Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра инженерной кибернетики

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Конспект лекций
для студентов специальности 5В070200 - Автоматизация и управление

Алматы 2015

СОСТАВИТЕЛИ: Н.В. Сябина, Л.Н.Рудакова. Технологии программирования. Конспект лекций для студентов специальности 5В070200 - Автоматизация и управление. – Алматы: АУЭС, 2015. – 72 с.

Настоящий конспект лекций составлен в помощь студентам второго курса при изучении теоретического материала по технологиям программирования и включает пятнадцать тем. В конце каждой темы приведены ссылки на дополнительную литературу для более глубокого освоения предмета. В приложениях содержится необходимый справочный и иллюстративный материал.

Конспект лекций предназначен для студентов специальности 5В070200 - Автоматизация и управление.

Ил. 14, табл. 9, библиогр. – 11 назв.

Рецензент: доцент Койлыбаева Р.К.

Печатается по плану издания некоммерческого акционерного общества «Алматинский университет энергетики и связи» на 2013 г.

Содержание

Лекция №1. Классификация программного обеспечения. Технологии программирования. Основные понятия и подходы

Цель – получить представление о современной классификации программного обеспечения, понятии «технологии программирования», а также об основных этапах их развития.

Программное обеспечение (ПО) – это совокупность программ, позволяющих осуществить автоматизированную обработку информации на компьютере, предназначенных для многократного использования и применения разными пользователями, а также программных документов, необходимых для их эксплуатации.

Условно программное обеспечение классифицируется как системное (общее) и прикладное (специальное).

Программы, работающие на системном уровне, обеспечивают взаимодействие программ базового уровня с прочими программами системы и непосредственно с аппаратным обеспечением.

Программное обеспечение прикладного уровня представляет собой комплекс прикладных программ, с помощью которых на данном рабочем месте выполняются конкретные задания.

Между тем, подразделение ПО на системное и прикладное является до некоторой степени устаревшим. Современное разделение предусматривает минимум три градации ПО: системное, промежуточное и прикладное.

Современная тенденция развития ПО состоит в снижении объема как системного, так и прикладного программирования. Основная часть работы программистов выполняется в промежуточном ПО. Снижение объема системного программирования обусловлено современными концепциями ОС, объектно-ориентированной архитектурой и архитектурой микроядра, в соответствии с которыми большая часть функций системы выносится в утилиты, которые можно отнести и к промежуточному ПО. Снижение объема прикладного программирования обусловлено тем, что современные продукты промежуточного ПО предлагают все больший набор инструментальных средств и шаблонов для решения задач своего класса. Значительная часть системного и практически все прикладное ПО пишется на языках высокого уровня, что обеспечивает сокращение расходов на их разработку/модификацию и переносимость.

Промежуточное ПО (middleware) определяется как совокупность программ, осуществляющих управление вторичными ресурсами (конструируемыми самим ПО), которые ориентированы на решение определенного класса задач. К такому ПО относятся менеджеры транзакций, серверы БД, серверы коммуникаций и другие программные серверы. С точки зрения инструментальных средств промежуточное ПО ближе к прикладному, так как не работает напрямую с первичными ресурсами, а использует для этого сервисы, предоставляемые системным ПО. С точки же зрения

алгоритмов и технологий разработки оно ближе к системному, так как всегда является сложным программным изделием многократного и многоцелевого использования, в котором применяются алгоритмы, сходные с применяемыми в системном ПО.

Технология программирования (ТП) - это совокупность методов и средств, используемых при разработке ПО. ТП представляет собой набор технологических инструкций:

1) указание последовательности выполнения технологических операций;

2) перечисление условий, при которых выполняются операции;

3) описания самих операций с исходными данными, результатами, инструкциями, нормативами, стандартами, критериями и методами оценки.

Технология определяет способ описания проектируемой системы (модели), используемой на конкретном этапе разработки. Различают технологии, используемые на конкретных этапах разработки (определенный метод), и технологии, охватывающие несколько этапов разработки (базовый метод или методология).

Среди основных этапов развития технологий программирования выделяют «стихийное» программирование, структурный, объектный и компонентный подходы к программированию [1].

Этап «стихийного» программирования. Характерной особенностью периода от момента появления первых ЭВМ до середины 60-х годов XX века является то, что сформулированные технологии программирования практически отсутствовали, а само программирование было искусством.

Первые программы имели простейшую структуру: программа на машинном языке и обрабатываемые данные. Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании. Использовалась *интуитивная* технология программирования. В результате появления ассемблеров вместо двоичных кодов стали использовать *символические имена данных* и *мнемоники* кодов операций, а программы стали более «читаемыми». Именно в этот период зародилась ставшая впоследствии фундаментальной для ТП *концепция модульного программирования* [1, 9], ориентированная на преодоления трудностей программирования в машинном коде. Создание языков программирования высокого уровня (FORTRAN, ALGOL) существенно упростило программирование вычислений, снизив уровень детализации операций, что позволило увеличить сложность программ.

В результате появления средств, позволяющих оперировать *подпрограммами*, были созданы огромные библиотеки расчетных и служебных подпрограмм. Типичная программа состояла из основной программы, *области глобальных данных* и набора подпрограмм (рисунок А.1а). Однако при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой, поэтому было предложено размещать в них *локальные* данные (рисунок А.1б).

