МИНИСТЕРСТВО СЕЛЬСКОГО ХОЗЯЙСТВА РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

Е.В. Попова, А.М. Кумратова

КУРС ЛЕКЦИЙ
по дисциплине

Комплексы проблемно-ориентированных программ

Краснодар, 2014

Модуль 1. Основы проблемно-ориентированных программных комплексов

Лекция 1.

Понятие и виды проблемно-ориентированных программных комплексов. Изучение типовых задач решаемых проблемно-ориентированными программными комплексами, различных подходов к их классификации. Рассмотрение примеров программных комплексов и основных областей их применения.

Прикладное программное обеспечение предназначено для разработки и выполнения конкретных задач

Прикладное программное обеспечение работает под управлением базового ПО, в частности операционных систем.

Пакеты прикладных программ (ППП) – это комплекс программ предназначенный для решений задач определенного класса (функциональная подсистема, бизнес-приложение).

ППП общего назначения – универсальные программные продукты, предназначенные для автоматизации разработки и эксплуатации функциональных задач пользователя информационных систем в целом.

Редактор – это ППП, предназначенный для создания и изменения текстов, документов, графических данных и иллюстраций. Предназначены для документооборота в фирме.

Редакторы делятся на:

- текстовые
- графические
- издательские системы.

Текстовые редакторы предназначены для обработки текстовой документации.

Например: Word, Word Perfect (принадлежит фирме Corel), ChiWriter, MultiEdit.

Графические редакторы предназначены для обработки графических документов, включая диаграммы, иллюстрации чертежи, таблицы.

Допускается управление размером фигур и шрифтов, перемещение фигур и букв, формирование любых изображений.

Например: Paintbrush, Corel DRAW, Adobe Illustrator, Adobe Photoshop

Издательские системы соединяют в себе возможности текстовых и графических редакторов, обладают развитыми возможностями по форматированию полос с графическими материалами и последующим выводом на печать.

Например: Page Maker (фирмы Adobe),Ventura Publisher (корпорации Corel)

Электронные таблицы - это ППП, предназначенный для обработки таблиц.

Например: Excel, Lotus 1-2-3

Система управления базами данных – это специальные ППП, предназначенные для создания внутримашинного информационного обеспечения.

База данных – это совокупность специальным образом организованных наборов данных, хранящихся на диске.

Управление базой данных включает в себя ввод данных, их корректировку, манипулирование данными, т.е. добавление удаление, извлечение, обновление. В зависимости от способа организации данных различают:

- сетевые
- иерархические
- распределенные
- реляционные СУБД

Например: FoxPro, Paradox, СУБД компании Oracle

Интегрированные пакеты – ППП, объединяющие в себе функционально различные программные компоненты ППП общего назначения.

Современные ИП включают:

- текстовые редакторы;
- электронную таблицу;
- графический редактор;
- СУБД;
- коммуникационный модуль.

Дополнительно могут включаться такие компоненты как система экспорта-импорта файлов, калькуляторы, календарь, системы программирования.

Например: Office, FrameWork

Case – технологии применяются при создании сложных информационных систем, требующих коллективной реализации проекта, в котором участвуют различные специалисты: системные аналитики, проектировщики и программисты.

CASE – технологии представляют собой методологию проектирования ИС, А также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех этапах разработки и сопровождения ИС и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методологиях структурного (в основном) или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.

Например: Erwin –средство концептуального моделирования БД, использующее методологию IDEF1X, Erwin реализует проектирование схемы БД, генерацию ее описания на языке целевой СУБВ (SQL Server).

Экспертные системы (ЭС)

Основу ЭС составляет ***база знаний***, в которую закладывается информация о данной предметной области.

Две основные формы представления знаний в ЭС: факты и правила.

Факты фиксируют количественные и качественные показатели явлений и процессов.

Правила описывают соотношения между фактами, обычно в виде логических условий связывающих причины и следствия.

Экспертные системы – это системы обработки информации знаний в узкоспециализированной области подготовки решений пользователей на уровне профессиональных экспертов.

Например: оболочки в экономике, Ш ЭДЛ (Диалог)

Методо-ориентированные ППП – в их основе лежит какой-либо экономико-математический метод решения задачи.

К ним относятся ППП:

- математического программирования (линейного, динамического, статистического);
- сетевого планирования и управления;
- теории массового обслуживания;
- математической статистики.

Проблемно-ориентированными ППП называются программные продукты, предназначенные для решения какой-либо задачи в конкретной функциональной области.

Проблемно-ориентированные ППП для промышленной сферы. Направление развития - создание интегрированных информационных систем, отвечающих новым требованиям:

1. они должны планировать производство усовершенствованными методиками (комплексный производственный график, потребности в материалах, мощностях), контролировать выполнение плана работ (управление запасами, клиентскими заказами, заказами-нарядами, заказами на закупку), составлять технологические карты, управлять финансовыми и трудовыми ресурсами, «непроизводственные» функции – контроль сервисного обслуживания, распределение готовой продукции и маркетинг;
2. ориентированы на архитектуру клиент-сервер, строятся на основе многозадачных, многопользовательских операционных систем (UNIX) и реляционных баз данных, разрабатываются на базе CASE-технологий и имеют графический пользовательский интерфейс;
3. современные системы способны поддерживать различные типы производства: изготовление «про запас», разработку и изготовление изделия

на заказ, сборку на заказ, мелко- и крупносерийные производства, производства с непрерывном циклом, смешанный тип.

Проблемно-ориентированные ППП непромышленной сферы, предназначенные для автоматизации деятельности фирм, не связанных с материальным производством (банки, биржи, торговля).

Банковские ППП, финансовые ППП, правовые ППП.

ППП различных предметных областей:

- ***ППП бухгалтерского учета.***

Например: 1С:Бухгалтерия, Парус, Монолит-Инфо, Бест, Инфобухгалтер, ППП БУ «Офис».

- ***ППП финансового менеджмента***

Например: ЭДИП (Центринвест Софт),
Альт Финансы (Альт)
Финансовый анализ (Инфософт)

- ***Оценка эффективности:***

Project Expert,
Альт-Инвест (Альт),
Инвестор (ИнЭк)

ППП правовых справочных систем. - инструмент работы с огромным объемом законодательной информации, поступающей непрерывным потоком.

Например: “Консультант Плюс”, “Гарант”

ППП глобальных сетей - обеспечение удобного, надежного доступа пользователя к территориально распределенным общесетевым ресурсам, базам данных, передаче сообщений и т.д. Используются для организации электронной почты, телеконференции, электронной доски объявлений, обеспечения секретности передаваемой информации в различных глобальных сетях ЭВМ.

Например: Explorer, Mail – электронная почта.

Обеспечение организации администрирования вычислительного процесса в локальных и глобальных сетях ЭВМ - это ППП, управляющие администрированием данных, коммутаторами, концентраторами, маршрутизаторами, трафиком сообщений.

Лекция 2.

Использование специализированных инструментальные средства и универсальных языков программирования для разработки проблемно-ориентированных программных комплексов. Обзор основных технологий создания проблемно-ориентированных программных комплексов. Рассмотрение специализированных инструментальные средства и универсальных языков программирования используемых для разработки проблемно-ориентированных программных комплексов.

Язык программирования – формализованный язык для описания алгоритма решения задачи на компьютере.

Поколения языков программирования

Языки программирования делят на 5 поколений.

1. **Начало 50-х годов**. Появились первые компьютеры. Язык ассемблера, созданный по принципу «одна инструкция – одна строка».
2. **Конец 50-х - начало 60-х годов.** Символический ассемблер, в котором появилось понятие переменной.
3. **60-е годы.** Универсальные языки высокого уровня. Качества языков: относительная простота, независимость от конкретного компьютера, возможность использования мощных синтаксических конструкций. Пишутся небольшие программы (инженерного, экономического характера). Специалисты из некомпьютерных областей.
4. **Начало 70-х годов – по настоящее время.** Языки 4-го поколения – предназначены для реализации крупных проектов, повышения их надежности и скорости создания. Ориентированы на специализированные области применения. Используются проблемно-ориентированные языки.
5. **Середина 90-х годов**. Системы автоматического создания прикладных программ с помощью визуальных средств разработки, без знания программирования. Главная идея – возможность автоматического формирования результирующего текста на универсальных языках программирования (который потом требуется откомпилировать). Инструкции вводятся в компьютер в наглядном виде с помощью методов, наиболее удобных для человека, незнакомого с программированием.

Классификация языков программирования представлена на рис. 1.

Системы программирования предназначены для совершенствования процесса разработки и отладки программ, т.е. для повышения эффективности и производительности труда программистов.

Системы программирования включают:

- входной язык системы программирования,
- транслятор,
- библиотеку стандартных подпрограмм,
- документацию.

Языки программирования, или алгоритмические языки, классифицируются:

- по степени их зависимости от вычислительной машины;

- по назначению (ориентации на ту или иную сферу применения);
- по специфике организац. структуры языковых конструкций и т.п. рис. 1.

Рисунок 1 – Схема классификации языков программирования

С учётом зависимости от ЭВМ языки программирования подразделяются на машинно-зависимые и машинно-независимые.

Структура и средства машинно-зависимых языков отражают (учитывают) специфику функционирования определённого класса ЭВМ. При программировании задач с помощью таких языков требуется знание не только сущности реализуемого алгоритма решения задачи, но и технических особенностей конкретной ЭВМ и специфики способов написания для неё программ.

К машинно-зависимым языкам относятся машинные языки, т.е. языки непосредственно используемые для управления работой отдельных устройств ЭВМ. Машинный язык представляет собой систему инструкций и данных, которые не требуют трансляции, могут непосредственно интерпретироваться и исполняться аппаратными средствами ЭВМ. Программирование на этих языках осуществлялось на ЭВМ первого и частично второго поколения.

К машинно-зависимым языкам программирования также относятся машинно-ориентированные языки, основные конструктивные средства которых позволяют учитывать особенности архитектуры и принципов работы определённой ЭВМ или ряда ЭВМ, но в отличие от машинных языков требуют предварительной трансляции на машинный язык программ, составленных с их помощью.

К данному виду языков программирования относятся: автокоды, языки символического кодирования и ассемблеры. Программирование на машинно-ориентированных языках (ассемблерах) характерно и для современных ПК, т.к. в языке ассемблера допускается использование средств, присущих языкам высокого уровня (макрорасширений, выражений, и т.п.).

