждения исключения «throw Error» и его перехвата «catch (Error)». Третий пример — это объявление функтора,

использующегося как аргумент для алгоритмов, в которые нужно передавать функцию.
Для класса-понятия рассмотренные выше служебные функции вообще не требуются.
Класс-значение (простейший класс в терминологии разработчика языка C++ Б. Страуструпа) — это класс, имеющий поля-данные.
Такому классу необходимы только конструкторы. Конструктор копии и перегрузку присваивания для них тоже можно определить, но делать этого не следует без особых причин, потому что функции, предлагаемые по умолчанию, всегда работают быстрее. Деструктор для такого класса не нужен, т. к. ему нечего делать.
Класс-контроллер — это класс, управляющий ресурсом. Именно такие классы обсуждаются в большинстве книг по программированию, оставляя первые два типа в тени.
Ресурс — это нечто вне класса. Чаще всего это память, но может быть и файл, канал связи и др.
ВАЖНО: для обеспечения устойчивости программы к сбоям ресурс, управляемый классом, должен быть **единственным**. И наоборот, если программе требуется ресурс, следует объявить класс, который будет им управлять. Смысл этих правил будет подробно рассмотрен в теме «Исключения» во второй части курса.
Рассмотренный выше пример класса для множества в массиве — это класс-значение. Его можно переделать в класс-контроллер, если перенести множество-массив в свободную память.

```
class Set {
private:
  const int Nmax;
  int n; char S;
  char *A;
public:
Set( ) : Nmax(26), n(0), S('R'), A(new char[Nmax]){ }
~Set( ) { delete [ ] A; }
…
```

В этом классе вместо массива хранится указатель на него. Конструктор инициализирует указатель адресом в свободной памяти, где располагается массив, а деструктор будет освобождать эту память при уничтожении объекта.
Для класса-контроллера определение конструктора копии и перегрузки присваивания является **обязательным**, потому что ресурс по умолчанию не копируется, и это приводит к тому, что в результате копирования или присваивания появляются два объекта, управляющие одним и тем же ресурсом. Этого можно не заметить, если для класса не определён деструктор. Тогда при уничтожении исходного объекта ситуация исправляется, и программа нормально работает. Но при окончательном удалении объектов их ресурсы остаются в памяти в качестве мусора. Если деструктор определён, тогда удаление исходного объекта удаляет и ресурс, и указатель в объекте копии становится недействителен. Удаление объекта-копии сопровождается попыткой повторного удаления уже не существующего ресурса, что приводит к аварийному завершению программы.
Для рассмотренного выше примера конструктор копии и перегрузка присваивания могут выглядеть так:

```
Set(const Set & other ) : Set( ) {
   n = other.n; S = other.S; strcpy(A, other.A); }
Set & operator = (const Set & other) {
   if (this != &other) {
      n = other.n; S = other.S; strcpy(A, other.A);
  }
  return *this;
}
```

Конструктор копии должен делать то же самое, что и конструктор по умолчанию, т. е. создавать в памяти массив для множества. Это можно обеспечить вызовом последнего в списке инициализации, чтобы заставить его выполниться перед собственно копированием. Такой приём называется *делегирование конструкторов*. Он доступен для версий C++17 и выше.
Перегрузка присваивания имеет дело с уже существующим объектом. Она начинается с проверки на **самоприсваивание**. Здесь используется указатель *this* = адрес объекта слева от знака присваивания. Если этот адрес не совпадает с адресом аргумента, производится копирование содержимого всех полей объекта. В обоих случаях предполагается, что множество представлено строкой в стиле Си, т. е. массивом символов, ограниченном нулём.
Создание копии ресурса при копировании объекта иногда может быть избыточным. Если при передаче аргументов в функцию проблема снимается заменой значения константной ссылкой, то при возврате результата из функции дублирование ресурса особенно неприятно, потому что создаваемый при этом временный объект в итоге уничтожается вместе с вновь созданным ресурсом.
Для снятия этой проблемы в версию C++11 были добавлены **перемещающий** конструктор и **перемещающее** присваивание, которые не создают копию ресурса, а просто передают его в результат. Для обеспечения такой возможности в язык была добавлена операция '&&' — правая ссылка (*r-value*), а существующий знак операции '&' стал обозначать левую, или

универсальную ссылку (*l-value*). Перемещающие варианты копирования и присваивания для рассматриваемого примера класса могут выглядеть так:

```
Set(Set && other ) : n(other.n), s(other.S), A(other.A) {
   other.A = nullptr; }
Set & operator = (Set && other) {
   if (this != &other) {
      n = other.n; S = other.S; A = other.A; other.A = nullptr;
  }
  return *this;
}
```

Перемещающий вариант копирования или присваивания выбирается компилятором в случае, когда копируется временный объект. Такой выбор можно форсировать, если применить служебную функцию *move*:

```
Set A(std::move(B));
 при создании нового объекта из существующего и
return std::move(C);
```

при возврате значения функции. Здесь *std*::*move*( ) — это не функция в обычном понимании, а средство преобразования универсальной ссылки в правую.

Обсудим теперь вопрос, класс какого типа следует использовать в эксперименте с множествами в памяти ЭВМ. Очевидно, что множество в машинном слове может быть реализовано только как класс-значение, а множество в списке — только как класс-контроллер. Для массива и массива битов возможны оба варианта. Есть основание предполагать, что вариант с классом-контроллером будет работать быстрее за счёт исключения ненужного копирования массивов.

Каждое объявление класса означает и объявление служебных функций-членов, предоставляемых системой по умолчанию. Для каждой из этих функций программист может, а часто и обязан предоставлять свою версию.

Возможный вариант к**ласса-множества на основе массива** с полным набором функций-членов

```
class Set {
private: // Закрытая часть класса — данные
static int N; // мощность универсума
  int n; // мощность множества
  char S, *A; // тег и память для множества
public: // Открытая часть — функции для работы с множеством
   Set operator | (const Set&) const; // объединение
   Set operator & (const Set&) const; // пересечение
   Set & operator |= (const Set&); // объединение и присваивание
   Set & operator &= (const Set&); // пересечение и присваивание
   Set operator ~ ( ) const; // дополнение до универсума
   void Show( ); // вывод множества на экран
   operator int ( ) { return n; } // получение мощности
   explicit Set(char); // конструктор множества
   Set( ); // ещё конструктор — по умолчанию
   Set(const Set &); // конструктор копии
   Set(Set &&); // конструктор копии с переносом (C++11)
   Set operator = (const Set &); // оператор присваивания
   Set operator = (Set &&); //оператор присваивания с переносом (C++11)
   ~Set( ) { delete [ ] A; } // деструктор
};
```

Имя класса *Set* — это имя нового типа данных. С его помощью будем объявлять в программе множества-объекты. Память для множества находится в закрытой части класса и доступна через член *A* — указатель на символы. Размер памяти не определён. Кроме этого, в закрытую часть помещены вспомогательные переменные-члены: мощность универсума *N*, текущая мощность множества *n* и символ-тег *S*, с помощью которого можно различать объекты-множества. Мощность универсума *N* объявлена со спецификатором «*static*». Это означает, что все объекты класса *Set* будут использовать единственную копию этой переменной. Переменная *N* должна быть дополнительно объявлена вне всех функций, чтобы ей была выделена память. При этом требуется установить и её значение:

```
int Set :: N = 26; // Мощность универсума для множества латинских букв.
```

В открытой части класса объявлены функции-члены, с помощью которых в программе-клиенте можно работать с множеством. Каждая функция-член имеет в качестве обязательного аргумента объект, для которого она вызывается. Данные-члены из закрытой части класса доступны в ней как обычные глобальные переменные, и их тоже не нужно передавать как аргументы.

Всё это позволяет свести количество аргументов функций-членов к минимуму или даже совсем от них отказаться, не засоряя при этом пространство глобальных имён.

Для работы с множествами-массивами предполагается использовать такой же синтаксис, как для машинных слов. С этой целью функции объединения, пересечения и дополнения множеств объявлены с именами, содержащими ключевое слово «*operator*», после которого следует знак соответствующей операции. Операции языка C++ «|», «&» и «~» определены так, чтобы их можно было использовать в выражениях, состоящих из данных типа *Set*. Такой приём называется перегрузкой операций. Чтобы это действительно можно было возможно, функции объявлены так, чтобы была обеспечена совместимость со встроенными операциями языка C++: все функции возвращают объект типа *Set*, а двуместные операции в качестве аргумента (второго, потому что первый — это сам объект) имеют константную ссылку на объект типа *Set*. Функции не меняют объект, для которого вызываются. Для контроля за этим в каждом из объявлений после списка параметров помещён спецификатор *const*.

Если мы объявляем для своего типа данных для операции пересечения перегрузку знака «&», а также перегрузку присваивания «=», то это не делает автоматически доступной комбинированную операцию «&=» — пересечение и присваивание. Такую операцию нужно тоже перегружать явно. Более того, рекомендуется обязательно сделать это и сделать согласованно. Проще всего этого добиться, реализую двуместную операцию через комбинированную:

```
Set& Set :: operator &= (const Set & B)
{ Set C(*this);
  n = 0;
  for (int i = 0; i < C.n; ++i) {
  for (int j = 0; j < B.n; j++)
        if (C.A[ i ] == B.A[ j ]) A[ n++ ] = C.A[ i ];
  }
  A[ n ] = 0; // ограничитель строки
  return *this;
}
Set Set :: operator & (const Set & B) const
{ Set C(*this);
  return (C &= B);
}
```

В первой функции объявляется множество *C*, которое конструктор копии заменяет текущим множеством, для которого вызвана операция (множеством слева от присваивания). Затем текущее множество делается пустым, и в нём формируется результат пересечения временного объекта *C* и множества *B*. Поскольку для этого используется двойной цикл по мощности множеств, временная сложность операции — квадратичная. Результат — текущий объект. Во избежание лишнего копирования можно в качестве результата вернуть ссылку на него.

Вторая функция — двуместная операция пересечения множеств — тоже сперва создаёт копию текущего объекта, а затем возвращает результат комбинированной операции с временным объектом в левой части. Возврат ссылки здесь недопустим: по выходе из функции временный объект автоматически уничтожится, и возвращённая ссылка станет недействительна.

Операция объединения множеств «|» реализуется похожим алгоритмом:

```
Set & Set :: operator |= (const Set & B)
{ for(int i = 0; i < B.n; ++i) {
  bool f = true;
  for (int j = 0; j < n; ++j)
        if (B.A[ i ] == A[ j ]) f = false;
  if (f) A[ n++ ] = B.A[ i ];
  }
  A[ n ] = 0;
  return *this;
}
Set Set :: operator | (const Set & B) const
{ Set C(*this);
  return (C |= B);
}
```

В текущий объект-множество добавляются недостающие элементы из *B*.

Операция вычисления дополнения может быть реализована так:

```
Set Set :: operator ~ ( ) const
{ Set C;
```

```
for (char c = 'A'; c <= 'Z'; ++c) { //Вариант для латинских букв!!!
  bool f = true;
  for (int j = 0; j < n; ++j)
        if (c == A[ j ]) { f = false; break; }
  if (f) C.A[ C.n++ ] = c;
  }
  C.A[ C->n ] = 0; //Эквивалентно '\0'
return C;
}
```

Здесь в качестве одного из операндов выступает множество-универсум. Результат — элементы универсума, которых нет в исходном множестве.

Поскольку количество повторений цикла по элементам универсума постоянно, временная сложность операции — $O(n)$. Однако, если учесть, что мощность универсума не может быть меньше мощности его подмножеств: $|U| \geq n$, более точной будет оценка $O(|U| * n)$, более пессимистическая по сравнению с $O(n^2)$.

Разумеется, нельзя обойтись без функции *Show*( ) для вывода множества на экран: это единственный способ увидеть результат обработки, поскольку сами множества из вызывающей программы недоступны.

```
void Set :: Show( ) { cout << '\n' << S << " = [" << A << "]"; }
```

Для определения мощности множества нужна специальная функция *power*( ). Эта функция просто возвращает значение закрытой переменной *n*, в которой другие функции поддерживают значение текущей мощности множества. Поскольку функция не только объявлена, но и определена внутри класса, она по умолчанию является *встроенной.* К объявлению функции неявно добавляется спецификатор *inline*. Это означает, что никакой функции не создаётся, вместо этого в каждую точку вызова просто подставляется значение закрытой переменной *n*. Таким образом, запрет на доступ к *n* обходится без всяких дополнительных расходов на вызов функции.

Все функции-члены, кроме перечисленных ранее, объявлять и определять не обязательно, компилятор делает это автоматически. Но в данном случае автоматически определяемые функции не подходят.

Так, при создании объекта класса *Set* выделяется память, после чего вызывается функция-конструктор *Set*( ). По умолчанию эта функция — пустая, она ничего с памятью не делает, и использовать такой объект невозможно. Поэтому конструктор надо определить явно. Сделаем так, чтобы он создавал пустое множество латинских букв, представленное строкой символов:

```
Set :: Set( ): n(0), S ('0'), A(new char[ N+1 ]) { A[ 0 ] = 0; }
```

В этом примере одни переменные инициализируются в заголовке, другие — в теле конструктора. Оба способа можно комбинировать произвольным образом, но нужно учитывать, что порядок инициализации переменных полностью определяется порядком их объявления в классе и не может быть изменён. В данном примере значение переменной *N* используется для создания строки *A*, поэтому важно, что переменная *N* объявлена в классе раньше, чем указатель *A*. Массив символов объявляется на 1 символ длиннее мощности универсума, чтобы резервировать место под ограничивающий нуль. Поскольку строка должна быть пустой, ограничивающий нуль записывается в её начало. Инициализировать остальную часть массива не обязательно.