В начале 60-х годов XX века разразился «кризис программирования»: разработчики сложного программного обеспечения срывали все сроки завершения проектов: проект устаревал раньше, чем был готов к внедрению, его стоимость увеличивалась, в результате многие проекты так никогда и не были завершены. Использование разработки *«снизу-вверх»* при отсутствии четких моделей описания подпрограмм и методов проектирования превращало создание программ в непростую задачу. Интерфейсы программ получались сложными, а при сборке программного продукта выявлялось большое количество ошибок согласования, исправление которых требовало серьезного изменения уже разработанных частей программ. При этом в программу часто вносились новые ошибки, в результате чего процесс тестирования и отладки программ занимал более 80% времени разработки, если вообще когда-нибудь заканчивался. Анализ причин возникновения ошибок позволил сформулировать новый подход к программированию - *структурный*.

Этап структурного программирования. Структурный подход к программированию, который развивался на втором этапе развития технологий в 60-70 годы XX века. В его основе лежит *декомпозиция* сложных систем с целью последующей реализации в виде отдельных небольших (до 40-50 операторов) подпрограмм, позже названная *процедурной* декомпозицией. Структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры, а проектирование осуществлялось *«сверху-вниз»* и подразумевало реализацию общей идеи. Были введены ограничения на конструкции алгоритмов, рекомендованы формальные модели их описания, а также специальный метод проектирования алгоритмов - *метод пошаговой детализации*. Принципы структурного программирования были заложены в основу *процедурных* языков программирования, которые включали основные *«структурные» операторы передачи управления*, поддерживали вложение подпрограмм, локализацию и ограничение *области «видимости»* данных (PL/1, ALGOL-68, Pascal, C). Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития *структурирования данных*, в языках появляется возможность определения пользовательских типов данных [1, 5, 9].

Стремление разграничить доступ к глобальным данным программы дало толчок к появлению и развитию *технологии модульного программирования* (рисунок А.2), что предполагало выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые *модули* (библиотеки). Связи между модулями осуществлялись через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) был запрещен. Эту технологию поддерживают современные версии Pascal, C, C++, Ада и Modula. Структурный подход в сочетании с модульным программированием позволяет получать надежные программы, размером не более 100 000 операторов. Существенным недостатком является тот факт, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за

раздельной компиляции модулей), а при увеличении размера программы возрастает сложность межмодульных интерфейсов, и предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Поэтому для разработки программного обеспечения большого объема было предложено использовать *объектный подход*.

Этап объектного программирования. На третьем этапе (80-е - 90-е годы XX века) был сформирован *объектный подход* к программированию. Широкое внедрение персональных компьютеров во все сферы человеческой деятельности привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества программного обеспечения [1, 9]. Развиваются методы и языки спецификации. Развивается концепция компьютерных сетей. Возникает необходимость в формировании совершенно нового подхода к разработке сложного программного обеспечения с интуитивно понятным и удобным пользовательским интерфейсом.

Технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности взаимодействующих путем передачи *сообщений* программных *объектов,* каждый из которых является *экземпляром* определенного *класса,* а классы образуют иерархию с *наследованием* свойств, была названа *объектно-ориентированным программированием* [1, 9]. Объектная структура программы (рисунок А.3) впервые была использована в языке имитационного моделирования сложных систем Simula, а затем использована в новых версиях универсальных языков программирования таких, как Pascal, C++, Modula, Java.

Достоинством объектно-ориентированного программирования является «*естественная*» декомпозиция программного обеспечения, существенно облегчающая разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных объектов программы. Кроме того, объектный подход предлагает новые способы организации программ, основанные на механизмах *наследования*, *полиморфизма*, *композиции*, *наполнения*, позволяющих конструировать из простых объектов сложные. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания *библиотек классов*. На основе объектного подхода были созданы среды, поддерживающие *визуальное программирование* (Delphi, C++ Builder, Visual C++), при использовании которого некоторая часть будущего продукта проектируется с применением визуальных средств добавления и настройки специальных библиотечных компонентов. В результате появляется заготовка будущей программы, в которую уже внесены коды.

Использование объектного подхода имеет много преимуществ. Однако его конкретная реализация в объектно-ориентированных языках программирования таких, как Pascal и C++ имеет существенные *недостатки*: отсутствуют стандарты компоновки двоичных результатов компиляции объектов в единое целое, даже в пределах одного языка программирования;

изменение в реализации одного программного объекта связано с перекомпиляцией модуля и перекомпоновкой всего программного обеспечения, использующего данный объект. Таким образом, сохраняется объективная зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Связи модулей нельзя разорвать, но можно стандартизировать их взаимодействие, на чем и основан *компонентный подход к программированию*.

В 90-е годы XX века персональные компьютеры стали подключаться к сети как терминалы. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Начался решающий этап полной информатизации и компьютеризации общества. Четвертый этап развития технологий программирования - *компонентный подход и CASE-технологии* - не завершился до настоящего времени.

Компонентный подход предполагает построение ПО из отдельных компонентов, которые взаимодействуют между собой через *стандартизованные двоичные интерфейсы.* Объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию [1, 9]. Компонентный подход лежит в основе технологий, разработанных на базе *COM* (*Component Object Model* - компонентная модель объектов), и технологии создания распределенных приложений *CORBA* (*Common Object Request Broker Architecture* - общая архитектура с посредником обработки запросов объектов), которые используют сходные принципы и различаются лишь особенностями реализации.

Кроме того, отличительной особенностью современного этапа развития технологии программирования являются создание и внедрение *автоматизированных технологий разработки и сопровождения программного обеспечения*, которые были названы *CASE-технологиями* (*Computer-Aided Software/System Engineering* - разработка программного обеспечения/программных систем с использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудно осуществимой: память человека уже не в состоянии фиксировать все детали, которые необходимо учитывать при разработке программного обеспечения. CASE-технологии поддерживают как структурный, так и объектный (компонентный) подходы к программированию [1, 9].