Язык ассемблера используется в системном программировании:

- программирование микропроцессоров;
- разработка операционных систем или их компонентов;
- разработка драйверов.

Машинно-независимые языки (или языки высокого уровня) не требуют от пользователя полного знания специфики ЭВМ, на которой реализуется программа решения задачи. Инструментальные средства этих языков программирования позволяют записывать программу в виде, допускающем её реализацию на ЭВМ с различными типами машинных операций, привязка к которым возлагается на соответствующий транслятор.

Решение задач на этих языках описывается в наглядном виде.

Обособленное, промежуточное положение между машинно-независимыми машинно-зависимыми языками занимает язык Си, создание которого явилось результатом попытки объединения достоинств, присущих языкам обоих классов.

Язык Си и его модификация в настоящее время используется для создания системных и прикладных программных продуктов, в которых решающее значение отводится быстродействию и минимизации объёмов памяти. На языке Си полностью написано ядро операционной системы UNIX, вследствие чего её легко можно было изменять и модернизировать.

Машинно-независимые языки классифицируются на процедурно-ориентированные и проблемно-ориентированные.

Процедурно-ориентированные (универсальные) языки эффективны при описании алгоритмов решения задач. Из языков этого класса наиболее известны: Фортран, Кобол, ПЛ/1, Бейсик, Паскаль, Ада.

Проблемно-ориентированные предназначены для описания процессов обработки информации в более узкой, специфической области. Наиболее известными языками этой группы являются: РПГ, Лисп, АПЛ, GPSS.

Объектно-ориентированные – языки, ориентированные на разработку программных приложений для широкого круга разнообразных по сфере приложения задач, имеющих общность в реализуемых компонентах. Объектно-ориентированный подход в программировании позволяет применять одни и те же (типовые) архитектурные и концептуальные решения для быстрого создания эффективных программных приложений.

Обзор языков программирования

Fortran (Фортран). - первый компилируемый язык, созданный Джимом Бэкусом в 50-е годы. Это первый процедурно-ориентированный язык высокого уровня, предназначенный для описания алгоритмов решения вычислительных задач научного и инженерно-технического характера. Создано большое количество библиотек и пакетов. Распространены версии

Фортран 4, Фортран 77, Фортран 90, ориентированные на решение математических задач. Версия Фортран F2K – в 2000 году. Стандартная версия Фортрана HPF - для параллельных суперкомпьютеров со множеством процессоров.

Cobol (Кобол) – общекоммерческий язык программирования, разработан в 1961 году ассоциацией CODASYL, для решения экономических задач. Создано много приложений на этом языке. Наибольшую зарплату в США получают программисты на Коболе.

Algol (Алгол) – компилируемый язык, созданный в 1960 году. Призван был заменить Фортран, но не получил распространения из-за сложной структуры. В 1968 году - версия Алгол –68 – не удалось создать своевременно хороших компиляторов.

Pascal (Паскаль) – создан в конце 70-х годов Никлаусом Виртом, предназначен для решения вычислительных и информационно-логических задач. Версии Турбо Паскаль и Паскаль плюс предоставляют возможность параллельного программирования.

Basic (Бейсик) – (многоцелевой язык символических инструкций для начинающих). Разработан в 1963 году группой студентов Дартмундского колледжа США в качестве учебного языка. По популярности занимает первое место в мире. Созданы более мощные версии Quick Basic и Visual Basic. Имеются компиляторы и интерпретаторы для этого языка.

C (Си) – язык программирования, разработанный Д. Ритчи в 1972 году в лаборатории Bell, для облегчения процесса переноса программного обеспечения с одной ЭВМ на другую. В языке сочетаются возможности языков высокого уровня и непосредственной адресации к аппаратным средствам ЭВМ на уровне языка ассемблера. Си во многом похож на Паскаль и имеет дополнительные средства для прямой работы с памятью (указатели). На этом языке в 70-е годы написано множество прикладных и системных программ и ряд известных операционных систем (Unix).

C++ (Си++) – объектно-ориентированное расширение языка Си, созданное Бьярном Страуструпом в 1980 году. Множество новых мощных возможностей. Создание сложных и надежных программ потребовало от разработчиков высокого уровня профессиональной подготовки.

Java (Джава, Ява) – язык создан компанией Sun в начале 90-х годов на основе Си++. Он признан упростить разработку приложений на основе Си++ путём исключения из него всех низкоуровневых возможностей. Главная особенность языка – компиляция не в машинный код, а в платформно-независимый байт-код (каждая команда занимает 1 байт). Занимает по популярности второе место в мире после Бейсика. Основной недостаток языка – невысокое быстродействие, так как язык Ява интерпретируемый. Язык предназначен для создания надёжных, переносимых, распределённых сетевых программных приложений, работающих в различных оконных системах в условиях архитектуры «клиент-сервер», а также для администраторов сети, использующих Java-приложения для улучшения интерактивных качеств Web-серверов.

Ada (Ада) – язык программирования сверхвысокого уровня, разработан в 1983 году по заказу Министерства обороны США. Назван в честь автора

идеи программного управления (Августы Ады Лавлейс – дочери английского поэта, Дж. Байрона). Язык разработан небольшой группой под руководством Жана Ишбиа. Структура самого языка похожа на Паскаль. Этот язык ориентирован на применение в системах реального времени и предназначен для разработки программного обеспечения встроенных вычислительных систем. Используется также как язык спецификаций для описания требований, программирования ввода-вывода, взаимодействия во времени, обнаружения динамических ошибок и ведения длительного сопровождения.

RPG (РПГ) – генератор отчетов – предназначен для создания и обработки файлов и формирования выходных документов.

Симскрипт – язык программирования, ориентированный на описание дискретных процессов. Имеет развитые средства обработки стихов, синхронизации параллельных процессов, воспроизведения изменений состояния моделируемого процесса.

GPSS – система программирования фирмы Westi, ориентированная на моделирование систем с помощью событий. В терминах этого языка легко описывается и исследуется класс моделей массового обслуживания, а также другие системы, работающие в реальном масштабе времени.

PL/1 (ПЛ/1) – язык программирования разработан фирмой IBM и опубликован в середине 60-х годов. Явился попыткой синтезировать лучшие свойства наиболее распространенных в то время языков программирования: Фортрана, Кобола, Алгола-60, а также включить ряд новых свойств с целью их замены. Разрабатывался как универсальный язык программирования, удобный для решения широкого класса вычислительных и информационных задач, поэтому он располагает большим набором средств обработки цифровой и текстовой информации. Допускает параллельную обработку программ, обеспечивает возможность работы с разнотипными данными, со сложными структурами данных (массивами, таблицами, картотеками, текстами), имеет большой набор встроенных функций и процедур. Эти достоинства сделали язык сложным для освоения, а компилятор с этого языка оказался малоэффективным, для него впоследствии был создан специальный оптимизирующий транслятор (оптимайзер). Западные фирмы предприняли попытки создания на базе ПЛ/1 версий для персонального компьютера (ПЛ/М – фирмы Intel, ПЛ/Z – фирмы Zilog и ПЛ/65 – фирмы Rockwell International).

Симула – язык программирования для моделирования дискретных процессов. Первая версия появилась в 1964 году как расширение языка Алгол-60, в 1967 году появилась версия Симула-67. Особенностью языка является наличие средств описания объектов моделирования в виде вложенных структур, называемых классами.

Снобол – непроцедурный язык программирования, предназначенный для описания задач преобразования и обработки текстовых данных. Первая версия языка разработана в США в 1962 году, в настоящее время используется версия Снобол-4.

Модула-2 – язык предложен Н.Виртом с целью обеспечения высокоуровневыми языковыми средствами коллективной разработки высоконадежных и эффективных программных систем. В язык вошли все наиболее удачные средства и конструкции языка Паскаль.

APL (АПЛ) – алгоритмический язык программирования высокого уровня, отличающийся большим набором операций над матрицами, векторами, строками, удобен для работы в режимах разделения времени и диалога. Применяется для разработки диалоговых программ, в первую очередь для задач, связанных со статистической обработкой больших массивов информации, представленной в матричном виде, требует специальной клавиатуры (содержащей набор специальных символов, реализующих различные функции формирования и преобразования матриц, векторов и т.д.).

Smalltalk (Смолток) – работа над языком началась в 1970 году в исследовательской лаборатории корпорации XEROX, закончилась спустя 10 лет, воплотившись в окончательном варианте интерпретатора SMALLTALK-80. Синтаксис языка компактен и базируется на понятии объекта. Отсутствуют в языке операторы или данные. Все, что входит в Смолток, является объектами, сами объекты общаются друг с другом исключительно с помощью сообщений. Сегодня версия Visual Age for Smalltalk активно развивается компанией IBM.

QBE – программирование на примере.

Forth (Форт) – результат попытки Чарльза Мура в 70-х годах создать язык, обладающий мощными средствами программирования, который мог бы быть реализован на компьютерах с небольшими объемами памяти, а компилятор мог бы выдавать очень быстрый и компактный код – то есть служил заменой ассемблеру. Из-за программного текста, записанного в непривычной форме, сильно затруднялся поиск ошибок.

LISP (Лисп) – интерпретируемый язык программирования, созданный в 1960 году Джоном Маккарти. Ориентирован на структуру данных в форме списка и позволяет организовывать эффективную обработку больших объемов текстовой информации. Язык нашел широкое применение в программировании систем искусственного интеллекта.

Prolog (Пролог) – язык логического программирования. Создан в начале 70-х годов Аланом Колмероэ. Главное назначение языка – разработка интеллектуальных программ и систем. Это специальный язык программирования, созданный специально для работы с базами знаний, основанных на фактах и правилах (один из элементов искусственного интеллекта). В языке реализован механизм возврата для выполнения обратной цепочки рассуждений, при котором предполагается, что некоторые выводы или заключения истинны, а затем эти предположения проверяются в базе знаний, содержащей факты и правила логического вывода. Если предположение не подтверждается, выполняется возврат и выдвигается новое предположение.

