**Ответы на экзаменационные вопросы по программированию 2017 (2 семестр)**

Оглавление

[1. Этапы решения задачи с использованием вычислительной техники 1](#_Toc485569030)

[2. Виды программной и эксплуатационной документации 1](#_Toc485569031)

[3. Понятие алгоритма. Алгоритмическая система. Алгоритмизация 1](#_Toc485569032)

[4. Средства записи алгоритмов. Словесная запись. Блок-схемы и структурограммы 1](#_Toc485569033)

[Средства записи алгоритмов 1](#_Toc485569034)

[Словесная запись алгоритмов 1](#_Toc485569035)

[Схемы алгоритмов 1](#_Toc485569036)

[Структурограммы 1](#_Toc485569037)

[5. Технология разработки алгоритмов 1](#_Toc485569038)

[6. Базовые алгоритмические структуры 1](#_Toc485569039)

[7. Парадигмы программирования. Развитие языков программирования 1](#_Toc485569040)

[Парадигмы программирования 1](#_Toc485569041)

[Развитие языков программирования 1](#_Toc485569042)

[8. Алфавит языков программирования Pascal и C++. Структура программы на языках Pascal и C++ 1](#_Toc485569043)

[Структура программы на языке Pascal 1](#_Toc485569044)

[Структура программы на языке C++ 1](#_Toc485569045)

[9. Стандартные типы данных я языках программирования Pascal и C++. Понятие операции. Преобразование типов 1](#_Toc485569046)

[Стандартные типы данных я языках программирования Pascal и C++ 1](#_Toc485569047)

[Преобразование типов 1](#_Toc485569048)

[10. Линейные и разветвленные алгоритмы в языках программирования Pascal и C++. Операторы if и множественного выбора 1](#_Toc485569049)

[Синтаксис 1](#_Toc485569050)

[Оператор множественного выбора. 1](#_Toc485569051)

[11. Организация циклов я языках программирования Pascal и C++. Цикл с предусловием, цикл с постусловием 1](#_Toc485569052)

[Repeat, или цикл с постусловием 1](#_Toc485569053)

[12. Организация циклов в языках программирования Pascal и C++. Цикл с параметром. Операторы break, continue, goto, exit 1](#_Toc485569054)

[13. Создание одномерных массивов в языках программирования Pascal и C++, особенности работы с ними 1](#_Toc485569055)

[14. Методы сортировки одномерных массивов в языках программирования Pascal и C++ 1](#_Toc485569056)

[15. Создание многомерных массивов в языках программирования Pascal и C++, особенности работы с ними 1](#_Toc485569057)

[16. Особенности сортировки матриц в языках программирования Pascal и C++ 1](#_Toc485569058)

[17. Строка как массив символов. Ввод/вывод строк, функции работы со строками 1](#_Toc485569059)

[Строковый тип данных паскаль 1](#_Toc485569060)

[Процедуры и функции для работы со строками 1](#_Toc485569061)

[18. Пользовательские типы данных в языке программирования C++. Структуры - особенности создания и работы. Понятие псевдонима структуры. Вложенные структуры, массивы структур 1](#_Toc485569062)

[Составные типы 1](#_Toc485569063)

[Структуры 1](#_Toc485569064)

[19. Пользовательские типы данных в языке программирования C++. Объединения и перечисления - особенности создания и работы 1](#_Toc485569065)

[Перечисления (enum) 1](#_Toc485569066)

[20. Структура жесткого диска. Файловые системы DOS и NTFS 1](#_Toc485569067)

[21. Организация ввода и вывода в языке программирования C++. Стандартные файлы ввода/вывода. Текстовые файлы. Методы обработки текстовых файлов 1](#_Toc485569068)

[22. Организация ввода и вывода в языке программирования C++. Двоичные файлы. Последовательный доступ к элементам двоичных файлов 1](#_Toc485569069)

[23. Организация ввода и вывода в языке программирования C++. Произвольный доступ к элементам двоичных файлов 1](#_Toc485569070)

[24. Обобщенная архитектура процессора IBM PC. Понятие об адресном пространстве, понятие о моделях памяти 1](#_Toc485569071)

[25. Функции в языке программирования C++. Объявление и определение функций, параметры функций 1](#_Toc485569072)

[26. Функции в языке программирования C++. Локальные и глобальные переменные. Строки, массивы и структуры в качестве параметров функций 1](#_Toc485569073)

[27. Функции в языке программирования C++. Рекурсия. Встраиваемые функции. Перегрузка функций, использование аргументов по умолчанию 1](#_Toc485569074)

[28. Указатели в языке программирования C++. Объявление и разыменовывание указателей 1](#_Toc485569075)

[29. Указатели в языке программирования C++. Нулевые указатели и указатели на тип Void 1](#_Toc485569076)

[30. Ссылки в языке программирования C++. Резервирование памяти в куче 1](#_Toc485569077)

[31. Указатели и одномерные массивы в языке программирования C++. 2](#_Toc485569078)

[32. Указатели и многомерные массивы в языке программирования C++ 2](#_Toc485569079)

[33. Строковые указатели в языке программирования C++. Указатели и структуры в языке программирования C++ 2](#_Toc485569080)

[34. Указатели и функции в языке программирования C++. Способы передачи параметров. Передача массивов в функцию посредством указателей 2](#_Toc485569081)

[Локальные и глобальные переменные 2](#_Toc485569082)

[Передача параметров по значению и по ссылке 2](#_Toc485569083)

[Переменные, в которых сохраняются параметры, передаваемые функции, также являются локальными для этой функции. Эти переменные создаются при вызове функции и в них копируются значения, передаваемые функции в качестве параметров. Эти переменные можно изменять, но все изменения этих переменных будут "забыты" после выхода из функции. Рассмотрим это на примере следующей функции, "меняющей" значения двух переданных ей переменных: 2](#_Toc485569084)

[35. Указатели и функции в языке программирования C++. Строки как аргументы функций 2](#_Toc485569085)

[36. Указатели и функции в языке программирования C++. Передача структур в функцию по указателю и по ссылке 2](#_Toc485569086)

[Указатель в Си 2](#_Toc485569087)

[37. Указатели и функции в языке программирования C++. Ссылка в качестве возвращаемого значения функции. Функции, возвращающие указатели 2](#_Toc485569088)

[38. Динамические структуры данных в языке программирования C++. Понятие о самоссылочных структурах. Формирование очереди. Формирование стека 2](#_Toc485569089)

[39. Основные определения объектно-ориентированного программирования 2](#_Toc485569090)

[40. Понятие о конструкторах и деструкторах в языке программирования C++. Конструкторы с параметрами, конструкторы по умолчанию 2](#_Toc485569091)

[41. Функции и объекты в языке программирования C++. Указатели на объекты. Передача объектов в функцию 2](#_Toc485569092)

[Указатели на объекты 2](#_Toc485569093)

[Ключевые термины 2](#_Toc485569094)

[Косвенная адресация – это обращение к области памяти не напрямую, по адресу, а через объект, которому в памяти соответствует определенный участок. 2](#_Toc485569095)

[42. Функции и объекты в языке программирования C++. Объекты в качестве возвращаемых значений. Дружественные функции 2](#_Toc485569096)

[43. Перегрузка операторов в языке программирования C++. Перегрузка унарных и бинарных операторов 2](#_Toc485569097)

[44. Наследование в языке программирования C++. Основные понятия, режимы доступа к элементам базового класса 2](#_Toc485569098)

[Простое наследование 2](#_Toc485569099)

[Множественное наследование 2](#_Toc485569100)

[45. Виртуальные функции в языке программирования C++. Основные понятия 2](#_Toc485569101)

[46. Объекты и файловые потоки в языке программирования C++. Потоковый ввод/вывод 2](#_Toc485569102)

[6.2. Ввод со стандартного потока ввода 2](#_Toc485569103)

[47. Объекты и файловые потоки в языке программирования C++. Текстовые файлы 2](#_Toc485569104)

[48. Объекты и двоичные файлы в языке программирования C++. Сохранение данных в двоичных файлах 2](#_Toc485569105)

[49. Шаблоны и родовые функции в языке программирования C++. Основные понятия 2](#_Toc485569106)

[Простой шаблон типа 2](#_Toc485569107)

[Шаблоны типа для списка 2](#_Toc485569108)

[50. Исключительные ситуации. Основные причины возникновения. Обработка исключений 2](#_Toc485569109)

[51. Библиотека стандартных шаблонов в языке программирования C++. Назначение, основные понятия 2](#_Toc485569110)

1. **Этапы решения задачи с использованием вычислительной техники**

Процесс решения задач на компьютере – это совместная деятельность человека и ЭВМ. На долю человека приходятся этапы, связанные с творческой деятельностью – постановкой, алгоритмизацией, программированием задач и анализом результатов, а на долю персонального компьютера – этапы обработки информации в соответствии с разработанным алгоритмом.

1. Постановка задачи. На этом этапе участвует человек, хорошо представляющий предметную область задачи (биолог, экономист, инженер). Он должен чётко определить цель задачи, дать словесное описание содержания задачи и предложить общий подход к её решению.
2. Выбор метода решения (математическое или информационное моделирование). Цель данного этапа – создать такую математическую модель решаемой задачи, которая могла быть реализована в компьютере. Существует целый ряд задач, где математическая постановка сводится к простому перечислению формул и логических условий.
3. Алгоритмизация задачи. На основе математического описания необходимо разработать алгоритм решения.
4. Программирование. Программой называется план действий, подлежащих выполнению некоторым исполнителем, в качестве которого может выступать компьютер. Программа позволяет реализовать разработанный алгоритм.
5. Ввод программы и исходных данных в ЭВМ (например, с клавиатуры с помощью редактора текстов). Для постоянного хранения осуществляется их запись на жёсткий диск или другое ПЗУ.
6. Тестирование и отладка программы. Исполнение алгоритма с помощью ЭВМ, поиск и исключение ошибок. При этом программисту приходится выполнять рутинную работу по проверке работы программы, поиску и исключению ошибок, и поэтому для сложных программ этот этап часто требует гораздо больше времени и сил, чем написание первоначального текста программы.

Отладка программы – сложный и нестандартный процесс, который заключается в том, чтобы протестировать программу на контрольных примерах.

1. Исполнение отлаженной программы и анализ результатов. На этом этапе программист запускает программу и задаёт исходные данные, требуемые по условию задачи.

*Полученные результаты анализируются постановщиком задачи, и на основании этого анализа вырабатываются соответствующие решения, рекомендации, выводы.*

1. **Виды программной и эксплуатационной документации**

Любая программа должна снабжаться программной документацией. Документирование программ регламентируется стандартами Единой системы программной документации (ЕСПД) вне зависимости от назначения и области применения.

Цель документирования – помощь стороннему пользователю в понимании и использовании программных средств.

Полнота и правильность документации – показатель квалификации программиста.

Следствия грамотного документирования: экономия времени пользователя; минимизация вероятности непредвиденных ситуаций.

Виды программ и программных документов для ЭВМ комплексов и систем независимо от их назначения и области применения установлены ГОСТ 19.101-77 и ГОСТ 19781-90.

Документация должна разрабатываться с самого начала проектирования ПО и непрерывно уточняться для обеспечения полного соответствия программному изделию.

**Виды программной документации:**

1. ЭКСПЛУАТАЦИОННАЯ (пользовательская) документация - это минимально необходимые сведения для эксплуатации ПО.

К такой документации относятся документы, которыми руководствуется пользователь при:

- *инсталляции*ПС (при установке ПС с соответствующей настройкой на среду применения ПС);

- при применении ПС для решения своих задач и при управлении ПС (например, когда данное ПС взаимодействует с другими системами).

Эти документы частично затрагивают вопросы сопровождения ПС, но не касаются вопросов, связанных с модификацией программ.

1. ТЕХНОЛОГИЧЕСКАЯ документация кроме эксплуатационной документации включает ряд документов, необходимых для развития и сопровождения ПО в течение всего жизненного цикла.
2. ИССЛЕДОВАТЕЛЬСКАЯ документация предназначена для анализа жизненного цикла ПО. Основная ее задача состоит в фиксировании и обобщении характеристик жизненного цикла ПО.

**Виды программной документации: внешняя; внутренняя.**

Формы внешней документации: схемы алгоритмов, инструкция для пользователей, образцы входных и выходных данных, полное описание процесса выполнения программы, словесное описание алгоритма, ссылки на источники информации и т.п.

Формы внутренней документации: комментарии в программном коде (например, «//вывод данных»).

1. **Понятие алгоритма. Алгоритмическая система. Алгоритмизация**
2. **Алгоритм** – это последовательность команд, выполнение которых приводит к решению поставленной задачи. Простым языком: это определенные действия, с указанной последовательностью, которые приводят нас к результату.

Анализ элементарных операций по обработке информации, встречающихся в реальных алгоритмах, показывает, что их можно разделить на две группы: арифметические и логические. Арифмети­ческие операции выполняют непосредственное преобразование ин­формации, а логические определяют направление процесса обра­ботки информации. В алгоритмах арифметические и логические операции чередуются в определенной последовательности. Если вы­полнение алгоритма сводится к арифметическим операциям, то та­кой алгоритм называется численным.

Свойства алгоритмов:

1. Дискретность – алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.
2. Детерминированность (или определённость). В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит формальный характер и не требует никаких дополнительных указаний или сведений о решаемой задаче.
3. Понятность – алгоритм для исполнителя должен включать только те команды, которые ему (исполнителю) доступны, которые входят в его систему команд. Алгоритм должен быть понятным!
4. Результативность (или конечность). Это свойство состоит в том, что при корректно заданных исходных данных алгоритм должен завершать работу (приводить к решению задачи) и выдавать результат за конечное число шагов.
5. Массовость – универсальность. Это означает, что алгоритм решения задачи разрабатывается в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применения алгоритма. Но это правило не всегда действует, в основном разрабатывается один или несколько алгоритмов для решения одной задачи.
6. **Понятие «алгоритмическая система»** дает формальный ответ на вопрос, что должно быть известно и доступно разработчикам алгоритмов. Алгоритмическаясистема – набор средств и понятий, позволяющих строить некоторое множество алгоритмов для решения определенного класса задач. Алгоритмическая система определяется наличием четырех составляющих ее частей:

1) множеством входных объектов или исходных данных, подлежащих обработке алгоритмами данной системы;

2) множеством выходных объектов или результатов выполнения алгоритмов данной системы;

3) системой команд исполнителя, то есть набором тех действий, которые может выполнять исполнитель, и которые мы можем описывать в алгоритмах, что собственно является ориентацией алгоритмической системы на конкретного исполнителя;

4) языком описания алгоритмов – языком исполнителя; язык, на котором описан алгоритм, должен быть понятен исполнителю и не должен включать в свой состав указания на невозможные для исполнителя действия, а также обращения к входным или выходным объектам, не принадлежащих к множеству входных или выходных объектов данной алгоритмической системы.

1. **Алгоритмизация** – процесс разработки и описания алгоритма решения какой-либо задачи.

Поскольку речь идет о разработке алгоритма для ЭВМ, то нужно сначала проанализировать возможность его машинной реализации, оценить ресурсы и возможности конкретной ЭВМ, имеющейся в распоряжении

Непосредственная разработка алгоритма начинается с осознания существа поставленной задачи, с анализа того, что нам известно, что следует получить в качестве результата, в какой форме нужно представить исходные данные и результаты вычислений. Следующая ступень – разработка общей идеи алгоритмического процесса и анализа этой идеи. После этого можно приступить к более детальной разработке уже задуманного конкретного алгоритма. И вот этот процесс разработки конкретного алгоритма, в соответствии с определением самого понятия «алгоритм», заключается в последовательном выполнении следующих пунктов:

1) разложении всего вычислительного процесса на отдельные шаги – возможные составные части алгоритма, что определяется внутренней логикой самого процесса и системой команд исполнителя;

2) установлении взаимосвязей между отдельными шагами алгоритма и порядка их следования, приводящего от известных исходных данных к искомому результату;

3) полном и точном описании содержания каждого шага алгоритма на языке выбранной алгоритмической системы;

4) проверке составленного алгоритма на предмет, действительно ли он реализует выбранный метод и приводит к искомому результату.

Ссылка на инфу <http://www.studfiles.ru/preview/1672255/page:35/>

1. **Средства записи алгоритмов. Словесная запись. Блок-схемы и структурограммы**

## **Средства записи алгоритмов**

Характер языка, используемого для записи алгоритмов, определяется тем, для какого исполнителя предназначен алгоритм. Возможности исполнителя алгоритмов определяют уровень используемых языковых средств, то есть степень детализации и формализации предписаний в алгоритмической записи. Если алгоритм предназначен для исполнителя-человека, то его запись может быть не полностью формализована и детализирована, но должна оставаться понятной и корректной. Для записи таких алгоритмов может использоваться естественный язык. Для записи алгоритмов, предназначенных для исполнителей-автоматов, необходимы строгая формализация средств записи и определенная детализация алгоритмических предписаний. В таких случаях применяют специальные формализованные языки.

Поскольку одним из пользователей языка описания алгоритмов, так или иначе, остается человек, то, говоря об уровне языка, имеют в виду также и уровень его доступности для человека.

<http://www.studfiles.ru/preview/1672255/page:36/#71>

## **Словесная запись алгоритмов**

Самой распространенной формой представления алгоритмов, адресуемых человеку, является обычная словеснаязапись. В этой форме могут быть выражены любые алгоритмы. Но если такой алгоритм предназначен для его дальнейшей реализации на вычислительном устройстве, то принято придерживаться более формализованного способа построения фраз с тщательно отобранным набором слов. Кроме того, необходимо указывать начало и конец алгоритма, отмечать момент ввода в вычислительное устройство значений исходных данных и вывода/печати полученного результата. В вычислительных алгоритмах широко используется общепринятая математическая символика, язык формул. Вводится необходимая в вычислительной практике операция присваивания

<http://www.studfiles.ru/preview/1672255/page:36/#72>

## **Схемы алгоритмов**

Схема алгоритма – это графический способ его представления с элементами словесной записи. Каждое предписание алгоритма изображается с помощью плоской геометрической фигуры – блока. Отсюда название: блок-схема. Переходы от предписания к предписанию изображаются линиями связи – линиями потоков информации, а направление переходов – стрелками. Различным по типу выполняемых действий блокам соответствуют различные геометрические фигуры. Приняты определенные стандарты графических изображений блоков:

|  |  |  |
| --- | --- | --- |
| Процесс (вычислительный блок) | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-mhUbLa.png | Выполнение операции или группы операций, в результате которых изменяются значение, форма представления или расположение данных |
| Решение (логический блок) | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-ZAyhRO.png | Выбор направления выполнения алгоритма в зависимости от некоторых условий |
| Модификация (заголовок цикла) | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-UgCnyG.png | Выполнение операций по управлению циклом – повторением команды или группы команд алгоритма |
| Пуск-останов (начало-конец) | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-votcGY.png | Начало или конец выполнения программы или подпрограммы |
| Предопределенный процесс (вызов подпрограммы) | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-0GyLgP.png | Вызов и использование ранее созданных и отдельно описанных алгоритмов (подпрограмм) |
| Ввод-вывод | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-JkxvNO.png | Общее обозначение ввода или вывода данных в алгоритме безотносительно к внешнему устройству |
| Соединитель | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-G_nqD7.png | Указание прерванной связи между блокам в пределах одной страницы |
| Межстраничный соединитель | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-RxsVGQ.png | Указание прерванной связи между блоками, расположенными на разных листах |

<http://www.studfiles.ru/preview/1672255/page:37/#73>

## **Структурограммы**

В практике структурного программирования для представления алгоритмов используются также структурограммы. Этот способ позволяет изображать схему передач управления в алгоритме не с помощью явного указания линий потоков информации, а с помощью представления вложенности структур – функциональных блоков, которые используются для описания выполняемых действий. Некоторые из используемых в этом способе блоков соответствуют их изображению в схемах алгоритмов. Для изображения алгоритмов в структурограммах используются следующие блоки.

|  |  |
| --- | --- |
| 1. Блок обработки (вычислений). Каждый блок является блоком обработки. Каждый прямоугольник внутри любого блока представляет собой также блок обработки. | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-5ADfDJ.png |
| 2. Блок следования. Этот блок объединяет ряд следующих друг за другом процессов обработки. | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-1iyr_x.png |
| 3. Блок решения. Этот блок применяется для обозначения структуры ветвления. Условие располагается в верхнем треугольнике, варианты решения – по сторонам треугольника, процессы обработки – в нижних прямоугольниках. Если блок решения является сокращенным (отсутствует одна из ветвей), то структурограмма видоизменяется соответствующим образом. | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-gClIaL.pnghttp://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-wBClWO.png |
| 4. Блок варианта реализует структуру многоальтернативного выбора. Варианты, которые можно сформулировать точно, размещаются слева, остальные объединяются в один, называемый выходом по несоблюдению условий, располагаемый справа. Правую часть можно оставить незаполненной или опустить. | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-mXMpUo.png |
| 5. Блок цикла с предусловием реализует циклическую структуру с проверкой условия в начале цикла. Условие продолжения цикла размещается в верхней полосе, сливающейся с левой, указывающей границу цикла. Данный блок может быть использован и для обозначения цикла с параметром, тогда вверху указывают закон изменения параметра цикла. | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-Y_Or8l.png |
| 6. Блок цикла с постусловие аналогичен блоку цикла с предусловием, но условие окончания цикла располагают внизу.  Каждый блок имеет форму прямоугольника и может быть вписан в любой внутренний прямоугольник любого другого блока. Блоки дополняются элементами словесной записи с использованием математической символики. На Рис. 9 .26 приведен пример структурограммы алгоритма Евклида. | http://www.studfiles.ru/html/2706/123/html_pyL6JInDtE.5aoW/img-L00D0m.png |

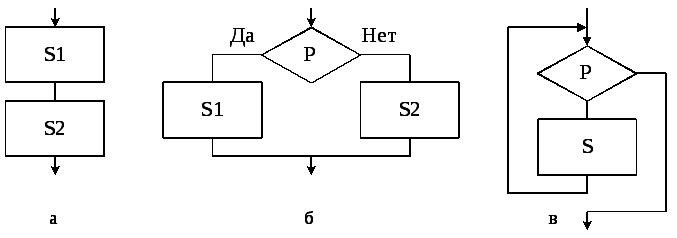
<http://www.studfiles.ru/preview/1672255/page:38/#74>

1. **Технология разработки алгоритмов**

Прежде всего, от алгоритма требуется, чтобы он правильно решал поставленную задачу. Но не менее важно, какой ценой это достигается. Речь идет о разумности затрат на его создание. С этой точки зрения алгоритм должен быть легким для понимания, простым для доказательства правильности и удобным для модификации. В рамках такой идеологии и сформировался так называемый структурный подход к конструированию и оформлению алгоритмов, позволяющий уменьшить количество ошибок в алгоритмах, упрощающий их контроль и модификацию.

По своей сути структурный подход есть отказ от беспорядочного стиля в алгоритмизации и программировании и определение ограниченного числа стандартных приемов построения легко читаемых алгоритмов и программ с ясно выраженной структурой. Теоретическим фундаментом этого подхода является теорема о структурировании, из которой следует, что алгоритм решения любой практически вычислимой задачи может быть представлен с использованием трех элементарных базисных управляющих структур: а) структуры следования или последовательности; б) структуры ветвления; в) структуры цикла с предусловием

Соответственно: а, б, в, где P– условие, S– оператор



Базисный набор управляющих структур является функционально полным, то есть с его помощью можно создать любой сколь угодно сложный алгоритм. Однако с целью создания более компактных и наглядных алгоритмов дополнительно используются следующие управляющие структуры: а) структура сокращенного ветвления; б) структура выбора; в) структура цикла с параметром; г) структура цикла с постусловием

Любой алгоритм может быть построен посредством композиции базисных и дополнительных структур:

- путем их последовательного соединения −образования последовательных конструкций;

- путем их вложения друг в друга −образования вложенных конструкций.

[http://www.studfiles.ru/preview/1672255/page:38/#75](http://www.studfiles.ru/preview/1672255/page:38/%2375)

1. **Базовые алгоритмические структуры**

Логическая структура любого алгоритма может быть представлена комбинацией трех базовых структур: следование (линейная структура), ветвление, цикл.

Характерной особенностью базовых структур является наличие в них **одного входа**и**одного выхода.**

1. Базовая структура**"следование".**Образуется последовательностью действий, следующих одно за другим

2. Базовая структура**"ветвление"**. Обеспечивает в зависимости от результата проверки условия (**да** или **нет**) выбор одного из альтернативных путей работы алгоритма. Каждый из путей ведет к **общему выходу**, так что работа алгоритма будет продолжаться независимо от того, какой путь будет выбран.

**3. Базовая структура "цикл".** Обеспечивает **многократное выполнение некоторой совокупности действий,** которая называется **телом цикла**.

**Таблица 1. Примеры использования базовых структур алгоритмов.**

|  |  |  |
| --- | --- | --- |
| **Пример структуры «следование»** | **Пример структуры «ветвление»** | **Пример структуры «цикл»** |
| http://www.studfiles.ru/html/2706/1071/html_Z_cKbr8za6.rpU4/img-X5pQa7.png | http://www.studfiles.ru/html/2706/1071/html_Z_cKbr8za6.rpU4/img-EHwvnx.png | http://www.studfiles.ru/html/2706/1071/html_Z_cKbr8za6.rpU4/img-pPHO6v.png  http://www.studfiles.ru/html/2706/1088/html_ApOb0gbvgb.LGiI/img-VuP9QI.png |

1. **Парадигмы программирования. Развитие языков программирования**

## Парадигмы программирования

**Парадигма – базовая модель конкретного способа организации информации.**Например, ОО-парадигма делает упор на поведении и обязанностях.

**Парадигма – способ построения системных абстракций на основе свойств общности и изменчивости.**В ОО-парадигме системы создаются на основе общности структуры и поведения и отклонений структуры и алгоритмов.