Технология COM фирмы Microsoft является развитием технологии *OLE I* (*Object Linking and Embedding* - связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Она определяет общую концепцию взаимодействия программ любых типов (библиотек, приложений, операционной системы), т. е. позволяет одной части программного обеспечения использовать функции (*службы*), предоставляемые другой (рисунок А.4).

Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется *DCOM* (*Distributed COM* - распределенная COM). На базе технологии COM и ее распределенной версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки ПО. На возможностях COM базируется технология *COM+*, которая обеспечивает поддержку распределенных приложений на компонентной основе и предназначена для поддержки систем обработки транзакций [1, 9].

Кроме того, к технологиям, реализующим компонентный подход, заложенный в COM, относятся:

а) *OLE-automation* - технология создания программируемых приложений, обеспечивающая программируемый доступ к их внутренним службам (например, MS Excel поддерживает ее, предоставляя другим приложениям свои службы);

б) *ActiveX* - технология, построенная на базе OLE-automation и предназначенная для создания как сосредоточенного на одном компьютере программного обеспечения, так и распределенного в сети. Предполагает использование визуального программирования для создания компонентов - *элементов управления ActiveX*, которые устанавливаются на компьютер дистанционно с удаленного сервера и применяются в клиентских частях приложений Интернет;

в) *MTS* (*Microsoft Transaction Server* - сервер управления транзакциями) - технология, обеспечивающая безопасность и стабильную работу распределенных приложений при больших объемах передаваемых данных;

г) *MIDAS* (*Multitier Distributed Application Server* - сервер многозвенных распределенных приложений) - технология, организующая доступ к данным разных компьютеров с учетом балансировки нагрузки сети.

Технология CORBA, разработанная группой компаний OMG (*Object Management Group* - группа внедрения объектной технологии программирования), реализует подход аналогичный COM на базе объектов и интерфейсов CORBA.

Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ, и потому технологию можно использовать для создания распределенного программного обеспечения в разнородной вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью посредника (*VisiBroker***)** и специализированного программного обеспечения.

К сожалению, в силу ряда причин разработчики были вынуждены отказаться от этой технологии [1, 9]. Сегодня CORBA, главным образом, используется для связывания компонентов, выполняемых внутри корпоративных сетей, в которых коммуникации защищаются брандмауэрами от внешнего мира. Технология CORBA используется также при разработке систем реального времени и встроенных систем, секторе, в котором CORBA действительно развивается. Однако в целом CORBA находится в упадке, и теперь ее нельзя назвать никак иначе, кроме как *нишевой* технологией.

Лекция №2. Особенности разработки сложных программных систем

Цель – получить представление о принципах разработки сложных программных систем и их жизненном цикле; изучить основные этапы разработки программного обеспечения и их особенности.

Большинство современных программных систем являются достаточно сложными. Эта сложность обуславливается многими причинами, главной из которых является *логическая сложность* решаемых ими задач.

Раньше компьютеры применяли в очень узких областях науки и техники, в первую очередь там, где задачи были хорошо детерминированы и требовали значительных вычислений. Сейчас, когда созданы мощные компьютерные сети, появилась возможность переложить на них решение сложных ресурсоемких задач, о компьютеризации которых раньше не задумывались. В процесс компьютеризации вовлекаются новые предметные области, а для освоенных областей усложняются уже сложившиеся постановки задач. Сложность разработки программных систем увеличивается за счет сложности формального определения требований к этим системам, отсутствия удовлетворительных средств описания поведения дискретных систем с большим числом состояний при недетерминированной последовательности входных воздействий, коллективной разработки, необходимости увеличения степени повторяемости кодов. Однако все эти факторы напрямую связаны со сложностью объекта разработки - программной системы [1, 9].

Подавляющее большинство сложных систем имеет *иерархическую* внутреннюю структуру. Связи элементов сложных систем различны как по типу, так и по силе, что и позволяет, рассматривать эти системы как некоторую *совокупность взаимозависимых подсистем*. Внутренние связи элементов таких подсистем сильнее, чем связи между подсистемами. Так, компьютер состоит из процессора, памяти и внешних устройств, а Солнечная система включает Солнце и планеты, вращающиеся вокруг него. Используя то же различие связей, каждую подсистему можно аналогично разделить на подсистемы до «элементарного» уровня. На этом уровне система, состоит из немногих типов подсистем, по-разному скомбинированных и организованных. Иерархии такого типа получили название *«целое-часть»*.

В природе существует еще один вид иерархии - иерархия *«простое-сложное»* или иерархия *развития (усложнения) систем в процессе эволюции*. В этой иерархии любая функционирующая система является результатом развития более простой системы. Именно этот вид иерархии реализуется *механизмом наследования* объектно-ориентированного программирования.

Будучи отражением природных и технических систем, программные системы являются иерархическими и обладают описанными выше свойствами. На этих свойствах иерархических систем строится *блочно-иерархический подход* к их исследованию или созданию, предполагающий сначала создание частей объекта (блоков и модулей), а затем сборку из них самого объекта.

Процесс разбиения сложного объекта на сравнительно независимые части получил название *декомпозиции.* При декомпозиции учитывают, что связи между отдельными частями должны быть слабее, чем связи элементов внутри частей. Чтобы из полученных частей можно было собрать разрабатываемый объект, в процессе декомпозиции необходимо определить все виды связей частей между собой.