Если требуется иметь несколько способов создания объекта, для каждого способа объявляется свой конструктор, отличающийся от других типом и/или количеством аргументов. В примере объявлен конструктор с одним символьным аргументом. Это может быть конструктор, генерирующий случайную строку латинских букв. Аргумент используется для инициализации тега. С помощью датчика случайных чисел генерируется *N* битов, и для каждого единичного бита соответствующий ему элемент добавляется в множество. Одновременно подсчитывается фактическая мощность множества *n*. По окончании генерации в строку добавляется ограничитель. Сгенерированное множество выводится на экран.

```
Set :: Set(char s): S(s), n(0), A(new char[ N+1 ])
{
  for (int i = 0; i < N; i++)
 if (rand() % 2) A[ n++ ] = i + 'A';
  A[n] = 0;
  cout << '\n' << S << " = [" << A << "]";
}
```

Следующие две функции-члена — конструктор копирования и перегрузку присваивания — для класса-значения обычно не определяют. Обе эти функции по умолчанию копируют один объект в другой по принципу «байт в байт». Если ничего другого не требуется, определять эти функции не нужно, так как компилятор наверняка сделает это лучше.

В данном же случае такое копирование не годится, потому что в классе есть указатель на дополнительную память, и копирование приведёт к тому, что указатели *A* в обоих объектах будут указывать на одну и ту же строку.

Конструктор копирования имеет единственный аргумент — константную ссылку на объект того же класса. Определить конструктор можно так:

```
Set :: Set(const Set & B) : S(B.S), n(B.n), A(new char[N+1])
{ char *s(B.A), *t(A);
  while(*t++ = *s++);
}
```

Здесь переменные *S* и *n* копируются обычным способом, а для указателя *A* выделяется память под новую строку, и в неё затем копируется содержимое старой. Переменную *N* копировать не нужно: она является общей для всех экземпляров объекта типа Set.

Функция-член для перегрузки присваивания отличается от копирования тем, что объект в левой части оператора уже существует. Более того, он может совпадать с аргументом (самоприсваивание). Поэтому первое, что функция должна сделать — проверить это. Затем текущий объект уничтожается и создаётся новый. В нашем случае можно ограничиться переносом содержимого строки в имеющуюся память. Поскольку результат операции присваивания может быть использован в выражении, например в цепочке присваиваний, функция должна возвращать значение объекта, для которого она вызвана. Это делается с помощью встроенного указателя *this*. Копировать тег нет смысла: устанавливается некоторое условное значение «*R*» (от слова «*result*»).

```
Set & Set :: operator = (const Set& B)
{    if (this != &B)
     { char *s(B.A), *t(A); n = B.n; S = 'R';
       while(*t++ = *s++);   }
return *this;
}
```

Конструктор копирования используется при инициализации объекта содержимым другого объекта в момент объявления, а также при передаче аргумента в функцию как параметра по значению и при возврате объекта как результата работы функции. Функция перегрузки присваивания вызывается соответствующим оператором. Бывают ситуации, когда эти функции в программе не нужны. Чтобы исключить трудно выявляемую ошибку в программе из-за использования функций по умолчанию, рекомендуется объявить ненужные функции в закрытой части класса. Можно даже не определять их. Невольное создание в программе ситуации, когда такая функция вызывается, будет ошибкой, выявляемой компилятором.

В стандарте C++11 добавлены конструктор копирования и перегрузка присваивания в варианте «**с переносом**». Они используются, если источником данных является временный объект. Вместо того, чтобы создавать копию данных, принадлежащих объекту, в варианте «с переносом» эти данные просто передаются объекту-приёмнику:

```
Set :: Set(Set && B)
        : S(B.S), n(B.n), A(B.A) //Копирование с переносом
{  B.A = nullptr; }
Set & Set :: operator = (Set&& B) //Присваивание с переносом
{    if (this != &B)
{ n = B.n; A = B.A;  S = 'R'; B.A = nullptr; }
return *this;
}
```

Указатель на строку в объекте-источнике обнуляется, чтобы сделать удаление этого объекта безопасным.

Последняя функция-член в объявлении класса — это деструктор, который автоматически вызывается при уничтожении объекта. В нём указано дополнительное действие, которое нужно выполнить перед освобождением памяти из-под объекта: уничтожить строку *A*. Поскольку деструктор определён внутри класса, он, как и *power*( ), тоже является встроенной функцией. Деструктор вызывается явно в операторе *delete* или неявно — при выходе из блока, в котором объект был определён. Объекты уничтожаются в порядке, обратном порядку их создания.

Программа, использующая объекты класса *Set* (программа-клиент), может выглядеть так:

```
#include <string>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
using namespace std;
#include "Set.h"  //Модуль с определением класса и его функций
int Set :: N = 26; // определение статического члена класса
const long q0 = 100000; // количество повторений цикла времени
int main( )
{ srand(time(nullptr));
```

```
  Set A('A'), B('B'), C('C'), D('D'), E;
  clock_t begin = clock( );
  for(long q =0; q < q0; ++q)
   { E = (A | B) & (C & ~D); }
  clock_t end = clock( );
  E.Out( );
  cout << " Middle power =" <<
  (A.int() + B.int( ) + C.int( ) + D.int( ) + E.int( )) / 5 <<
        " Time=" << end – begin<< " / " << q0 << endl;
  cin.get( );
  return 0;
}
```

В программе определяются пять множеств. Для исходных множеств *A*, *B*, *C* и *D* используется конструктор, генерирующий случайное множество с выводом на экран результата. Множество *E* генерируется конструктором по умолчанию как пустое. Затем множество вычисляется с использованием перегруженных операций, и результат выводится на экран. Далее вычисляются и выводятся средняя мощность всех множеств и время решения задачи.

Объявление класса *Set* и определения всех функций-членов находятся в подключаемом модуле *Set.h*. Определение статического члена — переменной *N* — помещено сразу после модуля. На самом деле оно является его частью. В самой программе никакой информации об устройстве модуля *Set.h* не имеется. Чтобы использовать другой способ хранения множеств в памяти, достаточно просто подменить модуль.

Вычисление множества *E* выполняется одним оператором присваивания, как в варианте для машинных слов. Но временная сложность этого вычисления не будет константной, она определяется функциями, реализующими операции над множествами и, следовательно, по-прежнему зависит от способа их хранения в памяти.

Контрольные вопросы.

1. Можно ли использовать ключевое слово *struct* в том же смысле, что и *class*?

```
class MyClass {
    int A;
public:
int FunA ( );
};
int FunB( MyClass A );
```

Может ли функция *FunA*( ) использовать переменную *A*, и если «да», то каким образом?

2. Тот же класс, сигнатура функции теперь

```
int FunB( const MyClass & S );
```

Каким образом функция *FunB*( ) может использовать переменную *A* объекта *S*?

3. Каким образом функция *FunB*( ) может использовать функцию *FunA*( ) объекта *S* в примере выше?

4.

```
 class MyClass {
    int A;
public:
MyClass ( int );
};
```

Можно ли создать массив объектов типа *MyClass*?

5.

```
        class MyClass {
    static int A;
public:
MyClass ( int );
};
```

Что означает атрибут *static* у переменной-члена *A* класса *MyClass*?

6. Как следует задавать значение статической переменной *A* в примере выше?
7. С каким ключевым словом следует объявить переменную в классе, чтобы она была общей для всех экземпляров объекта соответствующего типа?
8. С каким ключевым словом следует объявить функцию, чтобы она, не будучи членом класса, имела доступ ко его закрытым полям?
9. Можно ли объявить класс, который будет играть роль элемента множества списка и множества в целом?
10. Что лучше — объявить класс-элемент списка как структуру внутри класса-списка или объявить два отдельных класса, связав их дружбой?

11. К какому типу классов относится следующий класс?

```
template <class T> class less {
      public:
bool operator( )(T a, T b) { return a < b; }
};
```

12. К какому типу классов относится следующий класс?

```
template <class T, int p = 26> class MyClass {
 enum( N = p);
 int n;
 T Data[N];
public:
 MyClass( ): n(0) { Data[0] = 0; }
/…
};
```

13. К какому типу классов относится следующий класс?

```
template <class T, int p = 32> class MyClass {
 enum( N = p);
 int n;
 T * Data;
public:
 MyClass( ): n(0), Data(new T[N]) { Data[0] = 0; }
/…
};
```

14. К какому типу классов относится следующий класс?

```
template <class T, int p = 26> class MyClass {
 enum( N = p);
 int n;
 vector< T > Data;
public:
 MyClass( ): n(0) {  }
/…
};
```

15. К какому типу классов относится следующий класс?

```
template <class T, int p = 26> class MyClass {
 enum( N = p);
 int n;
 unique_ptr< T[ ] > Data;
public:
 MyClass( ): n(0), Data(make_unique(new T[N]) {  }
/…
};
```

16. Каковы преимущества объектно-ориентированного подхода к программированию по сравнению с процедурным?

8. Перегрузка операций

СПОСОБ: объявить функцию с именем «operator #», где за ключевым словом operator следует знак перегружаемой операции.
— Позволяет использовать более удобные средства работы с объектами, чем обычные функции.
Пример:
E = A | (B & C & ~D);
вместо E = or(A, and(B, and(C, not(D))));
или not(D, E); and(C, E, E); and(B, E, E); or(A, E, E);
ПРАВИЛА перегрузки операций:
— перегружать можно почти все существующие в языке знаки операций. Исключения: «.», «::», «?:», «#», sizeof;

— нельзя придумывать новые знаки операций;
— нельзя переосмысливать операции над встроенными типами: хотя бы один аргумент должен быть пользовательским;
— нельзя изменить количество аргументов (1 или 2) и применять аргументы по умолчанию. Но можно в двуместной операции игнорировать один из аргументов;
— нельзя изменить приоритеты и ассоциативность.
Пример: перегружены операции «+» и «*» и предложен знак «^» для возведения в степень комплексных чисел. Выражение *a = b + c*d^n* означает *a = (b + (c*d))^n*, а не *a = (b + (c*(d^n))*, как ожидал программист.
ВЫВОД: перегрузка должна быть естественной. Цель: улучшить читабельность текста программы, а не наоборот (показать крутизну программиста)! Если эта цель не достижима, от перегрузки следует отказаться.
СЛЕДСТВИЕ: нужно по возможности поддерживать семантику перегружаемых операций, чтобы можно было строить выражения привычным способом. А именно: результат операции должен быть пригоден как аргумент для следующих операций.
Можно ли отказаться от перегрузки совсем? Не получится:
— перегрузка присваивания определена для пользовательского типа (равно как и взятие адреса, запятая);
— перегрузка применяется в стандартных библиотеках:
«+» для *string* — конкатенация строк;
сдвиги — для объектов *iostream*.
Естественно применять арифметические операции для комплексных, рациональных, сверхдлинных чисел и т. п. (и это делается в доступных библиотеках).
— СПОСОБЫ объявления: функция-член, функция-друг, обычная функция. Какой способ выбрать?
Обычная функция возможна, если не требуется доступ к закрытой части класса.
Операция, манипулирующая адресом объекта, должна быть (нестатическим) членом: *operator=*, *operator*[ ], *operator*( ), *operator->*
Операции присваивание, взятие адреса и запятая обладают предопределёнными свойствами для объектов. Их можно сделать частными (запретить) или определить иначе.
Операция, первый аргумент имеет стандартный (другой существующий) тип, не может быть членом.
Пример: вывод множества на экран (первый аргумент — поток):

```
class Set {
  const int U = 26;
  int n; char S; char A[U + 1];
  friend ostream& operator <<(ostream & o, const set& src)
        { o << S << '=' << A << std::endl; return o; }
//…
}
```

Для данных *int i*++ означает то же, что и *i* += 1 или *i* = *i* + 1. Для пользовательских типов такой связи по умолчанию нет, все варианты надо определить явно.
ПРЕОБРАЗОВАНИЕ ТИПА
— конструктор с одним аргументом по существу является преобразователем типа и может вызываться неявно. Чтобы это запретить, его описывают с ключевым словом *explicit*.

```
  explicit Set(int power); // Конструктор случайного множества заданной мощности.
```

Преобразование в стандартный или существующий тип можно определить явно:
operator int (*const Set* &) { *return n*; } — преобразование к целому. Возможный смысл — выдача мощности для множества. Функция не содержит указания на тип возвращаемого значения!
Во избежание недоразумений для выдачи мощности лучше использовать вариант с явным именем функции: *Set*::*power*( ) { *return n*; }
ВОЗВРАТ ССЫЛКИ
Обязателен, если результат может быть слева от присваивания (=, [ ], ->).
Разумен, если возвращается аргумент-ссылка (как в перегрузке operator >>) или аргумент-объект (как в перегрузке присваивания):

```
Set& Set :: operator = (const Set & B)
{  if (this != &B) {
      n = B.n;  S = B.S;
      char *a(A), *b(B.A); while(*a++=*b++); //); ctrcpy (A, B.A);
   }
   return this*
}
```

ИНКРЕМЕНТ и ДЕКРЕМЕНТ — примеры для работы с множествами

1. Инкремент для объекта — указателя на элемент списка

```
Y* X::operator ++ ( ) //префиксный инкремент для объекта-указателя
{ p = p->next;    //
  return p;
}
Y* X::operator ++ (int)   //то же — постфиксный
{ X temp (*this);
  ++ *this;
  return temp;
}
```

2. Инкремент для множества-массива: увеличение мощности.

```
const Set & Set::operator ++ ( ) const        //префиксный вариант
{
    if(n < N) for (char c = 'A'; c <= 'Z'; ++c) //Цикл по универсуму
    { bool f = true;
      for (El * j = A; j && f; j = j->next) //Добавить отсутствующий
          if(c == j->e) f = false;
          if (f){  A = new El(c, A) , ++n;  break; }
    }
    return *this;
}
```

Постфиксный вариант совпадает с предыдущим примером.