**Парадигмы программирования** – это лишь различные инструменты, которые можно использовать при программировании. Каждый из этих инструментов по-своему хорош. Но это не значит, что они "эффективно универсальны". То есть, на самом деле, различные методики программирования дают разный выигрыш для решения задач разных классов. Этот выигрыш можно мерить по двум параметрам:

* эффективность программного обеспечения на современных ЭВМ
* общие затраты на разработку программного обеспечения

**Паради́гма программи́рования** — это парадигма, определяющая стиль программирования, иначе говоря — некоторый цельный набор идей и рекомендаций, определяющих стиль написания программ.

 парадигмой, имеющей наиболее "естественную" реализацию семантики на нынешних компьютерах, является **императивное программирование**. В эффективности реализации оно заведомо выигрывает по сравнению с любой другой методологией. Практически любой язык программирования в наши дни – это язык **определений**. Программы представляют из себя множество определений программных объектов (типов данных, функций, ...), которые как-то взаимосвязаны.

## Развитие языков программирования

Первые ЭВМ, созданные человеком, имели небольшой набор команд и встроенных типов данных, но позволяли выполнять программы на машинном языке.

***Машинный язык*** — единственный язык, понятный ЭВМ. Он реализуется аппаратно: каждую команду выполняет некоторое электронное устройство. Программа на машинном языке представляет собой последовательность команд и данных, заданных в цифровом виде.

Данные на машинном языке представлены числами и символами. Операции являются элементарными и из них строится вся программа. Ввод программы в цифровом виде производился непосредственно в оперативную память.

Естественно, что процесс программирования был очень трудоемким, разобраться в программе даже автору было довольно сложно, а эффект от применения ЭВМ был довольно низким.

Стремление программистов оперировать не цифрами, а символами, привело к созданию мнемонического языка программирования, который называют ***ассемблером***, мнемокодом, автокодом. Этот язык имеет определенный синтаксис записи программ, в котором, в частности, цифровой код операции заменен мнемоническим кодом.

Не очень заметный, казалось бы, шаг — переход к символическому кодированию машинных команд — имел на самом деле огромное значение. Программисту не надо было больше вникать в хитроумные способы кодирования команд на аппаратном уровне. Более того, зачастую одинаковые по сути команды кодировались совершенно различным образом в зависимости от своих параметров

Появилась также возможность использования макросов и меток, что также упрощало создание, модификацию и отладку программ. Появилось даже некое подобие переносимости — существовала возможность разработки целого семейства машин со сходной системой команд и некоего общего ассемблера для них, при этом не было нужды обеспечивать двоичную совместимость.

На сегодняшний день язык ассемблера используется для создания системных программ, использующих специфические аппаратные возможности данного класса ЭВМ.

***Вывод:*** ***Уровни языков программирования: языки программирования низкого уровня.***Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется *языком программирования низкого уровня*. «Низкий уровень» –это значит, что операторы близки к машинному коду и ориентированы на конкретный тип процессора. Языком самого низкого уровня является язык Ассемблера, который представляет каждую машинную команду в виде символьных условных обозначений, называемых *символьными* *мнемониками*.

Перевод программы на языке Ассемблера в машинный язык называется *транслитерацией*. Программа, написанная на языке низкого уровня, может быть использована только в такой среде, в которой она была создана. С помощью языков низкого уровня создаются очень эффективные и компактные программы, так как разработчик получает доступ ко всем возможностям процессора.

Следующий этап характеризуется созданием языков высокого уровня. Языкипрограммирования*высокого уровня* ближе и понятнее человеку, чем компьютеру. В программах, созданных на языках высокого уровня, особенности компьютерных систем не учитываются, перенос программ на уровне исходных текстов на другие платформы не создает трудностей, если в них создан транслятор этого языка. Программы разрабатывать на языках высокого уровня проще, а ошибок допускается меньше.

Эти языки являются универсальными (на них можно создавать любые прикладные программы) и алгоритмически полными, имеют более широкий спектр типов данных и операций, поддерживают технологии программирования. На этих языках создается неисчислимое множество различных прикладных программ.

***Принципиальными отличиями***языков высокого уровня от языков низкого уровня являются:

* использование переменных;
* возможность записи сложных выражений;
* расширяемость типов данных за счет конструирования новых типов из базовых;
* расширяемость набора операций за счет подключения библиотек подпрограмм;
* слабая зависимость от типа ЭВМ.

<http://www.studfiles.ru/preview/3582382/page:2/>

1. **Алфавит языков программирования Pascal и C++. Структура программы на языках Pascal и C++**

**Алфавитом языка программирования**

 называют набор символов (разрешенный к использованию и воспринимаемый компилятором), с помощью которого могут быть образованы величины, выражения и операторы данного языка.

Алфавит или множество литер языка программирования С++ основывается на множестве символов таблицы. Алфавит С++ включает:

1. Строчные прописные буквы латинского алфавита
2. Цифры от 0 до 9 (буквы-цифры)
3. Символ «\_» (буква)
4. Набор символов (“”, [], {}, +, -, %,/,\, :, ; и т.д.)

<http://www.studfiles.ru/preview/6070488/>

Алфавит Pascal:

* латинские строчные и прописные буквы,
* арабские цифры от 0 до 9,
* символ подчеркивания.
* пробел, основное назначение которого разделение ключевых слов и имен,
* управляющие символы (ASCII – коды от 0 до 31). Эти символы могут применяться при описании строчных и символьных констант. Управляющие символы с ASCII-кодом 9 (табуляция), а также 10 и 13 (замыкающее строку) используются в качестве разделителей при написании программ.
* **Специальные символы**, выполняющие определенные функции при построении различных конструкций языка: + – \* /{ } [ ] ( ) < > / ? ‘ : ; ^ # @ $
* **Составные символы**– группа символов, которые воспринимаются компилятором как единое целое:<= => := (\* \*) (. .) .

<http://www.studfiles.ru/preview/4693948/page:4/>

## Структура программы на языке Pascal

Любую программу, написанную на языке Паскаль, можно условно разделить на три основные части:

* раздел объявлений и соглашений (декларационная часть);
* раздел текстов процедур и функций;
* раздел основного блока.

Структуру программы можно представить следующим образом:

РАЗДЕЛ ОБЪЯВЛЕНИЙ И СОГЛАШЕНИЙ

|  |  |
| --- | --- |
| PROGRAM | Заголовок программы |
| {$ ... } | Глобальные директивы компилятора |
| USES | Подключаемые библиотеки |
| LABEL | Подраздел объявления глобальных меток |
| CONST | Подраздел объявления глобальных констант |
| TYPE | Подраздел объявления глобальных типов |
| VAR | Подраздел объявления глобальных переменных |

РАЗДЕЛ ТЕКСТОВ ПРОЦЕДУР И ФУНКЦИЙ

|  |  |
| --- | --- |
| PROCEDURE (FUNCTION) | Заголовок процедуры (функции) |
| LABEL | Подраздел объявления локальных меток |
| CONST | Подраздел объявления локальных констант |
| TYPE | Подраздел объявления локальных типов |
| VAR | Подраздел объявления локальных переменных |
| BEGIN | Основной блок процедуры или функции |
| END; |  |

РАЗДЕЛ ОСНОВНОГО БЛОКА ПРОГРАММЫ

|  |  |
| --- | --- |
| BEGIN | Основной блок программы |
| END. |  |

[http://www.studfiles.ru/preview/4693948/page:4/#6](http://www.studfiles.ru/preview/4693948/page:4/%236)

## Структура программы на языке C++

Большинство программ на С++ имеют следующий вид:

раздел подключения заголовочных файлов:

подключение библиотек (#include <iostream.h>)

заголовок программы (функции):

Программа C++ состоит из одного или более модулей, называемых функциями. Выполнение программы начи­нается с функции, имеющей имя main(), поэтому в про­грамме обязательно должна присутствовать функция с таким именем. Если в про­грамме нет такой функции, то в нет и законченной программы; компилятор в этом случае указывает, что функция main() не была определена.

Описание такой функции выполняют в разделе заголовка программы и записывают в виде:

void main()

{

тело функции:

описание переменных;

операторы;

return;

}

<http://www.studfiles.ru/preview/5403609/>

1. **Стандартные типы данных я языках программирования Pascal и C++. Понятие операции. Преобразование типов**

## Стандартные типы данных я языках программирования Pascal и C++

Данные отображают в программе окружающий мир. Цель программы состоит в обработке данных. Данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

1. внутреннее представление данных в памяти компьютера;
2. множество значений, которые могут принимать величины этого типа;
3. операции и функции, которые можно применять к данным этого типа.

В зависимости от требований задания программист выбирает тип для объектов программы. Типы Си++ можно разделить на простые и составные. К простым типам относят типы, которые характеризуются одним значением.

| **Тип** | **байт** | **Диапазон принимаемых значений** |
| --- | --- | --- |
| целочисленный (логический) тип данных | | |
| bool | 1 | 0   /   255 |
| целочисленный (символьный) тип данных | | |
| char | 1 | 0   /   255 |
| целочисленные типы данных | | |
| short int | 2 | -32 768    /    32 767 |
| unsigned short int | 2 | 0  /  65 535 |
| int | 4 | -2 147 483 648   /   2 147 483 647 |
| unsigned int | 4 | 0     /     4 294 967 295 |
| long int | 4 | -2 147 483 648    /    2 147 483 647 |
| unsigned long int | 4 | 0     /     4 294 967 295 |
| типы данных с плавающей точкой | | |
| float | 4 | -2 147 483 648.0  / 2 147 483 647.0 |
| long float | 8 | -9 223 372 036 854 775 808 .0   /   9 223 372 036 854 775 807.0 |
| double | 8 | -9 223 372 036 854 775 808 .0   /   9 223 372 036 854 775 807.0 |

[http://cppstudio.com/post/271/](http://cppstudio.com/post/271/%20)

<http://itandlife.ru/programming/cpp/tipy-dannyx-c/>

|  |  |  |  |
| --- | --- | --- | --- |
| Идентификатор | Длина (байт) | Диапазон значений | Операции |
| **Целые типы** | | | |
| integer | 2 | -32768..32767 | +, -, /, \*, Div, Mod, >=, <=, =, <>, <, > |
| byte | 1 | 0..255 | +, -, /, \*, Div, Mod, >=, <=, =, <>, <, > |
| word | 2 | 0..65535 | +, -, /, \*, Div, Mod, >=, <=, =, <>, <, > |
| shortint | 1 | -128..127 | +, -, /, \*, Div, Mod, >=, <=, =, <>, <, > |
| longint | 4 | -2147483648..2147483647 | +, -, /, \*, Div, Mod, >=, <=, =, <>, <, > |
| **Вещественные типы** | | | |
| real | 6 | 2,9x10-39 - 1,7x1038 | +, -, /, \*, >=, <=, =, <>, <, > |
| single | 4 | 1,5x10-45 - 3,4x1038 | +, -, /, \*, >=, <=, =, <>, <, > |
| double | 8 | 5x10-324 - 1,7x10308 | +, -, /, \*, >=, <=, =, <>, <, > |
| extended | 10 | 3,4x10-4932 - 1,1x104932 | +, -, /, \*, >=, <=, =, <>, <, > |
| **Логический тип** | | | |
| boolean | 1 | true, false | Not, And, Or, Xor, >=, <=, =, <>, <, > |
| **Символьный тип** | | | |
| char | 1 | все символы кода ASCII | +, >=, <=, =, <>, <, > |

<http://mif.vspu.ru/books/pascal/types.html>

Понятие операции

**Опера́ция** — конструкция [в языках программирования](http://dic.academic.ru/dic.nsf/ruwiki/1465), аналогичная по записи математическим операциям, то есть специальный способ записи некоторых действий. Наиболее часто применяются арифметические, [логические](http://dic.academic.ru/dic.nsf/ruwiki/824451) и [строковые](http://dic.academic.ru/dic.nsf/ruwiki/6807) операции. В отличие от функций, операции часто являются базовыми элементамиязыка и обозначаются различными символами пунктуации, а не алфавитно-цифровыми; они имеютспециальный инфиксный синтаксис и нестандартные правила передачи аргументов. Терминология, однако, несколько отличается от языка к языку.

Операции делятся по количеству принимаемых аргументов на:

* *унарные* — один аргумент (отрицание, унарный минус)
* *бинарные* — два аргумента (сложение, вычитание, умножение и т.д.)
* *тернарные* — три аргумента ("условие ? выражение1 : выражение2")

<http://dic.academic.ru/dic.nsf/ruwiki/106284>

## Преобразование типов

Преобразование типов — это процесс изменения типа значения. Например, можно преобразовать строку "1234" в число. Кроме того, можно преобразовать данные любого типа в тип **String**. Преобразование некоторых типов невозможно. Например, объект **Date** нельзя преобразовать в объект **ActiveXObject**.

Преобразование типов бывает *расширяющим* или *сужающим*: при расширяющих преобразованиях переполнение никогда не происходит и они всегда выполняются успешно, тогда как при сужающих преобразованиях может произойти потеря сведений и сбой.

В обоих случаях преобразование может быть явным (с идентификатором типа данных) или неявным (без идентификатора). Допустимые явные преобразования всегда выполняются успешно, даже если при этом происходит потеря данных. Неявные преобразования считаются успешными только в случае, если данные не потеряны; в противном случае преобразования оканчиваются сбоем и создаются ошибки компиляции или во время выполнения.

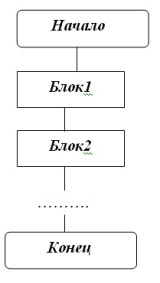
Преобразование с потерей точности происходит, когда для данных исходного типа не существует явного аналога в конечном типе. Например, строку "Fred" нельзя преобразовать в число. В этих случаях функция преобразования типов возвращает значение по умолчанию. Для типа **Number** значение по умолчанию — **NaN**; для типа **int** значение по умолчанию — нуль.

В некоторых случаях, например при преобразовании строк в числа, это занимает немало времени. Чем меньше преобразований применяется в программе, тем быстрее она работает.

<https://msdn.microsoft.com/ru-ru/library/hdz2a0he(v=vs.100).aspx>

1. **Линейные и разветвленные алгоритмы в языках программирования Pascal и C++. Операторы if и множественного выбора**

**Линейные алгоритмы**

Тип алгоритма определяется характером решаемой задачи в соответствии с его командами задачи. Различают три типа алгоритмов: линейные, разветвляющиеся, циклические.

**Линейными**называются алгоритмы, в которых все действия осуществляются последовательно друг за другом, при этом каждая команда выполняется только один раз строго после той команды, которая ей предшествует.

Таким, например, является алгоритм вычисления по простейшим безальтернативным формулам, не имеющий ограничений на значения входящих в эти формулы переменных. Как правило, линейные процессы являются составной частью более сложного алгоритма.

Линейный алгоритм составляется из команд присваивания, ввода, вывода и обращения к вспомогательным алгоритмам.

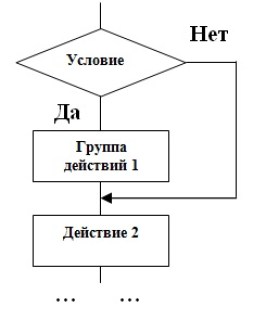
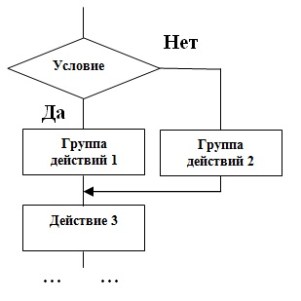
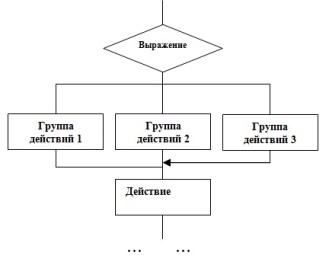
Основным элементарным действием в вычислительных алгоритмах является *присваивание значения переменной величине*. Если значение константы определено видом ее записи, то переменная величина получает конкретное значение только в результате присваивания.

**Присваивание** – это операция, которая значение выражения, стоящее справа от символа «=» запоминает в переменной или элементе массива, стоящем слева. При присваивании происходит преобразование типов данных, если они не совпадают.

Присваивание может осуществляться двумя способами:

* с помощью команды присваивания
* с помощью команды ввода
* Формат команды присваивания следующий:
* **Переменная := выражение**
* Знак «:=» нужно читать как «присвоить».
* Команда присваивания обозначает следующие действия, выполняемые компьютером:
* 1. вычисляется*выражение*;
* 2. полученное значение присваивается переменной.

три основных свойства команды присваивания:

* пока переменной не присвоено значение, она остается неопределенной;
* значение, присвоенное переменной, сохраняется в ней вплоть до выполнения следующей команды присваивания этой переменной;
* новое значение, присваиваемое переменной, заменяет ее предыдущее значение.
* **Разветвляющиеся алгоритмы**
* ***Разветвляющимся*** называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи, в зависимости от выполнения условий.
* Каждое из возможных направлений дальнейших действий называется ветвью.
* В блок-схемах разветвление реализуется специальным блоком «Решение». Этот блок предусматривает возможность двух выходов. В самом блоке «Решение» записывается логическое условие, от выполнения которого зависят дальнейшие действия.
* Общий вид команды ветвления в блок-схемах и на алгоритмическом языке следующий:
* **Различают несколько видов разветвляющихся алгоритмов:**
* 1. «*Обход*» — такое разветвление, когда одна из ветвей не содержит ни одного оператора, т.е. как бы обходит несколько действий другой ветви:
* 2.  «*Разветвление*» — такой тип разветвления, когда в каждой из ветвей содержится некоторый набор действий:
* 3. «*Множественный выбор*» — особый тип разветвления, когда каждая из нескольких ветвей содержит некоторый набор действий. Выбор направления зависит от значения некоторого выражения:

<http://prog.ytk.edu.ru/%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F-%D0%BB%D0%B5%D0%BA%D1%86%D0%B8%D1%8F-%E2%84%963/>

Операторы if

Оператор **if** управляет условным ветвлением. Тело оператора **if** выполняется, если значение выражения отлично от нуля. Существует две формы синтаксиса оператора **if**.

## **Синтаксис**

оператор-выбора:  
**if (**  выражение  **)**  оператор

**if (**  выражение  **)**  оператор  **else**  оператор

## **Оператор множественного выбора.**

Условный оператор **"switch case"** позволяет вам делать множественный выбор, когда количество вариантов очень много и писать для каждого свой if (else if) очень долго. Для этого придумали оператор "switch case", он имеет очень простой синтаксис:

// форма записи оператора множественного выбора switch

switch (/\*переменная или выражение\*/)

 {

 case /\*константное выражение1/\*:

 {

 /\*группа операторов\*/;

 break;

 }

 case /\*константное выражение2\*/:

 {

 /\*группа операторов\*/;

 break;

 }

 //.    .    .

 default:

                {

                 /\*группа операторов\*/;

                }

 }

1. **Организация циклов я языках программирования Pascal и C++. Цикл с предусловием, цикл с постусловием**

Операторы цикла используются для организации многократно повторяющихся вычислений. Любой цикл состоит из тела цикла, то есть тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла. Один проход цикла называется итерацией. Проверка условия выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле с предусловием), либо после тела цикла (цикл с постусловием). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу.

Переменные, изменяющиеся в теле цикла и используемые при проверке условия продолжения, называются параметрами цикла. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются **счетчиками цикла**.

**Цикл с предусловием в Pascal**  
В цикле с предусловием условие проверяется до шага цикла. Фактически, нужно вызывать приращение шага цикла в его теле. Этот цикл может быть и не зациклен, а просто выполняет некое действие, например проверку условия выхода из программы или иное действие.

Форма записи:

while   <условие выполнения>   do

begin

<оператор>;

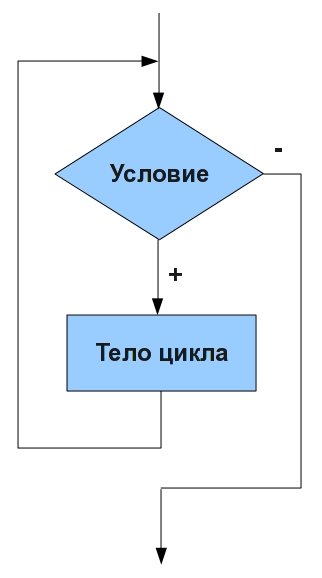
end;

Здесь while (пока), do {делать) - зарезервированные слова.

В качестве оператора может быть любой простой или составной оператор.

Блок-схема алгоритма работы такого оператора приведена на рис. 1.

Цикл while в Паскале применяется для создания повторений **с неизвестным заранее их числом**. Повторения ([итерации](http://labs.org.ru/pascal-2_3/#pascal2_4)) будут осуществляться, **пока истинно некоторое условие**.

Блок-схема, соответствующая циклу *while* в Паскале:  
[](http://labs.org.ru/wp-content/uploads/2016/09/1.jpg)Здесь оператор, стоящий после служебного слова do, образует тело цикла и будет выполняться, пока значение "условия" равно true (истина). Если операторов должно быть несколько, тогда необходимо применять оператор. Непосредственно условием цикла while может быть переменная или выражение. Операторы тела цикла while выполнятся только в том случае, если условие будет истинно, если условие ложно — они игнорируются, и программа продолжается с тех операторов, которые стоят уже после конструкции.

Пример 1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var x:Integer;  Begin  x:=1;    while (x<=10) do      begin        Write(x, ' ');        x:=x+1;      end;  End. |

.

## Repeat, или цикл с постусловием

**Repeat** — полная противоположность **while**. **Repeat** — это цикл, в котором условие стоит после тела .

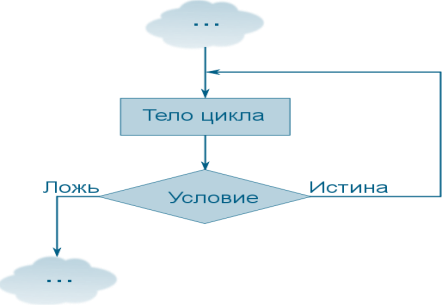
Операторы образуют тело цикла и будут выполняться до тех пор, пока значение "условие" не станет равно true, т.е. не станет истинным.  
Условием может быть переменная или выражение. Проверка условия осуществляется уже после первого прохождения операторов тела структуры, т.е. после первой итерации, т.о. цикл с постусловием в Паскаль обязательно выполнится хотя бы один раз, независимо от истинности условия.

Repeat имеет формат:

**repeat**{ повторяй … }  
<оператор 1>;  
< оператор 2>;  
…  
**until** {до…} <условие>

**Begin** и **end** не требуются.

блок-схема



Решение задачи. “Печатать «ноль» указанное количество раз”

**var** i,n:**integer**;

**begin**

write ('kolichestvo raz');

readln(n);

i:=1;

**repeat**

write(0);

i:=i+1

**until** i>n

**end**.

**Цикл с предусловием (while) в C++**  
  
Цикл с предусловием и имеет вид:  
**while ( выражение ) оператор**  
Выражение определяет условие повторения тела цикла, представленного простым или составным оператором. Выполнение цикла начинается с вычисления выражения. Если оно истинно (не равно false), выполняется оператор цикла. Если при первой проверке выражение равно false, цикл не выполнится ни разу. Тип выражения должен быть арифметическим или приводимым к нему. Выражение вычисляется перед каждой итерацией цикла. Когда выражение оказывается ложным, управление передается следующему после цикла оператору.  
Тело цикла может состоять не из одного оператора, а представлять собой блок.

Пример, в котором на экран выводятся значения переменной а, до тех пор пока ее значение не превышает 23:

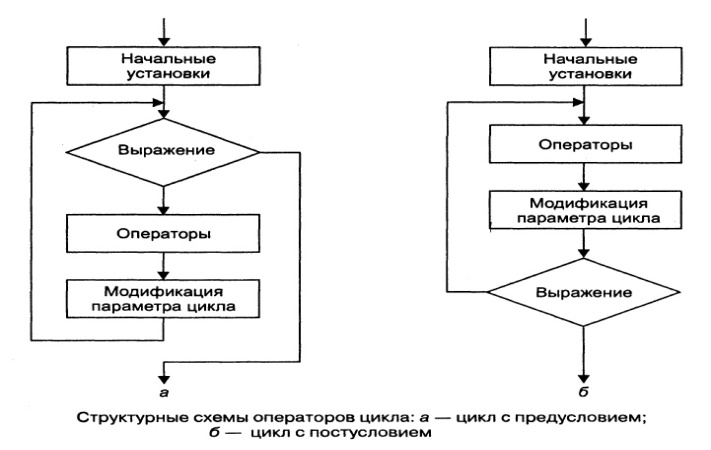
    int a = 1;   
    while( a < 23 )   
{   
    cout << a << "\n";   
    a += 4;   
}

**Цикл с постусловием (do … while) C++**  
  
Цикл с постусловием имеет вид:  
  
**do оператор while (выражение);**  
  
Сначала выполняется оператор или блок, составляющий тело цикла, а затем вычисляется выражение. Если оно истинно (не равно false), тело цикла выполняется еще раз. Цикл завершается, когда выражение станет равным false или в теле цикла будет выполнен какой-либо оператор передачи управления. Тип выражения должен быть арифметическим или приводимым к нему.

В примере

int a = 76;   
do   
{   
    cout << a << "\n";   
    a += 4;   
}   
    while( a < 23 )

на экран будет выведено число 76, несмотря на то что условие а < 23 явно не выполняется. В связи с этой особенностью, цикл с постусловием считается более опасным с точки зрения порождения возможных ошибок в программе, и если есть возможность, всегда лучше заменить его эквивалентным циклом с предусловием.