При создании сложных объектов процесс декомпозиции выполняется многократно: каждый блок, в свою очередь, декомпозируют на части, пока не получают блоки, которые сравнительно легко разработать. Этот метод разработки получил название *пошаговой детализации.* В процессе декомпозиции стараются выделить аналогичные блоки, которые можно было бы разрабатывать на общей основе. Таким образом, обеспечивают увеличение степени повторяемости кодов и снижение стоимости разработки.

Результат декомпозиции обычно представляют в виде *схемы иерархии,* на нижнем уровне которой располагают сравнительно простые блоки, а на верхнем - объект, подлежащий разработке [1, 6]. На каждом иерархическом уровне описание блоков выполняют с определенной степенью детализации, *абстрагируясь* от несущественных деталей. Как правило, для объекта в целом, удается сформулировать лишь общие требования, а блоки нижнего уровня должны быть специфицированы таким образом, чтобы из них действительно можно было собрать работающий объект. Другими словами, чем больше блок, тем более абстрактным должно быть его описание (рисунок А.5).

При соблюдении этого принципа разработчик сохраняет возможность осмысления проекта и принимает наиболее правильные решения на каждом этапе, что называют *локальной оптимизацией* (в отличие от глобальной оптимизации характеристик объектов, которая для действительно сложных объектов не всегда возможна).

Итак, в основе блочно-иерархического подхода лежат иерархическое упорядочение и декомпозиция. Важную роль играют следующие принципы:

а) *непротиворечивость* - контроль согласованности элементов;

б) *полнота* - контроль на присутствие лишних элементов;

в) *формализация* - строгость методического подхода;

г) *повторяемость* - необходимость выделения одинаковых блоков для удешевления и ускорения разработки;

д) *локальная оптимизация* - оптимизация в пределах уровня иерархии.

Совокупность языков моделей, постановок задач, методов описаний некоторого иерархического уровня принято называть *уровнем* проектирования. Различные взгляды на объект проектирования принято называть *аспектами* проектирования.

Достоинства блочно-иерархического подхода:

- упрощение проверки работоспособности отдельных блоков и системы в целом;

- возможность создания сложных систем;

- обеспечение возможности модернизации систем.

Использование блочно-иерархического подхода применительно к программным системам стало возможным только после конкретизации общих положений подхода и внесения изменений в процесс проектирования. При этом структурный подход учитывает только свойства иерархии: «целое - часть», а объектный дополнительно использует свойства иерархии: «простое - сложное».

Жизненным циклом программного обеспечения называется период от момента появления идеи создания некоторого программного обеспечения до момента завершения его поддержки фирмой-разработчиком или фирмой, выполняющей сопровождение [1, 9].

Состав процессов жизненного цикла регламентируется международным стандартом *ISO/IEC 12207: 1995 «Information Technologe - Software Life Cycle Processes»* («Информационные технологии - Процессы жизненного цикла программного обеспечения»). В стандарте описывается структура жизненного цикла программного обеспечения, определяются его процессы, не конкретизируя в деталях их реализацию.

Процесс жизненного цикла – это совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. На рисунке А.6 представлены процессы жизненного цикла по указанному стандарту. Согласно этой структуре, процессы разработки и сопровождения ПО относятся к основным процессам [1, 9]. Условно выделяют следующие *основные этапы разработки программного обеспечения*:

а) *постановка задачи* - формулируется назначение программного обеспечения, а также определяются основные требования к нему (*функциональные* и *эксплуатационные*). Результатами являются *техническое задание (ТЗ)*, фиксирующее принципиальные требования, и принятие основных проектных решений;

б) *анализ требований и разработка спецификаций* - выполняется анализ требований ТЗ, формулируется содержательная постановка задачи, строится модель предметной области, определяются подзадачи и выбираются или разрабатываются методы их решения, формируются тесты для поиска ошибок в проектируемом ПО с указанием ожидаемых результатов, т.е. разрабатывается общая *логическая модель* проектируемого ПО;

в) *проектирование* - определяются подробные *спецификации* разрабатываемого программного продукта, выполняется проектирование общей структуры, декомпозиция компонентов и проектирование компонентов. Результат - *детальная модель* разрабатываемого ПО со спецификациями компонентов всех уровней;

г) *реализация* - процесс поэтапного написания кодов программы на выбранном языке программирования (*кодирование*), их тестирование и отладка.

Сопровождение - это процесс создания и внедрения новых версий программного обеспечения. В соответствии со стандартом *ISO/IEC 12207* этап сопровождения был выделен в отдельный процесс жизненного цикла.

Причинами выпуска новых версий могут служить: необходимость исправления ошибок, выявленных в процессе эксплуатации предыдущих версий; необходимость совершенствования предыдущих версий; изменение среды функционирования (появление новых технических средств и/или программных продуктов, с которыми взаимодействует сопровождаемое программное обеспечение); в программный продукт вносят необходимые изменения, которые могут потребовать пересмотра уже принятых проектных решений.

Изменение жизненного цикла программного обеспечения стало возможным в результате использования при разработке программного обеспечения *CASE-технологий*, которые представляют собой совокупность методологий анализа, проектирования, разработки и сопровождения сложных программных систем, основанных как на структурном, так и на объектном подходах. В основу любой CASE-технологии положены *методология*, *метод*, *нотация* и *средства* [1, 9].