Комбинированные присваивания

Нужно обязательно объявлять оба варианта — с присваиванием и без. Разумно обеспечить согласованность: двуместную операцию определить через комбинированное присваивание.

Пример для множества на основе списка:

```
const Set & Set::operator &= (const Set & B)
{     Set C;
// std::swap(C, *this); «Естественный» способ приводит к бесконечной рекурсии!
std::swap(C.A,A); std::swap(C.S,S); std::swap(C.n,n); //Вместо Set C(*this);
    for (El * i = C.A; i; i = i->next)
        for (El * j = B.A; j; j = j->next)
          if (i->e == j->e)
            A = new El(i->e, A), ++n;
   }
    return *this;
}
Set Set::operator & (const Set& B) const
{   Set C(*this);   return (C &= B);  }
```

Перегрузка операций — естественный способ расширения языка в нужную предметную область.

Цель: получить эффективный и удобный в сопровождении программный код.

Перегружать можно почти всё.

Нельзя:

— придумывать новые знаки операций;

— изменить количество аргументов (один или два);

— изменить приоритет.

Полный перечень операций языка C++,
упорядоченный по уменьшению приоритета
(**выделены операции**, не пригодные для перегрузки)

1. :: **— уточнение области действия**

2. круглые скобки:

(выражение) — группировка для изменения приоритета
имя( ) — вызов функции (слева направо)
тип(выражение) — конструкция значения
[ ] — индексация
. — прямое членство
-> — косвенное членство
++ — постфиксный инкремент
– – — постфиксный декремент
const_cast, dynamic_cast, reinterpret_cast, static_cast, typeid — снятие атрибута *const*, преобразование типа, взятие идентификатора типа

3. ! ~ + – — одноместные «не», обратный код, плюс, минус
++ – – префиксные инкремент и декремент
& — взятие адреса
* — разыменование
(тип) выражение — приведение типа (в стиле Си)
sizeof — взятие размера
alignof, new, new[ ], delete, delete [ ], noexcept — взятие выравнивания, выделение и освобождение памяти, атрибут отсутствия исключений

4. **.* — разыменование члена — прямое**
–>* — то же, косвенное
5. 5. * / % — арифметические: умножение, деление, остаток от деления
6. + – — арифметические: сложение, вычитание
7. ≫ ≪ — сдвиги
8. < <= >= > — сравнения на меньше/больше
9. == != — проверка на совпадение
10. & — поразрядное «И»
11. ^ — исключающее «ИЛИ»
12. | — поразрядное «ИЛИ»
13. && — логическое «И»
14. || — логическое «ИЛИ»
15. **: ? — условная операция**
16. = *= /= %= += -= &= ^= |= ≪= ≫= — присваивания
17. throw — исключение
18. , — запятая

ВАЖНО: использование перегрузки операций — то же самое, что и вызовы соответствующих функций. Запись
E = A | (B & C & ~D);
для перегруженных операций на самом деле эквивалентна
E = A.operator | (B.operator & (C.operator &(D.operator ~( ));
Принципиальное отличие: аргументы функций вычисляются всегда и порядок их вычисления не определён.
Следствие: не рекомендуется во избежание недоразумений перегружать логические операции и запятую.
К перегрузкам можно отнести и преобразование типа. Конструктор с одним аргументом — преобразователь типа аргумента в тип объекта.
Set(10) — конструктор множества мощностью 10 = преобразователь целого числа в множество. Предохранение:
explicit Set(int);
Преобразование объекта в стандартный (любой другой) тип:
operator int( ) — функция-член для преобразования (множества) в целое. Не рекомендуется, потому что смысл не очевиден!
В общем случае: автоматическое преобразование типа объекта в месте, где ожидается другой тип (аргумент функции).
Альтернатива — **явные преобразования**
— преобразование «в стиле Си» — (тип)выражение;
— альтернатива для C++ — тип(выражение);
Рекомендуемый способ — *static_cast*<тип>(выражение) или reinterpret_cast<тип>(выражение).
Первый способ — это преобразование значения выражения к указанному типу, например, целого к вещественному. Второй способ означает замену типа без преобразования выражения, например, объявление указателя целым числом или наоборот.

Специальные способы (потенциально опасные!):
const_cast — снятие атрибута const/volatile,
dynamic_cast — преобразование указателя в иерархии типов;
Перегрузка круглых скобок: класс-функтор, используемый обычно как аргумент в качестве указателя на функцию:

```
class less{
  int A;
public:
  less(int a): A(a) { }
  bool operator( )(int B){return B < A;}
};
```

Использование:

```
   erase(arr, N, less(10)); //Удалить из массива элементы, меньшие 10.
```

Перегрузка присваивания — встроена в язык. Объявление класса автоматически означает объявление для него в качестве функции-члена перегрузки присваивания.
Типовое использование перегрузки операций:
— круглые скобки: функторы;
— арифметические: (комплексные, сверхдлинные числа);
— поразрядные (работа с множествами);
— сравнения (предикаты — функции с результатом типа bool);
— операции с указателями: умные указатели, итераторы;
— префиксный и постфиксный инкремент/декремент.
ВАЖНО. Префиксную и постфиксную формы категорически рекомендуется определять одновременно.
Пример:

```
Y* X::operator ++ ( ) //префиксный инкремент для объекта-указателя
{ p = p->next; //(Вариант для списка: инкремент перемещает указатель)
  return p;
}
Y* X::operator ++ (int)       //то же — постфиксный
{ X temp (*this);
  ++ *this; //Постфиксный реализован через префиксный !
  return temp;
}
```

ВАЖНО. Согласованность определений перегрузки для логически связанных операций.
Пример: двуместная операция и комбинированное присваивание.

```
const Set & Set::operator &= (const Set & B)
{   Set C;
   std::swap(C.A,A); std::swap(C.S,S); std::swap(C.n,n); //вместо Set C(*this);
   for (El * i = C.A; i; i = i->next)
      for (El * j = B.A; j; j = j->next)
         if (i->e == j->e)
            A = new El(i->e, A), ++n;
   }
   return *this;
}
Set Set::operator & (const Set& B) const
{   Set C(*this);   return (C &= B);  }
```

Класс-указатель
Для обычных указателей эквивалентны *p->m* == (**p*).*m* == *p*[0].m
Для пользовательских объектов-указателей это нужно обеспечить:

```
template < class Y >
class X { //Пример класса-указателя с полным набором операций с ним
   Y* p;
public:
```

```
  Y* operator -> ( ) { return p; }
  Y& operator *  ( ) { return *p; }
  Y& operator [ ] (int i) { return p[ i ]; }
  Y* operator ++ ( )
  Y* operator ++ (int)
};
```

Особый случай: класс для множества на основе списка.
Совместное использование двух классов-контроллеров:
— элемент списка;
— список в целом.
Вариант реализации:

```
class El{
  char e;
  El *next;
  El(char e, El *n = nullptr): e(e), next(n) { }
  ~El(){delete next;}
  El(const El &) = delete;
  El & operator=(const El &) = delete;
  El(El &&) = delete;
  El & operator=(El &&) = delete;
friend class Set;
};
class Set
{
private:
  static const int N; //Мощность универсума
  int n;   //Мощность множества
  char S;  //Тег
  El *A;   //Список элементов
public:
  Set();   //Пустое множество
  Set(char); //Случайное произвольной мощности (аргумент игнорируется)
  Set(const Set &);
  Set(Set &&);
  Set & operator = (const Set&);
  Set & operator = (Set &&);
  ~Set() { delete A; }
  void Show();
  int power() { return n; }
//…
};
```

Альтернативный вариант:

```
class Set
{
private:
  static const int N; //Мощность универсума
struct El{
  char e;
  El *next;
  El(char e, El *n = nullptr): e(e), next(n) { }
  ~El(){delete next;}
 };
  int n;   //Мощность множества
  char S;  //Тег
  El *A;   //Список элементов
public:
```

```
  Set();   //Пустое множество
//…
};
```

Парадоксальный вариант: в объявлении класса *Set* убираем звёздочку в объявлении поля *A*.

```
class Set
{
private:
  int n;   //Мощность множества
  El A; //Список элементов = полноценный тип
public:
  Set();   //Пустое множество
  Set(char); //Случайное произвольной мощности (аргумент игнорируется)
// Set(const Set &);
// Set(Set &&);
// Set & operator = (const Set&);
// Set & operator = (Set &&);
// ~Set() { delete A; }
  void Show();
  int power() { return n; }
//…
};
```

Список в таком варианте всегда имеет один элемент — головной, который содержится в *Set*; его поле для символа можно использовать под тег, а нулевое значения указателя *next* означает пустое множество.

Класс *Set* превратился в класс-значение, и ему нужны только конструкторы, а класс *El* — в полноценный класс-контроллер, и ему нужен полный набор служебных функций.

Эти функции автоматически используются при копировании, присваивании, уничтожении объекта типа *Set* без дополнительных указаний от программиста. Иногда (в сложных, сомнительных случаях) пишут:

```
  Set(const Set &) = default;
  Set(Set &&) = default;
//…
```

Существо изменения: отношение дружбы между классами *EL* и *Set* заменено отношением «содержит».

9. Перехват управления памятью: перегрузка new и delete

Специфика работы со списком: необходимость поэлементного заказа/освобождения памяти. Универсальная операция == дорогая!!!

Альтернатива: один раз выделить память под списки и самому управлять ею на уровне элементов. Способ: перегрузка *new* и *delete* для элемента списка.

Перегруженные *new* и *delete* — всегда статические функции-члены.

Пример программы для эксперимента с отслеживанием времени жизни объектов при работе с множествами-списками

1. Объявление класса «элемент списка» с перехватом управления свободной памятью (перегрузка операций new и delete)

```
class El{
  char e;
  El *next;
  static const int maxmup = 300;
  static El mem[maxmup]; //Свободная память для элементов списков
  static int mup, mup0;
public:
  El(): e('!'), next(nullptr){ }
  El(char e, El *n = nullptr): e(e), next(n) { std::cout << "+" << e; }
  ~El(){ if(this) { //Проверка обязательна
```

```
    if(next) { delete next;}
    std::cout << "-" << e;
    e = '*'; }
  else cout << "<Пусто!>";
  }
  static void* operator new(size_t) {
     return (mup < maxmup? &mem[mup++] : nullptr); }
  static void operator delete(void *, size_t) { }
  static void mark(){ mup0 = mup;}  //Фиксировать состояние памяти
  static void release() { mup = mup0; }  //Сбросить до фиксированного
  static void clear(){ mup = 0; }   //Очистить память полностью
  friend class Set;
  friend std::ostream & operator << ( std::ostream & o, El & S);
  friend static void memOut();
};
std::ostream & operator << (std::ostream & o, El & S)
{
  for (El* p = &S; p; p = p->next) o << p->e;
  return o;
}

El El::mem[El::maxmup];  //"Свободная память"
int El::mup = 0, El::mup0 = 0;

void memOut( )  //Вывод использованного содержимого "свободной памяти"
{
  std::cout << "\n\nПамять элементов списков (всего - " << std::dec << El::mup << ")\n";
  for(int i = 0; i < El::mup; ++i) cout << El::mem[i].e;
}

2. Объявление класса «множество-список» с автоматической нумерацией вновь создаваемых множеств
class Set
{
private:
 const int N = 26; //Мощность универсума
  static int num;      //Порядковый номер множества
  int n;   //Мощность множества
  char S;  //Тег
  El *A;   //Список элементов

public:
  Set();   //Пустое множество
  Set(char); //Случайное произвольной мощности (аргумент игнорируется)
  Set(const Set &);
  Set(Set &&);
  Set & operator = (const Set&);
  Set & operator = (Set &&);
  ~Set() {
    std::cout << "Удалено " << S <<"(" << std::dec << n << ") = [" << *A << "]";
    A->El::~El(); cout << std::endl; }   //Здесь нужен явный вызов деструктора

  void Show();
  int power() { return n; }
  void swap(Set & other) { std::swap(S, other.S); std::swap(n, other.n); std::swap(A, other.A);}
  Set & operator |= (const Set&);
  Set & operator &= (const Set&);
```

```
  Set operator | (const Set&) const;
  Set operator & (const Set&) const;
  Set operator ~ () const;
};
```