1. **Организация циклов в языках программирования Pascal и C++. Цикл с параметром. Операторы break, continue, goto, exit**

Ответ:

C++.Цикл с параметром

Цикл с управляющей переменной задается при помощи ключевого слова for. В таком цикле, будет выполнятся тело или нет, определяется путем проверки заданной управляющей переменной.

for( int i=5; i < 17; i++ )   
{   
    cout << i << "\n";   
}

В данном примере в качестве управляющей переменной задана переменная i, с начальным значением 5. В качестве условия прекращения цикла задано достижение управляющей переменной значения 17. В качестве действия над управляющей переменной задано увеличение ее на единицу при каждом выполнении тела цикла (оператор i++). В результате работы примера на экран будут выведены числа от 5 до 16 включительно. Цикл for, также как и цикл while, проверяет условие перед выполнением своего тела, т.е. если в последнем примере задать i=34, то тело цикла будет пропущено. В общем виде цикл for записывается так:

for (описание управляющей переменной; проверка условия; действие с управляющей переменной)

    {   
    тело цикла;   
}

Здесь приведена каноническая форма записи цикла for. На самом деле вариантов записи его очень много. В частности, описание управляющей переменной можно вынести за пределы цикла:

int i = 5;   
...   
for( ; i < 17; i++ )   
{   
    ...   
}

Pascal. Цикл с параметром

Оператор цикла **for** служит для организации цикла с известным чис­лом повторений и имеет две формы записи.

Первая форма оператора цикла с параметром имеет вид

**for < параметр\_цикла > := < нач\_знач > to < кон\_знач > do < тело цикла >;**

Вторая форма оператора цикла с параметром имеет вид

**for < параметр\_цикла >:= < нач\_знач > downto < кон\_знач > do < тело цикла >;**

Обе формы записи означают, что "для (**for**) значений параметра цикла от начального (< нач\_знач >) до (**to** или **downto**) конечного (< кон\_знач >) необходимо выполнять (**do**) действия операторов, образующих тело цикла"

Параметр цикла, его начальное и конечное значения должны иметь одинаковый тип. В качестве параметра цикла (управляющей переменной цикла) можно использовать любые данные из группы порядковых типов, кроме переменной вещественного типа. То есть в качестве параметра цикла могут быть целые, символьные, булевы, перечисляемые переменные и переменные тип-диапазон.

**Оператор break** имеет два назначения. Первое - это окончание работы оператора switch. Второе - это принудительное окончание цикла, минуя стандартную проверку условия. Когда оператор break встречается в теле цикла, цикл немедленно заканчивается и выполнение программы переходит на строку, следующую за циклом. Оператор break, как правило, используется в циклах, где особые условия могут вызвать немедленное завершение работы.

***continue*** — этот оператор, предназначенный для завершения текущей итерации цикла и переходу к концу тела цикла. Исходя из определения, если в цикле программы встречается оператор **continue**, то все последующие за ним операторы пропускаются, осуществляя переход к концу цикла. Далее все зависит от вида цикла – с параметром или с условием. Если цикл с параметром (цикл for) или цикл с предусловием (while), то сразу переходим в начало и проверяем условие входа в цикл; если цикл с послеусловием (repeat), то проверяется условие выхода из цикла.

*exit* — это оператор, предназначенный для досрочного выхода из процедуры или функции и возвращения в основную программу. Вызов оператора exit в основной программе приводит к её завершению. оператор exit можно использовать и вне цикла – и в этом его отличие от операторов break и continue, которые, как отмечалось, вызываются только внутри цикла.

*goto* — оператор безусловного перехода (перехода к определённой точке программы, обозначенной номером строки либо меткой. Как правило, оператор goto состоит из двух частей: собственно, оператора и метки, указывающей целевую точку перехода в программе: goto *метка*. Метка, в зависимости от правил языка, может быть либо числом (как, например, в классическом Бейсике), либо идентификатором используемого языка программирования. Для меток-идентификаторов метка, как правило, ставится перед оператором, на который должен осуществляться переход, и отделяется от него двоеточием (*метка*:).

Действие оператора перехода состоит в том, что после его исполнения следующими будут исполняться операторы программы, идущие в тексте непосредственно после метки (до следующего оператора перехода, ветвления или цикла).

1. **Создание одномерных массивов в языках программирования Pascal и C++, особенности работы с ними**

***Массив*** – это составной объект, образованный из элементов (*компонент*) одного и того же типа. Такой тип данных применяется в программировании для обозначения объектов, аналогичных числовым последовательностям в математике, где сразу целая группа чисел обозначается одним именем (чаще всего буквой), а для обращения к каждому отдельному числу данной последовательности используются различные *индексы* (номера элементов). В математике это может выглядеть, например, так: *a1, а2, а3, …, аn.*

Таким образом, в программировании *массив* – это последовательность однотипных элементов, имеющих общее имя, причем каждый элемент этой последовательности определяется порядковым номером (индексом) элемента.

***Одномерный массив*** – это фиксированное количество элементов одного типа, объединенных одним именем, причем каждый элемент имеет свой уникальный номер, и номера элементов идут подряд.

*В pascal (см. ссылку*<http://festival.1september.ru/articles/594645/>***)***

Тип *массив* описывается следующим образом:

*Type*

*Имя типа = Array [тип индекса (ов)] Of  тип элементов;*

*Var*

*Имя переменной: имя типа;*

Переменную типа *массив* можно описать сразу в разделе описания переменных Var:

*Var   Имя переменной: array [тип индекса (ов)] Of  тип элементов;*

Здесь

* *Array* – служебное слово (в переводе с английского означает «массив»);
* *Of* – служебное слово (в переводе с английского означает «из»).
* Тип индекса – любой порядковый тип, кроме типов *integer, longint.*
* Тип же самих элементов может быть любым, кроме *файлового* типа.
* Количество элементов массива называется его *размерностью****.*** Несложно подсчитать, что при последнем способе описания множества индексов *размерность массива равна: максимальное значение индекса – минимальное значение индекса + 1.*

Ввод и вывод массивов осуществляется поэлементно. Для ввода массива можно использовать любой цикл.

В С++ Нумерация ячеек массива всегда начинается с 0. Индекс ячейки – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

пример объявления одномерного массива:

int a[16] где, int—целочисленный [тип данных](http://cppstudio.com/obuchenie_cpp/tipy_dannyh);

а —  имя одномерного массива;  
16 — размер одномерного массива, 16 ячеек.

1. Массив является представителем *структурированного типа данных* в языке С++.
2. Элементы массива имеют одинаковые имя, тип и располагаются в памяти последовательно.
3. Элементы массива характеризуются индексами, значениями и адресуемой памятью.
4. Существует две основные формы объявления массивов: с указанием и без указания размера. Безразмерный массив объявляется, если: он инициализируется при объявлении, является *формальным параметром* функции, объявлен как ссылка на массив.
5. Инициализация массива заключается в присваивании начальных значений его элементам. В программах часто используется генерация массивов.
6. Для массивов нельзя выполнить операцию прямого присваивания.
7. Адресация элементов массива осуществляется с помощью индексированного имени. Обращаться к элементам массива можно также посредством механизма указателей.
8. Размер памяти, занимаемой массивом, зависит от реализации и вычисляется с помощью *операции sizeof*.
9. Массивы используются для решения прикладных задач.
10. **Методы сортировки одномерных массивов в языках программирования Pascal и C++**

**Сортировка выбором**основана на определении наибольшего (наименьшего) элемента, который переносится в начало или конец массива в зависимости от вида сортировки (по возрастанию или по убыванию). Затем эта процедура применяется ко всем оставшимся элементам, кроме уже перемещенных элементов, всего (N - 1 ) раз. Приведем пример операторов для сортировки элементов массива “Х” по возрастанию:

**for j: = 1 to N-1 do begin**{ цикл по числу "проходов" }

**k:= N-j+1;**{ k - номер последнего элемента в проверяемой части массива }

**m:= k;**{ m - номер элемента с наибольшим значением }

**for i:= 1 to N-j do**{цикл сравнения элементов в оставшейся части массива}

**if x[i] > x[m] then m: = i;**{ запоминаем значение "m" }

**b:= x[k]; x[k]:= x[m]; x[m]:= b**{ переставляем элементы }

**end;**

Здесь полагается, что последний элемент, расположенный в сортируемой части массива, имеет наибольшее значение. Это условие проверяется для оставшейся части массива и запоминается номер (индекс) элемента с действительно наибольшим значением. Затем производится перестановка наибольшего элемента с последним элементом в проверяемой части массива. Далее процесс повторяется с уменьшением числа рассматриваемых элементов на единицу.

**Сортировка обменом**(метод пузырька) основана на последовательном сравнении пары соседних элементов x[i] и x[i+1]. Если пара расположена не в требуемом порядке, то элементы переставляются. Например, при сортировке по возрастанию после первого "прохода" массива от первого до последнего элемента на последнем месте окажется наибольший элемент массива. Далее сортируется оставшаяся часть массива. С каждым очередным "проходом" наибольший элемент массива в оставшейся части массива будет занимать последнее место в проверяемой части массива. Наибольшее число проходов **j**равно "N - 1", причем число проверок при очередном проходе уменьшается на единицу:

**for j:= 1 to N-1 do**{ цикл по числу "проходов" }

**for i:= 1 to N-j do**{ цикл сравнения элементов в оставшейся части массива }

**if x[i] > x[i+1] then begin**{запоминаем значение x[i] и }

**b:=x[i]; x[i]:=x[i+1]; x[i+1]:=b end;**{ переставляем элементы }

Поскольку при сортировке сравниваются каждые два соседних элемента массива, то для упорядочения данных общее число "проходов" может быть меньше, чем "N - 1". Избежать лишних проходов можно используя оператор цикла с условием:

**j:= 0;**

**repeat j:= j+1; pr:= 0;**{ pr - признак необходимости "прохода" }

**for i:= 1 to N-j do**{цикл сравнения элементов в проверяемой части массива}

**if x[i] > x[i+1] then begin pr:= 1;**{ изменяем значение признака }

**b:=x[i]; x[i]:=x[i+1]; x[i+1]:=b end;**{ переставляем элементы }

**until pr = 0;**

Если при проходе проверяемой части массива не было перестановок, то pr=0 и процесс заканчивается. Оптимальным с математической точки зрения считается алгоритм с наименьшим числом перестановок. Однако при программировании необходимо учитывать также, что время выполнения логических операций, как правило, значительно превышает время выполнения арифметических операций. Таким образом, время выполнения программы определяется не только числом перестановок, но существенно зависит от количества выполнений логических операций.

**Сортировка вставками**основана на внедрении в отсортированную часть массива элемента, следующего за этой частью, если он удовлетворяет условию сортировки. На первом шаге сортировки второй элемент сравнивается с первым, на втором шаге третий элемент сравнивается с двумя первыми и т. д. Среди уже отсортированных i - 1 элементов массива вставляют i - й элемент без нарушения порядка, т. е. при вставке i - го элемента на j - е место (j < i) элементы с индексами >j и <i увеличивают свой номер на единицу. Приведем пример операторов для сортировки данных по возрастанию:

**for i := 2 to N do begin**{ цикл по числу шагов }

**b:=x[i];**{значение элемента, следующего за отсортированной частью массива}

**j:= 1;**

**while b > x[j] do j:= j+1;**{ определение номера j для вставки элемента}

**for k:=i downto j+1 do x[k]:=x[k-1];**{ увеличение индексов элементов }

**x[j]:= b**{ вставка значения b на место j - го элемента }

**end;**

В отличие от рассмотренных методов сортировки здесь процесс сравнения элементов заканчивается, как только вставляемый элемент удовлетворяет условию сортировки, т. к. элемент вставляется в отсортированную часть массива.

1. **Создание многомерных массивов в языках программирования Pascal и C++, особенности работы с ними**

**Многомерные массивы в C++**

Определение автоматических многомерных массивов почти полностью совпадает с определением одномерных массивов (смотри 14 вопрос), за исключением того, что вместо одного размера может быть указано несколько:

**const** **unsigned** **int** DIM1 = 3;

**const** **unsigned** **int** DIM2 = 5;

**int** ary[DIM1][DIM2];

этом примере определяется двумерный массив из 3 строк по 5 значений типа int в каждой строке. Итого 15 значений типа int.

При статической (определяемой на этапе компиляции) инициализации значения C-массива перечисляются в порядке указания размеров (индексов) в определении массива. Каждый уровень (индекс), кроме самого младшего, многомерного массива заключается в свою пару фигурных скобок. Значения самого младшего индекса указываются через запятую:

**const** **unsigned** **int** DIM1 = 3;

**const** **unsigned** **int** DIM2 = 5;

**int** ary[DIM1][DIM2] =

{

{ 1, 2, 3, 4, 5 },

{ 2, 4, 6, 8, 10 },

{ 3, 6, 9, 12, 15 }

};

Многомерный массив заполняется значениями с помощью вложенных циклов. Причём, как правило, количество циклов совпадает с размерностью массива: Для многомерного C-массива выделяется единый блок памяти необходимого размера: размер\_массива1 \* размер\_массива2 \* ... \* размер\_массиваN \* sizeof(тип\_элемента\_массива).

Значения располагаются последовательно. Самый левый индекс изменяется медленнее всего. Т.е. для трёхмерного массива сначала располагаются значения для первой (индекс 0) матрицы, затем для второй и т.д. Значения для матриц располагаются построчно (ср. со статической инициализацией массива выше).

Имя (идентификатор) многомерного C-массива является указателем на первый элемент массива (так же, как и для одномерных массивов)

В Pascal

Рассмотрим многомерные массивы в Паскале. Например, двухмерный массив можно объявить таким образом:

Var X1: array [1..10] of array [1..5] of integer;   
  
Этот оператор описывает двухмерный массив, который можно представить себе как таблицу, которая состоит из 10 строк и 5 столбцов. Также это можно объявить более компактно:

VarX1:[1..10,1..5] ofinteger;

Обычно используется именно такая форма объявления многомерных массивов. Как и в одномерных массивах, элементы и индексы могут иметь любой тип. Доступ к значениям элементов многомерного массива осуществляется через индексы, которые перечисляются через запятую. Например, **X1[3,4]** – значение элемента, лежащего на пересечении третьей строки и четвертого столбца.

Можно определить не непосредственно переменные типа многомерных массивов, а сначала определить соответствующий тип, а потом – переменные или типизированные константы данного типа как у одномерных массивов. Пример:

type Ar3 = array[1..4,1..3,1..2] of integer;

var A1, A2, A3;

const A3: Ar3 = (((0,1),(2,3),(4,5)),((6,7),(8,9),(10,11)),((12,13),(14,15),(16,17)),((18,19),(20,21),(22,23)));

Приведенный выше пример типизированной константы создает массив **A3**

1. **Особенности сортировки матриц в языках программирования Pascal и C++**

Если количество строк массива равно количеству столбцов, то такой массив называется квадратной матрицей. При работе с квадратной матрицей, в отличие от обычного двумерного массива, можно выделить:

· диагонали (главная, побочная);

· элементы, расположенные над и под диагоналями;

http://mirznanii.com/images/02/23/9212302.gif· четверти матрицы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 13 | 14 | 15 |
| 21 | 22 | 23 | 24 | 25 |
| 31 | 32 | 33 | 34 | 35 |
| 41 | 42 | 43 | 44 | 45 |
| 51 | 52 | 53 | 54 | 55 |

Здесь первая цифра номера элемента обозначает номер строки матрицы (i), вторая цифра – номер столбца (j)

Для определения элементов, входящих в любой из перечисленных разделов, существует формула, основными составляющими которой являются i – номер строки, j – номер столбца и N – размерность массива. Например, для определения элемента с номером 43, расположенного под побочной диагональю можно использовать формулу **i+j>N+1**, где i=4, j=3, N=5, таким образом, получаем 4+3>5+1.

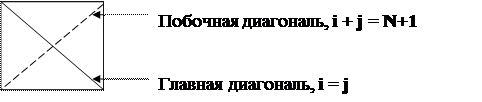
**4.1 Определение диагоналей массива** 

Рисунок 4. Диагонали двумерного массива

Таким образом, в матрице, представленной в п. 4, элементы с номерами 11, 22, 33, 44 и 55 являются элементами главной диагонали. Элементы с номерами 15, 24, 33, 42, 51 – элементы побочной диагонали.

Расположение элементов, находящихся над или под диагональю, определяется по отношению к одной из диагоналей.

|  |  |
| --- | --- |
| http://mirznanii.com/images/04/23/9212304.gif | http://mirznanii.com/images/05/23/9212305.gif |

Рисунок 5. Расположение элементов по отношению к диагоналям

Элементы 12, 13, 14, 15, 23, 24, 25, 34, 35 и 45 расположены над главной диагональю, 21, 31, 32, 41, 42, 43, 51, 52, 53, 54 расположены под главной диагональю. Элементы 11, 12, 13, 14, 21, 22, 23, 31, 32 и 41 расположены над побочной диагональю, 25, 34, 35, 43, 44, 45, 54, 53, 54, 55 расположены под побочной диагональю.

**4.2 Определение четвертей матрицы**

Относительно обеих диагоналей элемент массива может находиться в одной из четвертей.

12, 13, 14, 23 – элементы первой четверти

25, 34, 35, 45 – элементы второй четверти

43, 52, 53, 54 – элементы третьей четверти

21, 31, 32, 41 – элементы четвертой четверти

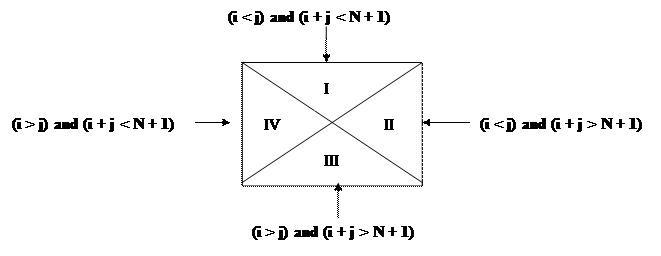


Рисунок 6. Определение четвертей матрицы

Используя правила, представленные на рисунке 6, очень легко можно программным путем формировать матрицы требуемого вида.

1. **Строка как массив символов. Ввод/вывод строк, функции работы со строками**

В С++

   Источник: <http://www.itmathrepetitor.ru/s-lekciya-4-rabota-so-strokami/>

Строкой называется массив символов, который заканчивается пустым символом ‘\0’. Строка объявляется как обычный символьный массив, например,                char  s1[10];         // строка длиной в девять символов                 char  \*s2;              // указатель на строку Различие между указателями s1 и s2 заключается в том, что указатель s1 является именованной константой, а указатель s2 – переменной.                 Строковые константы заключаются в двойные кавычки в отличие от символов, которые заключаются в одинарные кавычки. Например,               “This is a string.” Длина строковой константы не может превышать 509 символов по стандарту. Однако, многие реализации допускают строки большей длины.                 При инициализации строк размерность массива лучше не указывать, это выполнит компилятор, подсчитав длину строки и добавив к ней единицу. Например, char  s1[ ] = “This is a string.”;

Ввод-вывод строк.                 Для ввода строки с консоли служит функция                 char\*  gets ( char  \*str ); которая записывает строку по адресу str и возвращает адрес введенной строки. Функция прекращает ввод, если встретит символ ‘\n’ или EOF (конец файла). Символ перехода на новую строку не копируется. В конец прочитанной строки помещается нулевой байт. В случае успеха функция возвращает указатель на прочитанную строку, а в случае неудачи NULL.                 Для вывода строки на консоль служит стандартная функция                 int  puts ( const  char  \*s ); которая в случае удачи возвращает неотрицательное число, а в случае неудачи – EOF. Прототипы функций gets и puts описаны в заголовочном файле stdio.h.

Существуют варианты функций scanf и printf, которые предназначены для форматирования строк и называются соответственно sscanf и sprintf.                 Функция                 int  sscanf ( const  char  \*str, const  char  \*format, …); читает данные из строки, заданной параметром str, в соответствии с форматной строкой, заданной параметром format. В случае удачи возвращает количество прочитанных данных, а в случае неудачи – EOF

Функция                 int  sprintf (char  \*buffer, const  char  \*format, …); форматирует строку в соответствии с форматом, который задан параметром format и записывает полученный результат в символьный массив buffer. Возвращает функция количество символов, записанных в символьный массив buffer, исключая завершающий нулевой байт.

. Сравнение строк. Для сравнения строк используются функции strcmp и strncmp.                 Функция                 int  strcmp ( const  char  \*str1, const  char  \*str2); лексикографически сравнивает строки str1, str2 и возвращает –1, 0 или 1, если строка str1 соответственно меньше, равна или больше строки str2.                 Функция int strncmp ( const  char  \*str1, const  char  \*str2, size\_t  n); лексикографически сравнивает не более чем n первых символов из строк str1 и str2. Функция возвращает –1, 0 или 1, если первые n символов из строки str1 соответственно меньше, равны или больше первых n символов из строки str2.

  Копирование строк. Для копирования строк используются функции strcpy и strncpy.                 Функция                 char  \*strcpy ( char  \*str1, const  char  \*str2 ); копирует строку str2 в строку str1. Строка str2 копируется полностью, включая завершающий нулевой байт. Функция возвращает указатель на str1. Если строки перекрываются, то результат непредсказуем.                 Функция                 char  \*strncpy ( char  \*str1, const  char  \*str2, size\_t  n ); копирует n символов из строки str2 в строку str1. Если строка str2 содержит меньше чем n символов, то последний нулевой байт копируется столько раз, сколько нужно для расширения строки str2 до n символов. Функция возвращает указатель на строку str1.

**Строковый тип данных паскаль**

Для обработки строковой информации в Турбо Паскаль введен строковый тип данных. Строкой в Паскале называется последовательность из определенного количества символов. Количество символов последовательности называется длиной строки. Синтаксис:

var s: string[n];

var s: string;

n - максимально возможная длина строки - целое число в диапазоне 1..255. Если этот параметр опущен, то по умолчанию он принимается равным 255.

Строковые константы записываются как последовательности символов, ограниченные апострофами. Допускается формирование строк с использованием записи символов по десятичному коду (в виде комбинации # и кода символа) и управляющих символов (комбинации ^ и некоторых заглавных латинских букв).

Пример:

'Текстовая строка'  
#54#32#61  
'abcde'^A^M

Пустой символ обозначается двумя подряд стоящими апострофами. Если апостроф входит в строку как литера, то при записи он удваивается.

Переменные, описанные как строковые с разными максимальными длинами, можно присваивать друг другу, хотя при попытке присвоить короткой переменной длинную лишние символы будут отброшены.

Выражения типа char можно присваивать любым строковым переменным.

**Процедуры и функции для работы со строками**

В системе Turbo Pascal имеется несколько полезных стандартных процедур и функций, ориентированных на работу со строками. Ниже приводится список этих процедур и функций с краткими пояснениями.

Length(s:string):integer

Функция возвращает в качестве результата значение текущей длины строки-параметра

*Пример.*

n := length('Pascal'); {n будет равно 6}

Concat(s1,[s2,...,sn]:string):string

Функция выполняет слияние строк-параметров, которых может быть произвольное количество. Каждый параметр является выражением строкового типа. Если длина строки-результата превышает 255 символов, то она усекается до 255 символов. Данная функция эквивалентна операции конкатенации "+" и работает немного менее эффективно, чем эта операция. 

Copy(s:string; index:integer; count:integer):string

Функция возвращает подстроку, выделенную из исходной строки s, длиной count символов, начиная с символа под номером index.

Delete(var s:string; index,count:integer)

Процедура удаляет из строки-параметра s подстроку длиной count символов, начиная с символа под номером index.

Insert(source:string; var s:string;index:integer)

Процедура предназначена для вставки строки source в строку s, начиная с символа index этой строки.

Pos(substr,s:string):byte

Функция производит поиск в строке s подстроки substr. Результатом функции является номер первой позиции подстроки в исходной строке. Если подстрока не найдена, то функция возвращает 0.

Str(X: *арифметическое выражение*; var st: string)

Процедура преобразует численное выражение X в его строковое представление и помещает результат в st. 

Val(st: string; x: *числовая переменная*; var code: integer)

Процедура преобразует строковую запись числа, содержащуюся в st, в числовое представление, помещая результат в x. x - может быть, как целой, так и действительной переменной. Если в st встречается недопустимый (с точки зрения правил записи чисел) символ, то преобразование не происходит, а в code записывается позиция первого недопустимого символа. Выполнение программы при этом не прерывается, диагностика не выдается. Если после выполнения процедуры code равно 0, то это свидетельствует об успешно произошедшем преобразовании.

В дополнение приведем некоторые функции, связанные с типом char, но которые тем не менее часто используются при работе со строками. 

Chr(n: byte): char

Функция возвращает символ по коду, равному значению выражения n. Если n можно представить, как числовую константу, то можно также пользоваться записью #n. 

Ord(ch: char): byte;

В данном случае функция возвращает код символа ch. 

UpCase(c: char): char;

Если c - строчная латинская буква, то функция возвращает соответствующую прописную латинскую букву, в противном случае символ c возвращается без изменения.

1. **Пользовательские типы данных в языке программирования C++. Структуры - особенности создания и работы. Понятие псевдонима структуры. Вложенные структуры, массивы структур**

**Тип данных** – это множество допустимых значений, которые может принимать тот или иной *объект*, а также множество допустимых операций, которые применимы к нему. В современном понимании тип также зависит от внутреннего представления информации.

Таким образом, данные различных типов хранятся и обрабатываются по-разному. *Тип данных* определяет:

* внутреннее представление данных в памяти компьютера;
* объем памяти, выделяемый под данные;
* множество (диапазон) значений, которые могут принимать величины этого типа;
* операции и функции, которые можно применять к данным этого типа.

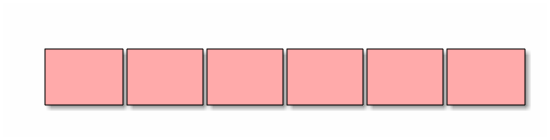
Исходя из данных характеристик, необходимо определять тип каждой величины, используемой в программе для представления объектов. Обязательное описание типа позволяет компилятору производить *проверку допустимости* различных конструкций программы. От выбора типа величины зависит последовательность машинных команд, построенная компилятором.

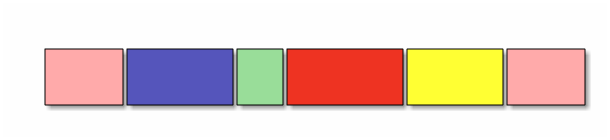
## Составные типы

### Структуры

#### Записи

**Структура (запись)** представляет собой набор данных, хранящихся в памяти в смежных адресах, но не обязательно принадлежащих одному типу. Это позволяет рассматривать саму структуру как универсальный тип для представления внутреннего устройства множества объектов.Изображение массива:

Изображение структуры (записи):



#### Фундаментальные структуры данных

Замечательно то, что массивы и записи можно **объединять**, создавая:

* Массивы записей.
* Записи, содержащие массивы.