Среди средств различают:

1) *CASE-средства анализа требований, проектирования спецификаций и структуры, редактирования интерфейсов*. Первое поколение *CASE-I*, в основном включают средства для поддержки графических моделей, проектирования спецификаций, экранных редакторов и словарей данных;

2) *CASE-средства генерации исходных текстов и реализации интегрированного окружения поддержки полного жизненного цикла разработки программного обеспечения*. Второе поколение *CASE-II* существенно отличается большими возможностями, обеспечивая контроль, анализ и связывание системной информации и информации по управлению процессом проектирования, построение прототипов и моделей системы, тестирование, верификацию и анализ сгенерированных программ.

Современные CASE-средства позволяют автоматизировать трудоемкие операции; повысить производительность труда программистов; улучшить качество создаваемого программного обеспечения; уменьшить время создания прототипа системы; автоматизировать формирование проектной документации для всех этапов жизненного цикла в соответствии с современными стандартами; частично генерировать коды программ для различных платформ разработки; применять технологии повторного использования компонентов системы и т. д.

Однако современные CASE-средства дороги, а их использование требует более высокой квалификации разработчиков. Следовательно, имеет смысл использовать их в сложных проектах, причем, чем сложнее разрабатываемое программное обеспечение, тем больше выигрыш от использования CASE-технологий. На сегодняшний день практически всё промышленно производимое сложное программное обеспечение разрабатывается с использованием CASE-средств.

Лекция №3. Структурное и неструктурное программирование. Основы алгоритмизации

Цель – получить представление об особенностях структурного и неструктурного программирования; изучить особенности представления алгоритмов, а также основные алгоритмические структуры.

Одним из способов обеспечения высокого уровня качества разрабатываемого программного обеспечения является *структурное программирование*.

В основе любой программы лежит *алгоритм*. Понятие «алгоритм» происходит от имени математика IX в. Аль Хорезми, который сформулировал правила выполнения арифметических действий. Первоначально под алгоритмом понимали только правила выполнения четырех арифметических действий над числами. В дальнейшем это понятие стали использовать вообще для обозначения последовательности действий, приводящих к решению любой поставленной задачи. *Алгоритм решения вычислительной задачи* - это формальное описание способа решения задачи путем разбиения ее на конечную по времени последовательность действий (этапов), понятных исполнителю. При этом должны быть четко указаны как содержание каждого этапа, так и порядок выполнения этапов. Отдельный этап алгоритма либо представляет собой другую, более простую задачу, алгоритм которой разработан ранее, либо должен быть достаточно простым и понятным без пояснений. Основными *свойствами* алгоритма являются:

1) детерминированность (определенность) - получение однозначного результата вычислительного процесса при заданных исходных данных, поэтому процесс выполнения алгоритма носит механический характер;
2) результативность - реализуемый по заданному алгоритму вычислительный процесс должен через конечное число шагов остановиться и выдать из исходных данных искомый результат;
3) массовость - алгоритм должен быть пригоден для решения всех задач данного типа;
4) дискретность - расчлененность вычислительного процесса на отдельные этапы.

Существует несколько способов записи алгоритмов: словесный, формульно-словесный, графический, язык операторных схем, алгоритмический язык.

Наибольшее распространение благодаря своей наглядности получил графический способ записи алгоритмов с помощью *блок-схем*.

Блок-схемой называется графическое изображение логической структуры алгоритма, в котором каждый этап процесса обработки информации представляется в виде геометрических символов (блоков), имеющих определенную конфигурацию в зависимости от характера выполняемых операций. Графические символы, их размеры и правила

построения схем алгоритмов определены Единой системой программной документации (ЕСПД), являющейся государственным стандартом (ГОСТ). Перечень символов, их наименование, отображаемые ими функции, форма и размеры определяются ГОСТами (таблица Б.1). Все формулы в блок-схеме записываются на языке математики, а не конкретном языке программирования.

Различают три вида вычислительного процесса, реализуемого программами: линейный, разветвленный и циклический. *Линейная структура* процесса вычислений предполагает, что для получения результата необходимо выполнить операции в определенной последовательности. При *разветвленной структуре* процесса вычислений конкретная последовательность операций зависит от значений одной или нескольких переменных. Для получения результата при *циклической структуре* некоторые действия необходимо выполнить несколько раз. Для реализации этих вычислительных процессов в программах используют соответствующие управляющие операторы.

Программы, написанные с использованием только структурных операторов передачи управления, называют *структурными,* чтобы подчеркнуть их отличие от программ, разрабатываемых с использованием низкоуровневых способов передачи управления.

После того, как в 60-х годах XX в. было доказано, что любой сложный алгоритм можно представить, используя три основные управляющие конструкции, в языках программирования высокого уровня появились управляющие операторы для их реализации [3, 4]. К *базовым* относят:

а) *следование* - обозначает последовательное выполнение действий;

б) *ветвление* - выбор одного из двух вариантов действий;

в) *цикл-пока* - определяет повторение действий, пока не будет нарушено некоторое условие, выполнение которого проверяется в начале цикла.

Кроме базовых, процедурные языки программирования высокого уровня используют три *дополнительные* конструкции, реализуемые через базовые:

а) *выбор* - выбор одного варианта из нескольких в зависимости от значения некоторой величины;

б) *цикл-до* - повторение действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле;

в) *цикл с заданным числом повторений* (счетный цикл) - повторение некоторых действий указанное количество раз.