3. Определение функций-членов класса set с отслеживанием появления
и уничтожения множеств и главная программа

```
Set::Set() : n(0), S('A'+num++), A(nullptr)
{ std::cout << "-> Создано " << S << "(" << n << ") = [" << *A << "] \n";}

Set::Set(char) : S('A'+num++), n(0)
{
  A = nullptr;
  for (int i = 0; i < N; i++)
    if (rand() % 2)
      A = new El(i + 'A', A), ++n;
  std::cout << "-> Создано " << S << "(" << n << ") = [" << *A << "] \n";
}

Set::Set(const Set & B) : n(B.n), S('A'+num++), A(nullptr)
{
  for(El * p = B.A; p; p = p->next) A = new El(p->e, A);
  std::cout << "-> Создано " << S << "(" << n << ") = [" << *A << "] из " << B.S << std::endl;
}

Set::Set( Set && B) : n(B.n), S('A'+num++), A(B.A)
{
  B.A = nullptr;
  std::cout << "-> ПРИНЯТО " << S << "(" << n << ") = [" << *A << "] из " << B.S << std::endl;
}

Set & Set::operator &= (const Set& B)
{
  Set C;
  for (El * i = A; i; i = i->next)
  {
    for (El * j = B.A; j; j = j->next)
      if (i->e == j->e)
        C.A = new El(i->e, C.A), ++C.n;
  }
  swap(C);
  std::cout << "; Получено " << S << "(" << n << ") = [" << *A << "] = " << C.S << "&" << B.S << std::endl;
  return *this;
}
Set Set::operator & (const Set& B) const
{
  Set C(*this);
  std::cout << "Вычисление " << C.S << " & " << B.S << std::endl;
  return C&=B;
}

Set & Set::operator |= (const Set & B)
{
  Set C(*this);
  for (El * i = B.A; i; i = i->next)
```

```
  {
    bool f = true;
    for (El * j = A; f && j; j = j->next)
      f = f && (i->e != j->e);
    if (f)
      C.A = new El(i->e, C.A), ++C.n;
  }
  swap(C);
  std::cout << "; Получено " << S << "(" << n << ") = [" << *A << "] = " << C.S << "|" << B.S << std::endl;
  return *this;
}
Set Set::operator | (const Set& B) const
{
  Set C(*this);
  std::cout << "Вычисление " << C.S << " | " << B.S << std::endl;
  return C|=B;
}
Set Set::operator ~ ( )const
{
  Set C;
  for (char c = 'A'; c <= 'Z'; ++c)
  {
    bool f = true;
    for (El * j = A; j && f; j = j->next)
      if(c == j->e) f = false;
    if (f)
        C.A = new El(c, C.A) , ++C.n;
    }
  std::cout << "; Получено " << C.S << "(" << C.n << ") = [" << *C.A << "] = ~" << S << std::endl;
  return C;
}

Set& Set::operator = (const Set & B)
{
  if (this != &B)
  {  std::cout << "\nУдалено " << S << "(" << n << ") = [" << *A << "]";
    delete A;
    A = nullptr;
    n = 0;
    for(El * p = B.A; p; p = p->next)
      A = new El(p->e, A), ++n;
     S = 'A'+num++;
  }
  std::cout << "; Создано " << S << "(" << n << ") = [" << *A << "] из " << B.S << std::endl;
  return *this;
}

Set& Set::operator = (Set && B)
{  std::cout << "\nУдалено " << S << "(" << n << ") = [" << *A << "]";
  swap(B);
  delete B.A; B.A = nullptr;
  S = 'A'+num++;
  std::cout << "; ПЕРЕДАНО " << S << "(" << n << ") = [" << *A << "] из " << B.S << std::endl;
  return *this;
}
```

```
void Set::Show( )
{
  std::cout << '\n' << S << "(" << hex << A << ") = [ ";
  for(El * p = A; p; p = p->next) std:: cout << p->e << " ";
  std::cout << "]\n\n";
}

const int Set :: N = 26;
int Set :: num = 0;

int main()
{
  {
  setlocale(0, "");
  El::release();
  Set A('A'), B('B'), C('C'), D('D');
  cout << "\nРезультат:\n";
  (Set(A) | Set(B) & Set(C) & ~Set(D)).Show( );
  cout << "\n=== pause ===\n";
  cin.get( );
  }
  memOut( );
  cout << "\nВсё!\n=== pause ===\n";
  cin.get();
  return 0;
}
```

4. Результат эксперимента

Можно отследить моменты появления и удаления как отдельных элементов, так и каждого множества в целом, в том числе и работу конструктора копии, копии с переносом, присваивания и комбинированных операций.

```
+A+B+E+K+L+M+N+O+P+Q+S+U+X-> Создано A(13) = [XUSQPONMLKEBA]
+A+D+E+G+I+K+L+M+O+P+R+S+U+V+W+Y-> Создано B(16) = [YWVUSRPOMLKIGEDA]
+B+C+D+E+F+G+J+T+V+Z-> Создано C(10) = [ZVTJGFEDCB]
+A+C+D+K+N+P+Q+U+V+W+X+Y-> Создано D(12) = [YXWVUQPNKDCA]

Результат:
+Y+X+W+V+U+Q+P+N+K+D+C+A-> Создано E(12) = [ACDKNPQUVWXY] из D
-> Создано F(0) = [ ]
+B+E+F+G+H+I+J+L+M+O+R+S+T+Z; Получено F(14) = [ZTSROMLJIHGFEB] = ~E
-> ПРИНЯТО G(14) = [ZTSROMLJIHGFEB] из F
Удалено F(14) = []<Пусто!>
+Z+V+T+J+G+F+E+D+C+B-> Создано H(10) = [BCDEFGJTVZ] из C
+Y+W+V+U+S+R+P+O+M+L+K+I+G+E+D+A-> Создано I(16) = [ADEGIKLMOPRSUVWY] из B
+A+D+E+G+I+K+L+M+O+P+R+S+U+V+W+Y-> Создано J(16) = [YWVUSRPOMLKIGEDA] из I
Вычисление J & H
-> Создано K(0) = []
+V+G+E+D; Получено K(4) = [DEGV] = J&H
Удалено J(16) = [YWVUSRPOMLKIGEDA]-A-D-E-G-I-K-L-M-O-P-R-S-U-V-W-Y
+D+E+G+V-> Создано L(4) = [VGED] из K
Удалено K(4) = [DEGV]-V-G-E-D
+V+G+E+D-> Создано M(4) = [DEGV] из L
Вычисление M & G
-> Создано N(0) = [ ]
+E+G; Получено N(2) = [GE] = M&G
Удалено M(4) = [DEGV]-V-G-E-D
```

+G+E-> Создано O(2) = [EG] из N
Удалено N(2) = [GE]-E-G
+X+U+S+Q+P+O+N+M+L+K+E+B+A-> Создано P(13) = [ABEKLMNOPQSUX] из A
+A+B+E+K+L+M+N+O+P+Q+S+U+X-> Создано Q(13) = [XUSQPONMLKEBA] из P
Вычисление Q | O
+X+U+S+Q+P+O+N+M+L+K+E+B+A-> Создано R(13) = [ABEKLMNOPQSUX] из Q
+G; Получено R(14) = [GABEKLMNOPQSUX] = Q|O
Удалено Q(13) = [XUSQPONMLKEBA]-A-B-E-K-L-M-N-O-P-Q-S-U-X
+G+A+B+E+K+L+M+N+O+P+Q+S+U+X-> Создано S(14) = [XUSQPONMLKEBAG] из R
Удалено R(14) = [GABEKLMNOPQSUX]-X-U-S-Q-P-O-N-M-L-K-E-B-A-G

S(00182730) = [X U S Q P O N M L K E B A G]

Удалено S(14) = [XUSQPONMLKEBAG]-G-A-B-E-K-L-M-N-O-P-Q-S-U-X
Удалено P(13) = [ABEKLMNOPQSUX]-X-U-S-Q-P-O-N-M-L-K-E-B-A
Удалено O(2) = [EG]-G-E
Удалено L(4) = [VGED]-D-E-G-V
Удалено I(16) = [ADEGIKLMOPRSUVWY]-Y-W-V-U-S-R-P-O-M-L-K-I-G-E-D-A
Удалено H(10) = [BCDEFGJTVZ]-Z-V-T-J-G-F-E-D-C-B
Удалено G(14) = [ZTSROMLJIHGFEB]-B-E-F-G-H-I-J-L-M-O-R-S-T-Z
Удалено E(12) = [ACDKNPQUVWXY]-Y-X-W-V-U-Q-P-N-K-D-C-A

=== pause ===
Удалено D(12) = [YXWVUQPNKDCA]-A-C-D-K-N-P-Q-U-V-W-X-Y
Удалено C(10) = [ZVTJGFEDCB]-B-C-D-E-F-G-J-T-V-Z
Удалено B(16) = [YWVUSRPOMLKIGEDA]-A-D-E-G-I-K-L-M-O-P-R-S-U-V-W-Y
Удалено A(13) = [XUSQPONMLKEBA]-A-B-E-K-L-M-N-O-P-Q-S-U-X

Память элементов списков (всего - 189)
****
Всё!
=== pause ===
-!

Контрольные вопросы.

1. Каким способом можно перегрузить операцию «|» для множеств — использовать функцию-член, функцию-друг, обычную функцию?
2. Каким способом можно перегрузить операцию «=»?
3. Каким способом можно перегрузить операцию «[ ]»?
4. Как различить перегрузку префиксного и постфиксного инкрементов «++»?
5. Можно ли при перегруженных «|» и «=» применять комбинированную операцию «|=»?
6. Какая сигнатура является корректной для двуместной операции объединения множеств?
7. Какая сигнатура является корректной для операции комбинированного присваивания — объединения множеств?
8. Какая сигнатура является корректной для перемещающего конструктора?
9. Можно ли ускорить выполнение оператора «*A* = *B* | *C*» с перегруженными операциями «=» и «|»?
10. Может ли функция-член для перегрузки операции возвращать ссылку на объект?
11. В каких случаях нужен перемещающий конструктор?
12. Как можно потребовать, чтобы следующая функция возвращала результат, используя перемещающий конструктор?

```
Set Set::And(const Set & rgt) const
{ Set Result;
  for(int i = 0; i < n; ++i)
    for(int j = 0; j < rgt.n ++j) if(A[i] == rgt.A[i])
         Result[Result.n++] = A[i];
  return Result;
}
```

13. Каким способом объект-множество лучше всего передавать в качестве аргумента в функцию?
14. В каком случае функция-член должна возвращать ссылку на объект?

15. Объявлен массив целых: *int A*[100]{0};
Какие элементы массива инициализируются нулём?

10. Понятие о шаблонах. Стандартная библиотека шаблонов

ШАБЛОНЫ — обобщённое программирование.
Алгоритм пригоден для разных (похожих) типов данных — написать шаблон.
Пример. Функция, меняющая местами содержимое своих аргументов.

```
template <class T>
void swap(T a, T b) { T c(a); a = b; b = c; }
```

Функция пригодна для любого типа T, для которого определены конструктор копии и перегрузка присваивания.
Важно:
— объявление шаблона само по себе ничего не добавляет в код. Определение функции появится в коде только при использовании шаблона — своё для каждого использованного типа данных.
Пример.

```
int a=0, b=1; double f=0, p=3.14; Set A(«A»), B(«B»);
swap<int>(a,b); swap(f,p); swap(A,B);
```

Здесь будут сгенерированы три варианта кода.

ШАБЛОННЫЙ КЛАСС — шаблонные все его функции-члены.
Пример: шаблонный класс «множество в массиве».

```
  template <class T = char, unsigned N = 26, char s0 = 'A'>
class set1{
  int n;
  T *A;
public:
  set1( ) : n(0), A(new T[N+1]) { }
  set1(int);
};
  template <class T = char, unsigned N = 26, char s0 = 'A'>
set1 :: set1(int k) : set1( )
{ for (int i = 0; i < N; ++i) if(rand( )%2) A[n++] = s0 + i;}
```

Аргументы шаблона — имена, указанные как *class* или *typename*, а также переменные (целые, указатели, ссылки) *const*, *constexpr* — вычисляемые на этапе компиляции.
Аргументы шаблона похожи на аргументы функции. Шаблон порождает программный код только по факту использования (инстанцирование шаблона).
При использовании шаблонного класса фактические аргументы шаблона обязательны (но могут использоваться значения по умолчанию).
Для шаблонных функций подстановка аргументов шаблона часто происходит автоматически (определяется из вызова функции).
Библиотека шаблонов — это набор исходных текстов (h-файлы), вставляемых в программу пользователя по мере надобности.
СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ
Для популярных типов данных библиотека содержит определения КОНТЕЙНЕРОВ — классов, специфицируемых пользовательским типом данных, и шаблоны функций, реализующих популярные алгоритмы.
По отношению к пользовательскому классу контейнер находится в отношении «содержит». Пользовательский класс должен иметь набор служебных функций, удовлетворяющий требованиям контейнера.
Концепция Стандартной библиотеки шаблонов:
КОНТЕЙНЕР (итератор)→ АЛГОРИТМ →(итератор) КОНТЕЙНЕР
Контейнеры используются как источник данных и хранилище для результатов вычислений. Для связи контейнеров с алгоритмами используются особые интерфейсные объекты — итераторы. Это вспомогательные классы, обобщающие понятие

указателя.