На основе массивов и записей строят:

* **Связанный список** - набор элементов одного типа, не обязательно следующих в памяти друг за другом и связанных между собой благодаря хранению адресов.
* **Графы** - множество вершин (узлов), соединённых рёбрами.
* **Деревья** - иерархически связанные элементы данных, частный случай графа.
* **Хэш-таблицы (ассоциативные массивы)** - наборы ‘’ключ-значение’‘.

#### Определение структуры

Сначала мы описываем новый структурный тип:

struct TAG

{

type1 field1;

type2 field2;

...

};

Потом создаём объекты:

**struct** TAG obj1, obj2,.. **\***pobj,arr[N];

**struct** BOOK

{

**char** title[40];

**char** author[30];

**int** pages;

**float** price;

};

**struct** BOOK mybook;

**typedef** **struct**

{

**char** title[40];

**char** author[30];

**int** pages;

**float** price;

} BOOK;

BOOK mybook;

#### Инициализация

Вариант 1.

**struct** BOOK b1**=**{"The C programming",

"K & R",

300,

100.23 };

Вариант 2.

**struct** BOOK

{

**char** title[40];

**char** author[30];

**int** pages;

**float** price;

}

b1 **=** {"The C programming",

"K & R",

300,

100.23 };

#### Свойства структур

* В памяти все поля структуры располагаются последовательно.
* Память под структуру выделяется при объявлении переменных.
* Доступ к полям производится с помощью операций **.** и **->**.
* Полем структуры может быть указатель на структуру.

#### Доступ к значениям полей

Для доступа к полю через структурный объект используется ‘’точка’‘

**struct** BOOK book, **\***pbook**=&**book;

book.price**=**123.50;

strcpy(book.title,"The C programming");

pbook**->**pages**=**300;

strcpy(pbook**->**author,"K&R");

Для доступа к полю через адрес объекта используется ‘’стрелочка’‘

**struct** BOOK book, **\***pbook**=&**book;

(**&**book)**->**price**=**123.50;

strcpy((**&**book)**->**title,"The C programming");

(**\***pbook).pages**=**300;

strcpy((**\***pbook).author,"K&R");

#### Вложенные структуры

Структуры можно вкладывать друг в друга (‘’матрёшки’‘)

**struct** TIME

{

**int** hh;

**int** mm;

**int** ss;

};

**struct** EVENT

{

**char** title[256];

**struct** TIME when;

};

**struct** EVENT event, **\***pevent**=&**event;

strcpy(event.title,"begin");

event.when.hh**=**18;

event.when.mm**=**0;

event.when.ss**=**0;

strcpy(pevent**->**title,"begin");

pevent**->**when.hh**=**18;

pevent**->**when.mm**=**0;

pevent**->**when.ss**=**0;

#### Присвоение структур

Допускается присваивать структурные объекты как обычные переменные при соблюдении следующих условий:

1. Объекты одного типа
2. В объекте нет полей-указателей

Если существуют поля-указатели, то после присваивания могут возникнуть двойные ссылки на один и тот же участок памяти.

Как же присваивать структуры? Через присвоение значения поля другому полю. При встрече с указателями нужно выделить память под копию данных, затем скопировать данные из одной области памяти в другую.

**struct** BOOK

{

**char** title[40];

**char** author[40];

**float** price;

};

**struct** BOOK book1**=**{"The C Programming",

"K&R", 300.0};

**struct** BOOK book2;

book2**=**book1; *// поэлементное копирование*

*// мы меняем только book1, но не book2!*

strcpy(book1.title,"The C++ Programming");

**struct** BOOK

{

**char** **\***title;

**char** **\***author;

**float** price;

};

**struct** BOOK book1**=**{"The C Programming",

"K&R", 300.0};

**struct** BOOK book2;

book2**=**book1; *// копируем ссылку!*

*// мы одновременно пытаемся поменять и book1 и book2*

*// а также "залезаем" в статическую память, вызывая сбой*

strcpy(book1.title,"The C++ Programming");

#### Структуры и функции

Структуры можно передавать в функцию как по **значению**, так и по **указателю**.

**void** **printBook**(**struct** BOOK book)

{

printf("Title: %s \nAuthor: %s \nPrice: %.2f\n",

book.title,book.author,book.price);

}

**void** **enterBook**(**struct** BOOK **\***book)

{

fgets(book**->**title,40,stdin);

fgets(book**->**author,40,stdin);

scanf("%f",**&**book**->**price);

}

В отличии от массивов, структуры можно возвращать из функций:

**struct** BOOK **printBook**(**struct** BOOK book)

{

printf("Title: %s \nAuthor: %s \nPrice: %.2f\n",

book.title,book.author,book.price);

**return** book;

}

#### Структуры и указатели

Структура может содержать указатель на себя в качестве поля:

**struct** BOOK

{

**char** **\***title;

**char** **\***author;

**float** price;

**struct** BOOK **\***next;

};

Благодаря полю **next** можно связывать структурные объекты между собой (создавать связанные списки).

1. **Пользовательские типы данных в языке программирования C++. Объединения и перечисления - особенности создания и работы**

**Объединения (union)**

Объединение (union) представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры, только вместо ключевого слова struct используется слово union. Длина объединения равна наибольшей из длин его полей, В каждый момент времени в переменной типа объединение хранится только одно значение, и ответственность за его правильное использование лежит на программисте.

Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется:

#include

int niain(){

  enum paytype {CARD, CHECK};

  paytype ptype;

  union payment{

    char card[25];

    long check;

  } info;

*/\* присваивание значений info и ptype \*/*

  switch (ptype){

    case CARD: cout << "Оплата no карте: " << info.card; break;

    case CHECK: cout << "Оплата чеком: " << info.check; break;

  }

  return 0;

}

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно:

#include

int niain(){

  enum paytype {CARD, CHECK};

  struct{

  paytype ptype;  
    union payment{

      char card[25];

      long check;

    }

  } info;

*/\* присваивание значений info и ptype \*/*

  switch (ptype){

    case CARD: cout << "Оплата no карте: " << info.card; break;

    case CHECK: cout << "Оплата чеком: " << info.check; break;

  }

  return 0;

}

Объединения применяются также для разной интерпретации одного и того же битового представления (но, как правило, в этом случае лучше использовать явные операции преобразования типов). В качестве примера рассмотрим работу со структурой, содержащей битовые поля:

struct Options{  
bool centerX:1;  
bool centerY:1;  
unsigned int shadow:2;  
unsigned int palette:4;  
}  
union{  
unsigned char ch;  
Options bit;  
}option = {0xC4};  
cout << option.bit.palette;  
option.ch &= 0xF0; // наложение маски

По сравнению со структурами на объединения налагаются некоторые ограничения. Смысл некоторых из них станет понятен позже:

* объединение может инициализироваться только значением его первого элемента;
* объединение не может содержать битовые поля;
* объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;
* объединение не может входить в иерархию классов.

**Перечисления (enum)**

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант. Формат:

*enum [ имя\_типа ] { список\_констант };*

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

enum Err {ERR\_\_READ, ERR\_\_WRITE, ERR\_CONVERT};  
Err error;  
switch (error){  
case ERR\_READ: */\* операторы \*/* break;  
case ERR\_WRITE: */\* операторы \*/* break;  
case ERR\_CONVERT: */\* операторы \*/* break;  
}

Константам ERR\_READ, ERR\_WRITE, ERR\_CONVERT присваиваются значения 0, 1 и 2 соответственно.

Другой пример:

enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};

Константам three и four присваиваются значения 3 и 4, константе eleven — 11.

Имена перечисляемых констант должны быть уникальными, а значения могут совпадать. Преимущество применения перечисления перед описанием именованных констант и директивой #define состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов.

При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить собственные операции.

**Примечание**

Диапазон значений перечислепия определяется количеством бит, необходимым для представления всех его значений. Любое значение целочисленного типа можно явно привести к типу перечислепия, по при выходе за пределы его диапазона результат не определен.

1. **Структура жесткого диска. Файловые системы DOS и NTFS**

**Физическая структура жесткого диска.**

Физическая структура жёстких дисков предполагает разбиение диска на рабочие поверхности, дорожки и секторы, номера которых образуют адрес, по которому на диске хранится информация.

Сектора

Любой жёсткий диск можно представить как огромный «чистый лист», на который можно записывать данные и откуда потом их можно считать. Чтобы ориентироваться на диске, всё его пространство разбивают на небольшие «клеточки» — сектора. Сектор — это минимальная единица хранения данных на диске, обычно его размер составляет 512 байт. Все сектора на диске нумеруются: каждый из n секторов получает номер от 0 до n–1. Благодаря этому любая информация, записанная на диск, получает точный адрес — номера соответствующих секторов. Так что диск ещё можно представить как очень длинную строчку (ленточку) из секторов. Можете посчитать, сколько секторов на вaшем диске размером в N гигабайт.

Разделы

Представлять жёсткий диск как единый «лист» не всегда бывает удобно: иногда полезно «разрезать» его на несколько независимых листов, на каждом из которых можно писать и стирать что угодно, не опасаясь повредить написанное на других листах. Логичнее всего записывать раздельно данные большей и меньшей важности или просто относящиеся к разным вещам.

Конечно, над жёстким диском следует производить не физическое, а логическое разрезание, для этого вводится понятие раздел (partition). Вся последовательность (очень длинная ленточка) секторов разрезается на несколько частей, каждая часть становится отдельным разделом. Фактически, нам не придётся ничего разрезать (да и вряд ли бы это удалось), достаточно объявить, после каких секторов на диске находятся границы разделов.

**Логическая структура жесткого диска.**

http://www.thalion.kiev.ua/idx.php/93/760/article/ //можешь почитать

Логическая структура жесткого диска – это способ организации информации. Если кратко, чтобы понять принцип, винчестер следует сопоставить с листом бумаги в клеточку. Каждая клеточка – это сектор, он содержит 0,5 Кб информации. Каждый сектор имеет свой порядковый номер. Когда мы обращаемся к какому-либо файлу, головка ищет по всей площади диска определенный сектор и обращается к нему.

Чтобы не было большой путаницы, винчестер разделяют на разделы. При этом никакого физического разделения не происходит. Все – образно, для удобства.

Запись данных на жесткие диски осуществляется не хаотически, а по определенным правилам и стандартам. На магнитную поверхность наносятся специальная разметка и другая служебная информация, например, создаются закрытые сектора, отвечающие за загрузку операционной системы и т.п. Т.е. создается так называемая логическая структура жесткого диска. Благодаря стандартизации логической структуры операционные системы могут работать с жесткими дисками разной емкости от различных производителей. Любая операционная система предлагает пользователем функцию форматирования жесткого диска (на профессиональном языке - высокоуровневое форматирование). Это самый простой и общедоступный вариант влияния на логическую структуру жесткого диска, который затрагивает только малую часть из необходимой служебной информации.

Логическая структура жёсткого диска предполагает деление общего дискового пространства на области, каждая из которых хранит специфическую информацию: MBR (Master Boot Record), BR (Boot Record), FAT1 и FAT2, Root Directory и область для данных.

***Файловая система* (*file system*) – способ организации данных в виде файлов на устройствах внешней памяти (жестких и оптических дисках, устройствах флеш-памяти и т. п.).**

***Файловая система* должна обеспечивать:**

1. **безопасное и надежное хранение данных (т. е. защищенное от несанкционированного использования и различного рода сбоев и ошибок);**
2. **программный интерфейс доступа к файлам;**
3. **организацию файлов в виде иерархии каталогов.**

***Windows* поддерживает несколько файловых систем для различных внешних устройств:**

**NTFS** - предпочитаемая файловая система в семействе **Windows 2003 Server**, **Windows XP**, **Windows 2000** и **Windows NT**. Она была разработана, чтобы удовлетворять требованиям быстродействующих файловых и сетевых серверов, а так же персональных **ЭВМ** и, при этом, обойти многие из ограничений, ранее сделанных в файловых системах **FAT16** и [**FAT32**](http://vsokovikov.narod.ru/New_MSDN_API/File_system/fat32.htm). Наиболее важными из этих требований следуют ниже:

* 1. **Восстанавливаемость данных. NTFS** является также и полностью восстанавливаемой файловой системой. Она предназначена, чтобы восстанавливать последовательность данных на диске после сбоя центрального процессора, системного аварийного отказа или ошибки ввода - вывода (**I/O**).
  2. **Сохранение устойчивости при сбоях.**Методы хранения избыточных данных могут использоваться **NTFS**, чтобы гарантировать, что, если данные разрушаются на одном физическом диске, то может извлечься сохраненная копия из зеркала диска.
  3. **Защита данных**. **NTFS** выполняет файлы и каталоги как защищенные объекты согласно архитектуре защиты (системы безопасности) объекта **Windows**. Доступ к объектам файла и каталога в **NTFS** может быть ограничен конкретным пользователям и их группам в среде этой архитектуры.

**Многочисленные потоки данных NTFS** файлы могут состоять из более чем одного потока. Дополнительные потоки могут содержать любой вид данных, хотя обычно это - данные, характеризующие файл или метаданные.

**Динамическое перераспределение плохих кластеров.**Когда операция чтения на томе **NTFS**, который - не защищен от ошибок, встречает испорченные данные в группе секторов, каждый сектор в группе (кластере), отмечается как плохой, и последующие попытки выполнить операции чтения в этом секторе приведут к возвращению ошибки.  файловая система, с каждым дефектным сектором, с которым она сталкивается, делает нижеследующее:

1. Восстанавливает неиспорченные данные из вторичного источника в томе.
2. Определяет местонахождение хорошего сектора и записывает восстановленные данные в него.
3. Перераспределяет дефектный сектор в новый хороший сектор так, чтобы все последующие попытки выполнить операции ввода-вывода (**I/O**) на дефектном секторе, должны автоматически переадресовываться в новый сектор.

**Поддержка сжатия и разреженного файла.**Тома **NTFS** поддерживают сжатие файла на базе отдельного файла. Алгоритм сжатия файла, используемый **NTFS** - это метод сжатия Lempel-Ziv. Это - ***не имеющий потерь (lossless)*** алгоритм сжатия, что означает, что данные, при сжатии и восстановлении из сжатого состояния файла, не теряются , в противоположность алгоритмам сжатия ***с потерями (lossy)***типа формата файла изображения **JPEG**, где некоторые данные теряются каждый раз, когда происходят сжатие и восстановление сжатых данных.

**Шифрование. NTFS** обеспечивает **Шифрующую файловую систему**, или (**EFS**) для криптографической защиты файлов и каталогов.

**Каталоги как точки повторной обработки тома.**Точки повторной обработки тома - это каталоги в томе, которые приложение может использовать, чтобы "собрать" другой том, то есть установить его для использования в месте, которое задает пользователь

<http://vsokovikov.narod.ru/New_MSDN_API/File_system/ntfs.htm>

**DOS**

Одна из важнейших функций DOS — организация файловой системы для хранения информации на дисках.

Вся информация на компьютере хранится в файлах. Файлом называют поименованную область на диске, в которой хранится однотипная информация (текст документа, текст или код программы, рисунок, база данных и т.д.). Доступ к этой информации можно получить через имя файла. Составное (полное) имя файла состоит из двух частей: имени (до 8 символов) и расширения (до 3-х символов).

Расширение отделяется от имени. Оно как правило характеризует тип файла.

Некоторые стандартные устройства ввода-вывода информации имеют свои имена, которые наряду с именами файлов могут быть использованы в командах MS DOS.

**Имена устройств:**

PRN — принтер;

CON — при вводе — клавиатура, при выводе — экран.

Имена файлов регистрируются на дисках в каталогах (папках). Каталог – это специальное место на диске, в котором хранятся имена файлов, сведения о размере файлов, атрибуты и т.д. Каждый каталог имеет имя, и он может быть зарегистрирован в другом каталоге. Если каталог Х зарегистрирован в каталоге Y, то говорят, что Х — подкаталог Y, а Y — над каталог или родительский каталог для Х. Требования к именам каталогов те же, что к именам файлов, но расширение не используется. На каждом магнитном диске имеется один главный, или корневой каталог. В нем регистрируются файлы и подкаталоги (каталоги первого уровня). В каталогах первого уровня регистрируются файлы и каталоги второго уровня и т.д. Получается иерархическая древообразная структура каталогов на магнитном диске, по которой с помощью специальных команд можно перемещаться для поиска необходимой информации. Каталог, с которым в настоящий момент работает пользователь, называется текущим каталогом. Имена накопителей на дисках.Иерархическое дерево каталогов может храниться на любом диске компьютера. Каждый диск имеет свое имя (буква с символ ":"):

A: — дискета (обычно большая);

B: — дискета (обычно маленькая);

C: — жесткий диск («винчестер»).

Жесткий диск может быть разбит на несколько логических дисков, они имеют имена D: E: F: и т.д.

Диск, с которым работает компьютер в данный момент, называется текущим диском.

<https://www.ronl.ru/lektsii/informatika/880742/>

1. **Организация ввода и вывода в языке программирования C++. Стандартные файлы ввода/вывода. Текстовые файлы. Методы обработки текстовых файлов**

Способы ввода/вывода в С++:

• Потоковый – через cin и cout;

• Форматный – через printf () и scanf ()

Синтаксис функции ввода printf():

printf (“управляющая\_строка”, [список аргументов])

список аргументов – последовательность выводимых на экран констант, переменных, выражений.

Особенность – в printf() управляющая строка является необязательным параметром.

Управляющая строка содержит управляющие коды и спецификации.

Объекты в составе управляющей строки:

- обычные символы – выводятся на экран без изменений

- спецификации преобразования - выводят на экран значения из прилагаемого списка аргументов (поочередно):

• Начало каждой спецификации с символа %

• Конец каждой спецификации символ преобразования

Минус – выравнивание текста по левому краю

Символы преобразования:

d -аргумент преобразования в десятичный предел

о - -// - // - восьмеричное представление

x - -//-//- 16-тиричное представление

с – значение аргумента является константой

s - -//-//- строкой символов

e - -//-//- величина типа float /double в форме с плавающей запятой

f – форма записи с фиксированной запятой

u – целое беззнаковое число

p – указатель (адрес)

основные управляющие символы

\a – кратковременный звуковой сигнал

\n – перевод строки

\t – горизонтальная табуляция

\b – возврат курсора на один шаг назад

\r – возврат каретки

Поток – логическое устройство, выдающее и принимающее информацию.

Стандартные потоки ВВ в С++ - клавиатура и экран интерпретируются как текстовые файлы.

Условие выполнения ВВ – наличие указателя, определяет позицию начала операции чтения/записи.

Потоки ВВ стандартно определяемые в С++: stdaux, stderr, stdin, stdout, stdprn

Iostream.h – заголовочный файл описания средств создания потоков ВВ

Cin – стандартный ввод – клавиатура

Cout – стандартный вывод – экран

Особенности представления текстовых файлов:

- на диске сплошная последовательность символов без разделения на строки

- при выводе на экран или в печать последовательность символов, разбитая на строки управляющим символом \n

Варианты работы с текстовыми файлами: посимвольный, построчный. Файловые функции ввода/вывода fprintf (), fscanf():

Аналогичны функциям printf(), scanf()

Имеют дополнительный аргумент – указатель на файловый поток FILE\*f

fopen () – открытие файла

fgets () – чтение строки символов буфер

fputs () – запись содержания буфера в файл

Fclose() – закрытие файла

Действия системы при закрытии файла:

- Размещение признака конца файла в таблице размещения файлов

- Установка всех требуемых атрибутов файла.

- Вывод текстового файла всегда буферизован

Стадии фактической записи в файл:

1.заполнение данными текстового буфера

2.собственно запись на диск.

Варианты инициализации процесса записи:

- автоматически при заполнении буфера

- принудительно командами записи (флешировании):

- flush () – для одного файла (fclose())

-flushall () – разом для всех открытых файлов

Доступ к элементам текстовых файлов (запись/чтение) производится только в последовательном режиме.

1. **Организация ввода и вывода в языке программирования C++. Двоичные файлы. Последовательный доступ к элементам двоичных файлов**

Преимущества хранения данных в двоичных файлах:

1. не требуется действий по преобразованию данных из текстового формата в двоичный. Следствие – увеличенная скорость извлечения из файла данных.
2. Компактность двоичной кодировки, отсутствие необходимости в использовании управляющих символов. Следствие – экономия памяти и дискового пространства.

Обработка двоичных файлов – по организации близка к обработке текстовых файлов: связывание логических и физических имен

Особенности работы:

1.открытие вызовом функции fopen()

2.обязательно добавление к режиму доступа буквы b

3. открытие бинарного файла для записи FILE\*fb = fopen()

fread () – чтение данных из потока (файла)

fwrite () – запись данных в поток (файл)

способы доступа к элементам: последовательный, произвольный.

Последовательный доступ к элементам двоичных файлов

Если файл открыт для записи и не содержит данных (пуст), то заполнение возможно только в последовательном режиме

Функция fwrite ()

- запись данных в поток

- прототип – в заголовочном файле stdio.h

- синтаксическое описание size\_t fwrite (const void\*ptr, size\_t, size\_t , FILE\*stream);

Параметры fwrite ():

- const void\*ptr – указатель на исходные данные, записываемые в файл

- size\_t size – размер в байтах одного элемента

Достоинство: наличие кол-ва записываемых элементов в виде аргумента. Как следствие функция может быть использована для записи в файл целиком массива (предварительно сформирован и заполнен.

Особенности функции filength():

- аргумент – дескриптор файла, открытого функцией fopen()

- результат – размер файлов в байтах

Дескриптор – число, уникально идентифицирующее чтение значений с диска (из файла) функцией fread()

* Прототип в фале stdio.h
* Синтаксис аналогичен fwrite (): size\_t read (const void\*ptr, size\_tsize, size\_tn, FILE\*stream);

Отличие – 1-й аргумент указатель на приемник данных из файла (адрес приемника).

1. **Организация ввода и вывода в языке программирования C++. Произвольный доступ к элементам двоичных файлов**

Произвольный доступ к компонентам файла позволяет считывать значения из любых позиций в файле, записывать информацию в любое место файла

Требования к компонентам файла с произвольным доступом:

- обязательно одинаковая длина (предусматривается на момент создания файла).

- первичная запись в файл возможна только в режиме последовательного доступа.

Организация произвольного доступа – функция fseek. Прототип в stdio.h

Особенность fseek:

- синтаксис int fseek (FILE\*stream, longoffset, int whence);

- параметры: FILE\*stream – указатель на открытый файловый поток

Long offset – число байтов

Аргументы whence:

SEEK\_SET – перемещение файлового указателя относительно начала файла

SEEK\_END – перемещение файлового указателя относительно конца файла

SEEK\_CUR – перемещение указателя относительно текущей позиции

1. **Обобщенная архитектура процессора IBM PC. Понятие об адресном пространстве, понятие о моделях памяти**

Состав современных микропроцессоров: АЛУ (арифметико-логическое устройство), регистры, дешифратор команд, блок управления и синхронизации

Внутренние шины цепей управления связаны с шинами адресов и данных.

**Входы и выходы управления**

Назначение АЛУ – арифметико-логические операции, функции сдвига

Назначение регистров – внутренние ЗУ ЦП для временного хранения обрабатываемой (управляющей) информации

Классификация регистров.

Регистры общего назначения

- РОН – восемь РГ произвольно используемых

16 разрядные – AX, DX, CX, BX, BP, SI, DI, SP

32 разрядные – EAX, EDX, ECX, EBX, EBP, ESI, EPI, ESP

Сегментные РГ:

- CS – РГ сегмента кода

- SS – РГ сегмента стека

- DS – РГ дополнительного сегмента

РГ системных флагов

РГ – счетчик команд (управляющий РГ) – IP

Назначение дешифратора

•интерпретация (декодирование) содержимого РГ команд

•выполнение преобразований команд

•активизация работы блока управления

Важнейшие характеристики объема микропроцессора

- разрядность внутренних РГ

- разрядность внешних шин адресов и данных

Особенности РС – совместимых микропроцессоров

16 разрядная внутренняя архитектура

16 разрядная ШД

16 битная длина РГ

Следствие 16 разрядности – максимально допустимое МП целое число – 216 – 1 = 65535 (64 кбайт – 1). Особенность шины адресов МП 8086 – наличие 20 линий (адресное пространство 220 = 1 Мбайт) →для обращения в любую точку адресного пространства 16рязрядной адресации недостаточно→решение – сегментная организация памяти→инструмент – 4 сегментах РГ – CS, SS, DS, ES

Адресное пространство с точки зрения про-мы разделено на блоки смежных адресов (сегментов)

Особенности организации интерфейса ЦП и памяти:

•от ЦП на ША – физический адрес нужной ячейки памяти

•со стороны ЦП – формирование сигнала чтения /записи: при записи ЦП помещает данные на ШД, при чтении память возвращает данные по ШД

Адресация внутри сегментов (сплошной = физический адрес) – только линейная относительно начала сегмента

Концептуальная модель распределения памяти в С++:

Стек куча

БВП•НОП→СП←СД•КП•ВП DOS

Старшие младшие

адреса памяти

БВП – буфера видеопамяти

НОП – неиспользованная оперативная память

СП – свободная память

СД – статические данные

КП – код программы

ВП DOS – вектор прерываний DOS

Размер области, занимаемой ВП (для IBMPC – 256)

Область под КП остается неизменной

Область по статические данные неизменна

Куча – организованная область памяти для работы с динамическими структурами

Объем зависит от запрошенного программой через функцию alloc

Стек – область добавления и удаления с доступными только последним (верхним) элементам

Объем изменяется в результате активизации локальных переменных функции

Модели памяти DOS

Tiny – крошечная, near (указатель)

Small – маленькая (модель по умолчанию) near, 64 Кб

Medium – средняя near,1 Мб

Compact – компактная, fat, 64 Кб

Large – большая fat, 1 Мб

Huge – огромная, fat, 1 Мб

Flat использует 32 разрядные сегменты near

В large статистические данные до 64 Кбайт

1. **Функции в языке программирования C++. Объявление и определение функций, параметры функций**

Функция – логически завершенный и оформленный фрагмент программы, вызываемый из других частей программы.