Перечисленные конструкции были положены в основу структурного программирования. Программы, написанные с использованием только структурных операторов передачи управления, называют *структурными,* чтобы подчеркнуть их отличие от программ, разрабатываемых с использованием низкоуровневых способов передачи управления. Недостатки схем:

а) низкий уровень детализации, что скрывает суть сложных алгоритмов;

б) использование неструктурных способов передачи управления, которые на схеме выглядят проще, чем эквивалентные структурные.

Пример 3.1 – Использование блок-схемы для описания алгоритма поиска в массиве A(n) элемента, равного заданному (рисунок 3.1).

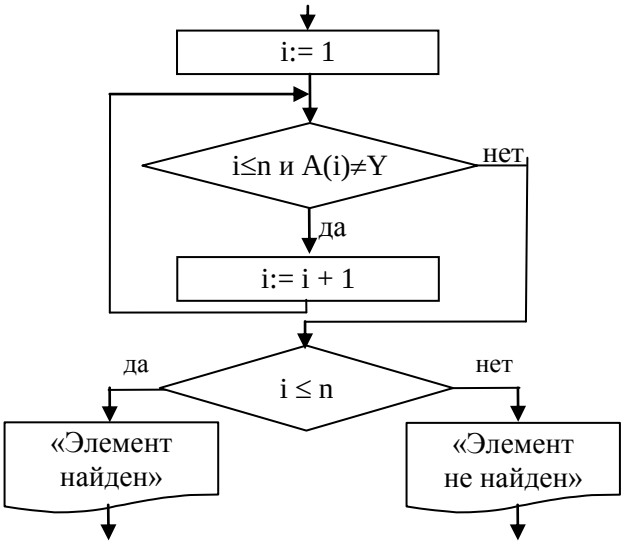


Рисунок 3.1 – Фрагмент блок-схемы алгоритма поиска

В приведенном примере используется структурный вариант алгоритма (цикл-пока). Элементы массива перебираются и поочередно сравниваются с заданным значением *Y*. В результате выводится соответствующее сообщение.

К недостаткам блок-схем можно отнести следующие:

а) низкий уровень детализации, что скрывает суть сложных алгоритмов;

б) использование неструктурных способов передачи управления, которые на схеме выглядят проще, чем эквивалентные структурные.

Кроме схем, для описания алгоритмов можно использовать *псевдокоды*, *Flow-формы* и *диаграммы Насси-Шнейдермана*, которые базируются на тех же основных структурах, допускают разные уровни детализации и делают невозможным описание неструктурных алгоритмов [1, 3, 8].

Псевдокод - формализованное текстовое описание алгоритма (текстовая *нотация* в нескольких вариантах, таблица В.1). Изначально ориентирует проектировщика только на структурные способы передачи управления, не ограничивают степень детализации проектируемых операций, позволяют соизмерять степень детализации действия с рассматриваемым уровнем абстракции и хорошо согласуются с методом пошаговой детализации.

Пример 3.2 – Использование псевдокода для описания алгоритма поиска в массиве A(n) элемента, равного заданному (фрагмент).

i:=1
Цикл-пока $i \le n$ *и* $A(i) \ne Y$
i:= i + 1
Все-цикл
Если $i \le n$
то *Вывести «Элемент найден»*
иначе *Вывести «Элемент не найден»*

Все-если

Flow-формы - графическая нотация описания структурных алгоритмов, иллюстрирующая вложенность структур. Каждому символу Flow-формы соответствует управляющая структура, изображаемая в виде прямоугольника и содержащая текст в математической нотации или на естественном языке. Для демонстрации вложенности структур символ Flow-формы вписывается в соответствующую область прямоугольника любого другого символа. В таблице В.1 приведены символы Flow-форм, соответствующие основным и дополнительным управляющим конструкциям.

Пример 3.3 – Использование Flow-форм для описания алгоритма поиска в массиве A(n) элемента, равного заданному (рисунок 3.2).

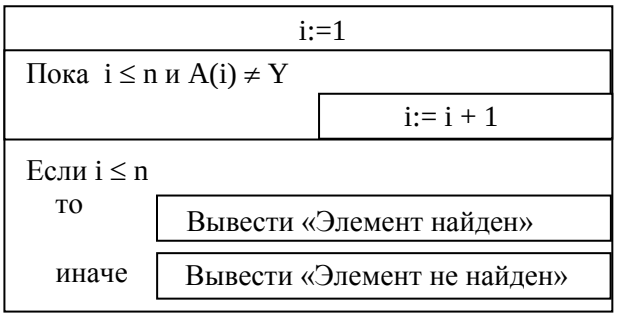


Рисунок 3.2 – Flow-форма алгоритма поиска (фрагмент)

Диаграммы Насси-Шнейдермана являются развитием Flow-форм лишь с той разницей, что область обозначения условий и вариантов ветвления изображают в виде треугольников (таблица В.1), обеспечивающих большую наглядность представления алгоритма.

Пример 3.4 – Использование диаграмм Насси-Шнейдермана для описания алгоритма поиска в массиве A(n) элемента, равного заданному (рисунок 3.3).

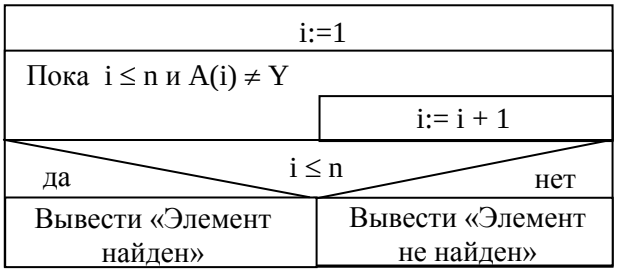


Рисунок 3.3 – Фрагмент диаграммы Насси-Шнейдермана

Так же, как при использовании псевдокодов описать неструктурный алгоритм, применяя Flow-формы или диаграммы Насси-Шнейдермана, невозможно (отсутствуют условные обозначения). В то же время, являясь графическими, эти нотации лучше отображают вложенность конструкций, чем псевдокоды. Недостаток: сложность построения изображений символов усложняет их практическое применение для описания больших алгоритмов.