Delphi – объектно-ориентированный язык программирования, созданный на базе языка Паскаль специалистами фирмы Borland. Обладая мощностью и гибкостью языков Си и Си++, превосходит их по удобству и простоте интерфейса при разработке приложений, обеспечивающих взаимодействие с базами данных и поддержку различного рода работ в рамках корпоративных сетей и сети Интернет.

SQL – основанные на исчислении отношений. Исп. в реляционных СУБД в качестве языка запросов к БД и языка программ-ия задач обработки данных.

Модуль 2. Использования языка программирования С++ для создания проблемно-ориентированных программных комплексов.

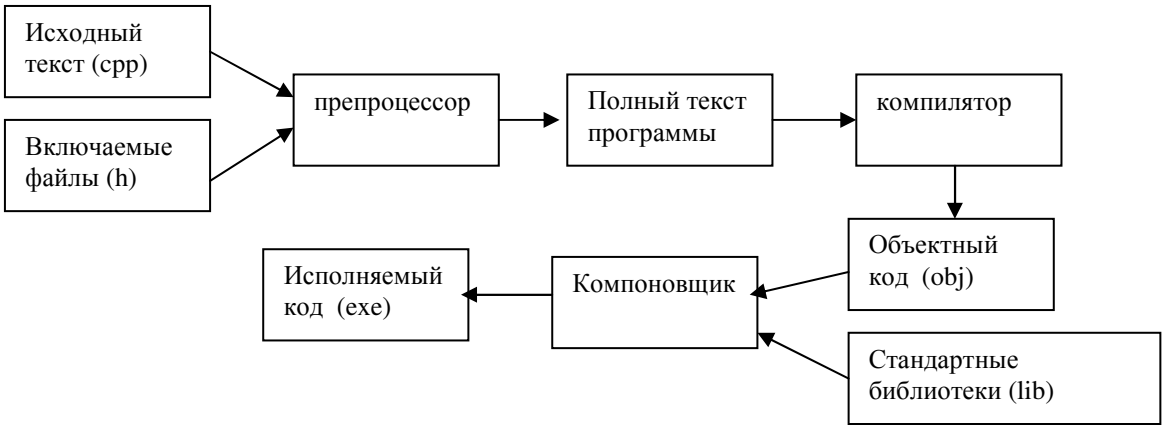
Лекция 3.

1. Программа на языке Си имеет следующую структуру:

#директивы препроцессора
. . . . . .
#директивы препроцессора
функция а ( )
    операторы
функция в ( )
    операторы
void main ( ) //функция, с которой начинается выполнение программы
    операторы
        описания
        присваивания
        функция
        пустой оператор
            составной
            выбора
            циклов
            перехода

Директивы препроцессора - управляют преобразованием текста программы до ее компиляции. Исходная программа, подготовленная на СИ в виде текстового файла, проходит 3 этапа обработки:

1) препроцессорное преобразование текста ;
2) компиляция;
3) компоновка (редактирование связей или сборка).



После этих трех этапов формируется исполняемый код программы. Задача препроцессора - преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора. Директива начинается с #. Например,

1) #define - указывает правила замены в тексте.

#define ZERO 0.0

Означает , что каждое использование в программе имени ZERO будет заменяться на 0.0.

2) #include< имя заголовочного файла> - предназначена для включения в текст программы текста из каталога «Заголовочных файлов», поставляемых вместе со стандартными библиотеками. Каждая библиотечная функция Си имеет соответствующее

описание в одном из заголовочных файлов. Список заголовочных файлов определен стандартом языка. Употребление директивы include не подключает соответствующую стандартную библиотеку, а только позволяют вставить в текст программы описания из указанного заголовочного файла. Подключение кодов библиотеки осуществляется на этапе компоновки, т. е. после компиляции. Хотя в заголовочных файлах содержатся все описания стандартных функций, в код программы включаются только те функции, которые используются в программе.

После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы.

Программа представляет собой набор описаний и определений, и состоит из набора функций. Среди этих функций всегда должна быть функция с именем main. Без нее программа не может быть выполнена. Перед именем функции помещаются сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип void: void main ( ). Каждая функция, в том числе и main должна иметь набор параметров, он может быть пустым, тогда в скобках указывается (void).

За заголовком функции размещается тело функции. Тело функции - это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Каждое определение, описание или оператор заканчивается точкой с запятой.

Определения - вводят объекты (объект - это именованная область памяти, частный случай объекта - переменная), необходимые для представления в программе обрабатываемых данных. Примером являются

```
int y = 10 ; //именованная константа
float x ;   //переменная
```

Описания - уведомляют компилятор о свойствах и именах объектов и функций, описанных в других частях программы.

Операторы - определяют действия программы на каждом шаге ее исполнения.

Пример программы на Си:

```
#include <stdio.h>  //препроцессорная директива
void main()         //функция
{                   //начало
printf("Hello! ");  //печать
}                   //конец
```

Контрольные вопросы

Из каких частей состоит программа на С++?

1. Чем определение отличается от объявления?
2. Перечислить этапы создания исполняемой программы на языке С++.
3. Что такое препроцессор?
4. Что такое директива препроцессора? Привести примеры директив препроцессора.
5. Составить программу, которая печатает текст «Моя первая программа на С++»

2. Базовые средства языка СИ++

2.1. Состав языка

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Алгоритмический язык также содержит такие элементы, только слова называют лексемами (элементарными конструкциями), словосочетания – выражениями, предложения – операторами. Лексемы образуются из символов, выражения из лексем и символов, операторы из символов выражений и лексем (Рис. 2)

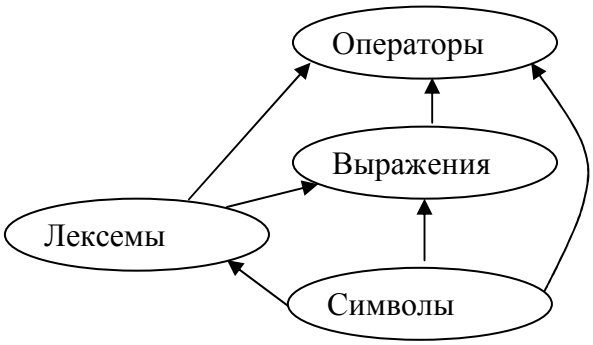


Рис. 2. Состав алгоритмического языка

Таким образом, элементами алгоритмического языка являются:

1) *Алфавит языка СИ++,* который включает
   - прописные и строчные латинские буквы и знак подчеркивания;
   - арабские цифры от 0 до 9;
   - специальные знаки “{ },| []()+-/%*.\’:;&?<>=!#^
   - пробельные символы (пробел, символ табуляции, символы перехода на новую строку).
2) Из символов формируются лексемы языка:
   - *Идентификаторы* – имена объектов СИ-программ. В идентификаторе могут быть использованы латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, PROG1, prog1 и Prog1 – три различных идентификатора. Первым символом должна быть буква или знак подчеркивания (но не цифра). Пробелы в идентификаторах не допускаются.
   - *Ключевые* (зарезервированные) слова – это слова, которые имеют специальное значение для компилятора. Их нельзя использовать в качестве идентификаторов.
   - Знаки операций – это один или несколько символов, определяющих действие над операндами. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в этой операции операндов.
   - *Константы* – это неизменяемые величины. Существуют целые, вещественные, символьные и строковые константы. Компилятор выделяет константу в качестве лексемы (элементарной конструкции) и относит ее к одному из типов по ее внешнему виду.
   - *Разделители* – скобки, точка, запятая пробельные символы.

Продолжение лекции 3

Типы данных, операторы и управляющие конструкции. Основные встроенные типы данных языка С++. Операторы их назначение, свойства, приоритет и примеры использования. Особенности управляющих конструкций языка С++.

2.1.1. Константы в Си++

Константа – это лексема, представляющая изображение фиксированного числового, строкового или символьного значения.

Константы делятся на 5 групп:

- целые;
- вещественные (с плавающей точкой);
- перечислимые;
- символьные;
- строковые.

Компилятор выделяет лексему и относит ее к той или другой группе, а затем внутри группы к определенному типу по ее форме записи в тексте программы и по числовому значению.

Целые константы могут быть десятичными, восьмеричными и шестнадцатеричными. Десятичная константа определяется как последовательность

десятичных цифр, начинающаяся не с 0, если это число не 0 (примеры: 8, 0, 192345). Восьмеричная константа – это константа , которая всегда начинается с 0. За 0 следуют восьмеричные цифры (примеры: 016 – десятичное значение 14, 01). Шестнадцатеричные константы – последовательность шестнадцатеричных цифр, которым предшествуют символы 0x или 0X (примеры: 0xA, 0X00F).

В зависимости от значения целой константы компилятор по-разному представит ее в памяти компьютера (т. е. компилятор припишет константе соответствующий тип данных).

Вещественные константы имеют другую форму внутреннего представления в памяти компьютера. Компилятор распознает такие константы по их виду. Вещественные константы могут иметь две формы представления: с фиксированной точкой и с плавающей точкой. Вид константы с фиксированной точкой:[цифры].[цифры] (примеры: 5.7, .0001, 41.).Вид константы с плавающей точкой: [цифры][.][цифры]E|e[+|-][цифры] (примеры:0.5e5, .11e-5, 5E3). В записи вещественных констант может опускаться либо целая, либо дробная части, либо десятичная точка, либо признак экспоненты с показателем степени.

Перечислимые константы вводятся с помощью ключевого слова enum. Это обычные целые константы, которым приписаны уникальны и удобные для использования обозначения. Примеры: enum { one=1, two=2, three=3,four=4};

enum {zero,one,two,three} – если в определении перечислимых констант опустить знаки = и числовые значения, то значения будут приписываться по умолчанию. При этом самый левый идентификатор получит значение 0, а каждый последующий будет увеличиваться на 1.

enum { ten=10, three=3, four, five, six};

enum {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} ;

Символьные константы – это один или два символа, заключенные в апострофы. Символьные константы, состоящие из одного символа, имеют тип char и занимают в памяти один байт, символьные константы, состоящие из двух символов, имеют тип int и занимают два байта. Последовательности, начинающиеся со знака \ , называются управляющими, они используются:

- Для представления символов, не имеющих графического отображения, например:
  \a – звуковой сигнал,
  \b – возврат на один шаг,
  \n – перевод строки,
  \t – горизонтальная табуляция.
- Для представления символов: \ , ’ , ? , ” (\\, \’ ,\? ,\”).
- Для представления символов с помощью шестнадцатеричных или восьмеричных кодов (\073, \0xF5).