Пример: работа с календарными датами: чтение из файла, корректировка, запись результата.

```
class Date{ //Класс «календарная дата» с перегруженным вводом/выводом
    //…
   public:
     istream& operator >>( istream& Date&);
     ostream& operator <<(ostream&, Date&);
   };
int main( )
    { vector <Date> e; //Рабочая память
      copy(istream_iterator<Date>(cin),
              istream_iterator<Date>( ), back_inserter(e)); //Чтение набора дат
      vector<Date>::iterator f = find(e.begin( ), e.end( ), «16.01.16»);
      if(f != e.end( )) *f = actual_date( ); //Замена искомой даты на текущую
      sort(e.begin(), e.end()); //Упорядочение
      unique_copy(e.begin(), e.end(), ostream_iterator<Date>(cout, «\n»));
      //Вывод результата с исключением дубликатов
   }
```

В примере — единообразный подход к работе с последовательностями в памяти, во входном и выходном файлах. Алгоритм *copy* читает данные из стандартного потока с помощью итератора ввода и помещает их в контейнер *vector*, используя итератор вставки. Далее в векторе алгоритмом *find* ищется заданная дата. Используется прямой итератор. Возвращается значение итератора, указывающее на искомую дату или на конец данных. Если дата найдена, в месте, указанном итератором, происходит её замена на актульную.
Далее с помощью алгоритма *sort* даты упорядочиваются, а алгоритмом *unique_copy* копируются в выходной поток. В последнем случае используется итератор вывода.

Библиотека STL — основные элементы:
— **контейнеры**;
— аллокаторы;
— **итераторы**: входные, выходные, прямые, двунаправленные, произвольного доступа; потоковые.
— **функторы** (plus, minus, multiplies, divides, modulus, negate) <functional>
— предикаты (less, equal_to, greather_equal, logical_and/or/not)
— **алгоритмы** (includes, set_intersection, set_difference, set_symmetric_difference, set_union, make_heap, pop_heap, push_heap, sort_heap, for_each, transform, unique, copy, fill, generate, reverse, rotate, random_shuffle, binary_search, merge, sort, stable_sort)

11. Контейнерные классы

Последовательные контейнеры: *vector*, *list*, *deque*, *array*, *forward_list*
Адаптеры *stack, queue, prioriry_queue*.
Ассоциативные контейнеры (множество и отображение): *map, multimap, set, multiset, unordered_set, unordered_multiset, unordered_map, unordered_multimap*.
Последовательные контейнеры и адаптеры впредь рекомендуются для работы с последовательностями, ассоциативные — откладываются до весны.
Объявление контейнера (на примере контейнера *vector*):

```
template <class T, class Allocator = allocator<T>>
class vector {…};
```

Обобщение для массива. Отличие: знает свой размер и никогда (при правильном использовании) не переполняется. При объявлении по умолчанию — пуст, заполняется в процессе работы. Является памятью прямого доступа, т. е. поддерживает индексирование. Есть функция at(int) — индексирование с проверкой корректности индекса.

```
vector <char> A; //Объявление пустого вектора (расширение — с конца).
vector <char> A(N, 0); //Вектор из N элементов, заполненный нулями.
```

Использование последнего вектора — такое же, как и обычного массива из *N* элементов.

В процесс выделения памяти можно вмешаться:
A.resize(N, 0); //Изменить размер на указанный, заполнить нулями.
A.reserve(N); //Резервировать память не менее чем под *N* элементов.
A.capacity( ); //Проверить текущий запас памяти.
A.shrink_to_fit( ); //Сбросить лишнюю память (C++11).
Аргумент шаблона *Allocator* — подключение своего альтернативного средства работы с памятью.
Если не нужна свободная память — упрощённый вариант (C++11):
array<char>(N, 0);
Это неперемещаемый массив с фиксированным размером, который должен быть указан.
Специальная альтернатива для массива символов — *string, wstring*.
Контейнер ***list*** — двунаправленный список. Обеспечивает константную вставку в начало и в середину, обмен частями между несколькими списками.
forward_list — упрощённая однонаправленная версия списка (C++11).
deque — гибрид массива и списка: список из экстентов. Допускает константную вставку в начало и индексацию. База для очереди и стека.
ПОЛЯ КОНТЕЙНЕРОВ (стандарт для использования в программах):
value_type — тип элемента
key_type — тип ключа (для ассоциативных)
key_compare — тип критерия сравнения (для ассоциативных)
size_type — тип индекса, счётчика и т. п.
iterator — итератор (указатель на элемент)
const_iterator — константный итератор (указатель на константу)
reverse_iterator — обратный итератор
const_reverse_iterator — константный обратный =
reference — ссылка на элемент
const_reference — константная ссылка на элемент
ФУНКЦИИ ДЛЯ ПРОСМОТРА (установка на начало/конец)
iterator begin ( ); //установка на первый элемент
const_iterator cbegin ( ) *const*;
iterator end ( ); // — на следующий за последним
const_iterator cend ( ) *const*;
reverse_iterator rbegin ( ); //установка на первый с конца
const_reverse_iterator rcbegin ( ) *const*;
reverse_iterator rend ( ); // — на следующий за последним с конца
const_reverse_iterator rcend ( ) *const*;
Пара соответствующих функций задаёт полуоткрытый интервал, в котором находятся элементы контейнера (начало и конец для параметра цикла: *for* (*auto x = A.begin*( ); *x* != *A.end*( ); ++*x*) *f*(**x*); //Обработать всё.
Альтернатива (C++11): *for*(*auto x* : *A*) *f*(*x*); //Здесь *x* — элемент контейнера
Сведения о размере:
size_type size ( ); — количество элементов в контейнере;
size_type max_size ( ); — максимально возможное количество;
bool empty ( ); — проверка «контейнер пуст».

ИТЕРАТОРЫ: сущность и классификация
Итератор — это класс-указатель, т. е. класс, для которого определены: разыменование прямое * и косвенное ->, инкремент/декремент ++ и - -, возможно, индексация [ ].
Конкретный набор операций связан с типом (областью применения) итератора:
— входные: только чтение, однопроходный: * и ++;
ВАРИАНТ: прямые, обратные;
— выходные: однократный проход для записи (только присваивание =);
ВАРИАНТ: итераторы вставки — в начало, в конец, универсальный (используют push_front( ), push_back( ), insert( ));
— двунаправленные: инкремент и декремент;
— произвольного доступа: индексация;
— потоковые (входные, выходные).
Применимость к контейнерам — для каждого указывается отдельно!
Template <class T> //Итератор = объект-указатель

```
Class SmartPtr
{ public:
    SmartPtr (T * realptr = nullptr);
    SmartPtr(const SmartPtr & Rhs);
    ~SmartPtr( );
    T& operator *( ) const;     //Разыменование прямое
    T* operator->( ) const;     //Разыменование косвенное
    T& operator[ ](int) const; //Прямой доступ (индексирование)
  private:
     T* pointee; //Реальный указатель
  };
```

Последовательные контейнеры — вставки (применимость)

Операция	функция	vector	deque	list
Вставка в начало	push_front	нет	да	да
Удаление из начала	pop_front	нет	да	да
Вставка в конец	push_back	да	да	да
Удаление с конца	pop_back	да	да	да
Вставка внутрь	insert	(да)	(да)	да
Удаление изнутри	erase	(да)	(да)	да
Произвольный доступ	[ ], at	да	да	нет

Для работы с последовательностью: просмотр, копирование и т. д. все контейнеры эквивалентны, поэтому рекомендуется самый экономичный из них — *vector*. Последний обеспечивает также прямой доступ к данным, что позволяет обрабатывать их в произвольном порядке. Но если предполагается добавлять или удалять данные из начала или середины — вне конкуренции *list*, выполняющий такие операции за константное время. Но произвольного доступа он не обеспечивает, и поиск данных в нём возможен только просмотром с начала. Контейнер *deque* — компромиссный вариант, обеспечивающий как эффективную вставку/удаление в начале, так и прямой доступ к данным.

НЕСТАНДАРТНЫЕ КОНТЕЙНЕРЫ (не подчиняющиеся общим правилам)
— *vector<bool>* — упакован по битам и нет доступа к биту через итератор (аналог массива битов);
— *bitset* — аналог машинного слова, любое к-во битов;
— *string* — оболочка для строки, вариант: *wstring* — для 16-битовых символов. Нестандартность — полный набор своих функций-членов, позволяющий обойтись без средств из библиотеки алгоритмов.
vector<bool> — поддерживает произвольное к-во битов, а *bitset* — фиксированное (по аналогии с *array*). Наборы операций похожи: инвертирование одного бита или всех битов контейнера, проверка/установка/копирование бита по номеру: *c.flip*( ), *c*[*pos*].*flip*( ), *c*[*pos*] = *val*, *c*[*pos*1] = *c*[*pos*2].
Контейнер *bitset* имеет более богатый набор функций: проверки *test*(*pos*), *all*, *any*, *none*, *count*, установки *set*, *reset*, *flip*, поразрядные операции &, &=, |, |=, ^, ^=, ~, сдвиги ≪=, ≫=, ввод и вывод ≪, ≫, преобразования to_string, to_ulong, to_ullong.

Сводка средств для работы с контейнером *string*/*wstring*

— *конструкторы*:

```
string( ); //умолчание (создаётся пустая строка)
    string(const char *); //из строки в стиле Си
    string(const char *, int n); //то же, с указанием размера
    string(string&); //копирование
```

— *присваивания*:

```
string& operator = (const string&);
    string& operator = (const char*);
    string& operator = (char);
```

Строка в стиле Си преобразуется в *string* по умолчанию;
обратное преобразование — только явно, функцией *c_str*( ).
— *операции над строками:*
= (присваивание), + (конкатенация), +=, ==, !=, <, ≤, >, ≥, [ ], ≪, ≫.
Размеры строк устанавливаются автоматически так, чтобы результат помещался.
Адрес нулевого элемента строки НЕ совпадает с указателем на строку.

S != &*S*[0].

Допустимость индекса в операции *S*[i] не проверяется. Для индексации с проверкой служит функция *S.at*(*i*), порождающая исключение *out_of_range*.

— *получение характеристик строк*:

size_type size ( ) *const*;

size_type length ( ) *const*;

size_type max_size ( ) *const*;

size_type capacity ( ) *const*;

bool empty ( ) *const*;

Функции для строк:

— присваивание подстроки

assign (*const string*& *str*); //=

assign (*const string*& *str*, *size_type pos*, *size_ type n*);

assign (*const char* * *s*, *size_ type n*);

— добавление подстроки

append (*const string*& *str*); //то же, что и операция «+»

append (*const string*& *str*, *size_type pos*, *size_ type n*);

append (*const char* * *s*, *size_ type n*);

— вставка подстроки

insert (*size_type pos*1, *const string*& *str*); //вставка целиком

insert (*size_type pos*1, *const string*& *str*, *size_type pos*2, *size_type n*); //вставка части

insert (*size_type pos*1, *const char* * *s*, *size_type n*);

— удаление подстроки

erase (*size_type pos* = 0, *size_type n* = *npos*);

//здесь и далее: *npos* — стат.член класса string, содержит максимальное значение *size_type*.

— очистка строки

void clear ( );

— замена подстроки

replace (*size_type pos*1, *size_type n*1, *const string*& *str*);

replace (*size_type pos*1, *size_type n*1, *const string*& *str*,

*size_type pos*2, *size_type n*2);

replace (*size_type pos*1, *size_type n*1, *const char* * *s*, *size_type n*2);

— обмен строк

swap (*string* & *str*);

— выделение подстроки

string substr (*size_type pos* = 0, *size_type n* = *npos*);

— преобразование в строку в стиле Си

const char * *c_str* ( ) *const*;

const char * *data* ( ) *const*; //не добавляет нуль

— копирование в массив

size_type copy (*char* * *s*, *size_type n*, *size_type pos* = 0);

//не добавляет нуль

— поиск подстрок

size_type find (*const string* & *str*, *size_type pos* = 0) *const*;

size_type find (*char c*, *size_type pos* = 0) *const*;

size_type rfind (*const string* & *str*, *size_type pos* = *npos*) *const*;

size_type rfind (*char c*, *size_type pos* = *npos*) *const*;

12. Вспомогательные средства STL

Пары и кортежи

template <*typename T*1, *typename T*2>

struct pair { *T*1 *first*; *T*2 *second*; }

Операции над парами:

pair <*T*1, *T*2> *p*;

pair <*T*1, *T*2> *p* (*val*1, *val*2);

pair <*T*1, *T*2> *p* (*rv*1, *rv*2);

pair <*T*1, *T*2> *p*(*p*2);

pair <*T*1, *T*2> *p*(*rv*);

Возможны копирование, перенос, сравнение, обмен;

make_pair(*val*1, *val*2); — создание пары из двух элементов в соответствии с их типами.

Доступ к элементам: *p.first, p,second*.

Вариант: *get*<0>(*p*), *get*<1>(*p*) — как для кортежей (*turple*).

Функциональные объекты:

— указатели на функцию;

— функторы.

Функтор — объект класса с перегруженной операцией ( ).