Следствия использования:

- разделение больших вычислительных задач на мелкие

- возможность использования написанных ранее программ

- большая ясность программ: сокрытие деталей для иных частей программы

- облегчение отладки (модификации)

Обязательные условия:

- описание (объявление) до момента первого обращения = создание прототипа

- определение функции – наполнение операндами

Объявление функции: тип имя\_функции (инф.пар1, инф.пар2, ….);

Инф.пар N – информация о параметре N функции т.е. варианты формата описания параметра:

•безымянный→тип

•именной→тип имя\_параметра

Пример описания параметров функции:

Именные void My\_Func(int Par1, double Par2, double Par3);

Безымянный void My\_Func(int, double, double);

Особенность – компилятор игнорирует имена параметров т.е. именные параметры для него то же, что и безымянные параметры

Сигнатура файла – информация о файле, содержащаяся в прототипе: число аргументов, типы аргументов.

Определение функции – придание функции смысла, наполнение тела функции операндами:

- повтор заголовка, объявленного в прототипе, без использования «;»

- описание тела функции

Void – функция выполняет действия, но не имеет возвращаемых значений. Для вызова достаточно имени без параметров имя\_функции ()

Классическая организация функции:

- с передачей параметров со стороны вызывающих программ

- с передачей вызываемой

Виды параметров:

- формальные - находятся в скобках

- при объявлении прототипа функции

- при определении функции – фактические подставляются на место формальных при вызове функций

Список параметров при необходимости использования более 1 параметра разделяются запятыми. Выход из функции – оператор return; без возвращаемого значения. С возвращаемым значением – return возвращаемое\_значение

Особенности типизации при использовании return:

- оптимальный вариант – типы фактических параметров ↔ типам формальных параметров

- передача значений совместимых типов (int во float и т.п.)

- расширяются автоматически

- может произойти искажение информации из-за округления

1. **Функции в языке программирования C++. Локальные и глобальные переменные. Строки, массивы и структуры в качестве параметров функций**

Локальные переменные – любые переменные, объявленные внутри функции (в т.ч. в функции main). Особенность – существуют в памяти ЭВМ только при активном состоянии функции

Глобальные переменные – переменные, объявленные вне функции (в т.ч. вне main). Особенность существуют в течении всего времени работы программы.

Работа программы при наличии в ней функции:

- последовательное выполнение операторов до появления оператора вызова функции

- при вызове функции – проталкивание текущего адреса программы в стек (= адрес возврата после завершения работы функции)

Стек – область памяти с непосредственно доступным только последним элементом

Функция выделяет в стеке память для хранения локальных переменных. Возврат из функции – удаление стековой памяти.

Особенности хранения глобальных переменных: запоминаются все

Особенность программы – может не содержать прототипа функции. Условие -определение функции до первого обращения к ней.

Строки могут быть переданы в виде параметра функции. Функции могут возвращать строки в программу.фактически в функцию передается только адрес 1-го элемента строки, связанный с именем – символы строки, являющиеся аргументом файла (через стек не передается)

Пример

Const int MAXLEN = 128

Void A\_Func(char s[], int len); //объявление прототипа Ф, обрабатывающей строку длины len

Char author [MAXLEN]=”Oleg Efimov”;

A\_Func(authour, MAXLEN);

Особенности программы

- A\_Func – любая функция обработки строк

- строки рассматриваются как элементы символьного массива

- при cons tint MAXLEN = 128 – могут быть написаны в 127 ячейках

Передача массива в Ф допустима для массивов.

Условие передачи многомерных массивов – указание в качестве параметров минимума информации, которой должно быть достаточно компилятору для получения передаваемых адресов памяти.

Варианты объявления функции (многомерный массив): очевидный, предпочтительный.

Структуры могут передаваться как параметры функций. Функции могут возвращать структуры в качестве результата.

Пример – объявление структуры, определяющей окружность путем указания ее центра и радиуса

Typdef struct circle {int coord\_x;

Int coord\_y;

Int radius;

} Circle;

При передаче структуры функции занимают большие объемы стекового пространства, как следствие возможно переполнение стека.

1. **Функции в языке программирования C++. Рекурсия. Встраиваемые функции. Перегрузка функций, использование аргументов по умолчанию**

Рекурсия = возврат. Рекурсивная функция – вызывающая сама себя.

Виды:

- включительно рекурсивная функция непосредственно вызывает сама себя

- взаимно рекурсивные функции – рекурсивно вызывают друг друга

Рекурсивный вызов (шаг рекурсии) – каждый вызов рекурсивной функции. Примеры задач – рекурсивных по своей природе – n!, an и т.п.

Общие особенности решения: разбиение на этапы

- решение базовой задачи

- разделение сложных задач на подзадачи

Особенность – часть, не являющаяся базовой задачей должна быть проще, чем исходная задача.

Пример вычисления факториала без применения рекурсии

N! = n\*(n-1)\*(n-2)\*…\*2\*1

Fact = 1

For(cnt = n; cht>=1; cht--)

Fact = fact\*cnt; //или fact\*=cnt

Суть решения рекурсивно-многократный вызов функции

…unsigned long fact (unsigned long i\_dat)

{if (i\_dat<1) return 1;

Else return i\_dat\*fact (i\_dat – 1);} //основная программа

Int\_tmain()

{cout<<”4!”<<fact(4)<<endl;

Cout<<”7!”<<fact(7)<<endl; return0;}

**Встраиваемые функции (ВФ)**

Вызов любой обычной функции всегда сопровождают:

- дополнительные действия по обращению к функциям

- передача параметров через стек

- передача возвращаемого значения

Следствие – ухудшение характеристик программ.

Встраиваемые функции – особый вид функций в С++:

- не вызываются в программе

- встраиваются непосредственно в месте вызова, число копий ВФ в программе равно кол-ву обращений

Достоинство ВФ – использование не связано с механизмом возврата/вызова – быстрая работа

Недостаток ВФ – значительный объем программного кода

Объявление ВФ – указание спецификатора inline перед определением функции.

Особенности inline: не являются командой для коппилятора.

Некоторые причины невозможности встраивания:

- функция содержит операторы цикла

- функция содержит операторы switch и goto

**Перегрузка функций**

Возможность существования нескольких функций с одним именем

требование – должны отличаться числом и/или типом аргументов→перегруженные функции

реализация перегрузки – задание всех требуемых вариантов реализации

достоинство – упрощение программы обращением к одному имени для выполнения близких действий

полиморфизм – существование в одной программе нескольких одноименных функций

Ограничение на возможность перегрузки:

1.функции отличаются только типом возвращаемого значения

Int OverloadFh (char \*str);

Char OverloadFh (char \*str);

Следствие – сообщение “Type mismatch in redeclaration”

2.аргументы функции отличаются только использованием ссылок

3.аргументы функций отличаются только применением модификатора const или volatile

Использование аргументов по умолчанию

Любые аргументы по умолчанию – только const или глобальные переменные

1. **Указатели в языке программирования C++. Объявление и разыменовывание указателей**

Указатель – переменная, содержащая адрес другой переменной. Адрес – местоположение ячейки памяти

Способы доступа к значению переменной

Int i – переменная целого типа

Int \*iPtr – указатель на переменную целого типа, сохраненную в определенном месте памяти интерпретируется как \*iPtr есть нечто типа int

Указатель – переменная, подобная переменной типа int или float

Переменная типа указателя – содержит адрес указаний на определенное место памяти.

Значения, сохраненные в определенном месте памяти, являются данными, адресуемыми указателям.

Следствия использования неинициализированных указателей: разыменовывание – использование указателей для получения доступа к данным (операция косвенной адресации)

Int i; int \*Ptr; iPtr = &i // & - унарная операция адресации.

Следствие разыменовывания – возможно выполнение с указателем тех же действий, что и над самой переменной

& возвращает адрес объекта в памяти. Может быть использован для вычисления адресов переменных и функций. Не может быть использован для выражений и символьных const, в виде #define name”…”

Переменные разных типов – занимают разное число ячеек в памяти – указатели могут ссылаться только на объекты заданного типа (исключение тип void) – компилятор всегда выдает проверку соответствия типов

Float \*Ptr;

Char c;

iPtr = &c //ошибка компиляции

1. **Указатели в языке программирования C++. Нулевые указатели и указатели на тип Void**

нулевой указатель – не адресует в данный момент достоверного значения в памяти. Значение нулевого указателя всегда = 0 (единственный адрес, к которому нет доступа).

Определение нулевого указателя #define NULL в заголовочном файле stdef.h или stdlib.h.

пример проверки с использованием нулевого указателя if(iPtr!=NULL) оператор ; //оператор выполняется

указатель = NULL – не адресует достоверных данных.

Особенности инициализации:

- глобальные указатели инициализируются = 0

- локальные указатели требуют явной инициализации float \*Ptr = NULL;

Особая группа указателей – типа void (указатель на void) – указывают на неопределенный тип данных void \*SomePlace;

Особенности указателей типа void:

- не ограничиваются определенными типами данных

- могут адресовать

Применение указателей на void:

- для адресации буферов

- для заполнения блоков памяти

- для чтения аппаратных РГ

- для передачи указателей в ф-ии, независимые от типа данных.

int i ; float f ; int \*iPtr; float \*fPtr; void \*iPtr; iPtr = &i; fPtr = &i –

ошибка трансляции

**типы указателей**

- ближний (near) – 16 -разрядный. Float near \*fPtr. Хранит только смещение в текущем сегменте

- дальний (far) 32 разрядный. Включает значение адреса сегмента float far \*fPtr.

Ссылка на переменную может рассматриваться как указатель

1. **Ссылки в языке программирования C++. Резервирование памяти в куче**

*Ссылки* представляют особый вид данных, напоминающих указатели. Будучи объявлены в функции, они должны быть связаны с адресами конкретных объектов и после этого изменять свои значения не могут. В дальнейшем в рамках этой функции они выступают как синонимы своих объектов – такое ощущение, что одному и тому же объекту присвоено несколько имен:

int x;

int &rx=x; //объявление и инициализация ссылки

*Ссылка* rx является эквивалентом идентификатору x, т.е. *операторы* x=5 и rx=5 абсолютно идентичны. В этом варианте особой пользы от ссылки rx довольно мало – ее имя длиннее основного имени переменной. Однако при программировании в среде Borland C++ Builder довольно часто приходится иметь дело с надоедающе длинными обозначениями свойств объектов, и тогда применение разумной ссылки сокращает время набора программы:

TColor old\_pc,&pc=TForm1->Image1->Canvas->Pen->Color;

............

//запоминание цвета пера

old\_pc=pc; //вместо old\_pc=TForm1->Image1->Canvas->Pen->Color;

//смена цвета пера

pc=clRed; //вместо TForm1->Image1->Canvas->Pen->Color=clRed;

............

//восстановление цвета пера

pc=old\_pc; //вместо TForm1->Image1->Canvas->Pen->Color=old\_pc;

Однако главное преимущество ссылок проявляется при спецификации параметров функций. Если формальный *параметр* объявлен в заголовке функции как *ссылка*, то упрощается его использование в теле функции (в отличие от указателей к именам ссылок ничего добавлять не надо) и становится более естественным вызов функций (вместо формальных параметров-ссылок указываются имена переменных).

Если размер объекта или массива заранее неизвестен (например, изображение), или размер объекта слишком большой, чтобы создавать его внутри функции, значит настало время воспользоваться механизмом динамической памяти С++, использующую отдельную область памяти называемой кучей.

Для этого вам необходимо знать всего два оператора:

new - выделение памяти, если выделение памяти не произошло возвращается нулевой указатель;

delete - освобождение памяти, не во всех компиляторах после освобождения памяти указателю присваивается 0.

умные указатели

В С++ нет автоматической сборки мусора. Другими словами, если указатель на выделенную память потерян, то она становится недоступной. Это называется утечкой памяти. Другой крайностью является попытка освободить одну и ту же память более одного раза, что приводит к системной ошибке. Частично или полностью эти проблемы решаются созданием классов, реализующих "умные указатели". Для примера ниже рассмотрены классы из библиотек STL и Boost.

класс auto\_ptr

Класс auto\_ptr из библиотеки STL имеет следующие ограничения:

объектом может владеть только один указатель,

объектом не может быть массив,

нельзя использовать адресную арифметику.

Единственное предназначение этого класса автоматизировать уничтожение выделенной ранее памяти. Таким образом, данный класс используется, когда время существование выделенного объекта можно ограничить определенным блоком. Аналогичными свойствами обладает и класс scoped\_ptr из библиотеки Boost. Но в отличие от auto\_ptr для этого класса запрещена передача объекта от одного указателя к другому. Делая код более безопасным, данные классы не наносят ущерб размеру или скорости программы.

класс shared\_ptr

Класс shared\_ptr из Boost обладает расширенными возможностями:

объект может иметь несколько владельцев;

можно указать дополнительный класс, отвечающий за уничтожение объекта

new/delete и классы

Оператор new можно перегрузить для новых классов, а оператор delete для классов автоматически вызывает деструктор.

альтернативы new/delete

Как альтернативой можно воспользоваться С функциями (объявлены в stdlib.h) как malloc и free. Эти функции нельзя смешивать с операторами динамической памяти, то есть нельзя выделить память оператором new и освободить ее функцией free или наоборот.

Также не следует забывать системные возможности управления памятью, например в Windows API есть функции для управления кучей (HeapCreate, HeapAlloc, HeapFree,...) и виртуальной памятью (VirtualAlloc, VirtualFree,...).

1. **Указатели и одномерные массивы в языке программирования C++.**

<http://cpp.com.ru/kr_cbook/ch5kr.html> **Указатель** - это переменная, содержащая адрес переменной. Указатели широко применяются в Си - отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и **массивы** тесно связаны друг с другом

В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен с помощью указателя. Вариант с указателями в общем случае работает быстрее, но разобраться в нем, особенно непосвященному, довольно трудно.

Объявление

int a[10];

Определяет массив *a* размера 10, т. е. блок из 10 последовательных объектов с именами a[0], a[1], ..., a[9].

http://cpp.com.ru/kr_cbook/krfigs/fig5_3.jpg

Запись *a[i]* отсылает нас к *i*-му элементу массива. Если *pa* есть указатель на *int*, т. е. объявлен как

int \*pa;

то в результате присваивания

pa = &a[0];

*pa* будет указывать на нулевой элемент *a*, иначе говоря, *pa* будет содержать адрес элемента *a[0]*.

Теперь присваивание

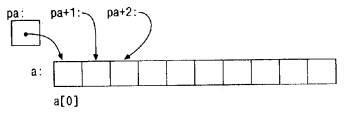
x = \*pa;

будет копировать содержимое *a[0]* в *x*.

Если *pa* указывает на некоторый элемент массива, то *pa+1* по определению указывает на следующий элемент, *pa+i* - на *i*-й элемент после *pa*, a *pa-i* - на *i*-й элемент перед *pa*. Таким образом, если *pa* указывает на *a[0]*, то

\*(pa+1)

есть содержимое *a[1]*, *a+i* - адрес *a[i]*, a *\*(pa+i)* - содержимое *a[i]*.



Сделанные замечания верны безотносительно к типу и размеру элементов массива *a*. Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы *pa+1* указывал на следующий объект, a *pa+i* - на *i*-й после *pa*.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания

pa = &a[0];

*ра* и *a* имеют одно и то же значение. Поскольку имя массива является синонимом расположения его начального элемента, присваивание *pa=&a[0]* можно также записать в следующем виде:

pa = a;

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. *Указатель - это переменная*, поэтому можно написать *pa=a* или *pa++*. Но *имя массива не является переменной*, и записи вроде *a=pa* или *a++* не допускаются.

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес. Мы можем воспользоваться отмеченным фактом и написать еще одну версию функции *strlen*, вычисляющей длину строки.

/\* strlen: возвращает длину строки \*/

int strlen(char \*s)

{

int n;

for (n = 0; \*s != '\0'; s++)

n++;

return n;

}

Так как переменная *s* - указатель, к ней применима операция ++; *s++* не оказывает никакого влияния на строку символов функции, которая обратилась к *strlen*. Просто увеличивается на 1 некоторая копия указателя, находящаяся в личном пользовании функции *strlen*. Это значит, что все вызовы, такие как:

strlen("3дравствуй, мир"); /\* строковая константа \*/

strlen(array); /\* char array[100]; \*/

strlen(ptr); /\* char \*ptr; \*/

правомерны.

Формальные параметры

char s[];

и

char \*s;

в определении функции эквивалентны. Мы отдаем предпочтение последнему варианту, поскольку он более явно сообщает, что *s* есть указатель. Если функции в качестве аргумента передается имя массива, то она может рассматривать его так, как ей удобно - либо как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется уместным и понятным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если *a* - массив, то в записях

f(&a[2])

или

f(a+2)

функции *f* передается адрес подмассива, начинающегося с элемента *a*[2]. Внутри функции *f* описание параметров может выглядеть как

f(int arr[]) {...}

или

f(int \*arr) {...}

Следовательно, для *f* тот факт, что параметр указывает на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в "обратную" сторону по отношению к нулевому элементу; выражения *p[-1]*, *p[-2]* и т.д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед *p[0]*. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

1. **Указатели и многомерные массивы в языке программирования C++**

В Си имеется возможность задавать прямоугольные многомерные массивы, правда, на практике по сравнению с массивами указателей они используются значительно реже. В этом параграфе мы продемонстрируем некоторые их свойства.

Рассмотрим задачу перевода даты "день-месяц" в "день года" и обратно. Например, 1 марта - это 60-й день невисокосного или 61-й день високосного года. Определим две функции для этих преобразований: функция *day\_of\_year* будет преобразовывать месяц и день в день года, a *month\_day* - день года в месяц и день. Поскольку последняя функция вычисляет два значения, аргументы месяц и день будут указателями. Так вызов

month\_day(1988, 60, &m, &d)

присваивает переменной *m* значение 2, а *d* - 29 (29 февраля).

Нашим функциям нужна одна и та же информация, а именно таблица, содержащая числа дней каждого месяца. Так как для високосного и невисокосного годов эти таблицы будут различаться, проще иметь две отдельные строки в двумерном массиве, чем во время вычислений отслеживать особый случай с февралем. Массив и функции, выполняющие преобразования, имеют следующий вид:

static char daytab[2][13] = {

{0, 31, 28, 31. 30, 31, 30, 31, 31, 30, 31, 30, 31},

{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}

}

/\* day\_of\_year: определяет день года по месяцу и дню \*/

int day\_of\_year(int year, int month, int day)

{

int i, leap;

leap = year % 4 == 0 && year % 100 !=0 || year % 400 == 0;

for (i = 1; i < month; i++)

day += daytab[leap][i];

return day;

}

/\* month\_day: определяет месяц и день по дню года \*•/

void month\_day(int year, int yearday, int \*pmonth, int \*pday)

{

int i, leap;

leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;

for (i = 1; yearday > daytab[leap][i]; i++)

yearday -= daytab[leap][i];

\*pmonth = i;

\*pday = yearday;

)

Напоминаем, что арифметическое значение логического выражения (например выражения, с помощью которого вычислялось *leap*) равно либо нулю (ложь), либо единице (истина), так что мы можем использовать его как индекс в массиве *daytab*.

Массив *daytab* должен быть внешним по отношению к обеим функциям *day\_of\_year* и *month\_day*, так как он нужен и той и другой. Мы сделали его типа *char*, чтобы проиллюстрировать законность применения типа *char* для малых целых без знака.

Массив *daytab* - это первый массив из числа двумерных, с которыми мы еще не имели дела. Строго говоря, в Си двумерный массив рассматривается как одномерный массив, каждый элемент которого - также массив. Поэтому индексирование изображается так:

daytab[i][j] /\* [строка] [столбец] \*/

а не так:

daytab[i,j] /\* НЕВЕРНО \*/

1. **Строковые указатели в языке программирования C++. Указатели и структуры в языке программирования C++**

http://studopedia.org/8-23930.html

Строковые указатели являются не строками, а указателями, т.е. адресами, которые определяют местонахо­ждение первого символа строки, сохраненной где-то в памяти. Строковые указатели объявляются как char \* или, чтобы операторы не могли изменить адресуемые ими данные, как const char \*

Вот несколько примеров:

char \*stringPtr; /\* Строковый указатель \*/

const char \*fixedString; /\* Указатель на фиксированную строку \*/

char far •stringPtr; /\* Дальний указатель на строку \*/

В строковых указателях нет ничего особенного - они просто указывают на массив значений типа char и ведут себя аналогично другим указателям (см. главу 5). Существует много функций, которые обрабатывают строки, адресуемые строковыми указателями, и объявления char \* очень распространены.

Чтобы *выделить память в куче* для строкового *указателя*, вызовите функцию malloc() и задайте размер, в который не забудьте включить один дополнительный байт для завершающего нуля. Оператор

stringPtr = (char \*) malloc(81);

резервирует 81 байт памяти в куче и присваивает указателю stringPtr адрес первого байта. Строка может со­держать до 80 символов плюс завершающий нуль. По окончании работы со строкой вызовите функцию free(), чтобы вернуть зарезервированную память обратно в кучу, сделав ее доступной для будущих вызовов функции malloc():

free(stringPtr);

Вспомните, что массивы и указатели эквивалентны. Чтобы обратиться к отдельным символам строки, нуж­но использовать индексные выражения. Например, чтобы отобразить третий символ в строке, адресуемой ука­зателем stringPtr, можно написать:

printf("%c", stringPtr[2]);

При подобном индексировании строк лишь на вас лежит ответственность за то, что в указанном месте на­ходится допустимый символ. Если адресуемая строка содержит меньше трех значащих символов, то это выра­жение, вероятно, отобразит некорректную информацию.

http://cppstudio.com/post/5377/

Структура — это агрегатный тип данных, так как может содержать в себе разнотипные элементы.

|  |
| --- |
| struct Name  {      type atrib;      // остальные элементы структуры  } structVar1, structVar2, ...; |

где,

* struct — ключевое слово, которое начинает определение структуры
* Name — имя структуры
* type — тип данных элемента структуры
* atrib — элемент структуры
* structVar1-2 — структурные переменные

Объявление структуры всегда должно начинаться с ключевого слова struct. Необязательно, чтобы структура имела имя, но тогда такая структура обязательно должна иметь структурные переменные, объявленные между закрывающей фигурной скобкой и точкой с запятой, строка 5. Обязательно в объявлении структуры должны присутствовать фигурные скобочки, они обрамляют тело структуры, в котором объявляются её атрибуты (элементы), строка 3. Структурные переменные, при объявлении структуры, указывать необязательно, строка 5.

Так как структура — это тип данных, то, для того, чтобы использовать этот тип данных, необходимо объявить структурную переменную, а вместо типа данных указать имя структуры.

|  |  |
| --- | --- |
| 1 | struct\_name structVariable; |

http://www.c-cpp.ru/books/ukazateli-na-struktury

1. **Указатели и функции в языке программирования C++. Способы передачи параметров. Передача массивов в функцию посредством указателей**

http://informatics.mccme.ru/mod/book/view.php?id=563&chapterid=304

## Локальные и глобальные переменные

Внутри функции могут быть объявлены переменные, как, например, переменные p и i в функции power. Такие переменные называются локальными, и они определены только внутри этой функции. Переменные, определенные вне тела любой функции (ранее мы такие переменные вообще не рассматривали) называются глобальными, и они определены всюду начиная с места определения и до конца файла. Глобальными переменными можно пользоваться во всех функциях, но с современной точки зрения использовать глобальные переменные рекомендуется как можно реже.

Локальные переменные создаются каждый раз при входе в функцию и уничтожаются при выходе из нее. Таким образом, значения, сохраненные в локальных переменных, пропадут после завершения работы функции.

В различных функциях можно определять переменные с одним и тем же именем. Это будут различные переменные. Также можно объявлять в функциях переменные, имена которых совпадают с именами глобальных переменных. Тогда будет создана новая локальная переменная, а изменить значение глобальной переменной с таким именем будет невозможно. Пример:

     #include<iostream>  
     using namespace std;  
     int i;                // i - глобальная переменная  
     void f1() {  
         int i;            // Определена локальная переменная i  
         i=5;              // Изменяется значение локальной переменной  
         cout<<i<<endl;    // Будет напечатано 5  
     }  
      void f2() {  
         i=3;              // Изменяется значение глобальной переменной  
     }  
     int main(){  
         i=1;              // Изменяется значение глобальной переменной  
         f1();             // f1() не меняет значение глобальной переменной  
         cout<<i<<endl;    // Будет напечатано 1  
         f2();             // f2() меняет значение глобальной переменной  
         cout<<i<<endl;    // Будет напечатано 3  
         return 0;      }

### **Передача параметров по значению и по ссылке**

### Переменные, в которых сохраняются параметры, передаваемые функции, также являются локальными для этой функции. Эти переменные создаются при вызове функции и в них копируются значения, передаваемые функции в качестве параметров. Эти переменные можно изменять, но все изменения этих переменных будут "забыты" после выхода из функции. Рассмотрим это на примере следующей функции, "меняющей" значения двух переданных ей переменных:

     #include<iostream>  
     using namespace std;  
     void swap(int a, int b)  
     {  
         int t;  
         t=b;  
         b=a;  
         a=t;  
     }  
     int main()  
     {  
         int p=3,q=5;  
         swap(p,q);  
         cout<<p<<" "<<q<<endl;  
         return 0;  
     }

При вызове функции swap создаются новые переменные a и b, им присваиваются значения 3 и 5. Эти переменные никак не связаны с переменными p и q и их изменение не изменяет значения p и q. Такой способ передачи параметров называется передачей параметров по значению.