Строковая константа – это последовательность символов, заключенная в кавычки. Внутри строк также могут использоваться управляющие символы. Например: “\nНовая строка”,

“\n\”Алгоритмические языки программирования высокого уровня \”” .

2.2. Типы данных в Си++

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. В Си++ определено 6 простых типов данных:

int (целый)
char (символьный)
wchar_t (расширенный символьный) } целочисленные
bool (логический)
float(вещественный) } с плавающей точкой
double (вещественный с двойной точностью) (число=мантисса x $10^{к}$)

Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов

short (короткий)
long (длинный)
signed (знаковый)
unsigned (беззнаковый)

Лекция 4.

Массивы, указатели и структуры. Одномерные и двумерные массивы. Указатели, адресная арифметика, операции * и &. Связь указателей и массивов, массивы указателей и указатели на массивы. Структуры, описание и обращение к полям, указатели на структуры.

3. Массивы

В языке Си/Си++ ,кроме базовых типов, разрешено вводить и использовать производные типы, полученные на основе базовых. Стандарт языка определяет три способа получения производных типов:

- массив элементов заданного типа;
- указатель на объект заданного типа;
- функция, возвращающая значение заданного типа.

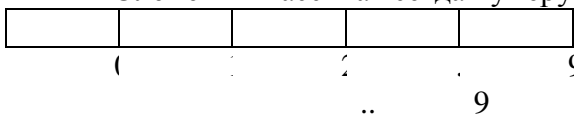
Массив – это упорядоченная последовательность переменных одного типа. Каждому элементу массива отводится одна ячейка памяти. Элементы одного массива занимают последовательно расположенные ячейки памяти. Все элементы имеют одно имя - имя массива и отличаются индексами – порядковыми номерами в массиве. Количество элементов в массиве называется его размером. Чтобы отвести в памяти нужное количество ячеек для размещения массива, надо заранее знать его размер. Резервирование памяти для массива выполняется на этапе компиляции программы.

3.1. Определение массива в Си/Си++

int a[100];//массив из 100 элементов целого типа

Операция sizeof(a) даст результат 400, т. е.100 элементов по 4 байта.

Элементы массива всегда нумеруются с 0.



.. 9

Чтобы обратиться к элементу массива, надо указать имя массива и номер элемента в массиве (индекс):

a[0] – индекс задается как константа,
a[55] – индекс задается как константа,
a[I] – индекс задается как переменная,
a[2*I] – индекс задается как выражение.

Элементы массива можно задавать при его определении:

int a[10]={1,2,3,4,5,6,7,8,9,10} ;

Операция sizeof(a) даст результат 40, т. е.10 элементов по 4 байта.

int a[10]={1,2,3,4,5};

Операция sizeof(a) даст результат 40, т. е.10 элементов по 4 байта. Если количество начальных значений меньше, чем объявленная длина массива, то начальные элементы массива получат только первые элементы.

int a[]={1,2,3,4,5};

Операция sizeof(a) даст результат 20, т. е.5 элементов по 4 байта. Длин массива вычисляется компилятором по количеству значений, перечисленных при инициализации.

3.2. Обработка одномерных массивов

При работе с массивами очень часто требуется одинаково обработать все элементы или часть элементов массива. Для этого организуется перебор массива.

Перебор элементов массива характеризуется:

- направлением перебора;
- количеством одновременно обрабатываемых элементов;
- характером изменения индексов.

По направлению перебора массивы обрабатывают :

- слева направо (от начала массива к его концу);
- справа налево (от конца массива к началу);
- от обоих концов к середине.

Индексы могут меняться

- линейно (с постоянным шагом);
- нелинейно (с переменным шагом).

3.2.1. Перебор массива по одному элементу

Элементы можно перебирать:

1) Слева направо с шагом 1, используя цикл с параметром

```
for(int I=0;I<n;I++){обработка a[I];}
```

2) Слева направо с шагом отличным от 1, используя цикл с параметром

```
for (int I=0;I<n;I+=step){обработка a[I];}
```

3) Справа налево с шагом 1, используя цикл с параметром

```
for(int I=n-1;I>=0;I--){обработка a[I];}
```

4) Справа налево с шагом отличным от 1, используя цикл с параметром

```
for (int I=n-1;I>=0;I-=step){обработка a[I];}
```

3.2.2 Формирование псевдодинамических массивов

При описании массива в программе надо обязательно указывать количество элементов массива для того, чтобы компилятор выделил под этот массив нужное количество памяти. Это не всегда бывает удобно, т. к. число элементов в массиве может меняться в зависимости от решаемой задачи. Динамические массивы реализуются с помощью указателей (см. далее).

Псевдодинамические массивы реализуются следующим образом:

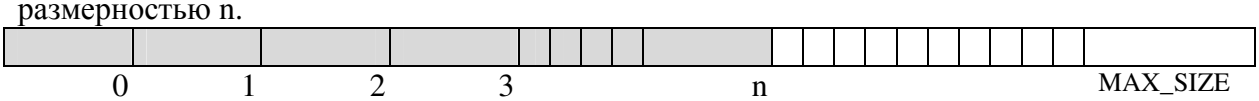
1) при определении массива выделяется достаточно большое количество памяти:

```
const int MAX_SIZE=100;//именованная константа
int mas[MAX_SIZE];
```

2) пользователь вводит реальное количество элементов массива меньшее N.

```
int n;
cout<<"\nEnter the size of array<"<<MAX_SIZE<<":";cin>>n;
```

3) дальнейшая работа с массивом ограничивается заданной пользователем размерностью n.



Т. о. используется только часть массива.

3.2.3. Использование датчика случайных чисел для формирования массива.

Датчик случайных чисел (ДСЧ) – это программа, которая формирует псевдослучайное число. Простейший ДСЧ работает следующим образом:

1) Берется большое число К и произвольное $x_0 \in [0,1]$.
2) Формируются числа x1=дробная_часть(x0*K); x2=дробная_часть(x1*K); и т. д.

В результате получается последовательность чисел x0, x1, x2,. . . беспорядочно разбросанных по отрезку от 0 до 1. Их можно считать случайными, а точнее псевдослучайными. Реальные ДСЧ реализуют более сложную функцию f(x).

В Си++ есть функция

int rand() – возвращает псевдослучайное число из диапазона 0..RAND_MAX=32767, описание функции находится в файле <stdlib.h>.

Пример формирования и печати массива с помощью ДСЧ:

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<" ";
}
}
```

В этой программе используется перебор массива по одному элементу слева направо с шагом 1.

Задача 1

Найти максимальный элемент массива.

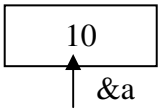
```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<" ";
}
int max=a[0];
for(I=1;I<n;I++)
if (a[I]>max)max=a[I];
cout<<"\nMax="<<max";
}
```

В этой программе также исп. перебор массива по одному элементу слева направо с шагом 1.

4.1. Понятии указателя

Указатели являются специальными объектами в программах на Си++. Указатели предназначены для хранения адресов памяти.

Пример: Когда компилятор обрабатывает оператор определения переменной, например, int i=10;, то в памяти выделяется участок памяти в соответствии с типом переменной (int=> 4байта) и записывает в этот участок указанное значение. Все обращения к этой переменной компилятор заменит на адрес области памяти, в которой хранится эта переменная.



Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями. Указатель не является самостоятельным типом, он всегда связан с каким-то другим типом.

Указатели делятся на две категории: указатели на объекты и указатели на функции. Рассмотрим указатели на объекты, которые хранят адрес области памяти, содержащей данные определенного типа .

В простейшем случае объявление указателя имеет вид:

тип *имя;

Тип может быть любым, кроме ссылки.

Примеры:

```
int *i;
double *f, *ff;
char *c;
```

Размер указателя зависит от модели памяти. Можно определить указатель на указатель: int**a;

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

Примеры:

```
1. int i;          //целая переменная
const int ci=1; //целая константа
int *pi;           //указатель на целую переменную
const int *pci;//указатель на целую константу
```

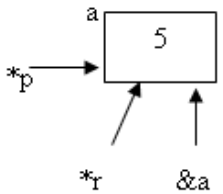
Указатель можно сразу проинициализировать:

```
int *pi=&i;      //указатель на целую переменную
const int *pci=&ci;;//указатель на целую константу
```

1. 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 указателя, а если он находится слева от типа (т. е. слева от *), то он запрещает изменение значения, на которое указывает указатель.



Для инициализации указателя существуют следующие способы:

1) Присваивание адреса существующего объекта:

1) с помощью операции получения адреса

```
int a=5;
int *p=&a; или int p(&a);
```

2) с помощью проинициализированного указателя

```
int *r=p;
```

3) адрес присваивается в явном виде

```
char*cp=(char*)0x B800 0000;
```

где 0x B800 0000 – шестнадцатеричная константа, (char*) – операция приведения типа.

4) присваивание пустого значения:

```
int*N=NULL;
int *R=0;
```

Лекция 5.

Использование функций стандартной библиотеки. Классификация функций стандартной библиотеки. Функции ввода-вывода, потоки, форматированный ввод-вывод. Работа с текстовыми и бинарными файлами. Функции работа с динамической памятью. Обработка строк и блоков памяти.

Динамические переменные

Все переменные, объявленные в программе размещаются в одной непрерывной области памяти, которую называют сегментом данных (64К). Такие переменные не меняют своего размера в ходе выполнения программы и называются статическими. Размера сегмента данных может быть недостаточно для размещения больших массивов информации. Выходом из этой ситуации является использование динамической памяти. Динамическая память – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы.

Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.

Для создания динамических переменных используют операцию new, определенную в СИ++:

указатель = new имя_типа[инициализатор];

где инициализатор – выражение в круглых скобках.

Операция new позволяет выделить и сделать доступным участок динамической памяти, который соответствует заданному типу данных. Если задан инициализатор, то в этот участок будет занесено значение, указанное в инициализаторе.

int*x=new int(5);

Для удаления динамических переменных используется операция delete, определенная в СИ++:

delete указатель;

где указатель содержит адрес участка памяти, ранее выделенный с помощью операции new.

delete x;

В языке Си определены библиотечные функции для работы с динамической памятью, они находятся в библиотеке <stdlib.h>:

1) void*malloc(unsigned s) – возвращает указатель на начало области динамической памяти длиной s байт, при неудачном завершении возвращает NULL;
2) void*calloc(unsigned n, unsigned m) – возвращает указатель на начало области динамической для размещения n элементов длиной m байт каждый, при неудачном завершении возвращает NULL;
3) void*realloc(void *p,unsigned s) –изменяет размер блока ранее выделенной динамической до размера s байт, p – адрес начала изменяемого блока, при неудачном завершении возвращает NULL;
4) void *free(void *p) – освобождает ранее выделенный участок динамической памяти, p-адрес начала участка.

Пример:

```
int *u=(int*)malloc(sizeof(int)); // в функцию передается количество требуемой памяти в байтах, т. к. функция возвращает значение типа void*, то его необходимо преобразовать к типу указателя (int*).
free(u); //освобождение выделенной памяти
```

Модуль 3. Особенности объектно-ориентированного программирования в С++

Лекция 6.

Инкапсуляция. Принцип и назначение и роль инкапсуляции при разработке программного обеспечения. Особенности реализации инкапсуляции в С++, права доступа к членам, друзья классов. Примеры практического использования инкапсуляции в практике программирования.

Наследование. Назначение и область применения наследования при разработке программного обеспечения. Особенности реализации механизмов наследования в С++. Управление правами доступа при наследовании. Множественное и виртуальное наследование. Примеры использования различных аспектов механизма наследования в практике программирования.

Полиморфизм и виртуальные функции. Понятие, назначения и область применения полиморфизма. Особенности реализации полиморфизма в С++. Виртуальные функции, определение, назначение особенности реализации. Таблица виртуальных методов. Чисто виртуальные методы и абстрактные классы. Примеры использования полиморфизма в практике программирования.

1. Существующие парадигмы программирования.
2. Основные принципы ООП.
3. Объекты и классы.
4. Абстракции и иерархия.

1. Языки высокого уровня (Algol 68, Fortran, PL/1 и т.д.) облегчили трудоемкую работу по созданию машинного кода, который стал делать *компилятор*. Программы стали короче и понятнее.

Потом задачи усложнились, и программы снова стали слишком громоздкими. Программы стали разбивать на *процедуры* или *функции*, которые решают свои задачи. Написать, откомпилировать и отладить маленькую функцию можно легко и быстро, а потом собрать все функции воедино.

Такое программирование стало *процедурным программированием* и стало *парадигмой*.

Появились библиотеки процедур и функций. Как сделать программу нагляднее, какую часть кода надо выделить в отдельную процедуру и как лучше связать их между собой, т.е. как выявить структуру программы.

Появились новые языки.

Удачная структура данных может облегчить их обработку. Некоторые удобно представить в виде массива, а другие в виде стэка или дерева. Рост сложности и размеров программ потребовал развития структурирования данных и появления новых типов данных, которые могут определяться программистом.

Идея объединения данных и всех процедур их обработки в единый модуль – *парадигма модульного программирования*.

Сначала такой модуль был более или менее случайным набором данных и подпрограмм. В такие модули собирали подпрограммы, которые, как казалось, скорее всего будут изменяться совместно. Программы составлялись

из отдельных модулей. Эффективность таких программ тем выше, чем меньше модули зависят друг от друга.

Необходимо четко отделить процедуры, которые будут вызываться другими модулями – открытые процедуры. Отделять их будем от вспомогательных, которые обрабатывают данные, заключенные в модуль. Данные, занесенные в модуль также делятся на *открытые* и *закрытые*. Удобно разбить программу *на модули*. Таким образом, чтобы она превратилась в совокупность взаимодействующих объектов.

Так возникло ООП.

Это *современная парадигма* программирования.

Все программы состоят из двух частей (описание и сама программа). Любая программа может быть концептуально организована либо вокруг её кода “кодовое воздействие на данные”, либо вокруг данных “управляемый данными доступ к коду”.

При первом процедурном подходе программу определяет последовательность операторов её кода. Второй подход организует программу вокруг данных, т.е. вокруг объектов и набора хорошо организованных интерфейсов.

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

1).ООП использует в качестве базовых элементов объекты, а не алгоритмы.

2).Каждый объект будет экземпляром какого-либо определенного класса.

3).Классы реализованы (организованы) иерархически.

Основное достоинство ООП – сокращение числа межмодульных связей, изменение объемов информации, которая передается между модулями и возможность повторного использования кодов.

Недостатки – снижение быстродействия из-за более сложной организации программы.

В основу ООП положены 4 главных принципа:

- *абстрагирование;*
- *инкапсуляция;*
- *наследование;*
- *полиморфизм.*

Программа будет состоять из трех файлов

|| Товар (название, код, единица измерения, продажная стоимость единицы).

1-ый файл: *“file.h”* - заголовочный файл.
2-ой файл: *“file.cpp”* - файл реализации.
3-ий файл: *“demofile.cpp”* - демонстрационный файл
здесь main

iostream
class имя класса
{
}; поля данных класса.

“Конструктор” выделяет память, *“деструктор”* – наоборот. Конструктор никогда не возвращает значение.

{ private :
public :

Создание приложений в интегрированной среде разработчика.

Абстрагирование – это метод решения сложных задач. Описывая поведение сложного объекта, мы выделяем только те стороны, которые нас интересуют с точки зрения решаемых задач. Т.е. строим его приближенную модель. Модель не может описать реальный объект полностью. Мы выделяем только те характеристики, которые важны для задачи. Нам надо абстрагироваться от несущественных деталей объекта.

Уровень абстракции.

Надо выбрать правильный уровень абстракции, чтобы не получилась слишком простая модель, когда потеряется что-то важное. Нельзя выбирать слишком высокий уровень абстракции, так как он дает слишком приблизительное упрощенное описание объекта.

Слишком низкий уровень абстракции делает объект слишком сложным, перегруженным деталями.

Абстрагирование – это взгляд на объект ни как он есть на самом деле, а с точки зрения наблюдателя и интересующих его характеристик данного объекта.

Характеристики – это свойства объекта, т.е. то, что, касается его состояния или определяет его поведение, выделяется в единую программную единицу или некий абстрактный класс. Объектно-ориентированное проектирование основано на абстрактном объединении объектов, решении одних и тех же задач, в классы. В виде класса можно представить любую общепринятую абстракцию данных.

СТЭК – гора тарелок, растет вверх и уменьшается сверху.

```
class CStack {private :
                int size;      // максимальный размер
                int top_el;   // индекс верхнего элемента
                int *stack;  // динамич. массив элементов
              public:
                   cstack (int);  // конструктор
                  ~ cstack (    );  // деструктор
              };
class CPerson {private :
                    char name [ 25 ];  // Ф.И.О.          ( элементы
```

```
        short  int age;              // возраст              данных
               char sex;            // пол                    или
             public :                                         поля)
                  CPerson (  );
         void  *Member_of (  );                        (элементы функций
         void  AddSpisok (void*);                            или
         void  RemoveFromSpisok (void*);                  методы)
         };
```

Тип данных
```
CStack       stack [10 ];
CPerson     *person = new CPerson;
// Тип типа CPerson – class, описанный программистом //
void   *firma = person -> Member_of (  );
```

Инкапсуляция – ограничение прав доступа. Объекты в задаче сохраняют конкретные данные, тип которых определяется полями класса. Каждый объект в задаче играет роль, определяет его “поведение”. То, что может делать каждый объект (кроме сохранения значений своих полей) задается элементами функции. Особенностью класса является инкапсуляция одной конструкции, как данных, так и методов функций, которые обрабатывают эти данные, контролируемым образом. Это защита данных и операций от неконтролируемого доступа.

Так элементы *private* оказываются автоматически доступными только для методов самого класса, но сокрытыми для другой части программы.

Элементы *public* определяют интерфейс класса с другими частями программы и другими классами.

Абстрагирование и инкапсуляция дополняют друг друга. Абстрагирование направлено на наблюдение за объектом, а инкапсуляция занимается внутренним устройством объекта – это сокрытие некоторых элементов абстракции (которые не затрагивают существенных характеристик объекта как целого).

Надо выделить две части в описании абстракции. Первая – это интерфейс (взаимодействие). Это основные характеристики состояния, поведения объекта. Он описывает внешнее поведение объекта типа *class*. Вообще описывает абстракцию поведения всех объектов данного класса. В этой части собрано все, что касается взаимодействия этого объекта с любыми другими объектами.

“ То есть то, как сделано снаружи ” Бутч.

Вторая – реализация, представляет собой недоступные извне элементы реализации абстракции (внутренняя организация абстракции и механизмы реализации её поведения). Скрывает все детали.

“ То есть то, как сделано изнутри ” Бутч.

Наследование классов – это основа ООП, позволяет увеличить масштабируемость программ, реализовать повторное использование кода.

Программа, которая использует объекты базового класса может быть расширена без изменения старого кода путем построения производных классов от базового. Использование новых объектов в производных классах. Это достигается соглашением о наследовании: объекты производного класса содержат все элементы базового класса (поля и методы), как если бы эти элементы были описаны в самом производном классе.

Отпадает необходимость многократного переписывания одних и тех же определений базового класса, появляется возможность пользоваться ими, как они есть. В базовом классе надо определить какие права доступа мы предоставляем производным класса

private
public
protected

```
class   CBase  { protected :
                   int a;
                 public :
                   void  Draw ( );
                  };
class  CProizv : public  CBase {
                   int b;
                 public :
                   void  Draw ( );
                  };
```

// (class CBase – базовый класс; class CProizv – производный класс.)

Производный класс может наследовать базовый как *private,* как *public* и как *protected*. Это влияет на возможность будущего расширения иерархии классов и интерпретации целей самого наследования.

Полиморфизм – это положение теории типов, согласно которому имена (переменных) могут обозначать объекты разных, но имеющих общего родителя, классов. Следовательно, любой объект, обозначаемый полимофным именем, может по-своему реагировать на некий общий набор операций. Один интерфейс – много методов.

Может понадобиться программа, которой требуется три типа СТЭКОВ (для хранения целых, вещественных чисел, для хранения символов).

Алгоритм, который реализует все стэки, будет один и тот же, хотя хранимые даны различны.

Полиморфизм позволяет определить общий для всех типов данных набор стэковых функций, использовать одно и то же имя. Дальнейшее – забота компилятора – выбрать специфический метод для использования в каждой конкретной ситуации.

Применение полиморфизма позволяет решить проблему добавления новых функциональностей. При этом существующий программный код не подвергается никаким изменениям. Мы добавим новый код к уже существующему.

Классы.

class CMyClass
(тип) (название)