Пример.

```
class Linear
    { private:
            double slope, y0;
      public:
            Linear (double s1 = 1, double y = 0) : slope(s1), y0(y) { }
            double operator( ) (double x) { return y0 + slope * x; }
     };
```

Использование:

```
Linear f1, f2(2.5, 10.0);
      double y1 = f1(12.5), y2 = f2(0.4);
```

Пример 2.

```
// functor.cpp – использование функторов
     #include <iostream>
     #include <list>
#include <iterator>
    #include <algorithm>
    template<class T>  // класс-функтор определяет operator( )( )
    class TooBig
   {
     private:
        T cutoff; //Память функтора
     public:
        TooBig(const T & t) : cutoff(t) { }
        bool operator( )(const T & v) { return v > cutoff; }
    };
void outint(int n) {std::cout << n << " ";}
int main( )
    {
       using std::list;
       using std::cout;
       using std::endl;
       using std::for_each;
       using std::remove_if;
    TooBig<int> f100(100); // limit = 100
         int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
         list<int> L1(vals, vals + 10); // конструктор из диапазона
         list<int> L2(vals, vals + 10);
// C++11 разрешает инициализировать непосредственно:
//  list<int> L1 = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
//  list<int> L2 {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
   cout << "Оригиналы:\n";
```

```
    for_each(L1.begin( ), L1.end( ), outint);
    cout << endl;
    for_each(L2.begin(), L2.end(), outint);
    cout << endl;
    L1.remove_if(f100);      // исп. поименованный функциональный объект
    L2.remove_if(TooBig<int>(200));   // сконструированный функц. объект
    cout <<"Усечённые:\n";
    for_each(L1.begin( ), L1.end( ), outint);
    cout << endl;
    for_each(L2.begin( ), L2.end( ), outint);
    cout << endl;
    std::cin.get( );
    return 0;
  }
```

Концепции функторов:

— **генератор** (без аргументов)

— **унарная функция** (один аргумент)

— **бинарная функция** (два аргумента)

— предикат (один аргумент, возврат — bool)

— бинарный предикат (то же, два аргумента)

Примеры из *STL*:

plus, minus, multiplies, divides, modulus, negate — функторы;

equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or, logical_not — бинарные предикаты.

Функтор может применяться как АДАПТЕР функции с неподходящим количеством аргументов.

— **лямбда**

В стандартном C++, например, при использовании алгоритмов стандартной библиотеки C++ *sort* и *find*, часто возникает потребность в определении функций-предикатов рядом с местом, где осуществляется вызов алгоритма. В языке существует только один механизм для этого: возможность определить класс функтора (передача экземпляра класса, определённого внутри функции, в алгоритмы запрещена (*Meyers*, *Effective STL*)). Зачастую данный способ является слишком избыточным и многословным и лишь затрудняет чтение кода. Кроме того, стандартные правила C++ для классов, определённых в функциях, не позволяют использовать их в шаблонах и таким образом делают их применение невозможным.

Очевидным решением проблемы явилось разрешение определения лямбда-выражений и лямбда-функций в C++11. Лямбда-функция определяется следующим образом:

```
[ ](int x, int y) { return x + y; }
```

Тип возвращаемого значения этой безымянной функции вычисляется как *decltype*(*x*+*y*). Тип возвращаемого значения может быть опущен только в том случае, если лямбда-функция представлена в форме *return expression*. Это ограничивает размер лямбда-функции одним выражением.

Тип возвращаемого значения может быть указан явно, например:

```
 [ ](int x, int y) -> int { int z = x + y; return z; }
```

В этом примере создаётся временная переменная *z* для хранения промежуточного значения. Как и в обычных функциях, это промежуточное значение не сохраняется между вызовами.

Тип возвращаемого значения может быть полностью опущен, если функция не возвращает значения (то есть тип возвращаемого значения — *void*)

Также возможно использование ссылок на переменные, определённые в той же области видимости, что и лямбда-функция. Набор таких переменных обычно называют замыканием. Замыкания определяются и используются следующим образом:

```
std::vector<int> someList;
    int total = 0;
    std::for_each(someList.begin(), someList.end(),
                  [&total](int x) {  total += x;  });
    std::cout << total;
```

Здесь вычисляется и выводится сумма всех элементов в списке. Переменная *total* хранится как часть замыкания лямбда-функции. Так как она ссылается на стековую переменную *total*, она может менять её значение.

Переменные замыкания для локальных переменных могут быть также определены без использования символа ссылки &. Это означает, что функция будет копировать их значения, что вынуждает пользователя заявлять о намерении сослаться на локальную переменную или скопировать её.

Для лямбда-функций, гарантированно исполняемых в области их видимости, возможно использование всех стековых

переменных без необходимости явных ссылок на них:

```
std::vector<int> someList;
     int total = 0;
     std::for_each(someList.begin( ), someList.end( ),
                          [&](int x) {  total += x; });
```

Способы внутренней реализации могут различаться, но предполагается, что лямбда-функция сохранит указатель на стек функции, в которой она создана, а не будет работать с отдельными ссылками на переменные стека.
Если вместо [&] используется [=], все используемые переменные будут скопированы, что позволяет использовать лямбда-функцию вне области действия исходных переменных.
Способ передачи по умолчанию можно также дополнить списком отдельных переменных. Например, если необходимо передать большинство переменных по ссылке, а одну по значению, можно использовать следующую конструкцию:

```
int total = 0;
    int value = 5;
    [&, value](int x) { total += (x * value); }
                        //вызов лямбда-функции с передачей значения
```

Это вызовет передачу *total* по ссылке, а *value* — по значению.
Если лямбда-функция определена в функции-члене класса, она считается дружественной этому классу. Такие лямбда-функции могут использовать ссылку на объект типа класса и обращаться к его внутренним полям:

```
[ ](SomeType *typePtr) { typePtr->SomePrivateMemberFunction(); }
```

Это будет работать, только если областью создания лямбда-функции является функция-член класса SomeType.
Особым образом реализована работа с указателем *this* на объект, с которым взаимодействует текущая функция-член. Он должен быть явно обозначен в лямбда-функции:

```
[this]( ) { this->SomePrivateMemberFunction( ); }
```

Использование формы [&] или [=] лямбда-функции делает *this* доступным автоматически.
Тип лямбда-функций зависит от реализации; имя этого типа доступно только компилятору. Если необходимо передать лямбда-функцию в качестве параметра, она должна быть шаблонного типа, либо сохранена с использованием *std::function*. Ключевое слово *auto* позволяет локально сохранить лямбда-функцию:

```
auto myLambdaFunc = [this]( ) { this->SomePrivateMemberFunction(); };
```

Управление инициализацией объектов — класс *initializer_list*
Пример использования, поясняющий суть дела.

```
class Set{
   enum { N = 26};
   int n = 0;
   char A[N+1]; //Множество в массиве символов
public:
  Set(initializer_list<char> in)
     { for(auto p = in.begin( ); p != in.end( ); ++p) A[n++] = p; }
  //Вариант: for(auto x : in) A[n++] = x;
```

Использование при объявлении объекта:

```
  Set A{ 'a', 'b', 'd', 'r', 'x' };
   //Используется конструктор с соответствующим типом аргумента.
```

Для формального аргумента функции *initializer_list* в качестве фактического аргумента при вызове используется список инициализации, который функция просматривает как обычный последовательный контейнер.

13. Интеллектуальные указатели

Проблема работы с указателями в программах на Си/C++ состоит в том, что указатель живёт своей жизнью независимо от того, на что он указывает. Если значение указателя изменяется или он исчезает по выходе из области своего определения, ресурс, на который он указывал, может стать недоступен для управления и превратиться в неудаляемый мусор. Решить эту проблему пытались ещё в самом первом стандарте языка с помощью указателя *auto_ptr*, управляющего ресурсом, так, чтобы во всех перечисленных выше случаях изменения или исчезновения указателя ресурс тоже автоматически удалялся. Но сделать из *auto_ptr* надёжный инструмент не получилось, в первую очередь, из-за отсутствия в языке семантики

перемещения.
В стандарте C++17 *auto_ptr* исключён. Вместо него введено три варианта интеллектуального указателя:
unique_ptr
shared_ptr
week_ptr
Класс *unique_ptr* — концепция исключительного владения. Гарантирует, что объект и связанные с ним ресурсы принадлежат только одному указателю. Современный аналог *auto_ptr*.
ПРОБЛЕМА утечки ресурсов:

```
void f( )
     { ClassA * ptr = new ClassA;
  try {
      // Работаем с объектом, здесь возможно исключение
      }
       catch(…) {
           delete ptr; throw; //В случае сбоя освободить ресурс
      }
      delete ptr; //освобождаем ресурс (сбоя не было)
    }
```

АЛЬТЕРНАТИВА (используем интеллектуальный указатель):

```
#include <memory>
     void f( )
     { std::unique_ptr<ClassA> ptr(new ClassA); //=make_unique<ClassA>( );
     // Работаем с объектом
    } //delete и catch не нужны
```

НЕЛЬЗЯ инициализировать присваиванием:

```
    std::unique_ptr<ClassA> ptr = new ClassA;
```

По умолчанию создаётся пустой указатель.
Можно очистить (освободив ресурс!):

```
ptr = nullptr;
     ptr.reset( );
```

Можно освободить от владения, передав объект обычному указателю:

```
ClassA * sp = ptr.release( );
```

Проверка владения:

```
if ( ptr ) //→ указатель не пуст!
     if ( ptr != nullptr ) …
     if (ptr.get( ) != nullptr) …
```

Передача владения:

```
std:: string * sp = new string("Hello");   //Обычный указатель
     std::unique_ptr<std::string> up1(sp);   //Так можно
     std::unique_ptr<std::string> up2(sp);
        //ОШИБКА: один ресурс на два указателя. Этого следует избегать!
 std::unique_ptr<std::string> up2(up1);
           //Так НЕЛЬЗЯ. Ошибка компиляции
    std::unique_ptr<std::string> up2(std::move(up1));
          //ОК: передача владения через копирование
 up2 = up1;     //НЕЛЬЗЯ: ошибка компиляции
     up2 = std::move(up1);
```

//ОК: передача владения через присваивание; к объекту **up*2 применяется *delete*
Передача владения при вызове функций:

```
void sink(std::unique_ptr<ClassA> up)
```

//**Функция-сток**: получение и использование указателя; по выходе из функции ресурс освобождается

```
{ /*…*/ }
      std::unique_ptr<ClassA> up (new ClassA); //Создание ресурса
      //…
     sink(std::move(up)); //Вызов с передачей владения в функцию sink
```

```
    std::unique_ptr<ClassA> source ( ); //Источник данных через указатель
    { std::unique_ptr<ClassA> up (new ClassA);
            //Работаем с данными
      return up;  //std::move( ) по умолчанию!
    }
     void g( )
     { std::unique_ptr<ClassA> p;
       for ( int i = 0; i < 10; ++i)
        { p = source( );
//p получает владение возвращённым объектом. Предыдущий — удаляется
       //…
        }
    } //удаляется последний объект, которым владел p.
```

unique_ptr как член класса

ПРИМЕР проблемы утечки ресурсов

```
class ClassB { //Класс с двумя ресурсами — потенциальная проблема!
        private:
          ClassA * ptr1, *ptr2;
        public: //утечка памяти первого объекта в случае сбоя со вторым
          ClassB( int v1, int v2) : ptr1(new ClassA(v1)), ptr2(new ClassA(v2)) { }
          ClassB( const ClassB & x) : ptr1(new ClassA(*x.ptr1)),
                                      ptr2(new ClassA(*x.ptr2)) { }
           const ClassB & operator = ( const ClassB & x) {
              *ptr1 = *x.ptr1; *ptr2 = *x.ptr2;
               return *this;
           }
          ~ClassB ( ) { delete ptr2; delete ptr1; }
        }
```

РЕШЕНИЕ: применить *unique_ptr*

```
class ClassB {
      private:
        std:: unique_ptr<ClassA>  ptr1, ptr2;
      public: //утечка невозможна
         ClassB( int v1, int v2) : ptr1(new ClassA(v1)), ptr2(new ClassA(v2)) { }
         ClassB( const ClassB & x) : ptr1(new ClassA(*x.ptr1)),
                                     ptr2(new ClassA(*x.ptr2)) { }
         const ClassB & operator = ( const ClassB & x) {
             *ptr1 = *x.ptr1; *ptr2 = *x.ptr2; //копируются объекты
             return *this;     }
      // ~ClassB ( ) { delete ptr1; delete ptr2; }
      //деструктор не нужен, по умолчанию всё будет как надо
     };
```

unique_ptr — работа с массивами

По умолчанию для удаления применяется *delete*. Язык не различает указатели на скаляр и на массив.

```
std::unique_ptr<std::string> up(new std::string[10]);  //Ошибка времени выполнения (неправильная инициализация)
```

ПРАВИЛЬНО:

```
std::unique_ptr<std::string[ ]> up(new std::string[10]);
            //Указатель на массив!
```

Для доступа к объекту вместо * и -> следует применять [ ].

```
std::cout ≪ "Первая строка" ≪ *up ≪ std::endl;  //ОШИБКА, так нельзя
     std::cout ≪ "Первая строка" ≪ up[ 0 ] ≪ std::endl;  //ОК
```

Ответственность за корректность индекса — на программисте.