Чтобы функция могла изменять значения переменных, объявленных в других функциях, необходимо указать, что передаваемый параметр является не просто константной величиной, а переменной, необходимо передавать значения по ссылке. Для этого функцию swap следовало бы объявить следующим образом:

     void swap(int & a, int & b)

Амперсанды перед именем переменной означают, что эта переменная является не локальной переменной, а ссылкой на переменную, указанную в качестве параметра при вызове функции. Теперь при вызове swap(p,q) переменные a и b являются синонимами для переменных p и q, и изменение их значений влечет изменение значений p и q. А вот вызывать функцию в виде swap(3,5) уже нельзя, поскольку 3 и 5 — это константы, и сделать переменные синонимами констант нельзя.

Однако в языке C (не C++) вообще не было такого понятия, как передача параметров по ссылке. Для того, чтобы реализовать функцию, аналогичную swap в рассмотренном примере, необходимо было передавать адреса переменных p и q, а сама функция при этом должна быть объявлена, как

     void swap(int \* a, int \* b)

Поскольку в этом случае функция swap знает физические адреса в оперативной памяти переменных p и q, то разыменовав эти адреса функция swap сможет изменить значения самих переменных p и q.

Теперь вспомним, что в C++ массивы и указатели — это почти одно и то же. А поскольку передача массива по значению, то есть создание копии всего массива является трудоемкой операцией (особенно для больших массивов), то вместо массива всегда передается указатель на его начало. То есть два объявления функции void f(int A[10]) и void f(int \* A) абсолютно идентичны. В любом случае функция f получит указатель на начало массива, а, значит, будет способна изменять значения его элементов.

Передача массивов в функцию посредством указателей

http://www.c-cpp.ru/books/peredacha-massivov-v-funkcii

1. **Указатели и функции в языке программирования C++. Строки как аргументы функций**

<http://cppstudio.com/post/7216/>

<https://code-live.ru/post/cpp-functions/>

Передача символьной строки в функцию подобна передаче любого массива в качестве параметра. Внутри функции вам нужно просто указать тип массива (char) и левую и правую скобки массива. Вам не надо указывать размер строки. Например, следующая программа SHOW\_STR.CPP использует функцию show\_ string для вывода символьной строки на экран:

#include <iostream.h>

void show\_string(char string[])

{  
cout << string << endl;  
}

void main(void)

{  
show\_string(«Привет, C++!»);  
show\_string(«Учусь программировать на C++»);  
}

Как видите, функция show\_string трактует параметр символьной строки как массив:

void show\_string(char string[])

Так как символ NULL указывает конец строки, функция не требует параметр, который задает количество элементов в массиве. Вместо этого функция может определить последний элемент, просто найдя в массиве символ NULL.

Как вы уже знаете, функции C++ часто используют символ NULL для определения конца строки. Следующая программа STR\_LEN.CPP создает функцию с именем string\_length, которая ищет символ NULL в строке для определения количества символов, содержащихся в строке. Далее функция использует оператор return для возврата длины строки вызвавшей функции. Программа передает несколько различных символьных строк в функцию, отображая длину каждой из них на экране:

#include <iostream.h>

int string\_length(char string[])

{  
int i;  
for (i = 0; string[] != ‘\0’; i++); // Ничего не делать, но перейти к  
// следующему символу return(i); Длина строки  
}

void main(void)

{  
char title[] = «Учимся программировать на языке C++»;  
char lesson[] = «Символьные строки»;  
cout << «Строка » << title << » содержит » << string\_length(title) << » символов» << endl;  
cout << «Строка » << lesson << » содержит » << string\_length(lesson) << » символов» << endl;  
}

Как видите, функция запускается с первого символа строки (элемент 0) и затем исследует каждый элемент до тех пор, пока не встретит NULL. Рассматривая программы на C++, вы встретите целый ряд функций, которые подобным образом просматривают символьные строки в поисках символа NULL.

1. **Указатели и функции в языке программирования C++. Передача структур в функцию по указателю и по ссылке**

<http://webhamster.ru/mytetrashare/index/mtb0/339>

<http://ermak.cs.nstu.ru/cprog/html/052.htm>

**указатель –**  адресная информация о расположении информационного ресурса, через которую пользователь может обратиться к нему. При изменении содержимого объекта через указатель на него всегда возникает проблема **синхронизации (разделения)** ресурса между несколькими пользователями, имеющими адресную информацию о нем. Синонимом указателя в информационных технологиях является **ссылка.** Иногда она имеет все внешние признаки объекта, например, ярлык файла на рабочем столе, который внешне выглядит как файл, а на самом деле ссылается на файл-оригинал.

В языках программирования термины объект (значение), указатель и ссылка имеют примерно аналогичный смысл, но касаются способов доступа и передачи значений переменных.

        терминология **ссылка, значение** касается фундаментальных свойств переменных в языках программирования. Имя переменной в различных контекстах может восприниматься как ее значение (содержимое памяти), так и ссылка на нее (адрес памяти, указатель) Например, при присваивании левая часть рассматривается как ссылка, а правая – как значение

        при передаче формальных параметров при вызове процедур (функций) практически во всех языках программирования реализованы способы передачи **по ссылке** и **по значению;**

        в Паскале и Си определено понятие **указатель** как переменная особого вида, содержащая адрес размещения в памяти другой переменной. Использование указателей позволяется создавать динамические структуры данных, в которых элементы взаимно ссылаются друг на друга;

        и, наконец, в Си существует расширенная интерпретация указателя, именуемая **адресной арифметикой,** которая позволяет интерпретировать значение любого указателя как адрес не отдельной переменной, а памяти в целом, где она размещена.

### Указатель в Си

Передавать данные между программами, данные от одной части программы к другой (например, от вызывающей функции к вызываемой) можно двумя способами:

        создавать в каждой точке программы (например, на входе функции) копию тех данных, которые необходимо обрабатывать;

        передавать информацию о том, где в памяти расположены данные. Такая информация, естественно, является более компактной, чем сами данные, и ее условно можно назвать указателем. Получаем «дилетантское» определение указателя: **указатель** **- переменная, содержащая информацию о расположении в памяти другой переменной.**

Наряду с указателем в программировании также используется термин **ссылка.**  Ссылка – содержанием ссылки также является адресная информация об объекте (переменной), но внешне она выглядит как переменная (синоним оригинала).

Пример передачи структуры в функцию по указателю, и пример передачи массива структур.

Имеется некая структура console\_menu:

// Структура

typedef struct {

char text[100];

int value;

} console\_menu;

Она используется в двух следующих функциях:

// Функция принимает одну структуру по указателю

set\_one\_item(console\_menu \*menu)

{

sprintf(menu->text,"One item");

menu->value=50;

}

// Функция принимает массив структур

set\_console\_menu(console\_menu menu[])

{

sprintf(menu[0].text,"One item in array");

menu[0].value=5;

sprintf(menu[1].text,"Two item in array");

menu[1].value=100;

}

Вызов этих функций выглядит так:

void main(void)

{

console\_menu omnu;

console\_menu cmnu[10];

// Передача по указателю одной структуры (передается адрес структуры)

set\_one\_item(&omnu);

// Передача массива структур

set\_console\_menu(cmnu);

printf("%s %d\n",omnu.text, omnu.value);

printf("%s %d\n",cmnu[0].text,cmnu[0].value);

printf("%s %d\n",cmnu[1].text,cmnu[1].value);

}

1. **Указатели и функции в языке программирования C++. Ссылка в качестве возвращаемого значения функции. Функции, возвращающие указатели**

http://www.c-cpp.ru/books/vozvrashchenie-obektov-funkciyami

Функция может возвращать объект в точку вызова. В качестве примера рассмотрим программу:  
  
#include <iostream.h>  
class myclass {  
int i;  
public:  
void set\_i(int n) {i=n;}  
int get\_i() {return i;}  
};  
myclass f(); // возвращение объекта типа myclass  
int main()  
{  
myclass o;  
о = f ();  
cout << о.get\_i() << "\n";  
return 0;  
}  
myclass f()  
{  
myclass x;  
x.set\_i(1);  
return x;  
}

Когда функция возвращает объект, автоматически создается временный объект, содержащий возвращаемое значение. Именно этот объект фактически возвращается функцией. После того, как значение возвращено, этот объект уничтожается. Уничтожение временного объекта может вызывать неожиданные побочные эффекты в некоторых ситуациях. Например, если возвращае­мый функцией объект имеет деструктор, освобождающий динамически зарезервированную па­мять, то эта память будет освобождена даже в том случае, когда объект, получающий возвраща­емое значение, будет продолжать использовать ее. Как будет продемонстрировано дальше в этой книге, имеются способы преодолеть эту проблему, для чего используется перегрузка оператора присваивания и определение конструктора копирования.

http://www.c-cpp.ru/books/vozvrat-ukazateley

Хотя функции, возвращающие указатели, обрабатываются точно так же, как и функции с другим типом, следует рассмотреть несколько важных понятий.

Указатели на переменные - это не целые числа и не беззнаковые целые. Это адрес памяти некоторого типа данных. Причина такого разделения состоит в том, что когда выполняются арифметические действия с указателем, изменения происходят с учетом базового типа, то есть, если указатель на целое увеличивается, он будет содержать значение на 2 больше по сравнению с предыдущим (предполагается использование 2-байтных целых). Каждый раз при увеличении указателя он указывает на следующий элемент базового типа. Поскольку каждый тип данных может иметь различную длину, компилятор должен знать, на какой тип данных указывает указатель, чтобы правильно осуществить переход к следующему элементу данных.

Ниже показана функция, возвращающая указатель на строку в месте, где найдено соответствие символов:  
  
char \*match(char с, char \*s)  
{  
register int count;  
count = 0;  
while(c!=s[count] && s[count]) count++;  
return(&s[count]);  
}  
  
Функция match() пытается вернуть указатель на позицию в строке, где первый раз найден символ с. Если не найдено соответствие, возвращается указатель, содержащий NULL. Ниже показана небольшая программа, использующая match():  
  
#include <stdio.h>  
#include <conio.h>  
  
char \*match(char c\*, char \*s);  
  
int main(void)  
{  
char s[80], \*p, ch;  
gets (s) ;  
ch = getche();  
p = match(ch, s);  
if (p) /\* совпадение \*/  
printf("%s ", p);  
else  
printf("No match found.");  
return 0;  
}  
  
Данная программа осуществляет чтение строки, а затем символа. Если символ содержится в строке, то выводится строка, начиная с момента совпадения. Иначе выводится «No match found».

1. **Динамические структуры данных в языке программирования C++. Понятие о самоссылочных структурах. Формирование очереди. Формирование стека**

) <http://www.apmath.spbu.ru/ru/staff/smirnovmn/files/devcpp_4.pdf>

Часто в серьезных программах надо использовать данные, размер и структура которых должны меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы. Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту. В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью ссылок. Каждый элемент(узел) состоит из двух областей памяти: поля данных и ссылок. Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменные - указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются нулевые ссылки(NULL).

https://nmetau.edu.ua/file/teoriya\_algoritmov\_metodichka.pdf

Если главный интерес представляет последовательный перебор набора элементов их можно организовать в виде связного списка —базовой структуры данных, в которой каждый элемент содержит информацию, необходимую для получения следующего элемента. В языке C++ указатели служат удобной реализацией абстрактной концепции связных списков. Определение 3.1 Связный список—это набор элементов, причем каждый из них является частью узла (node), который также содержит ссылку (link) на узел. Связный список содержит либо null ссылки, либо ссылки на узлы, ко

торые содержат элемент и ссылку на связный список. Интерфейс абстрактного типа данных «список» приведен в листинге 3.1.

Листинг 3.1 Интерфейс АТД «Список»

template <class Item, class Key> class List

{

private:// программный код, зависящий от реализации

public:

List() //Конструктор

~List(); //Деструктор

void add(Item); //Вставка узла в конец списка

Item \* search(Key); //Поиск элемента (элементов) с заданным ключом

Item \* insert(Item, Item); // Вставка узла в заданную позицию

bool remove(Item); //Удаление узла

void print(); //Печать списка

};

Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.

Узлы определяются ссылками на узлы, поэтому связные списки иногда называют самоссылочными структурами. Более того, хотя узел обычно ссылается на другой узел, возможна ссылка на самого себя, поэтому связные списки могут представлять собой циклические структуры. Обычно под связным списком подразумевается реализация последовательного расположения набора элементов. Начиная с некоторого узла, мы считаем его первым элементом последовательности. Затем прослеживается его ссылка на другой узел, который дает нам второй элемент последовательности и т.д. Поскольку список может быть циклическим, последовательность иногда представляется бесконечной. Обычнопринимается одно из следующих соглашений для ссылки последнего узла:

•это пустая (null) ссылка, не указывающая на какой-либо узел;

•ссылка указывает на фиктивный узел (dummy node), который не содержит элементов;

•ссылка указывает на первый узел, что делает список циклическим.

В каждом случае отслеживание ссылок от первого узла до последнего формирует последовательное расположение элементов. Массивы также задают последовательное расположение элементов, но оно реализуется косвенно, за счет позиции в массиве. Кроме того, массивы поддерживают произвольный доступ по индексу, что невозможно для списков

http://cppstudio.com/post/5159/

Очередь — структура данных, которая, как и стек, имеет ограничения, по добавлению и удалению элементов. Чтобы понять очереди, представьте очередь в  магазин: человек в начале очереди (тот, кто пришел первый) будет обслуживаться первым, вновь пришедшие люди будут добавляться в конец очереди. Таким образом, первый человек в очереди обслуживается первым, последний в очереди, обслуживается последним.Сокращенно очереди обозначаются как: FIFO — First In, First Out (первым пришел, первым ушел).

Очереди часто используются в программировании сетей, операционных систем и других ситуациях, в которых различные процессы должны разделять ресурсы, такие как процессорное время.

Немного терминологии:

* **enqueue** — добавление элемента в очередь;
* **dequeue** — удаления элемента из очереди.

Хотя концепция очереди может быть простой, в программировании очередь не так проста, как стек. Давайте вернемся к примеру с очередью в магазин. Скажем, один человек покидает очередь. Тогда что? Все в очереди должны пройти вперед, верно? Теперь, представьте себе, что только один человек может двигаться одновременно. Таким образом, второй человек делает шаг вперед, чтобы заполнить место, оставленное от первого лица, а затем третьим лицом делается шаг вперед, чтобы заполнить место, оставленное после второго человека, и так далее. Теперь представьте, что никто не может уйти или быть добавленным в очередь, пока все не шагнул вперед. Как вы уже поняли, очередь будет двигаться очень медленно. Конечно, не трудно запрограммировать очередь, и она будет работать, но очень не просто сделать очередь, которая бы работала очень быстро!

Есть несколько основных способов реализации очереди. Во-первых, просто создать массив и переложить все элементы, чтобы поместить элемент в очередь или извлечь его. Это тот случай, о котором мы говорили, он является очень медленным.

Медлительность предыдущего метода заключается в том, что чем больше элементов в очереди, тем больше времени занимает перемещение. Есть еще один метод, не смещая элементы в очереди, выполняются функции постановки и удаления элементов из очереди. Представьте себе, снова, очередь в магазин. Представим, что не очередь движется к продавцу, а продавец движется к очереди, таким образом позиция начала очереди постоянно изменяется, стремится к концу очереди. Таким образом люди, которые стоят в очереди не делают шаг вперед или назад, что экономит время.

<http://cppstudio.com/post/5159/> (снизу дичь)

https://prog-cpp.ru/data-stack/

Стеком называется упорядоченный набор элементов, в котором размещение новых и удаление существующих происходит с одного конца, называемого вершиной.

Дисциплина обслуживания — это совокупность правил (упорядочение и алгоритм) обслуживания элементов динамической структуры данных.

В зависимости от дисциплины обслуживания различают те или иные структуры динамических данных.

Принцип работы стека сравнивают со стопкой листов бумаги: чтобы взять второй сверху, нужно снять верхний.

В стеке реализуется дисциплина обслуживания LIFO:

* **LAST** - последний
* **INPUT** - вошел
* **FIRST** - первый
* **OUTPUT** - вышел

Различают аппаратный и программный стек.

Аппаратный стек используется для хранения адресов возврата из функций и их аргументов.  
Программный стек – это пользовательская модель (структура) данных.

**Операции для работы со стеком**

Над стеком реализованы следующие операции:

* инициализация стека init(s), где s — стек
* помещение элемента в стек push (s, i), где s — стек, i — помещаемый элемент;
* удаление элемента из стека i=pop(s);
* определение верхнего элемента без его удаления i=stkTop(s), которая эквивалентна операциям i=pop (s); push (s, i);
* получение вершины стека (количества элементов) i=gettop(s), где s — стек
* печать стека stkPrint(s), где s — стек
* определение пустоты стека isempty(s)  
  возвращает true если стек пустой и false в противном случае.

**Способы реализации стека**

Существует несколько способов реализации стека:

* с помощью одномерного массива;
* с помощью связанного списка;
* с помощью класса объектно-ориентированного программирования.

Пример реализации стека

Стек можно реализовать в виде следующей структуры:

#define NMAX 100  
struct stack {

  float elem[NMAX];

  int top;  
};

NMAX — максимальное количество элементов в стеке;  
elem — массив из NMAX чисел типа float, предназначенный для хранения элементов стека;  
top — индекс элемента, находящегося в вершине стека.

**Инициализация стека**

Индекс элемента, находящегося в вершине стека, равен 0.

void init(struct stack \*stk) {

  stk->top = 0;  
}

**Помещение элемента в стек**

void push(struct stack \*stk, float f) {

  if(stk->top < NMAX) {

    stk->elem[stk->top] = f;

    stk->top++;

  } else

    printf("Стек полон, количество элементов: %d !\n", stk->top);  
}

**Удаление элемента из стека**

float pop(struct stack \*stk) {

  float elem;

  if((stk->top) > 0) {

    stk->top--;

    elem = stk->elem[stk->top];

    return(elem);

  } else {

    printf("Стек пуст!\n");

    return(0);

  }  
}

**Извлечение вершины стека**

float stkTop(struct stack \*stk) {

  if((stk->top) > 0) {

    return( stk->elem[stk->top-1]);

  } else {

    printf("Стек пуст!\n");

    return(0);

  }  
}

**Получение верхнего элемента стека без его удаления**

int gettop(struct stack \*stk) {

 return(stk->top);}

**Определение пустоты стека**

int isempty(struct stack \*stk) {

  if((stk->top) == 0)    return(1);

  else return(0);  
}

**Вывод элементов стека**

void stkPrint(struct stack \*stk) {

  int i;

  i=stk->top;

  if(isempty(stk)==1) return;

  do {

    i--;

    printf("%f\n", stk->elem[i]);

  }while(i>0);  
}

1. **Основные определения объектно-ориентированного программирования**

http://mycsharp.ru/post/23/2013\_06\_20\_ponyatie\_obektno-orientirovannogo\_programmirovaniya\_(oop)\_klassy\_i\_obekty.html

Понятие **объектно-ориентированное программирование (ООП)** означает один из самых эффективных подходов к современному программированию.  
  
Раньше программисты, в большинстве случаев, использовали функциональный или процедурный принцип программирования. Все программы, большие и маленькие, писались в одном файле. С течением времени программы становились всё сложнее и больше, что доставляло проблемы разработчикам при поддержке таких программ и внесении изменений. Эту проблему решает объектно-ориентированное программирование. ООП позволяет объединить данные и методы, относящиеся к одной сущности, и работать с ними, как с одним целым.  
  
**Классы и объекты. В чем разница?**  
  
ООП привносит нам два ключевых понятия: **Класс** и **Объект**. **Класс** – это абстрактный тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д. Описав класс, мы можем создать его экземпляр – **объект**. Объект – это уже конкретный представитель класса.  
  
Пример  
  
Допустим, нам в программе необходимо работать со странами. Страна – это абстрактное понятие. У нее есть такие характеристики, как название, население, площадь, флаг и другое. Для описания такой страны будет использоваться класс с соответствующими полями данных. Такие страны, как Россия и Украина будут уже объектами (конкретными представителями типа страна).  
  
**Основные принципы объектно-ориентированного программирования**  
  
ООП основывается на нескольких базовых принципах, каждому из которых будет посвящен отдельный урок, а пока коротко рассмотрим их.  
  
**Инкапсуляция** – позволяет скрывать внутреннюю реализацию. В классе могут быть реализованы внутренние вспомогательные методы, поля, к которым доступ для пользователя необходимо запретить, тут и используется инкапсуляция. Больше об инкапсуляции читайте в уроке [Инкапсуляция в Си-шарп. Модификаторы доступа](http://mycsharp.ru/post/38/2014_02_23_inkapsulyaciya_v_si-sharp_modifikatory_dostupa.html).  
  
**Наследования** – позволяет создавать новый класс на базе другого. Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником. Например, есть базовый класс животное. В нем описаны общие характеристики для всех животных (класс животного, вес). На базе этого класса можно создать классы наследники Собака, Слон со своими специфическими свойствами. Все свойства и методы базового класса при наследовании переходят в класс наследник. Больше о наследовании читайте в уроке [Наследование в Си-шарп](http://mycsharp.ru/post/28/2013_07_21_nasledovanie_v_si-sharp_konstruktor_bazovogo_klassa.html).  
  
**Полиморфизм** – это способность объектов с одним интерфейсом иметь различную реализацию. Например, есть два класса, Круг и Квадрат. У обоих классов есть метод GetSquare(), который считает и возвращает площадь. Но площадь круга и квадрата вычисляется по-разному, соответственно, реализация одного и того же метода различная. Больше о полиморфизме читайте в уроке [Полиморфизм в Си-шарп](http://mycsharp.ru/post/32/2013_08_27_polimorfizm_v_si-sharp_chto_eto_takoe_.html).  
  
**Абстракция** – позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере (для поставленной задачи) описывают объект. Например, создавая класс для описания студента, мы выделяем только необходимые его характеристики, такие как ФИО, номер зачетной книжки, группа. Здесь нет смысла добавлять поле вес или имя его кота/собаки и т.д.  
  
**Все так запутано**  
  
Конечно, для тех, кто совсем не знаком с ООП, все это может показаться сложным, и сразу понять, что к чему, будет сложно. Но с дальнейшими уроками подход ООП для вас будет проясняться, и станет понятно, что всё очень логично.

1. **Понятие о конструкторах и деструкторах в языке программирования C++. Конструкторы с параметрами, конструкторы по умолчанию**

<http://purecodecpp.com/archives/1764>

***Конструктор (от construct – создавать)*** — это особый метод класса, который выполняется автоматически в момент создания объекта класса. То есть, если мы пропишем в нем, какими значениями надо инициализировать поля во время объявления объекта класса, он сработает без «особого приглашения». Его не надо специально вызывать, как обычный метод класса.

Пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #include <iostream>  using namespace std;    class SomeData  {  private:  int someNum1;  double someNum2;  char someSymb[128];  public:  SomeData() //это конструктор:  {  someNum1 = 111; // присваиваем начальные значения полям  someNum2 = 222.22;  strcpy\_s(someSymb, "СТРОКА!");  cout << "\nКонструктор сработал!\n";  }    void showSomeData()  {  cout << "someNum1 = " << someNum1 << endl;  cout << "someNum2 = " << someNum2 << endl;  cout << "someSymb = " << someSymb << endl;  }  }obj1; //уже на этом этапе сработает конструктор (значения запишутся в поля)    int main()  {  setlocale(LC\_ALL, "rus");  obj1.showSomeData();    SomeData obj2; //тут сработает конструктор для второго объекта класса  obj2.showSomeData();  }  В строках 11 — 17 определяем конструктор: имя должно быть идентично имени класса; конструктор НЕ имеет типа возвращаемого значения (void в том числе). Один объект объявляется сразу во время определения класса — строка 25. При запуске программы, конструктор этого объекта сработает даже до входа в главную функцию |

**Деструктор (от destruct — разрушать) —** так же особый метод класса, который срабатывает во время уничтожения объектов класса. Чаще всего его роль заключается в том, чтобы освободить динамическую память, которую выделял конструктор для объекта. Имя его, как и у конструктора, должно соответствовать имени класса. Только перед именем надо добавить символ ~

Добавим деструктор в предыдущий код. И создадим в классе два конструктора: один будет принимать параметры, второй — нет.

#include <iostream>

using namespace std;

class SomeData

{

private:

int someNum1;

double someNum2;

char someSymb[128];

public:

SomeData()

{

someNum1 = 0;

someNum2 = 0;

strcpy\_s(someSymb, "СТРОКА ПО УМОЛЧАНИЮ!");

cout << "\nКонструктор сработал!\n";

}

SomeData(int n1, double n2, char s[])

{

someNum1 = n1;

someNum2 = n2;

strcpy\_s(someSymb, s);

cout << "\nКонструктор с параметрами сработал!\n";

}

void showSomeData()

{

cout << "someNum1 = " << someNum1 << endl;

cout << "someNum2 = " << someNum2 << endl;

cout << "someSymb = " << someSymb << endl;

}

~SomeData()

{

cout << "\nДеcтруктор сработал!\n";

}

};

int main()

{

setlocale(LC\_ALL, "rus");

SomeData obj1(1, 2.2, "СТРОКА ПАРАМЕТР"); // сработает конструктор с параметрами

obj1.showSomeData();

SomeData obj2; // сработает конструктор по умолчанию

obj2.showSomeData();

}

Деструктор определен в строках 34 — 37. Для простоты примера он просто отобразит строку в том месте программы, где сработает. Строка 43 — объявляем объект класса и передаем данные для записи в поля. Тут сработает конструктор с параметрами. А в строке 46 — сработает конструктор по умолчанию.

Видим, что деструктор сработал автоматически и дважды (так как в программе было два объекта класса). Он срабатывает тогда, когда работа программы завершается и уничтожаются все данные.

Важное:

* Конструктор и деструктор должны быть **public**;
* Конструктор и деструктор не имеют типа возвращаемого значения;
* Имена класса, конструктора и деструктора должны совпадать;
* Конструктор может принимать параметры. Деструктор не принимает параметры;
* При определении деструктора перед именем надо добавить символ ~ ;
* Конструкторов может быть несколько, но их сигнатура должна отличаться (количеством принимаемых параметров, например);
* Деструктор в классе должен быть определен только один.