```
# include <iostream.h>
class            CMyClass
{ public :
    int num;
    int umnoz (  );   // прототип функции, которую умножили на 2.
void set_num (int );  // прототип функции, кот. устанавл. значение num.
void show (  ) { cout<<num; } // встроенная функция.
};
int  CMyClass :: umnoz (  )  {return num*2;}
void CMyClass :: set_num (int n )
     {num = n;}
int main (  )
{ CMyClass  a,b,c;  // экземпляр типа класса CMyClass a
  a .set_num (5 );
  cout << a . umnoz (  ) << " \ n " ;
  a .show (  );        // печать  значения
( или cout << a .num ; )
  return Ø ;
}
```

Элементы класса – это данные, которые инкапсулируют состояние объектов, и функции, которые представляют собой код, реализующий поведение объектов.

Элементы данных аналогичны структурам в языке СИ. Они не могут быть объявлены как *auto*, *extern* или *register*. Они могут быть перечислениями и объектами ранее объявленных классов. Они не могут быть представителями самого класса. Элемент данных может быть указателем или ссылкой на сам класс.

Элементы функции.

Функция – элемент класса, которая объявляется внутри класса. Определение функции может находиться внутри класса (встроенная функция). Компилятор будет генерировать её встроенное расширение на месте вызова.

Если определение функции максимально вне определения класса, то к её имени добавляется *префикс*, состоящий из имени класса и операции расширения области видимости " *::* " .

Модуль 4. Шаблоны типов и функций

Лекция 7.

Понятие шаблонов и область из применения. Понятие, назначения и область применения шаблонов. Синтаксис описания шаблонов, особенности реализации шаблонов в C++. Шаблоны типов и шаблоны классов. Аргументы шаблонов, использование различных видов аргументов. Механизмы реализации шаблонов компилятором. Особенности организации исходного программного кода и использовании шаблонов.

Разработка собственных шаблонов функций и типов. Назначение и особенности использования шаблонов функций. Примеры использования шаблонов функций. Назначение и особенности использования шаблонов типов. Примеры использования шаблонов типов. Применение шаблонов при создании различного рода контейнеров, включая списки, множества, стек и др.

ШАБЛОНЫ ФУНКЦИЙ

Шаблоны являются инструментом ООП, позволяющим использовать одни и те же функции или классы для обработки разных типов данных. Концепция шаблонов может быть реализована как по отношению к функциям, так и по отношению к классам.

Допустим, требуется написать функцию вычисления модуля чисел.

```
int abs(int n)
{
if (n<0) return –n;
else return n;
}
```

Описанная функция берет аргумент типа int и возвращает результат того же типа. Если нужно найти модуль числа типа float, то придется писать ещё одну функцию:

```
float abs(float n)
{
if (n<0) return –n;
else return n;
}
```

Тело функций при этом ничем не отличается. Эти функции могут быть перегружены и иметь одинаковые имена, но все равно для каждой из них нужно писать отдельное определение. Многократное переписывание таких функций-близнецов утомляет и способствует порождению ошибок. А если где-то нужно исправить алгоритм, придется исправлять его в теле каждой функции.

Шаблоны функций в C++ как раз и существуют для того, чтобы можно было написать алгоритм всего один раз и заставить его работать с различными типами данных возвращая результаты разного типа.

Следующий пример показывает, как пишется шаблон функции, вычисляющей модуль числа и как он потом используется.

```
template <class T> // Это шаблон функции
T abs(T n ) //
{ //
if (n<0) return –n; //
else return n; //
} //
int main
{
int a = –10, b;
float x = 3, y;
```

b = abs(a); // теперь функция abs( ) может работать с любым типом данных
y = abs(x);
}

Заданная таким образом функция abs( ) может работать с любыми типами данных, если для них определен оператор < и унарный оператор -. Типы данных определяются функцией при передаче аргумента.

Сутью концепции шаблонов функций является представление использующегося функцией типа не в виде какого-то специфического, а с помощью названия, вместо которого может быть подставлен любой тип. Ключевое слово template сообщает компилятору о том, что определяется шаблон функции.

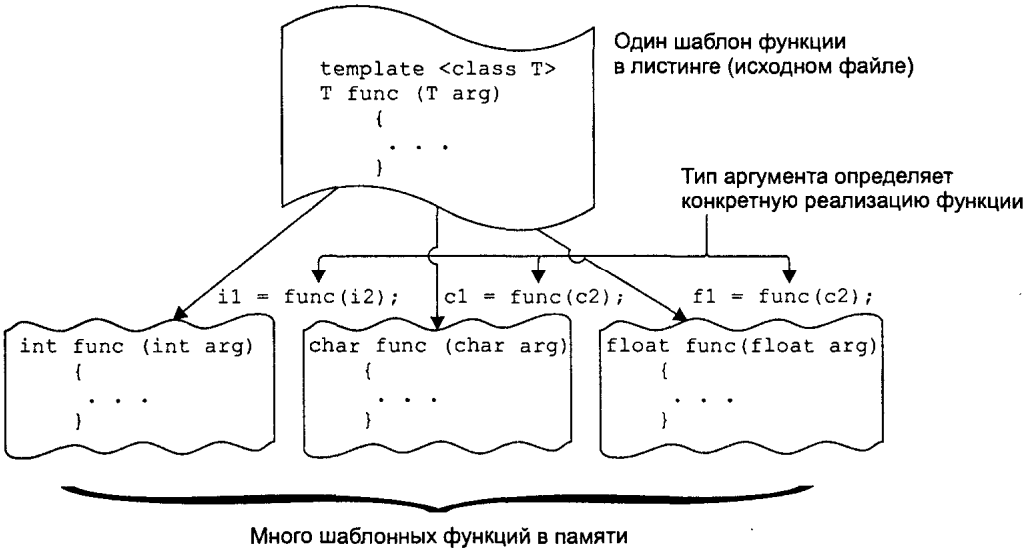


Рисунок 3. Сущность шаблона функции.

Генерация кода при определении шаблона не происходит до тех пор, пока функция не будет реально вызвана в ходе исполнения программы. Когда компилятор увидит вызов такой функции, он сгенерирует код для функции, поставив в неё нужный тип данных. Компилятор принимает решение о том, как именно компилировать функцию, основываясь только на типе данных используемого в шаблоне аргумента. Тип данных, возвращаемый функцией, не играет при этом роли.

В шаблоне функции можно использовать несколько шаблонных аргументов. Шаблонные функции можно перегружать.

Когда компилятор встречает вызов какой-то функции, для его разрешения он следует такому алгоритму

- Сначала ищется обычная функция с соответствующими параметрами;
- Если таковой не найдено, компилятор ищет шаблон, из которого можно было бы генерировать функцию с точным соответствием параметров;
- Если этого сделать невозможно, компилятор вновь рассматривает обычные функции на предмет возможных преобразований типа параметров

ШАБЛОНЫ КЛАССОВ

Шаблонный принцип можно расширить и на классы. В этом случае шаблоны используются, когда класс является хранилищем данных.

Пусть, например, имеется класс типа стек, для хранения чисел типа int.

class Stack
{
private:
int st[max]; // целочисленный массив
int top; // индекс вершины стека
public:

```
Stack( ); // конструктор
void push(int var); // аргумент типа int
int pop( ); // возвращает значение типа int
};
```

Если теперь нужно будет хранить в стеке значения типа float, придется написать новый класс.

```
class Stack
{
private:
float st[max]; // массив
int top; // индекс вершины стека
public:
Stack( ); // конструктор
void push(float var); // аргумент типа float
float pop( ); // возвращает значение типа float
};
```

Подобным же образом пришлось бы создать классы для хранения данных каждого типа. Для преодоления этого ограничения и используются шаблоны классов.

```
template <class Type>
class Stack
{
private:
Type st[max]; // массив любого типа
int top; // индекс вершины стека
public:
Stack( ); // конструктор
void push(Type var); // аргумент любого типа
Type pop( ); // возвращает значение любого типа
};
```

Здесь *Stack* является шаблонным классом. Идея шаблонных классов во многом сходна с идеей шаблонных функций. Шаблоны классов отличаются от шаблонов функций способом реализации. Для создания шаблонной функции она вызывается с аргументами нужного типа. Классы реализуются с помощью определения объекта, использующего шаблонный аргумент.

```
Stack <float> s1;
```

Такое выражение создаст переменную s1. В нашем случае это будет стек, в котором хранятся числа типа float. На рисунке показано, как шаблоны классов и определения конкретных объектов приводят к занесению этих объектов в память.

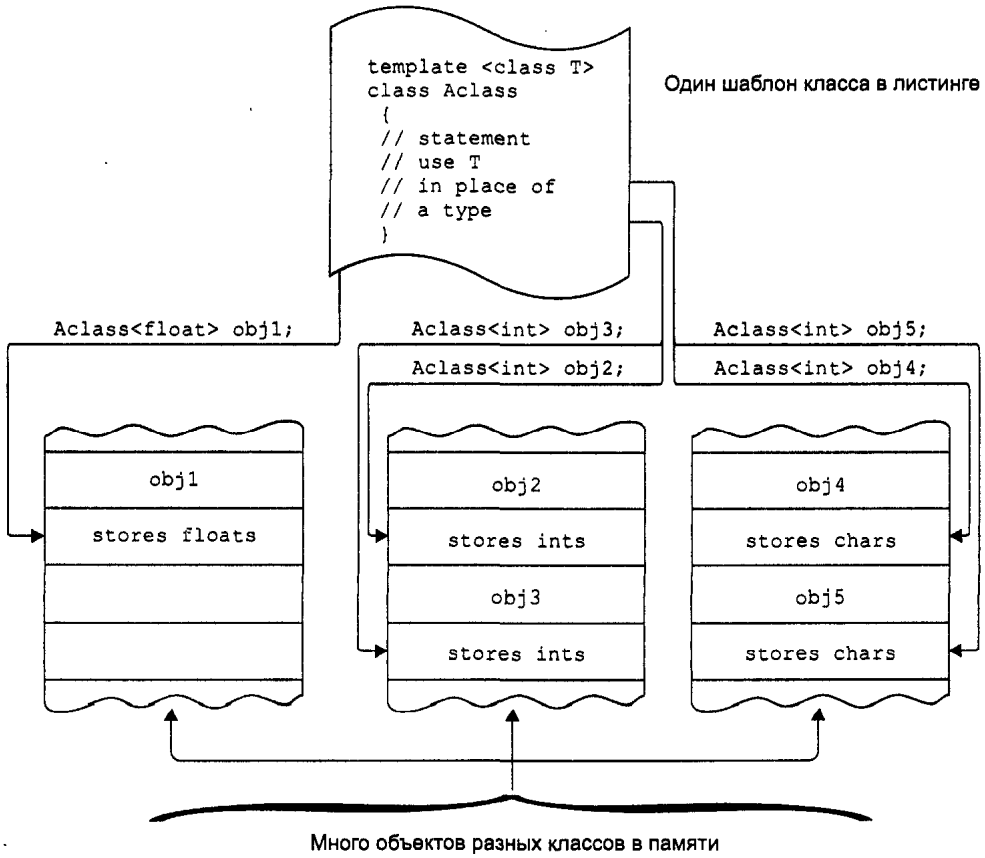


Рисунок 4. Шаблон класса.

Если методы класса определяются вне его спецификации, то они становятся, по сути, шаблонами функций, и определять их надо именно так. Например, конструктор класса стек из предыдущего примера будет описан так:

```
template <class Type>
Stack<Type>::Stack( )
{
top = -1;
}
```

то есть выражение *template <class Type>* должно предварять не только определение класса, но и каждый определённый вне класса метод.

В библиотеке C++ Standard Template Library реализовано множество шаблонов стандартных типов данных (списки, векторы, очереди, реализованные как шаблоны классов) и стандартных методов (накопление, поиск и сортировка и др.). Подробнее узнать о них можно из литературы.

УПРАВЛЕНИЕ ИСКЛЮЧЕНИЯМИ

Исключения позволяют применить объектно-ориентированный подход к обработке возникающих в классах ошибок. Под исключениями понимаются ошибки, возникающие во время работы программы. Они могут быть вызваны различными обстоятельствами, такими как выход за пределы массива, ошибка открытия файла, инициализация объекта некорректным значением и т.д.

В нормальной ситуации вызовы методов классов не приводят ни к каким ошибкам. Но иногда в программе возникает ошибка, которую обнаруживает *сам метод*. Например, это может быть проверка на выход за пределы индексов массива. И тогда метод информирует программу о случившемся – генерирует исключительную ситуацию. В приложении при этом создается отдельная секция кода, в которой задаются операции по обработке ошибок - этот блок называют обработчиком исключительных ситуаций (см рисунок).

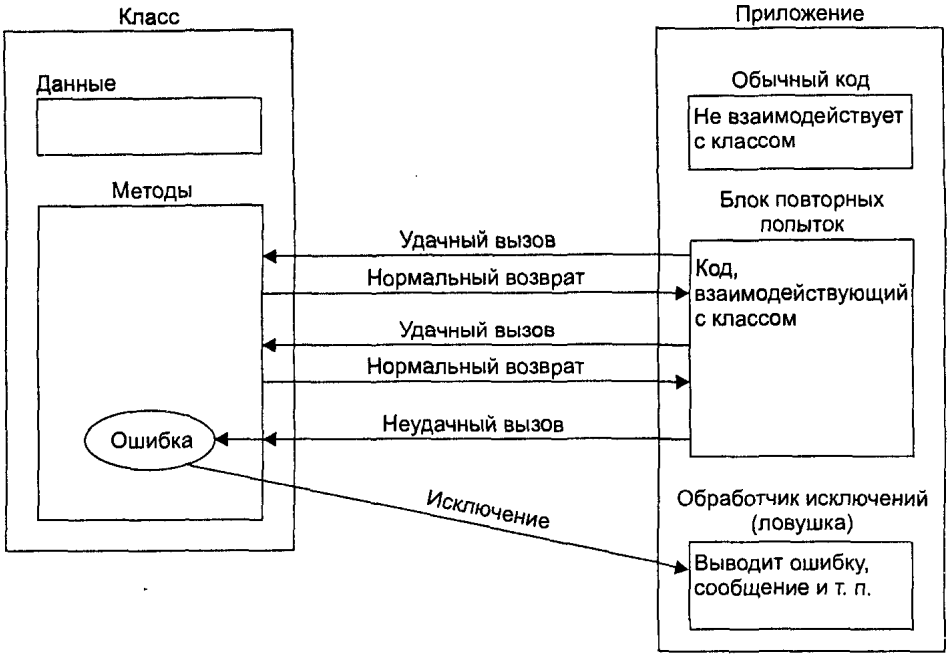


Рисунок 5. Механизм исключений.

Механизм исключений использует три служебных слова: catch, throw и try. Ниже приводится пример программы, демонстрирующий механизм исключений.

```
#include <iostream>
using namespace std;
const int MAX = 3; //в стеке максимум 3 целых числа
//////////////////////////////////////////////////////////
class Stack
{
private:
int st[MAX]; //стек: целочисленный массив
int top; //индекс вершины стека
public:
class Range //класс исключений для Stack
{ //внимание: тело класса пусто
};
//-----------------------------------------------------
Stack() //конструктор
{ top = -1; }
//-----------------------------------------------------
void push(int var)
{
if(top >= MAX-1) //если стек заполнен,
throw Range(); //генерировать исключение
st[++top] = var; //внести число в стек
}
//-----------------------------------------------------
int pop()
{
if(top < 0) //если стек пуст,
throw Range(); //исключение
return st[top--]; //взять число из стека
}
};
//////////////////////////////////////////////////////////
```

```
int main()
{
Stack s1;
try
{
s1.push(11);
s1.push(22);
s1.push(33);
// s1.push(44); //Опаньки! Стек заполнен
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;
cout << "4: " << s1.pop() << endl; //Опаньки! Стек пуст
}
catch(Stack::Range) //обработчик
{
cout << "Исключение: Стек переполнен или пуст"<<endl;
}
cout << "Приехали сюда после захвата исключения (или нормального выхода" << endl;
return 0;
}
```

В приведенном примере класс исключение описывается внутри класса *Stack* следующим образом

```
class Range
{
};
```

Тело класса пусто, в данном случае он создается исключительно ради имени класса. Оно используется для связывания выражения генерации исключения *throw* с улавливающим блоком *catch.*

В классе *Stack* исключение возникает, когда приложение пытается извлечь значение из пустого стека, или положить значение в уже заполненный стек. Чтобы сообщить приложению о том, что оно выполнило недопустимую операцию с объектом, методы этого класса проверяют условия с использованием if и генерируют исключение, если условие выполняется. В примере исключение генерируется в двух местах, с помощью выражения

```
throw Range( );
```

Все выражения, в который могут произойти ошибки, заключены в фигурные скобки, перед которыми стоит слово *try.* Этот блок называется блоком повторных попыток.

Код, в котором содержатся операции по обработке ошибок, заключается в фигурные скобки и начинается со слова catch. В скобках указывается имя класса обрабатываемого исключения. В примере это *catch(Stack::Range).* Если класс ошибки не важен (то есть нужно обработать любую ошибку), то в скобках указываются три точки *catch(…).*

Классы исключений не обязательно объявлять внутри класса, они могут и не принадлежать другим классам. Можно в качестве классов исключений использовать и встроенные типы данных.

Можно спроектировать класс и таким образом, чтобы он генерировал несколько исключений. В приведенном ниже примере описан класс стека, который генерирует разные исключения для ситуаций пустого и заполненного стека.

```
#include <iostream>
using namespace std;
const int MAX = 3; //в стеке может быть до трех целых чисел
//////////////////////////////////////////////////////////
```