Лекция №4. Алгоритмические языки и предъявляемые к ним требования. Процедурные языки

Цель – получить представление об основных характеристиках алгоритмических языков и их классификации; изучить особенности использования процедурных языков.

Языки программирования, которые используются при записи алгоритмов, обладают рядом характеристик, которые позволяют классифицировать, сравнивать и выбирать их с учетом целей разработки программы. К таким характеристикам относятся *мощность*, *уровень* и *целостность* [11].

Мощность языка характеризуется разнообразием задач, алгоритмы которых можно записать, используя этот язык. Поэтому, очевидно, что самым мощным является язык процессора, так как любая задача в конечном итоге записывается на языке компьютера.

Уровень языка определяется сложностью решения задач с использованием этого языка. Чем проще записывается решение, тем более непосредственно выражаются сложные операции и понятия, тем меньше объем получаемых исходных программ и, наконец, тем выше уровень языка.

Целостность языка обусловлена свойствами *экономии, независимости* и *единообразия* понятий.

Экономия понятий предполагает достижения максимальной мощности языка при условии использования минимального числа понятий. *Независимость* понятий означает, что правила использования одного и того же понятия в разных контекстах также должны быть одними и теми же, кроме того, между ними не должно быть взаимного влияния. *Единообразие* понятий требует единого согласованного подхода к описанию и использованию всех понятий.

Традиционно при классификации языков программирования используется такая характеристика, как уровень языка (рисунок 4.1). На нижних уровнях размещаются машинно-ориентированные языки, а на верхних – машинно-независимые.

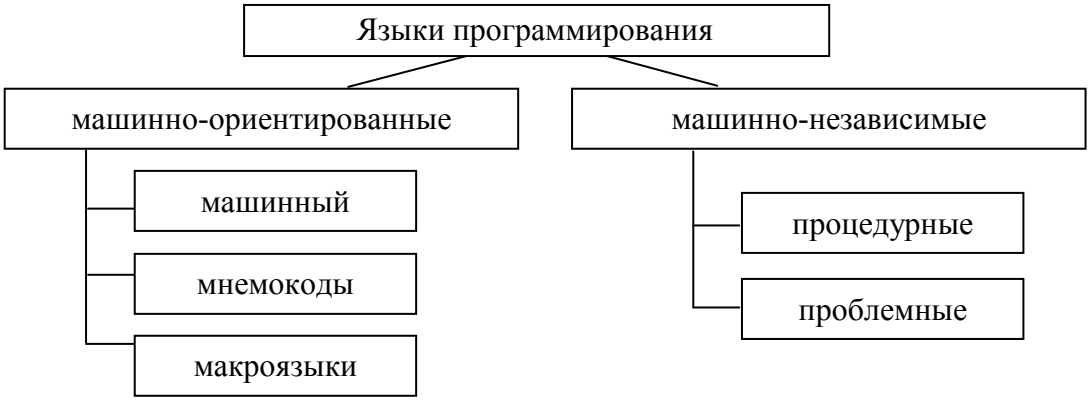


Рисунок 4.1 – Классификация языков программирования по уровню

Машинно-ориентированные языки позволяют в полной мере учитывать особенности процессора и получать программы с высокой степенью быстродействия. Однако они не способны обеспечить мобильность (переносимость) программ между разнотипными компьютерами. Под *мнемокодами* подразумеваются языки ассемблера без макросредств, к *макроязыкам* относятся языки ассемблера с макросредствами.

Машинно-независимые языки еще называются языками *высокого уровня*. *Процедурные* языки - это алгоритмические языки, которые предназначены для описания процедуры решения задачи, то есть программист должен указать компьютеру, *что* и *как* следует сделать для решения задачи. Среди процедурных выделяют группу *универсальных* языков, пригодных для решения любых задач (например, Паскаль, C/C++, Ада). *Непроцедурные (проблемные)* языки позволяют указать компьютеру, *что* нужно сделать для решения задачи, а *как* это сделать - система программирования решает автоматически. Среди проблемных языков выделяют языки *СУБД, объектно-ориентированные, веб-программирования, функциональные, логические* и другие. Каждая из этих групп отличается от других не уровнем, а принципами программирования.

Таким образом, классификация, которая показана на рисунке 4.1, содержит 5 уровней языков: 0 – машинные, 1 – мнемокоды, 2 – макроязыки, 3 – процедурные, 4 – проблемные языки. Но в связи с тенденцией универсализации языков, эта классификация уже не является строгой, поскольку языки ассемблеров приобрели средства, присущие языкам высокого уровня, а язык C++ вообще имеет признаки как высокоуровневого, так и низкоуровневого языка. Поэтому класс языка имеет смысл определять по уровню его основополагающих средств.

При использовании любого языка необходимо следить за тем, чтобы все записываемые с его помощью предложения (строки) были корректны с точки зрения алфавитных конструкций (*синтаксически*) и имело определенный смысл (*семантику*). Переводом программы с исходного языка, на котором она написана, на машинный занимается программа - *транслятор*.