1. **Функции и объекты в языке программирования C++. Указатели на объекты. Передача объектов в функцию**

**Указатель** – именованный *объект*, предназначенный для хранения адреса области памяти (объекта, непоименованной области оперативной памяти либо точки входа в функцию).

**Указатели на объекты**

С*интаксис* определения указателя на *объект*:

Тип\*Описатель;

При определении указателя специфицируется имя указателя-переменной (в дальнейшем *указатель*) и *тип объекта*, на который он ссылается.

Тип задает *тип объекта*, *адрес* которого будет содержать определяемая *переменная* и может соответствовать базовому, пустому (свободному, родовому, то есть типу void ), перечислению, *структурному типу* и типу объединения. Реально *указатель* на void ни на что не указывает, но обладает способностью указывать на область любого размера после его типизирования каким-либо объектом.

Описатель – это *идентификатор*, определяющий имя объявляемой переменной типа *указатель* или конструкция, которая организует непосредственно *доступ* к памяти. *Описателю* обязательно должна предшествовать звездочка (\*).

Примеры определения указателей:

int \*P;

/\*указатель Р может содержать адрес объекта типа int\*/

float \*s;

/\*указатель s может содержать адрес объекта типа float\*/

*Синтаксис объявления* указателя на *объект* *базового типа*:

*Указатель* может быть константой или переменной, а также указывать на константу или переменную.

При объявлении указателя его можно сразу проинициализировать (задать *значение*):

int \*pi=&i; //указатель на целую переменную

const int \*pci=&ci; //указатель на целую константу

int \*const cpi=&i;

//указатель-константа на целую переменную

const int \*const cpc=&ci;

//указатель-константа на целую константу

Если модификатор const относится к указателю (т.е. находится между именем указателя и \*), то он запрещает изменение значения указателя, а если он находится слева от типа (т.е. слева от \*), то он запрещает изменение значения, на которое указывает *указатель*.

*Способы инициализации указателя*

1. Присваивание указателю адреса области памяти существующего объекта:
   * с помощью операции получения адреса:
   * int a=5;

int \*p=&a;

* + с помощью проинициализированного указателя

int \*r=p;

1. Присваивание указателю адреса области памяти в явном виде:

char \*cp=(char\*)0х В800 0000;

где 0х В800 0000 – шестнадцатеричная константа, ( char\* ) – *операция приведения* типа.

1. Присваивание указателю пустого значения:

int \*N=NULL; или int \*N=0;

*Спецификатор указателя при форматированном выводе*

Если на экран необходимо вывести *адрес*, следует применять *спецификатор* %p.

%p – *спецификатор* указателя.

Этот *спецификатор* формата заставляет функцию printf() выводить на экран *адрес*, формат которого совместим с типом адресации, принятой в компьютере.

*Операции с указателями*

* *разыменование* (\*) – получение значения величины, адрес которой хранится в указателе;
* взятие адреса (&);
* присваивание;
* арифметические операции
  + сложение указателя только с константой,
  + вычитание: допускается разность указателей и разность указателя и константы,
  + *инкремент* (++) увеличивает значение указателя на величину sizeof(тип);
  + *декремент* (--) уменьшает значение указателя на величину sizeof(тип);
* сравнение;
* приведение типов.

*Пример 1*. Демонстрация ситуации, когда указатели различных типов указывают на одно и то же *место* в памяти. Однако при разыменовании получаются разные результаты.

// Выбор данных из памяти с помощью разных указателей

// Использование функций приведения типов

#include "stdafx.h"

#include <iostream>

using namespace std;

int \_tmain(int argc, \_TCHAR\* argv[]){

unsigned long L=12345678;

char \*cp=(char\*)&L;

int \*ip=(int\*)&L;

long \*lp=(long\*)&L;

cout <<"\n&L = "<<&L;

cout <<"\nL = "<<L;

cout <<"\n\*cp = "<<\*cp;

cout <<"\n\*ip = "<<\*ip;

cout <<"\n\*lp = "<<\*lp;

system("pause");

return 0;

}

Указатели одного и того же типа можно сравнивать с помощью стандартных операций сравнения. При этом сравниваются значения указателей, а не значения величин, на которые данные указатели ссылаются.

*Пример 2.*

#include "stdafx.h"

#include <iostream>

using namespace std;

int \_tmain(int argc, \_TCHAR\* argv[]){

int x=10;

int y=10;

int \*xptr=&x;

int \*yptr=&y;

//сравниваем указатели

if (xptr == yptr)

cout << "Указатели равны\n";

else

cout << "Указатели неравны\n";

//сравниваем значения, на которое указывает указатель

if (\*xptr == \*yptr) {

cout << "Значения равны\n";

} else {

cout << "Значения неравны\n";}

system("pause");

return 0;

}

В приведенном примере результат первой *операции* сравнения будет ложным, а второй – истинным, поскольку переменные x и y имеют одно и то же *значение*.

### Ключевые термины

### ***Косвенная адресация* – это обращение к области памяти не напрямую, по адресу, а через*****объект*, которому в памяти соответствует определенный участок.**

*Разыменование*– это операция получения значения объекта, *адрес* которого хранится в указателе;

*Указатель* – это именованный *объект*, предназначенный для хранения адреса области памяти.

*Указатель на**константу* – это *указатель* на такой *объект*, *значение* которого нельзя изменить в процессе выполнения программы.

*Указатель-константа на константу* – это *указатель*, для которого невозможно изменение как самого указателя, так и значения адресуемого объекта.

Объекты можно передавать в функции в качестве аргументов, в точности так, как передаются данные других типов. Однако следует помнить, что в C++ по-умолчанию параметры передаются по значению. Это означает, что внутри функции (а точнее, в стеке) создаётся копия объектааргумента, и эта копия, а не сам объект, будет далее использоваться функцией. Благодаря этому функции могут произвольно изменять переданные значения, не влияя на оригинал.

Итак, при передаче объекта в функцию создаётся новый объект, а когда работа функции завершается, копия переданного объекта будет разрушена. Как всегда, при разрушении объектов, при этом будет вызван деструктор копии. И здесь может наблюдаться очередной побочный эффект: если переданный в качестве параметра объект содержит в себе указатель на динамически выделенную область памяти, деструктор копии её освободит. Но так как копия создавалась побитовым копированием, деструктор копии высвободит область памяти, на которую указывал объект-оригинал. Исходный объект будет по-прежнему "видеть" свои данные по указанному адресу, однако для системы эта память будет считаться свободной. Рано или поздно она будет выделена какому-то другому объекту, и данные окажутся затёрты.

Кроме возможности преждевременного разрушения объекта-оригинала, к аварийной ситуации приведёт вызов его деструктора (в конце работы программы или при выходе из соответствующей области видимости), который попытается освободить уже свободную память. Та же проблема возникает при использовании объекта в качестве возвращаемого значения.

Во всех трёх случаях (при присваивании объекта, при использовании его как параметра и при передаче в качестве возвращаемого значения) если деструктор высвобождает динамически выделенную память, то разрушение временного объекта приведёт к преждевременному разрушению данных оригинала.

Частично проблема может быть решена перегрузкой оператора присваивания для данного класса. Кроме того, для объектов, которым противопоказано побитовое копирование, рекомендуется создавать особую разновидность конструктора — т. н. конструктор копирования (в некоторых источниках также можно встретить название "конструктор копии"). Конструктор копирования выполняет именно то действие, которое заложено в его названии: позволяет программисту лично проконтролировать процесс создания копии.

Любой конструктор копирования имеет следующую форму:

имя\_класса ( const имя\_класса & obj )

{

... //тело конструктора

}

*&obj — это ссылка на объект, известная ещё как скрытый указатель.*

Во второй строке примера переменная p3 и объявляется и определяется, а в третьей строке переменной p1 всего лишь присваивается значение. Иными словами, конструктор копирования вызывается для конкретной переменной за время её жизни только один раз, а присваивать значения ей можно многократно. В логике работы конструктора копирования и оператора присваивания настолько много общего, что часто рекомендуют описывать одну операцию в терминах другой. Фактически операция присваивания неявно используется в конструкторе копирования. Однако конструктор копирования может добавлять дополнительные действия по инициализации переменных в довесок к тем действиям, которые должен выполнять оператор присваивания.

1. **Функции и объекты в языке программирования C++. Объекты в качестве возвращаемых значений. Дружественные функции**

Для того, чтобы функция могла возвращать объект, нужно:

* 1. Объявить функцию с возвращаемым значением типа класса;
  2. Возвращать объект с помощью обычного оператора return.

Если возвращаемый объект содержит деструктор, то возникают похожие проблемы, связанные с «неожиданным» разрушением объекта.

**Пример.**

Class B {

public:

B ( ) { cout << ‘Раб-та к-ра/ n’; }

~B ( ) { cout << ‘Раб-та дестр-ра/ n’; }

}

B f( )

{

B obj;

count << ‘Раб-та ф-ции/ n’;

return obj;

}

main ( ) {

B obj-m;

obj-m = f ( );

}

Конструктор вызывается два раза: для obj и obj-m.

Деструктор вызывается три раза:

1. Разрушает obj;
2. Разрушает obj-m;
3. Вызывается для так называемого **временного объекта**, который является копией возвращаемого объекта. Формируется эта копия, когда функция возвращает объект. После того, как функция возвратила своё значение, выполняется деструктор временного объекта. Понятно, что если деструктор высвобожд-е динамич-й памяти, то разрушение временного объекта приведёт к разрушению возвращаемого.

Избежать рассмотренных побочных эффектов можно с помощью ссылочной переменной (ссылки), при использовании которой в качестве параметра не происходит создания копии объекта. В это случае передаётся не сам объект, а его адрес.

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

Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие *спецификаторов* доступа, место размещения ее объявления в классе безразлично.

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

class monster; // Предварительное объявление класса

class hero

{

...

void kill(monster &);

};

class monster

{

...

friend int steal\_ammo(monster &);

/\* Класс hero должен быть определен ранее \*/

friend void hero::kill(monster &);

};

int steal\_ammo(monster &M){return --M.ammo;}

void hero::kill(monster &M){M.health = 0; M.ammo = 0;}

1. **Перегрузка операторов в языке программирования C++. Перегрузка унарных и бинарных операторов**

Перегрузка операций

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Таблица 5.1. | | | | | | |
| . | .\* | ?: | :: | # | ## | sizeof |

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

* сохраняются количество аргументов, *приоритеты операций* и правила ассоциации (справа налево или слева направо) по сравнению с использованием в стандартных типах данных;
* нельзя переопределить операцию по отношению к стандартным типам данных;
* функция-операция не может иметь аргументов по умолчанию;
* функции-операции наследуются (за исключением =);

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

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

тип operator операция ( список параметров) { тело функции }

#### **Перегрузка унарных операций**

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

class monster

{... monster & operator ++() {++health; return \*this;}}

monster Vasia;

cout << (++Vasia).get\_health();

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

class monster

{... friend monster & operator ++( monster &M);};

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

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

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

class monster

{... monster operator ++(int){monster M(\*this); health++; return M;}};

monster Vasia;

cout << (Vasia++).get\_health();

#### **Перегрузка бинарных операций**

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

class monster

{ ...

bool operator >( const monster &M)

{

if( health > M.get\_health()) return true;

return false;

}

};

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

bool operator >(const monster &M1, const monster &M2)

{ if( M1.get\_health() > M2.get\_health()) return true;

return false;

}

1. **Наследование в языке программирования C++. Основные понятия, режимы доступа к элементам базового класса**

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

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

Классы, находящиеся ближе к началу *иерархии*, объединяют в себе наиболее общие черты для всех нижележащих классов. *По* мере продвижения вниз *по* *иерархии классы* приобретают все больше конкретных черт. 

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

class имя : [private | protected | public] базовый\_класс{тело класса};

Если *базовых классов* несколько, они перечисляются через запятую. Перед каждым может стоять свой модификатор *наследования*. По умолчанию для классов он *private*, а для структур - public.

Если задан модификатор *наследования* public, оно называется *открытым*. Использование модификатора *protected* делает *наследование* *защищенным*, а модификатора *private* - *закрытым*. Это не просто названия: в зависимости от вида *наследования* классы ведут себя по-разному. Класс может наследовать от структуры, и наоборот.

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

Из таблицы, *private* элементы *базового класса* в *производном классе* недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы *базового класса*.

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

Доступ к элементам public при *наследовании* становится соответствующим ключу доступа.

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

class Base{...

public: void f();};

class Derived : private Base{...

public: Base::void f();};

**Наследование**— один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляциейи полиморфизмом), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса наследуются новым классом.

Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса 1

## **Простое наследование**

Класс, от которого произошло наследование, называется базовым или родительским (англ. base class). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class).

В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, который описан в программе, имеет поля, функции, но не используется для создания объекта. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник ВУЗа», от которого наследуются классы «аспирант», «профессор» и т.д. Т.к. производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

## **Множественное наследование**

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Из других языков, предоставляющих эту возможность, можно отметить Python. Множественное наследование поддерживается в языке UML.

Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов.

Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживает возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним классом. Этот механизм позволяет во многом заменить множественное наследование — методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Рассмотрим **правила наследования** различных методов.

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

* Если в *конструкторе* *производного класса* явный вызов *конструктора* *базового класса* отсутствует, автоматически вызывается *конструктор* *базового класса* по умолчанию (то есть тот, который можно вызвать без параметров). Это использовано в первом из конструкторов класса *daemon*.
* Для *иерархии*, состоящей из нескольких уровней, *конструкторы* *базовых классов* вызываются начиная с самого верхнего уровня. После этого выполняются *конструкторы* тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется *конструктор класса*.
* В случае нескольких *базовых классов* их *конструкторы* вызываются в порядке объявления.

1. **Виртуальные функции в языке программирования C++. Основные понятия**

*Виртуальная функция —* это функция, которая может быть переопределена классом-наследником, для того чтобы тот имел свою, отличающуюся, реализацию. В языке C++ используется такой механизм, как таблица виртуальных функций  
(кратко vtable) для того, чтобы поддерживать связывание на этапе выполнения программы. Виртуальная таблица — статический массив, который хранит для каждой виртуальной функции указатель на ближайшую в иерархии наследования реализацию этой функции. Ближайшая в иерархии реализация определяется во время выполнения посредством извлечения адреса функции из таблицы методов объекта.  
  
Давайте теперь посмотрим на простой пример использования виртуальных функций в C++

**class** ClassA

{

**public**:

ClassA() {data = 10;}

**virtual** **void** **set**()

{

std::cout << "ClassA is increasing" << std::endl;

data++;

}

**int** **get**()

{

set();

**return** data;

}

**protected**:

**int** data;

};

**class** ClassB : **public** ClassA

{

**public**:

**void** **set**()

{

std::cout << "ClassB is decreasing" << std::endl;

data--;

}

};

В приведенном примере у нас есть класс ClassA, имеющий два метода: get() и set(). Метод get()помечен как виртуальная функция; в классе ClassB его реализация меняется. Целое число data помечено ключевым словом protected, чтобы класс-наследник ClassB имел доступ к нему.

**int** **main**()

{

ClassA classA;

ClassB classB;

std::cout << "ClassA value: " << classA.get() << std::endl;

std::cout << "ClassB value: " << classB.get() << std::endl;

**return** 0;

}

Вывод:

ClassA **is** increasing

ClassA **value**: 11

ClassB **is** decreasing

ClassB **value**: 9

Базовый класс иерархии типа обычно содержит ряд виртуальных функций, которые обеспечивают динамическую типизацию. Часто в самом базовом классе сами виртуальные функции фиктивны и имеют пустое тело. Определенное значение им придается лишь в порожденных классах. Такие функции называются ***чистыми виртуальными функциями***.

***Чистая виртуальная функция*** — это функция-член класса, тело которой не определено.

В базовом классе такая функция записывается следующим образом:

**virtual ПрототипФункции = 0;**

***Например***

 virtual void func() = 0;

Чистая виртуальные функции используются для того, чтобы отложить решение задачи о реализации функции на более поздний срок. В терминологии ООП это называется ***отсроченным методом***. Класс, имеющий по крайней мере одну чистую виртуальную функцию, называется ***абстрактным классом***. Для иерархии типа полезно иметь абстрактный базовый класс. Он содержит общие свойства иерархии типа, но каждый порожденный класс реализует эти свойства по-своему.

Для рассмотренного выше примера (класс Фигура) функцию вычисления площади целесообразно задать чистой виртуальной функцией, которую переопределяет каждый наследуемый класс.

#include <iostream>  
using namespace std;  
class figure {

  protected:

    double x, y;

  public:

    figure(double a=0, double b=0) {x = a; y = b;}

    virtual double area()=0 // чистая виртуальная функция  
};  
class rectangle : public figure {

  public:

    rectangle(double a=0, double b=0) : figure(a, b) {};

    double area() {return(x\*y);}  
};  
class circle : public figure {

  public:

    circle(double a=0) : figure(a, 0) {};

    double area() {return(3.1415\*x\*x);}  
};  
int main() {

  ...

}

1. **Объекты и файловые потоки в языке программирования C++. Потоковый ввод/вывод**

В операционных системах корпорации Microsoft (начиная с версии *MS-DOS* 2.0) и всех "клонах" *UNIX* существует возможность "потокового ввода-вывода" данных, например, на *консоль*, на принтер или в *файл*.

Принцип "потокового ввода-вывода" следующий:

1. В оперативной памяти средствами операционной системы создаётся некоторый "промежуточный буфер" для хранения данных, читаемых из *файла, устройства* или записывания на него этих данных;
2. Средствами программы, созданной прикладным программистом, происходит чтение или запись информации (символов) в этот буфер;
3. Средствами операционной системы осуществляется "синхронизация" этого буфера ("потока данных") с файлом или устройством;
4. При создании или открытии файла для него выделяется буфер в оперативной памяти компьютера, а после закрытия файла этот буфер очищается.

В *буфер* заносятся только символьные данные. Как следствие, при потоковом выводе нельзя (или "практически нельзя") изменить атрибуты выводимых символов, их шрифтовое и абзацное оформление и т.д. Кроме того, в стандартные *потоки ввода-вывода* нельзя выводить "двоичные" файлы (то есть файлы, имеющую кодировку, отличную от *ASCII* и совместимой с ней кодировки). В операционных системах корпорации Microsoft при потоковом вводе-выводе для вывода на *консоль* кириллических символов возможна только *кодировка* *OEM* 866 (кодовая страница *MS-DOS* для русскоязычных пользователей). Это в настоящее время резко ограничивает применение потокового ввода-вывода для создания программ в современном программировании. Однако этот подход широко востребован в *UNIX* и её клонах, работающих в режиме командной строки.

Вследствие вышесказанного *автор* просто обязан рассказать о технологии потокового ввода-вывода. Тем более что при помощи функций sprintf и sscanf (см. *Приложение* №1) возможна "эмуляция потокового вывода" информации в строку текста, откуда её можно вывести средствами *API* системных библиотек ввода-вывода (таких как *Windows* *API*, GTK+, Qt и др.).

В качестве "стандартных потоков", присутствующих в операционной системе всегда и никогда не удаляемых из оперативной памяти, используются следующие потоки (см. [таблицу 6.1](http://www.intuit.ru/studies/courses/640/496/lecture/11252?page=1#table.6.1)):

* *Стандартный поток ввода* (обозначение: stdin, cin и др.) - используется для ввода символьных данных в программу. По-умолчанию этот поток закреплён за клавиатурой компьютера;
* *Стандартный поток вывода* (обозначается как: stdout, cout и др.) - используется для вывода символьной информации, полученной в результате работы программы в "штатном режиме". По-умолчанию этот поток закреплён за экраном дисплея;
* Стандартный поток ошибок (обозначение: stderr, cerr и др.) - используется для вывода символьных *диагностических сообщений*, ошибок и предупреждений, возникших в результате работы программы. По-умолчанию этот поток закреплён за экраном дисплея;

Примечание: стандартный поток и поток ошибок разделены в связи с тем, что при *перенаправлении вывода* часто совсем не нужно записывать в результаты работы программы диагностические сообщения. Эти сообщения будут лишними, например, при формировании таблицы базы данных в виде текстового файла;

* Стандартный поток печати (обозначение: stdprn и др.) - используется для вывода результатов работы программы на печать. По-умолчанию этот поток закреплён за текущим принтером в системе, подключённым к порту *LPT1*. В настоящее время этот поток почти не используется, поскольку чаще проще и безопаснее перенаправить *стандартный поток вывода* на принтер, чем разделять потоки отдельно для экрана и отдельно для принтера.

Все остальные потоки создаются или уничтожаются с помощью функций открытия и закрытия файлов, на период чтения/записи/добавления информации в эти файлы.

### 6.2. Ввод со стандартного потока ввода

#### 6.2.1. Ввод средствами языка Си

Ввод со стандартного потока в Си осуществляется при помощи следующих функций:

* для ввода одиночного символа - функцию getchar ;
* для ввода строки символов без ограничения на длину - функцию gets ;
* для "форматированного ввода" символов и их преобразования в "двоичные значения" переменных - функцию scanf ;

С синтаксисом и правилами использования этих функций можно ознакомиться в приложении №I к данной лекции. Автор хочет отметить, что использование функций ввода со стандартного потока является небезопасным, а, следовательно, и нежелательным способом ввода данных из стандартного потока. По возможности, заменяйте эти функции функциями потокового чтения данных с "явным" указанием потоков и "длины" прочтённой строки.

**Пример 6.1**

/\* Файл ex06001.c \*/

/\* Функция иллюстрирует потоковый ввод-вывод (как это делать нельзя)\*/

#include <stdio.h>

#define STR\_LENGTH 3 // Длина строки 3 символа

void main()

{

char str[STR\_LENGTH]; // Текстовый буфер

char \*s; // Временная переменна

s = gets( str );

puts( s );

}

Например, функция, приведённая в примере 6.1, при вводе строки: "aaaaaaaa\n", - вызовет аварийное завершение с ошибкой "переполнение буфера". Поскольку длина текстового буфера - 2 символа (последний символ - "нулевой", знак окончания строки '\0' ), то лишние символы, введённые с клавиатуры, попадают в "запредельную" *системную область* данных. В этом случае происходит операция (прерывание) "отказ системы", и адрес этой *системной области* оказывается доступной "взломщику системы". Функции, приведённые в примерах 6.2 и 6.3, лишены этих недостатков;

**Пример 6.2**

/\* Файл ex06002.c \*/

/\* Функция иллюстрирует потоковый ввод-вывод (как это делать надо)\*/

/\* Пример тестировался в системе программировани

Borland C/C++ 3.10 \*/

#include <stdio.h>

#include <conio.h>

#define BUF\_LENGTH 7 // Длина строки 3 символа

#define STR\_LENGTH 2 // Длина буфера без дескриптора

// Всё те же 2 символа

void main()

{

char str[BUF\_LENGTH]; // Текстовый буфер

char \*s, c = '\0'; // Временная переменная.

int icsize; // Временная переменная.

icsize = BUF\_LENGTH - 2;

str[0] = (char) icsize;

s = cgets( str );

puts( s );

puts( "\nPress any key to continue...");

while( !( c = getch()) ); // Цикл пока не нажата клавиша

}

#### 6.2.2. Ввод средствами языка C++

Консольный потоковый ввод в языке C++ осуществляется при помощи операторов ">>", а также при помощи функций get, *peek* и ignore. *Стандартный поток ввода* в языке C++ обозначается как cin (см. [таблицу 6.1](http://www.intuit.ru/studies/courses/640/496/lecture/11252?page=1#table.6.1)).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 6.1. Обозначение потоков в разных языках программирования. | | | | |
| **Обозначения потоков** | | | | |
| **Поток** | **Ввод (стандартный)** | **Вывод (стандартный)** |  |  |  |
| Обозначение в C++ | cin | cout |  |  |  |

C помощью оператора ">>" можно осуществить как неформатированный, так и форматированный ввод-вывод. Однако автор советует для ввода использовать неформатированный ввод из потока с помощью функции cin.get, а затем "выделить и ввести" нужные двоичные значения при помощи функции sscanf (см. пример 6.4.).

**Пример 06.004**.

/\* File ex06004.cpp \*/

/\* Пример ввода-вывода средствами C++\*/

#include <iostream.h>

#include <stdio.h>

#include <conio.h>

#include <string.h>

#ifndef STR\_LENGTH

#define STR\_LENGTH 35 // Длина строки 34 символа + завершающий символ '\0'

#endif

#ifndef INT\_WIDTH

#define INT\_WIDTH 5 // Ширина под поле целого числа - 4 символов

#endif

void main( void )

{

const char format\_in[] = "%5d"; // Формат ввода

const char format\_out[] = "\nВведённое значение: %5d"; // Формат вывода

char str[STR\_LENGTH], c='\0';

int iValue;

(void) memset( str, '\0', STR\_LENGTH ); // Обнуляем строку;

cout << "\nВведите число: ";

cin.get( str, INT\_WIDTH );

if( !sscanf( str, format\_in, &iValue ) ) // Если ошибка ввода

{

cerr << "\nНичего не введено или неправильный формат данных!\nПрограмма завершена с ошибкой";

return;

}

(void) memset( str, '\0', STR\_LENGTH ); // Обнуляем строку;

if( sprintf( str, format\_out, iValue ) == EOF ) // Если ошибка вывода

{

cerr << "\nНичего не выведено или неправильный формат данных!\nПрограмма завершена с ошибкой";

return;

}

cout << str; // Выводим сформированную строку

cerr << "\nПрограмма завершена нормально";

cerr << "\nPress any key to continue...";

while( !(c = getch()) ); // Цикл пока не нажата клавиша

}

В этом примере для ввода символьной строки использовалась функция cin.get, для вывода - оператор "<<", а для форматных преобразований - функции sprintf и sscanf.

Описание функций sscanf и sprintf см. в приложении №I к данной лекции.

Замечание: из личного опыта автора родился следующий совет: при работе с консолью, когда не требуется программирование фильтров и *перенаправления потоков*, лучше использовать консольные функции ввода-вывода. Для программирования фильтров, наоборот, требуется ввод-вывод исключительно файловыми потоковыми функциями. В обоих случаях для форматного преобразования необходимо использовать только функции sprintf и sscanf, предварительно проверив (т.н. "валидация") формат читаемых и записываемых символов. Во всех случаях нужно жёстко контролировать итоговую длину строки ввода-вывода.