Нельзя инициализировать массив объектами производного типа (как и для обычных массивов).

Можно передавать в указатель собственные средства для удаления объекта.

По умолчанию: *class default_delete <T>* или *class default_delete <T[ ]>*.

```
template<typename T, typename D = default_delete <T>> //Первичный шаблон
```

```
class unique_ptr {
        public:
            T& operator* ( ) const;
            T* operator -> ( ) const noexcept;
    // …
    };
    template<typename T, typename D>
            //Частичная специализация для массивов
    class unique_ptr <T[ ], D>{
       public:
          T& operator [ ] ( ) const;
  // …
    };
```

Пример. Работа с множествами в массивах:

```
class set1{
       enum {N = 26}; //Ещё один способ определить константу
       int n;
       unique_ptr<char[ ]> A; //3а ресурс теперь отвечает указатель
     public:
        set1( ) : n(0), A(new char[N+1]) { }
    //…
    };
```

Класс *set*1 СОДЕРЖИТ ресурс. Деструктор не нужен.

Ещё об интеллектуальных указателях

Класс *unique_ptr* — дешёвое и эффективное средство контроля связанного с ним ресурса. Гарантирует, что объект и связанные с ним ресурсы принадлежат только одному указателю и живут ровно столько, сколько живёт сам указатель. Никаких накладных расходов. Объект *unique_ptr* — это просто оболочка над обычным указателем, в которой запрещён конструктор копии.

Более сложными являются варианты

shared_ptr — разделяемый указатель;

week_ptr — слабый указатель.

Разделяемый указатель можно копировать, получая группу указателей на один ресурс. Освобождение ресурса происходит, когда исчезает последний из указателей на него. Такое поведение обеспечивается блоком контроля, хранящим счётчик активных указателей на ресурс.

Слабый указатель необходим, чтобы удерживать блок контроля, не захватывая ресурс. Разделяемый и слабый указатели — средство для специфических применений, т. е. там, где это оправдано существом задачи.

14. Бинарное отношение на множестве. Графы

Произвольное бинарное отношение на множестве и граф как подходящая модель для этого.

В практических задачах с множествами, как правило, приходится учитывать и обрабатывать их связи. Для строгого математического описания любых связей между элементами двух множеств вводится понятие бинарного отношения. Бинарное отношение между множествами A и B называется подмножество R их прямого произведения $A \times B$. В случае $A = B$ мы говорим просто об отношении R на множестве A.

$$a\ R\ b = \{<a, b>: a \in A, b \in B\} \text{ или } a\ R\ b = \{<a, b>: a, b \in A\}$$

Подходящей моделью для бинарного отношения является граф.

Граф — это пара множеств $G = <V, E>$, где V — произвольное множество, а $E = \{\{u, v\}: u, v \in V, u \neq v\}$ — множество пар из элементов множества V. Если пара $\{u, v\}$ представляет собой множество мощностью 2, граф называется неориентированным, а если это последовательность $<u, v>$ — ориентированным. Будем обозначать мощность множества вершин $|V| = n$, а мощность множества рёбер $|E| = m$. Очевидно, что справедливо ограничение $m = O(n^2)$.

Вершины $\{u, v\}$, образующие ребро, называются ***смежными***, а само ребро — ***инцидентным*** по отношению к образующим его вершинам, а вершины, в свою очередь, ***инцидентны*** ребру. Количество рёбер, инцидентных вершине, называется её ***степенью***. Вершина, не входящая ни в одно ребро, имеет степень 0 и называется изолированной. В ориентированном графе

различают также количество рёбер, входящих в вершину — ***полустепень захода*** — и количество выходящих рёбер — ***полустепень выхода***.

Последовательность попарно смежных вершин образует ***путь*** в графе. ***Длина пути*** равна количеству входящих в него рёбер. Если в последовательности, образующей путь, все вершины различны, путь называется ***элементарным***. Путь, начало и конец которого совпадают, называется ***циклом***. Связный граф без циклов называется ***деревом***, несвязный — ***лесом***.

Если любая пара вершин графа связана путём, граф называется ***связным***. Если для любой пары вершин находятся, по крайней мере, два пути, множества вершин которых не пересекаются, граф — ***двусвязный***.

В связном ориентированном графе (орграф) путь между некоторыми вершинами может быть только в одну сторону. Если же любая пара вершин орграфа связана путями в обе стороны, такой граф называется ***сильно связным***.

Граф с пустым множеством вершин называется ***пустым***, а граф, в котором имеются все возможные рёбра, — ***полным*** графом или ***кликой***.

Граф, множество вершин которого можно разбить на два непустых непересекающихся подмножества таким образом, что концы любого ребра будут принадлежать разным подмножествам, называется ***двудольным***.

Если граф каким-либо из перечисленных свойств не обладает, можно ставить задачу отыскания ***компонент*** — максимальных подграфов, обладающих нужным свойством, например, компонент связности, двусвязности, максимальных клик и т. п.

Графы $G = < V, E >$ и $G' = < V', E' >$ называются ***изоморфными***, если существует биекция $f: V \to V'$ такая, что для любой пары вершин $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

На свойстве изоморфизма строятся все возможные способы хранения графа в памяти. Перечислим наиболее употребительные из них:

1. Вершины хранятся в массиве, каждый элемент которого — множество рёбер в форме вектора битов. Единичные биты соответствуют рёбрам, инцидентным данной вершине. Альтернатива — массив рёбер, каждое из которых задано вектором инцидентных вершин, которых может быть ровно две. Это — **матрица инциденций** размером $m \times n$. Это расточительный способ, потому что матрица большей частью состоит из нулей. Но он достаточно компактен и удобен для некоторых задач, например для отыскания вершинного или рёберного покрытия. Способ является естественным для неориентированного графа. Для орграфа следует различать начала и концы рёбер, например, так: «1» — ребро выходит из вершины, «2» — ребро входит, «3» — и входит, и выходит (петля).
2. Вершины хранятся в массиве, каждый элемент которого — множество смежных вершин в форме массива (вектора) битов. Это **матрица смежности** размерами $n \times n$, она может содержать 0 и 1 в любой пропорции. Так, полному графу соответствует единичная матрица. Способ удобен для орграфов. Неориентированные графы хранятся как дважды ориентированные, т. е. их матрица смежности всегда симметрична; она может храниться только верхним треугольником.
3. Вершины хранятся в массиве, каждый элемент которого, кроме самой вершины, множество смежных вершин, обычно в форме списка. Каждый элемент списка содержит поле с номером смежной вершины — индексом массива вершин. Это — **списки смежности**. Но поскольку списки — не обязательная форма, можно применить и векторы; более правильным будет название способа **набор множеств смежности.** Способ удобен, если количество рёбер в графе не очень велико, и требует порядка $O(n + m)$ ячеек памяти.
4. Массив рёбер, каждое из которых задано парой номеров инцидентных вершин, — **массив (вектор) пар**. Требует $2 \times m$ ячеек памяти. Способы 3 и 4 также естественны для орграфов. Если требуется неориентированный граф, он хранится как дважды ориентированный: для каждого ребра $<u, v>$ обязательно хранится и противоположно ориентированное ребро $<v, u>$.
5. Разветвляющийся список из вершин, в котором рёбра реализованы посредством указателей. Этот способ применяется главным образом для ациклических орграфов (деревьев), а в общем случае малопригоден без каких-либо дополнений. Интересной реализацией такого способа является структура Вирта. В её основе — список из вершин. Каждая вершина дополнена списком смежности, каждый элемент которого хранит указатель на смежную вершину — элемент списка вершин. Структура похожа на списки смежности, но не использует массив и номера его элементов для идентификации смежных вершин.

Контрольные вопросы.

1. Граф какого вида больше подходит в качестве модели бинарного отношения на произвольном конечном множестве?
2. Путь в графе — это ...
3. Длина пути между парой вершин в ненагруженном графе — это…
4. Цикл в графе называется элементарным, если…
5. Можно ли для представления графа в памяти использовать машинное слово?
6. Какое машинное представление графа может быть сделано самым компактным?
7. Представление графа в виде множеств смежности считается оптимальным…

15. Деревья

Дерево в общем случае — это связный граф без циклов (обычно — не ориентированный), абстрактная структура данных, представляющая собой множество ***вершин***, или ***узлов***, на которых определены попарные связи — *рёбра*. Будем рассматривать частный случай — корневые упорядоченные деревья. Они отличаются тем, что одна из вершин объявлена ***корнем***. Если в дереве есть корень, его рёбра становятся ориентированными, поскольку у любой последовательности попарно связанных вершин — *пути*, включающем корень, появляется направление от корня или к корню. Для любого узла *v* все вершины дерева на пути в корень, находящиеся ближе к корню, называется *предками*, а дальше от корня — *потомками*. Из пары узлов, связанных ребром, узел ближе к корню — *отец*, дальше от корня — *сын*. У каждого узла может быть несколько сыновей, которые называются *братьями*, или *дочерними узлами,* но только один отец. Корень — это единственный в дереве узел, у которого нет отца. Узлы, у которых нет сыновей, называются *листьями*.
Количество рёбер на пути из корня в узел дерева называется *глубиной* узла, количество рёбер на самом длинном пути в лист — *высотой*. Высота дерева — это высота его корня. Разность между высотой дерева и глубиной узла — это *уровень* узла. Дерево упорядочено, если упорядочены сыновья любого его узла. Сыновья упорядочены, если их перестановка меняет дерево. Из корневых упорядоченных деревьев наиболее часто используются ***двоичные***, или ***бинарные***. Каждый узел двоичного дерева может иметь не более двух сыновей — левого и правого, причём единственный сын узла — обязательно левый или правый. Более сложный вариант — ***троичное*** дерево, где у каждого узла — не более трёх сыновей: левый, средний, правый — в любой комбинации. Каждый из сыновей может рассматриваться как корень соответствующего поддерева, возможно, пустого.

15.1. Представление дерева в памяти

Поскольку дерево — это граф, годятся все способы представления в памяти, используемые для графов, с одной оговоркой: в структурах для графов общего вида группы смежных вершин сами по себе не упорядочены.
Но есть способы, специфичные именно для деревьев, в частности, для упорядоченных деревьев.
— пара (тройка) массивов. Вершины нумеруются от 0. В нулевом столбце массива располагается корень. Для каждого узла указываются индексы левого и правого сыновей;
— разветвляющийся список. Каждый узел располагается в свободной памяти и хранит указатели на своих сыновей.
— одномерный массив произвольного содержания. В его нулевом элементе располагается корень. Для каждого *i*-го узла (счёт от 0) сыновья располагаются в позициях 2*(*i* +1) – 1 и 2*(*i* +1), а отец — в позиции (*i*/2). Дерево в данном случае «прячется» в алгоритме обработки массива. Можно также получить дерево из массива, взяв в качестве корня его средний по порядку элемент, и построив поддеревья тем же способом из левой и правой половин массива. Здесь важен именно массив, используемый как память прямого доступа.
— альтернативные способы (обсуждаются ниже).

Разветвляющийся список — это естественный и часто применяемый способ для представления дерева в памяти, хотя, как показано ниже, далеко не самы экономный. Следует только заметить, что упомянутый выше способ «пара массивов» — это то же самое. Он получается при перехвате управления памятью при размещении узлов, как это проделывалось в своё время при работе с множествами в списках.
Узлы дерева — объекты, связи между которыми осуществляются через указатели. Для создания дерева достаточно объявить класс «узел дерева», членами которого должны быть указатели на узлы того же типа: «левый» и «правый» (у троичного дерева — «левый», «средний» и «правый»). В узле могут быть и другие данные-члены. Минимально необходимым является тег — метка или номер узла, с помощью которого можно различать узлы в процессе их обработки. Для работы с деревом в целом удобно иметь особый класс «дерево», в котором собираются данные, относящиеся к дереву в целом, и функции-члены для работы с деревом. Чтобы эти функции имели доступ к данным узла, достаточно объявить класс «дерево» дружественным для класса «узел».

```
//Класс «узел дерева»
class Node { char d;     //тег узла
   Node * lft;   // левый сын
//    Node  * mdl;   средний сын (если нужно)
   Node * rgt;  // правый сын
public:
      Node( ) : lft(nullptr), rgt(nullptr) { } // конструктор узла
      ~Node( ){ if delete rgt; // деструктор (уничтожает поддерево)
                  // delete mdl;
                  delete lft; } // удаление узлов в порядке обратном их созданию
```

```
    Node(const Node &) = delete;
    Node(Node &&) = delete;
    Node & operator = (const Node &) = delete;
    Node & operator = (Node &&) = delete;
friend class Tree;  // дружественный класс «дерево»
 } ;
// Класс «дерево в целом»
class Tree
{   Node * root;    // указатель на корень дерева
    char num, maxnum;   //счётчик тегов и максимальный тег
  const int trwide = 80;     // ширина изображения дерева
  int maxrow, offset;      //максимальная глубина, смещение корня
  char ** SCREEN; // память для выдачи на экран
  void clrscr( );   // очистка рабочей памяти
  Node* MakeNode(int depth); // создание поддерева
  void OutNodes(Node * v, int r, int c); // выдача поддерева
  Tree (const Tree &);   // конструктор копии (фиктивный )
  Tree (Tree &&);          //перемещающий конструктор (C++11)
  Tree operator = (const Tree &) const;   // присваивание
  Tree operator = (Tree &&) const; // перемещающее присваивание (C++11)
public:
  Tree(char num, char maxnum, int maxrow);
  ~Tree( );
  void MakeTree() // ввод — генерация дерева
    { root = MakeNode(0); }
  bool exist( ) { return root != nullptr; } // проверка «дерево не пусто»
  int DFS( ); // обходы дерева
  int BFS( );
  void OutTree( );    // выдача на экран
};
```

Кроме данных, в классе *Tree* объявлены скрытые функции-члены: вспомогательные функции, которые не входят в интерфейс и предназначены только для вызова из других функций-членов (эти функции при желании можно объявить членами класса «узел»). Конструкторы копирования и перегрузки присваивания сделаны скрытыми умышленно: попытка создать в программе ситуацию, в которой эти функции могут быть вызваны, приведёт к ошибке на этапе компиляции «нарушение защиты». Современные компиляторы рекомендую более радикальный и надёжный способ — пометить эти функции «=delete».
Конструктор дерева инициализирует параметры разметки и создаёт рабочую память — матрицу символов, необходимую для выдачи изображения дерева на экран.

```
Tree ::  Tree(char nm, char mnm, int mxr):
 num(nm), maxnum(mnm), maxrow(mxr), offset(trwide/2), root(nullptr),
 SCREEN(new char * [maxrow])
 {   for(int i = 0; i < maxrow; ++i) SCREEN[ i ] = new char[trwide]; }
```

Деструктор дерева уничтожает матрицу символов и запускает деструктор узла для корня.

```
Tree :: ~Tree( ) { for(int i = 0; i < maxrow; ++i) delete [ ]SCREEN[i];
                   delete [ ] SCREEN; delete root; }
```

Обратите внимание на то, как создаётся и уничтожается матрица.