```
class Stack
{
private:
int st[MAX]; //стек: массив целых чисел
int top; //индекс верхушки стека
public:
class Full { }; //класс исключения
class Empty { }; //класс исключения
//--------------------------------------------------------
Stack() //конструктор
{ top = -1; }
//--------------------------------------------------------
void push(int var) //занести число в стек
{
if(top >= MAX-1) //если стек полон,
throw Full(); //генерировать исключение Full
st[++top] = var;
}
//--------------------------------------------------------
int pop() //взять число из стека
{
if(top < 0) //если стек пуст,
throw Empty(); //генерировать исключение Empty
return st[top--];
}
};
////////////////////////////////////////////////////////////
int main()
{
Stack s1;
try
{
s1.push(11);
s1.push(22);
s1.push(33);
// s1.push(44); //Опаньки: стек уже полон
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;
cout << "4: " << s1.pop() << endl; //Опаньки: стек пуст
}
catch(Stack::Full)
{
cout << "Ошибка: переполнение стека" << endl;
}
catch(Stack::Empty)
{
cout << "Ошибка: стек пуст" << endl;
}
return 0;
}
```

Существуют и более сложные конструкции использования исключений. Такие как, например, исключения с аргументами. Они предназначены для передачи в программу дополнительных сведений о том, что привело к возникновению исключительной ситуации. Познакомиться с ними можно, обратившись к специальной литературе.

ВОПРОСЫ К ЛЕКЦИИ 7

1. Шаблоны позволяют удобным способом создать семейство
а) переменных;
б) функций;
в) классов;
г) программ.
2. Истинно ли утверждение, что шаблоны автоматически создают разные версии класса в зависимости от введенных пользователем данных?
3. Напишите шаблон функции, всегда возвращающей свой аргумент, умноженный на 2.
4. Шаблонный класс:
а) работает с разными типами данных;
б) генерирует идентичные объекты;
в) генерирует классы с различным числом методов
5. Истинно ли утверждение, что шаблон может иметь несколько аргументов?
6. Реальный код шаблонной функции генерируется при:
а) объявлении функции в исходном коде;
б) вызове функции в исходном коде;
в) запуске функции во время работы программы.
7. В каких случаях целесообразни прибегать к шаблонным классам?
8. При работе с механизмом исключений в C++ используются следующие ключевые слова: ____, _____, ____ .
9. Напишите выражение, генерирующее исключение, используя класс BoundsError.
10. Исключения передаются
а) из блока-ловушки в блок повторных попыток;
б) из выражения, создавшего исключительную ситуацию, в блок повторных попыток;
в) из точки, где возникла ошибка, в блок-ловушку.
11. Истинно ли утверждение о том, что программа может продолжить свое выполнение после возникновения исключительной ситуации?

ЗАДАНИЕ К ЛЕКЦИИ 7

1. Создать шаблон функции, возвращающей среднее арифметическое всех элементов массива. Аргументами функции должны быть имя и размер массива (типа int).
   Создать шаблон функции, возвращающей значение максимального элемента массива. Аргументами функции должны быть имя и размер массива (типа int).
   Создать шаблон функции, обменивающей местами значения двух передаваемых ей по ссылке аргументов.
   Создать шаблон функции, осуществляющей сортировку данных массива. Аргументами функции должны быть имя и размер массива (типа int).
   Продемонстрировать работу шаблонов на данных различных типов, в том числе, на данных вновь созданного класса – вектор на плоскости, определив для него операции сравнения.
2. Написать шаблон класса для работы с очередью FIFO. Определить функции включения и исключения элементов. Добавить механизм обработки исключений при превышении размера очереди и при попытке удалить данные из пустой очереди. Это можно сделать, добавив элемент данных – счетчик текущего числа элементов. Исключения генерируются, если счетчик превысил размер массива или если он стал меньше 0.

Модуль 5. Библиотека STL

Лекция 8.

Назначение и область применения библиотеки. Основные особенности и характерные черты библиотеки STL. Обзор функциональных возможностей и основных областей применения библиотеки. Рассмотрение групп шаблонов библиотеки с краткой характеристикой назначения и возможностей каждой из групп.

Шаблоны потоков в STL. Характеристика возможностей и назначения потоков ввода-вывода библиотеки STL. Шаблоны используемые для консольного ввода-вывода. Шаблоны используемые для организации работы с файлами. Форматированный ввод-вывод. Буферизация при работе с потоками. Применение потоков библиотеки для работы с объектами пользовательских классов.

Шаблоны контейнеров в STL. Назначение и область применения контейнеров библиотеки STL. Принципы построения шаблонов контейнеров. Особенности применения итераторов и распределения памяти. Основные контейнеры библиотеки STL (включая динамический массив, строка, множество, хеш-таблица, очередь, стек и др.)

Стандартная библиотека шаблонов STL

Практическая деятельность программистов в течение нескольких десятков лет привела широкому распространению ряда способов организации структур данных, например, массив, список, очередь и т.д. Эти структуры данных стали стандартными. Для использования стандартных структур данных при решении различных задач была разработана стандартная библиотека шаблонов, предназначенных для формирования контейнерных классов.

Контейнер – это объект, содержащий набор других объектов, организованный определенным образом.

Назначение контейнеров – управление наборами (коллекциями) объектов определенного типа.

Примеры контейнеров: массив, список.

Работа с контейнерами поддерживается с помощью контейнерных классов.

Для каждого типа контейнера в соответствующем контейнерном классе определены методы для работы с его элементами.

Свойства контейнеров:

- для каждого типа контейнера в контейнерном классе определены методы для работы с его элементами независимо от типа элементов

Достоинства:

- повышение надежности, переносимости и универсальности
- уменьшение сроков и стоимости разработки программ

Недостатки:

- снижение быстродействия
- трудность изучения

Классификация контейнеров:

- последовательные
- ассоциативные

Последовательные контейнеры

- векторы (vector)
- двусторонние очереди (deque);
- списки (list)
- стеки (stack)
- очереди (queue)

- очереди с приоритетами (priorityqueue)

В последовательном контейнере элемент занимает определенную позицию, которая зависит только от времени и места вставки.

Ассоциативные контейнеры

- словари (map)
- словари с дубликатами (multimap);
- множества (set);
- множества с дубликатами (multiset)
- битовые множества (bitset)

Позиция элемента зависит от его значения по определенному критерию сортировки. Это ускоряет поиск нужного элемента

Требования к контейнерам

1. При вставке элемента создается его копия внутри контейнера
2. Расположение элементов в контейнера в определенном порядке
3. Необходимость отслеживания операций

Общие операции и методы

begin()
end()

rbegin()
rend()
size()
max_size()

empty( )
сравнение <>
присваивание =

swap()
clear( )

Операция	Метод	Вид последовательного контейнера		
		vector	deque	list
Вставка в начало	push_front()	-	+	+
Удаление из начала	pop_front()	-	+	+
Вставка в конец	push_back()	+	+	+
Удаление из конца	pop_back()	+	+	+
Вставка в произвольное место	insert()	±	±	+
Удаление из произвольного места	erase ()	±	±	+
Произвольный доступ к элементу	[] или at()	+	+	-

Выводы:

1. Вектор эффективно реализует произвольный доступ, добавление и удаление элемента из конца
2. Двусторонняя очередь эффективно реализует произвольный доступ, а также добавление элементов в оба конца.и удаление элементов из обоих концов.
3. Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к элементам.

Массивы и двусторонние очереди – на самостоятельную проработку

Списки

Свойства и возможности списков

- Список не предоставляет произвольного доступа к своим элементам
- Вставка и удаление элементов в любое место списка выполняется за одно и то же время
- Нет операций, связанных с емкостью и перераспределением памяти
- Много специальных методов для перемещения элементов.

Для передвижения по структуре используется итератор – указатель на текущий элемент.

Модуль 6. Использование библиотеки STL при разработке проблемно-ориентированных программных комплексов

Лекция 9.

Методы и примеры использования шаблонов потоков. Методы использования потоков при создании проблемно-ориентированных программных комплексов. Основные преимущества использования потоков. Практические приемы и рекомендации использования потоков. Промеры практического использования шаблонов потоков при разработке прикладного программного обеспечения.

Методы и примеры использования шаблонов контейнеров. Методы использования шаблонов контейнеров при создании проблемно-ориентированных программных комплексов. Основные преимущества использования шаблонов контейнеров. Практические приемы и рекомендации использования контейнеров. Промеры практического использования шаблонов контейнеров при разработке прикладного программного обеспечения.

Пример работы со списком.

```
include "stdafx.h"
#include<list>//включение заголовочного файла шаблонного класса списка
#include<algorithm>
usingnamespacestd;//стандартное пространство имен
int _tmain()
{
// TODO: Please replace the sample code below with your own.
//Console::WriteLine(S"Hello World");
        list<int> L1, L2, L3;//определение 3-х пустых списков
        //Заполнение списков значениями
        for(unsigned i=0; i<5; i++)
                L1.push_front(i);
        for(unsignedi=0; i<3; i++)
                L2.push_back(i+10);
        L3=L2;
        list<int>::iterator it;
printf("L1: ");
        for(it=L1.begin(); it !=L1.end(); it++)
                {
                        printf("%d ",*it);
                }
printf("L1.size %d", L1.size());
printf("L2: ");
for(it=L2.begin(); it !=L2.end(); it++)
                {
                        printf("%d ",*it);
                }
printf("L2 size %d", L2.size());
printf("L3: ");
        for(it=L3.begin(); it !=L3.end(); it++)
                {
                        printf("%d ",*it);
                }
printf("L3 size %d ", L3.size());
        it = L1.begin(); it++; it++;
```

```
        L1.splice(it,L2);
        printf("L1: ");
        for(it=L1.begin(); it !=L1.end(); it++)
                {
                        printf("%d ",*it);
                }
printf("L1.size %d", L1.size());
        L1.sort();
        printf("L1: ");
        for(it=L1.begin(); it !=L1.end(); it++)
                {
                        printf("%d ",*it);
                }
        L1.merge(L3);
        printf("L1: ");
        for(it=L1.begin(); it !=L1.end(); it++)
                {
                        printf("%d ",*it);
                }
printf("Смена порядка следования элементов списка L1 на обратный");
        L1.reverse();

        for(it=L1.begin(); it !=L1.end(); it++)
                {
                        printf("%d ",*it);
                }
printf("Удаление дубликатов");

        L1.unique();

        for(it=L1.begin(); it !=L1.end(); it++)
                {
                        printf("%d ",*it);
                }
}
```

Ассоциативные контейнеры обеспечивают быстрый доступ к данным за счет того, что они построены на основе сбалансированных деревьев поиска.

Типы ассоциативных контейнеров:
- словарь (map)$;
- словарь с дубликатами (multimap);
- множество (set);
- множество с дубликатами (multiset);
- битовые множества (bitset);

Словарь – набор пар «ключ/значение». Каждый ключ в одном экземпляре.
Словарь с дубликатами – словарь с возможностью дублирования ключей.

Множество – набор элементов, где элементы сортируются в соответствии с их значениями. Дубликаты не разрешаются.

Множество с дубликатами – множество, в котором могут быть элементы с одинаковыми значениями.

Битовые множества – массивы битов фиксированного размера.
Достоинства битовых множеств:
- произвольный фиксированный размер битового поля
- поддержка присваивания значениям отдельным битам.
- запись битовых полей в виде последовательностей нулей и единиц.

Пример работы со словарями (телефонная книга)

```
/ generated using an Application Wizard.
#include "stdafx.h"
#include<fstream>
#include<iostream>
#include<string>
#include<map>
usingnamespace std;
typedef map <string, long, less <string>>map_dic;
int _tmain()
{
// TODO: Please replace the sample code below with your own.
        map_dic dic1;
        ifstream in("phonebook.txt");
stringfam;
        long num;
        for (unsigned k=0; k<3; k++)
        {
                in>>num;
                in.get();
                getline(in, fam);
                dic1[fam]=num;
                cout<<num<<" "<<fam<<endl;
        }
        dic1["Николаев Николай Николаевич"]=1234567;
        dic1["Сергеев Сергей Сергеевич"]=4444444;
        map_dic::iteratorit;
        for(it = dic1.begin(); it != dic1.end(); it++)
                cout<< it->first << " " << it->second <<endl; getline (in, fam);
        if(dic1.find(fam) !=dic1.end())
                cout<<"Found!!!"<<endl;
        fam="Petrov;
        dic1.insert(make_pair(fam, 6666666));
        dic1.erase(fam);

        in.close();
        return 0;
}
```