1. **Объекты и файловые потоки в языке программирования C++. Текстовые файлы**

Текстовыми называют файлы, состоящие из любых символов. Они организуются *по* строкам, каждая из которых заканчивается символом "конец строки". Конец самого файла обозначается символом "*конец файла*". При записи информации в текстовый *файл*, просмотреть который можно с помощью любого текстового редактора, все данные преобразуются к символьному типу и хранятся в символьном виде.

Для работы с файлами используются специальные типы данных, называемые потоками4.*Поток* ifstream служит для работы с файлами в режиме чтения. *Поток* ofstream служит для работы с файлами в режиме записи. Для работы с файлами в режиме как чтения, так и записи служит *поток* iofstream.

В программах на C++ при работе с текстовыми файлами необходимо подключать библиотеки iostream и fstream.

Для того, чтобы записать данные в текстовый *файл*, необходимо:

1. Описать переменную типа ofstream.
2. Открыть файл с помощью функции open.
3. Вывести информацию в файл.
4. Закрыть файл.

Для того, чтобы считать данные из текстового файла, необходимо:

1. Описать переменную типа ifstream.
2. Открыть файл с помощью функции open.
3. Считать информацию из файла, при считывании каждой порции данных необходимо проверять достигнут ли конец файла.
4. Закрыть файл.

#### 7.2.1 Запись информации в текстовый файл

Для того, чтобы начать работать с текстовым файлом, необходимо описать переменную типа ofstream. Например, с помощью оператора

ofstream F;5

будет создана переменная F для записи информации в файл. На следующем этапе файл необходимо открыть для записи. В общем случае оператор открытия потока будет иметь вид:

F.open("file", mode);

Здесь F — переменная, описанная как ofstream, file — имя файла на диске, mode — режим работы с открываемым файлом.

Файл может быть открыт в одном из следующих режимов:

* ios::in — открыть файл в режиме чтения данных, этот режим является режимом по умолчанию для потоков ifstream;
* ios::out — открыть файл в режиме записи данных (при этом информация в существующем файле уничтожается), этот режим является режимом по умолчанию для потоков ofstream;
* ios::app — открыть файл в режиме записи данных в конец файла;
* ios::ate — передвинуться в конец уже открытого файла;
* ios::trunc — очистить файл, это же происходит в режиме ios::out;
* ios::nocreate — не выполнять операцию открытия файла, если он не существует6;
* ios::noreplace — не открывать существующий файл.

Параметр mode может отсутствовать, в этом случае файл открывается в режиме по умолчанию для данного потока7.

После удачного открытия файла (в любом режиме) в переменной F будет храниться true, в противном случае false. Это позволит проверять корректность операции открытия файла.

Открыть файл (в качестве примера возьмём файл abc.txt) в режиме записи можно одним из следующих способов:

//Первый способ.

ofstream f;

f.open ( " abc.txt ", ios::out );

//Второй способ,

//режим ios::out является режимом по умолчанию для потока ofstream

ofstream f;

f.open ( " abc.txt " )

//Третий способ объединяет описание переменной типа поток

//и открытие файла в одном операторе

ofstream f( " abc.txt ", ios::out );

После открытия файла в режиме записи будет создан пустой файл, в который можно будет записывать информацию. Если нужно открыть файл, чтобы в него что-либо дописать, то в качестве режима следует использовать значение ios::app.

После открытия файла в режиме записи, в него можно писать точно также, как и на экран, только вместо стандартного устройства вывода cout необходимо указать имя открытого для записи файла.

Например, для записи в поток F переменной a оператор вывода будет иметь вид:

F<<a;

Закрытие потока осуществляется с помощью оператора:

F.close();

Обратите внимание, что в текстовый файл записываются не только вещественные числа, но и символы табуляции. Таким образом, в конце файла после последнего числа находится символ табуляции. По этой причине может возникнуть проблема при чтении информации (п. 7.2.2), так как символ табуляции будет интерпретирован как вещественное число. Чтобы этого избежать в программе была применена следующая конструкция:

if (i<n-1) f<<a<<"\t"; else f<<a;

Здесь реализована проверка ввода последнего числа. После него символ табуляции отсутствует.

#### 7.2.2 Чтение информации из текстового файла

#### Для того, чтобы прочитать информацию из текстового файла, необходимо описать переменную типа ifstream. После этого файл необходимо открыть для чтения с помощью оператора open. Если переменную назвать F, то первые два оператора будут такими:

ifstream F;

F.open("abc.txt", ios::in);

После открытия файла в режиме чтения, из него можно считывать информацию точно так же, как и с клавиатуры, только вместо стандартного устройства ввода cin необходимо указать имя потока, из которого будет происходить чтение данных.

Например, для чтения из потока F в переменную a оператор ввода будет иметь вид:

F>>a;

файле считаются разделёнными, если между ними есть хотя бы один из символов: пробел, табуляция, символ конца строки.

Для проверки, достигнут или нет конец файла, служит функция

F.eof();

Здесь F — имя потока, функция возвращает логическое значение: true — если достигнут конец файла, если не достигнут функция возвращает значение false.

Следовательно, цикл для чтения содержимого всего файла можно записать так.

while ( ! F.eof ( ) )

{

F>>a;

if ( ! F.eof ( ) ) //Проверяем, достигнут ли конец файла, если нет — печатаем значение a

{

cout<<a<<" \t ";

n++;

}

}

1. **Объекты и двоичные файлы в языке программирования C++. Сохранение данных в двоичных файлах**

Двоичные файлы отличаются от текстовых тем, что представляют собой последовательность байтов, содержимое которых может иметь различную природу. Это могут быть байты, представляющие числовую информацию в машинном формате, байты с графическими изображениями, байты с аудиоинформацией и т.п. Содержимое таких байтов может случайно совпасть с *управляющими кодами* таблицы ASCII, но на них нельзя реагировать так, как это делается при обработке текстовой информации. Естественно, что единицей обмена с такими данными могут быть только порции байтов указанной длины.

Создание двоичных файлов с помощью функции *fopen* отличается от создания текстовых файлов только указанием режима обмена – "rb" (двоичный для чтения), "rb+" (двоичный для чтения и записи), "wb" (двоичный для записи), "wb+" (двоичный для записи и чтения):

FILE \*f1;

.........

f1=fopen(имя\_файла, "режим");

Обычно для обмена с двоичными файлами используются функции *fread* и fwrite:

c\_w = fwrite(buf, size\_rec, n\_rec, f1);

Здесь

* buf – указатель типа void\* на начало буфера в оперативной памяти, из которого информация переписывается в файл;
* size\_rec – размер передаваемой порции в байтах;
* n\_rec – количество порций, которое должно быть записано в файл;
* f1 – указатель на блок управления файлом;
* c\_w – количество порций, которое фактически записалось в файл.

Считывание данных из двоичного файла осуществляется с помощью функции *fread* с таким же набором параметров:

c\_r = fread(buf, size\_rec, n\_rec, f1);

Здесь

* c\_r – количество порций, которое фактически прочиталось из файла;
* buf – указатель типа void\* на начало буфера в оперативной памяти, в который информация считывается из файла.

Обратите внимание на значения, возвращаемые функциями *fread* и fwrite. В какой ситуации количество записываемых порций может не совпасть с количеством записавшихся данных? Как правило, на диске не хватило места, и на такую ошибку надо реагировать. А вот при чтении ситуация, когда количество прочитанных порций не совпадает с количеством запрашиваемых порций, не обязательно является ошибкой. Типичная картина – количество данных в файле не кратно размеру заказанных порций.

Двоичные файлы допускают не только последовательный обмен данными. Так как размеры порций данных и их количество, участвующее в очередном обмене, диктуются программистом, а не смыслом информации, хранящейся в файле, то имеется возможность пропустить часть данных или вернуться повторно к ранее обработанной информации. Контроль за текущей позицией доступных данных в файле осуществляет система с помощью указателя, находящегося в блоке управления файлом. С помощью функции fseek программист имеет возможность переместить этот указатель:

fseek(f1,delta,pos);

Здесь

* f1 – указатель на блок управления файлом;
* delta – величина *смещения в байтах*, на которую следует переместить указатель файла;
* pos – позиция, от которой производится смещение указателя (0 или SEEK\_SET – от начала файла, 1 или SEEK\_CUR – от текущей позиции, 2 или SEEK\_END – от конца файла)

Кроме набора функций { *fopen*/fclose, *fread*/fwrite } для работы с двоичными файлами в библиотеке BC предусмотрены и другие средства – \_dos\_open /\_\_dos\_close, \_dos\_read /\_dos\_write, \_create /\_close, \_read /\_write. Однако знакомство со всеми возможностями этой библиотеки в рамках настоящего курса не предусмотрено.

Операторы:

fclose(f1); //закрытие файла

f1=fopen("c\_bin","rb"); //открытие двоичного файла для чтения

могут быть заменены обращением к единственной функции freopen, которая повторно открывает ранее открытый файл:

f1=freopen("c\_bin","rb");

Основное правило, которого надо придерживаться при обмене с двоичными файлами звучит примерно так – как данные записывались в файл, так они должны и читаться.

#### 10.1.3. Структурированные файлы

Структурированный файл является частным случаем двоичного файла, в котором в качестве порции обмена выступает структура языка C, являющаяся точным аналогом записи в Паскале. По сравнению с предыдущим примером использование записей позволяет сократить количество *обращений к функциям* *fread*/fwrite, т.к. в одном обращении участвуют все поля записи.

Инициализация структурированного файла выполняется точно таким же способом, как и подготовка к работе двоичного файла.

1. **Шаблоны и родовые функции в языке программирования C++. Основные понятия**

Одним из самых полезных видов классов является контейнерный *класс*, т.е. такой *класс*, который хранит объекты каких-то других типов. Списки, массивы, ассоциативные массивы и *множества* - все это *контейнерные классы* . К*онтейнерные классы* обладают тем интересным свойством, что тип содержащихся в них объектов не имеет особого значения для создателя контейнера, но для пользователя конкретного контейнера этот тип является существенным. Следовательно, тип содержащихся объектов должен быть параметром контейнерного класса, и создатель такого класса будет определять его с помощью типа-параметра. Для каждого конкретного контейнера (т.е. объекта контейнерного класса) *пользователь* будет указывать каким должен быть тип содержащихся в нем объектов.

### **Простой шаблон типа**

*Шаблон* типа для класса задает способ построения отдельных классов, подобно тому, как описание класса задает способ построения его отдельных объектов. Можно определить *стек*, содержащий элементы произвольного типа:

template<class T>

class stack {

T\* v;

T\* p;

int sz;

public:

stack(int s) { v = p = new T[sz=s]; }

~stack() { delete[] v; }

void push(T a) { \*p++ = a; }

T pop() { return \*--p; }

int size() const { return p-v; }

};

*Префикс* template<class T> указывает, что описывается *шаблон* типа с параметром T, обозначающим тип, и что это обозначение будет использоваться в последующем описании. После того, как *идентификатор* T указан в префиксе, его можно использовать как любое другое имя типа. *Область видимости* T продолжается до конца описания, начавшегося префиксом template<class T>. Отметим, что в префиксе T объявляется типом, и оно не обязано быть именем класса. Так, ниже в описании объекта sc тип T оказывается просто *char*.

Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном типа), и его можно использовать как все имена класса. Например, ниже определяется *объект* sc класса stack<char>:

stack<char> sc(100); // стек символов

В частности, применение шаблона типа не предполагает каких-либо средств динамической поддержки помимо тех, которые используются для обычных "ручных" классов. Не следует так же думать, что оно приводит к сокращению программы.

Имея *определение* шаблонного класса stack, можно следующим образом определять и использовать различные стеки:

stack<shape\*> ssp(200); // стек указателей на фигуры

stack<Point> sp(400); // стек структур Point

Важно составлять *определение* шаблона типа таким образом, чтобы его зависимость от глобальных данных была минимальной. Дело в том, *шаблон* типа будет использоваться для порождения функций и классов на основе заранее неизвестного типа и в неизвестных контекстах. Практически любая, даже слабая зависимость от контекста может проявиться как проблема при отладке программы пользователем, который, вероятнее всего, не был создателем шаблона типа. К совету избегать, насколько это возможно, использований глобальных имен, следует относиться особенно серьезно при разработке шаблона типа.

### Шаблоны типа для списка

На практике при разработке класса, служащего коллекцией объектов, часто приходится учитывать взаимоотношения использующихся в реализации классов, *управление памятью* и необходимость определить *итератор* *по* содержимому коллекции. Часто бывает так, что несколько родственных классов разрабатываются совместно . В качестве примера мы предложим семейство классов, представляющих односвязные списки и шаблоны типа для них.

#### 8.3.1 Список с принудительной связью

Вначале определим простой список, в котором предполагается, что в каждом заносимом в список объекте есть поле связи. Потом этот список будет использоваться как строительный материал для создания более общих списков, в которых объект не обязан иметь поле связи. Сперва в описаниях классов будет приведена только общая часть, а реализация будет дана в следующем разделе. Это делается затем, чтобы вопросы проектирования классов не затемнялись деталями их реализации.

Начнем с типа slink, определяющего поле связи в односвязном списке:

struct slink {

slink\* next;

slink() { next = 0; }

slink(slink\* p) { next = p; }

};

Теперь можно определить класс, который может содержать объекты любого, производного от slink, класса:

class slist\_base {

// ...

public:

int insert(slink\*); // добавить в начало списка

int append(slink\*); // добавить к концу списка

slink\* get(); // удалить и возвратить начало списка

// ...

};

Такой класс можно назвать списком с принудительной связью, поскольку его можно использовать только в том случае, когда все элементы имеют поле slink, которое используется как указатель на slist\_base. Само имя slist\_base (базовый *односвязный список*) говорит, что этот класс будет использоваться как базовый для односвязных списочных классов. Как обычно, при разработке семейства родственных классов возникает вопрос, как выбирать имена для различных членов семейства. Поскольку имена классов не могут перегружаться, как это делается для имен функций, для обуздания размножения имен перегрузка нам не поможет.

Здесь все нормально, но поскольку определение класса slist\_base дано через структуру slink, приходится использовать явное приведение типа для преобразования значения типа slink\*, возвращаемого функцией slist\_base::get(), в name\*. Это некрасиво. Для большой программы, в которой много списков и производных от slink классов, это к тому же чревато ошибками. Нам пригодилась бы надежная по типу версия класса slist\_base:

template<class T>

class Islist : private slist\_base {

public:

void insert(T\* a) { slist\_base::insert(a); }

T\* get() { return (T\*) slist\_base::get(); }

// ...

};

Имя Islist (intrusive singly *linked list*) обозначает *односвязный список* с принудительной связью. Этот шаблон типа можно использовать так:

void f(const char\* s)

{

Islist<name> ilst;

ilst.insert(new name(s));

// ...

name\* p = ilst.get();

// ...

delete p

}

1. **Исключительные ситуации. Основные причины возникновения. Обработка исключений**

***Устойчивость*** - это способность программной системы должным образом реагировать на *исключительные ситуации*. *Обработка* ***исключительных ситуаций*** - процесс, направленный на достижение *устойчивости*.

#### Три закона программотехники

##### Первый закон (закон для разработчика)

*Корректность* **системы - недостижима. Каждая последняя найденная ошибка является предпоследней**.

Этот *закон* отражает сложность нетривиальных систем. Разработчик всегда должен быть готов к тому, что в работающей системе имеются ситуации, в которых система работает не в точном соответствии со своей *спецификацией*, так что от него может требоваться очередное изменение либо системы, либо ее *спецификации*.

##### Второй закон (закон для пользователя)

**Не бывает некорректных систем. Каждая появляющаяся ошибка при эксплуатации системы - это следствие незнания** *спецификации* **системы**.

Есть два объяснения справедливости второго *закона*. Несерьезное объяснение состоит в том, что любая система, что бы она ни делала, при любом постусловии *корректна* по отношению к предусловию False, поскольку невозможно подобрать ни один набор входных данных, удовлетворяющих этому предусловию. Так что все системы *корректны*, если задать False в качестве их предусловия. Если вам пришлось столкнуться с системой, предусловие которой близко к False, то лучшее, что можно сделать, это отложить ее в сторону и найти другую.

##### Третий закон (закон чечако)

**Если** *спецификацию* **можно нарушить, - она будет нарушена. Новичок (чечако) способен "подвесить" любую систему**.

Какой бы *надежный код* ни был написан, сколь бы тщательной ни была *отладка*, в версии, переданной в эксплуатацию и на сопровождение, при запусках будут встречаться нарушения *спецификаций*. Причиной этого являются выше упомянутые законы *программотехники*. В системе остается последняя ошибка, находятся пользователи, не знающие *спецификаций*, и если *спецификацию* можно нарушить, то это событие когда-нибудь да произойдет. В таких *исключительных ситуациях* продолжение выполнения программы либо становится невозможным (попытка выполнить неразрешенную операцию деления на ноль, попытки записи в защищенную область памяти, попытка открытия несуществующего файла, попытка получить несуществующую *запись* *базы данных*), либо в возникшей ситуации применение алгоритма приведет к ошибочным результатам.

#### Обработка исключений в языках C/C++

Для стиля программирования на языке C характерно описание методов класса как булевых функций, возвращающих true в случае нормального завершения метода и false - при возникновении *исключительной ситуации*. Вызов метода встраивался в If -оператор, обрабатывающий ошибку в случае неуспеха завершения метода:

bool MyMethod(...){...}

if !MyMethod(){// обработка ошибки}

{//нормальное выполнение}

Недостатки этой схемы понятны. Во-первых, мало информации о причине возникновения ошибки, поэтому либо через поля класса, либо через аргументы метода нужно передавать дополнительную информацию. Во-вторых, блок обработки встраивается в каждый вызов, что приводит к раздуванию кода.

Поэтому в C/C++ применяется схема try/catch блоков, суть которой в следующем. Участок программы, в котором может возникнуть *исключительная ситуация*, оформляется в виде охраняемого try-блока. Если при его выполнении возникает *исключительная ситуация*, то происходит прерывание выполнения try-блока c классификацией исключения. Это исключение начинает обрабатывать один из catch-блоков, соответствующий *типу исключения*. В C/C++ применяются две такие схемы. Одна из них - *схема с возобновлением* - соответствует так называемым структурным, или С-исключениям. Вторая схема - *без возобновления* - соответствует С++ исключениям. В первой схеме обработчик исключения - catch-блок - возвращает управление в некоторую точку try-блока. Во второй схеме управление не возвращается в try-блок.

#### Выбрасывание исключений. Создание объектов Exception

В теле try-блока может возникнуть *исключительная ситуация*, приводящая к *выбрасыванию исключений*. Формально *выбрасывание исключения* происходит при выполнении оператора throw. Этот оператор, чаще всего, выполняется в недрах операционной системы, когда система команд или функция API не может сделать свою работу. Но этот оператор может быть частью программного текста try-блока и выполняться, когда в результате проведенного анализа становится понятным, что дальнейшая нормальная работа невозможна.

Синтаксически оператор throw имеет вид:

throw[выражение]

Выражение throw задает объект класса, являющегося наследником *класса Exception*. Обычно это выражение new, создающее новый объект. Если оно отсутствует, то повторно выбрасывается текущее исключение. Если исключение выбрасывается операционной системой, то она сама классифицирует исключение, создает объект соответствующего класса и автоматически заполняет его поля.

При выполнении оператора throw создается объект te, класс TE которого характеризует текущее исключение, а поля содержат информацию о возникшей *исключительной ситуации*. Выполнение оператора throw приводит к тому, что нормальный процесс вычислений на этом прекращается. Если это происходит в охраняемом try-блоке, то начинается этап *"захвата" исключения* одним из обработчиков исключений.

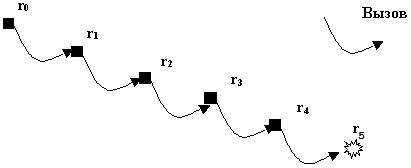
#### Захват исключения

Блок catch - обработчик исключения имеет следующий синтаксис:

catch (T e) {...}

Класс T, указанный в заголовке catch -блока, должен принадлежать *классам исключений*. Блок catch с формальным аргументом e класса T потенциально способен захватить текущее исключение te класса TE, если и только если объект teсовместим по присваиванию c объектом e. Другими словами, потенциальная способность захвата означает допустимость присваивания e = te, что возможно, когда класс TE является потомком класса T. Обработчик, класс T которого является *классом Exception*, является *универсальным обработчиком*, потенциально он способен захватить любое исключение, поскольку все они являются его потомками.

Потенциальных захватчиков может быть много, исключение захватывает лишь один - тот из них, кто стоит первым в списке проверки. Каков порядок проверки? Он довольно естественный. Вначале проверяются обработчики в порядке следования их за try -блоком, и первый потенциальный захватчик становится активным, захватывая исключение и выполняя его обработку. Отсюда становится ясно, что порядок следования в списке catch -блоков крайне важен. Первыми идут наиболее *специализированные обработчики*, далее по мере возрастания универсальности. Так, вначале должен идти обработчик исключения DivideByZeroException, а уже за ним - ArithmeticException. *Универсальный обработчик*, если он есть, должен стоять последним. За этим наблюдает *статический контроль типов*. Если потенциальных захватчиков в списке catch -блоков нет (сам список может отсутствовать), то происходит переход к списку обработчиков охватывающего блока. Напомню, что try -блок может быть вложен в другой try -блок. Когда же будет исчерпаны списки вложенных блоков, а потенциальный захватчик не будет найден, то произойдет подъем по стеку вызовов. На рис. 23.5 показана цепочка вызовов, начинающаяся с точки "большого взрыва" - процедуры Main.



#### Параллельная работа обработчиков исключений

Обработчику исключения - catch-блоку, захватившему исключение, - передается текущее исключение. Анализируя свойства этого объекта, обработчик может понять причину, приведшую к возникновению *исключительной ситуации*, попытаться ее исправить и в случае успеха продолжить вычисления. После завершения catch-блока выполняются операторы текущего метода, следующие за конструкцией try-catch-finally.

|  |
| --- |
| Зачастую обработчик исключения не может исправить ситуацию или может выполнить это лишь частично, предоставив решение оставшейся части проблем вызвавшему методу - предшественнику в цепочке вызовов. Механизм, реализующий такую возможность - это тот же механизм исключений. Как правило, в конце своей работы обработчик исключения выбрасывает исключение, выполняя оператор throw. При этом у него есть две возможности: повторно выбросить текущее исключение или выбросить новое исключение, содержащее дополнительную информацию. |

Таким образом, обработку возникшей *исключительной ситуации* могут выполнять несколько обработчиков, принадлежащие разным уровням цепочки вызовов.

1. **Библиотека стандартных шаблонов в языке программирования C++. Назначение, основные понятия**

Библиотека стандартных шаблонов (STL) (англ. Standard Template Library) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Библиотека стандартных шаблонов до включения в стандарт C++ была сторонней разработкой, вначале — фирмы HP, а затем SGI. Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальной части стандартной библиотеки (потоки ввода-вывода (iostream), подраздел Си и др.).

Каждый производитель компиляторов C++ обязательно поставляет какую-либо реализацию этой библиотеки, так как она является очень важной частью стандарта и широко используется.

Архитектура STL была разработана Александром Степановым и Менг Ли.

В библиотеке выделяют пять основных компонентов:

**Контейнер** (англ. container) — хранение набора объектов в памяти.

**Итератор** (англ. iterator) — обеспечение средств доступа к содержимому контейнера.

**Алгоритм** (англ. algorithm) — определение вычислительной процедуры.

**Адаптер** (англ. adaptor) — адаптация компонентов для обеспечения различного интерфейса.

**Функциональный объект** (англ. functor) — сокрытие функции в объекте для использования другими компонентами.

Разделение позволяет уменьшить количество компонентов.

Стандартная библиотека шаблонов языка С++ объединяет в себе контейнерные типы данных, алгоритмы для их обработки данных, итераторы для обращения к элементам или последовательностям в контейнер. STL также содержит набор шаблонов для обеспечения стандартного ввода-вывода.

Слово «стандартная» обозначает, что данная библиотека является частью стандарта языка С++ и должна рассматриваться, как первая альтернатива при выборе методов и средств работы с данными и потоками ввода-вывода.

Слово «шаблонов» обозначает, что вся библиотека построена на шаблонах классов и функций, что обеспечивает возможность унифицированной работы с различными типами данных. Использование шаблонов в библиотеке позволяет не только одинаково обрабатывать встроенные типы С++, но и работать с пользовательскими типами данных, которые не были известны в момент разработки библиотеки.

**STL обладает рядом преимуществ:**

· Код библиотеки написан профессиональными программистами, проверен и отлажен. Вам не придётся искать ошибки в реализации контейнеров или алгоритмов STL. Скорее, ошибки будут связаны с неверным пониманием концепций STL, но это вопрос опыта использования.

· Код библиотеки написан очень эффективно с точки зрения использования оперативной памяти и быстродействия для типовых вариантов применения.

· Библиотека предлагает унифицированный интерфейс, однообразный для всех контейнеров и алгоритмов, что после получения навыка использования библиотеки позволяет значительно повысить читаемость программы.

· Использование библиотеки позволяет приступить сразу к решению проектных задач, не задумываясь о реализации низкоуровневых контейнеров и алгоритмов. В случае работы проектной группы это позволяет избежать дублирования кода у различных разработчиков.

· Библиотека хорошо документирована и описана в книгах. В случае разработки собственных контейнеров и алгоритмов документация будет, скорее всего, значительно беднее, что повысит стоимость подготовки нового специалиста.

· Код, написанный с использованием STL легко переносится на другие компиляторы, операционные системы и платформы.

**К недостаткам STL можно отнести:**

· Неприспособленность к работе со структурными типами данных.

· Низкая эффективность (быстродействие, память) при решении частных задач, где возможны целевые оптимизации кода.

· Неадекватный интерфейс шаблона для работы со строками

· Сложность управления пулом памяти при работе с контейнерами STL.