15.2. Обходы дерева как рекурсивной структуры данных

Чтобы обработать каким-либо образом множество узлов дерева, его нужно обойти. Каждый узел дерева является корнем поддерева, а его сыновья — тоже корнями поддеревьев. Поэтому алгоритм обхода, запускаясь для узла, должен обработать информацию в узле и запустить такой же алгоритм для каждого из непустых поддеревьев. Существует три способа сделать это, отличающиеся лишь порядком шагов:

1. Прямой обход:
— обработать узел;
— посетить в прямом порядке каждого сына (левого, среднего, правого).
2. Обратный обход:
— посетить в обратном порядке каждого сына (левого, среднего, правого);
— обработать узел.

3. Внутренний, или симметричный обход:
— посетить во внутреннем порядке левого сына;
— обработать узел;
— посетить во внутреннем порядке правого сына (остальных сыновей).
Минимальная обработка узла может состоять в присвоении соответствующему в нём полю номера в порядке посещения (разметка) или в выдаче номеров на экран, если они уже имеются, или в формировании последовательности из номеров посещённых узлов. Очевидно, что не существует иных способов отличить один порядок обхода узлов от другого.
При разметке дерева в прямом порядке номер любого узла — наименьший, а при обратном — наибольший в соответствующем поддереве, а диапазон использованных номеров равен мощности поддерева. При разметке внутренним способом номер узла больше любого номера в левом поддереве и меньше любого номера в правом.

15.3. Создание дерева

Для создания дерева в памяти тоже применяется алгоритм обхода. Первым шагом этого алгоритма является проверка необходимости создания узла.
Если ответ положительный, узел создаётся и в нём заполняются информационные поля. В частности, может быть выполнен шаг разметки. Далее заполняются поля указателей на каждого сына: для получения значения указателя алгоритм запускается рекурсивно. Результат — указатель на вновь созданный узел или нуль, если узел не создан.
Проверка необходимости создания узла может быть выполнена тремя способами:
1. Запрос на ввод с клавиатуры. Приглашение ко вводу может содержать какую-либо информацию о месте предполагаемого узла в дереве. Ожидаемый ответ — «да» или «нет» (1 или 0, *Y* или *N*, и т. п.). Вместо ответа «да» можно вводить произвольную информацию (но не разметку!) для размещения в узле, особый ввод, например пустой, может означать «нет».
2. Чтение очередного элемента заранее заготовленной последовательности из массива, линейного списка или файла. Такая последовательность сама по себе тоже является способом размещения дерева в памяти, а алгоритм ввода просто преобразует её в форму разветвляющегося списка.
3. Обращение к датчику случайных чисел с целью генерации дерева. Датчик должен быть управляемым. Простой датчик с равновероятной выдачей 0 или 1 будет создавать пустые или очень маломощные деревья — из 1, 2, 3 узлов, так как вероятность того, что узел будет создан, очень быстро падает с ростом его глубины: для корня она составляет всего 0.5, для сыновей — 0.25 (0.5^2) и т. д. Нужен датчик, который бы обеспечивал вероятность создания корня близкую к 1 и уменьшал её с ростом глубины узла.
***Пример* такого датчика:** Y = depth < rand() % 6 + 1;
Здесь *depth* — глубина узла: для корня она 0, для произвольного узла — на 1 больше, чем у отца. Очевидно, что для корня *Y* = 1, а для узла на глубине больше 5 — всегда 0.
Функция-член для генерации случайного дерева может выглядеть так:

```
Node * Tree :: MakeNode(int depth)
{ Node * v = nullptr;
  int Y = (depth < rand( )%6+1) && (num <= 'z');
//  Вариант: cout ≪ "Node (" ≪ num ≪ ',' ≪ depth ≪ ")1/0: "; cin ≫ Y;
if (Y) {   // создание узла, если Y = 1
 v = new Node;
 v->d = num++;     // разметка в прямом порядке (= «в глубину»)
 v->lft = MakeNode(depth+1);
//     v->d = num++;          //вариант — во внутреннем
 v->rgt = MakeNode(depth+1);
//     v->d = num++;          // вариант — в обратном
  }
  return v;
}
```

Эта функция запускается из встраиваемой функции-члена *MakeTree*(), результат её работы присваивается полю *root*.
Вместо генерации случайного значения *Y* можно организовать ввод его с клавиатуры. Соответствующая альтернатива помещена в комментарий.
Функция создаёт дерево прямым обходом по той простой причине, что невозможно создать узел дерева, если не создан его отец. Но вот считать узел «пройдённым» можно когда угодно. Поэтому для разметки узла в алгоритме можно использовать три точки (две из них закомментированы): до обхода поддеревьев, после левого поддерева и перед правым и по окончании обхода поддеревьев. Нужный вариант разметки можно обеспечить, включив инициализацию в соответствующей точке и выключив — в остальных.
Значение глубины узла *depth*, необходимое для датчика, известно при входе в функцию и может быть использовано в любом месте. А вот данные, зависящие от поддеревьев: высота узла, количество листьев, количество потомков и т. п., могут быть

известны только тогда, когда оба поддерева уже обработаны, т. е. они доступны только при обратном обходе.

15.4. Вывод изображения дерева на экран монитора

Чтобы получить наглядное представление о способе разметки дерева, нужно вывести его на экран в виде диаграммы. Можно обойтись для этого текстовым режимом, если принять следующее соглашение. В середине первой строки текста вывести метку корня дерева. В следующей строке — расположить метки левого и правого сыновей в серединах левой и правой половины строки и т. д. Если дерево — троичное, метку среднего сына можно разместить прямо под корнем, и т. д., уменьшая смещение сыновей относительно корня в два раза по отношению к предыдущему ряду. Удобно воспользоваться рекурсивной функцией обхода дерева, которая выдаёт метку узла в некоторой точке экрана (*r*, *c*), а для сыновей добавляет 1 к номеру ряда и смещения к номеру столбца. Смещение удобно вычислять сдвигом некоторой константы *offset* на номер ряда, который совпадает с глубиной узла.

Для выдачи метки в нужную точку экрана можно использовать функцию позиционирования курсора *gotoxy*(*r*, *c*) из библиотеки *conio.h*, предварительно очистив экран функцией *clrscr*(). Но поскольку эти функции есть не во всех оболочках, можно обойтись без них, использовав промежуточную буферную память в виде матрицы символов, как это сделано ниже в примере. Для того чтобы понять разметку дерева, достаточно вывести узлы 5–6 верхних уровней. Для улучшения читабельности картинки рекомендуется вместо числовых меток использовать буквы латинского алфавита.

Функция-член для вывода изображения дерева на экран может выглядеть так:

```
void Tree :: OutTree( )
{     clrscr( );
OutNodes(root, 1, offset);
for (int i = 0; i < maxrow; ++i)
{ SCREEN[ i ][ trwide-1 ] = 0;
      cout << '\n' << SCREEN[ i ];
}
cout << '\n';
}
```

Она запускает закрытую функцию-член *clrscr*( ), которая готовит матрицу символов, заполняя её точками:

```
void Tree :: clrscr( )
{ for(int i = 0; i < maxrow; ++i)
memset(SCREEN[ i ], '.', trwide);
}
```

Далее выполняется закрытая функция *OutNodes*( ), расставляющая метки вершин дерева в матрице символов:

```
void Tree :: OutNodes(Node * v, int r, int c)
{ if (r && c && (c<trwide)) SCREEN[ r – 1 ][ c – 1 ] = v->d; // вывод метки
if (r < maxrow) {
if (v->lft) OutNodes(v->lft, r + 1, c – (offset >> r)); //левый сын
// if (v->mdl) OutNode(v->mdl, r + 1, c);     — средний сын (если нужно)
if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset >> r)); //правый сын
}
}
```

Затем матрица символов построчно выводится на экран.

Может получиться следующая картинка:

```
.......................................a.......................................
...................b.......................................h...................
.........c...................e...................i...................j.........
 .............d.........f.........g.....................................
................................................................................
................................................................................
```

Рис. 15.1. Пример выдачи изображения дерева на экран

Вместо вычисления смещения (*offset* >> *r*) в функции иногда удобнее использовать глобальный массив (*M*[*r*]), в котором смещения можно подобрать более точно, чтобы смежные поддеревья не перекрывались: *int M*[ ] {40, 20, 10, 5, 2, 1, 0}. Особенно это важно для троичных деревьев. Запас смещений должен быть достаточен для максимально используемой для вывода глубины узла *maxrow*.

Примеры: Двоичные деревья с разными вариантами разметки

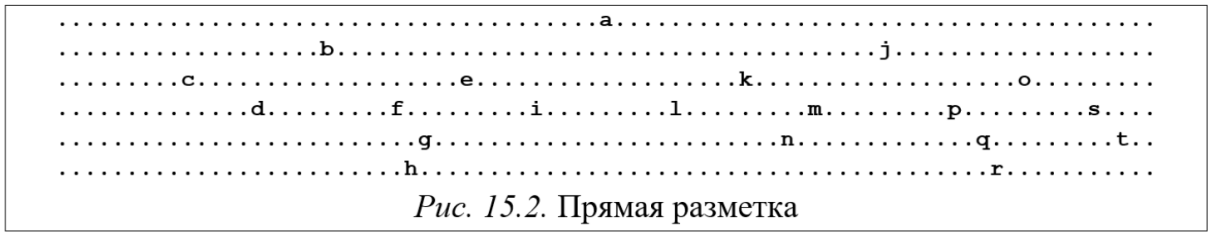
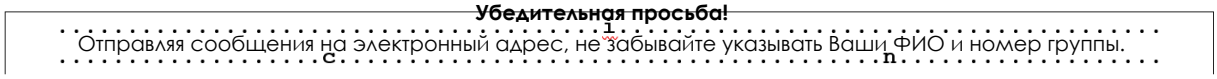


Рис. 15.2. Прямая разметка

Рис. 15.3. Обратная разметка



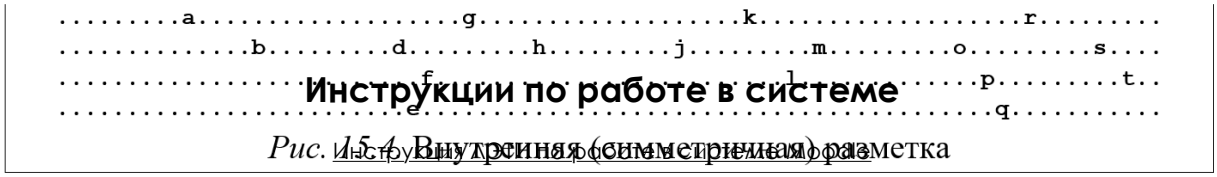


Рис. 15.4. Внутренняя (симметричная) разметка

Вывод на экран с использованием набора смещений

```
//Данные – ширина экрана и набор смещений
const int trwide = 80, offset[ ]{ 20, 10, 5, 2, 1, 0 };
void Tree :: OutNodes(Node * v, int r, int c)
{ if (r && c && (c < trwide)) SCREEN[ r – 1 ][ c – 1 ] = v->d; // вывод метки
  if (r < maxrow) {
  if (v->lft) OutNodes(v->lft, r + 1, c – (offset[ r ])); //левый сын
// if (v->mdl) OutNode(v->mdl, r + 1, c);     —) средний сын (если нужно
  if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset[ r ])); //правый сын
  }
}
```

Смысл: подбором смещений можно отсрочить момент наложения узлов на изображении дерева. Особенно это важно для троичных деревьев.