В исходных текстах программ, как правило, используются *комментарии*, т. е. пояснительный текст, оформленный определенным образом и никоим образом не влияющий на ход выполнения программы. Для *идентификации* (обозначения) всех объектов, вводимых в программу, используются *имена* (идентификаторы). Под *объектами* понимаются переменные, константы, типы данных, функции и т. д. Для каждого языка четко определены правила, согласно которым вводятся обозначения. *Ключевые* (служебные) слова имеют однозначно определенный смысл и могут использоваться только так, как это задано в языке. Ключевые слова не могут быть переопределены, т. е. их нельзя использовать в качестве имен, вводимых программистом [1, 6, 7].

Лекция №5. Введение в язык С++. Структура и этапы создания программы на языке С++. Стандарты языка С++

Цель – получить представление о языке программирования С++, его особенностях, структуре программ и процессе их создания.

Язык программирования высокого уровня С++ был разработан в США в начале 80-х годов сотрудником компании Bell Laboratories Бьерном Страуструпом (Bjarne Stroustrup) в результате расширения и дополнения языка С средствами, необходимыми для объектно-ориентированного программирования. Среди современных языков С++ относится к классу универсальных и по праву считается господствующим языком, используемым для разработки коммерческих программных продуктов. Пожалуй, лишь такой язык программирования, как Java может составлять ему конкуренцию. Разновидностью С++ является С# - новый язык, разработанный Microsoft для сетевой платформы. Несмотря на ряд принципиальных отличий, языки С++ и С# совпадают примерно на 90%. Особенно эффективно применение С++ в написании системных программ-трансляторов, операционных систем, экранных интерфейсов. В этом языке сочетаются лучшие свойства Ассемблера и языков программирования высокого уровня. Программы, выполненные на языке С++, по быстродействию сравнимы с программами, написанными на Ассемблере, но более наглядны, просты в сопровождении и легко переносимы с одного компьютера на другой. К основным особенностям языка относят следующие:

- С++ предлагает большой набор операций, многие из которых соответствуют машинным командам и поэтому допускают прямую трансляцию в машинный код, а их разнообразие позволяет выбирать различные наборы для минимизации результирующего кода;
- базовые типы данных С++ совпадают с типами данных Ассемблера, на преобразования типов налагаются незначительные ограничения;
- объем С++ невелик, т.к. практически все выполняемые функции оформлены в виде подключаемых библиотек, также С++ полностью поддерживает технологию структурного программирования и обеспечивает полный набор соответствующих операторов;
- С++ широко использует указатели на переменные и функции, кроме того, поддерживает арифметику указателей, и тем самым позволяет осуществлять непосредственный доступ и манипуляции с адресами памяти; удобным средством для передачи параметров являются ссылки;
- С++ содержит в себе все основные черты объектно-ориентированных языков программирования: наличие объектов и инкапсуляцию данных, наследование, полиморфизм и абстракцию типов.

При написании программ на языке С++ используются следующие понятия: алфавит, константы, идентификаторы, ключевые слова, комментарии, директивы [2, 5].

Алфавитом называют присущий данному языку набор символов, из которых формируются все конструкции языка. Язык С++ оперирует со следующим набором символов: латинские прописные и строчные буквы (A, B, C, ..., x, y, z); арабские цифры (0, 1, 2, ..., 7, 8, 9); символ подчеркивания («_»); специальные символы (список специальных символов языка С++ приведен в таблице Г.1); символы-разделители (пробелы, комментарии, концы строк и т.д.).

С помощью перечисленных символов формируются *имена*, *ключевые* (служебные) *слова*, *числа*, *строки* символов, *метки*.

Идентификаторы (имена) обязательно начинаются с латинской буквы или символа подчеркивания «_», за которыми могут следовать в любой комбинации латинские буквы и цифры. С++ различает прописные и строчные буквы. Не допускается использование для написания имен специальных символов и символов-разделителей. Например,

_x, B12, Stack - правильно;

Label.4, Root-3 - неправильно.

Существуют некоторые соглашения относительно использования прописных и строчных букв в идентификаторах. Например, имена переменных содержат только строчные буквы, константы и макросы – прописные. С символа подчеркивания обычно начинаются имена системных зарезервированных переменных и констант, а также имена, используемые в библиотечных функциях. Поэтому во избежание возможных конфликтов и взаимопересечений с множеством библиотечных имен не рекомендуется использовать знак подчеркивания в качестве первого символа имени.

Некоторые идентификаторы, имеющие специальное значение для компилятора, употребляются как *ключевые слова*. Их употребление строго определено, и они не могут использоваться иначе. Список зарезервированных слов в С++ приведен в таблице Г.2.

Числа, обозначающие целые и вещественные значения, записываются в десятичной системе счисления. Перед любым числом может стоять знак «+» или «-». В вещественном числе целая часть числа отделяется от его дробной части точкой. Вещественные числа, содержащие десятичную точку, должны иметь перед ней или после нее, по крайней мере, по одной цифре.

Имя *метки перехода* представляет собой символьно-цифровую конструкцию, например, *metkal, pass, cross15*, и в программе не объявляются.

Строка символов — это последовательность символов, заключенная в кавычки. Например, «*Строка символов*».

Различают два вида *комментариев*. Любая последовательность символов, заключенная в ограничивающие скобки /* */, в языках С/С++ рассматривается как многострочный комментарий, например,

*/*Главная программа*/.*

В языке С++ дополнительно имеется еще один вид комментария – однострочный: все символы, следующие за знаком // (двойной слеш) до конца строки, рассматриваются как комментарий, например, *//Главная программа.*

В основном, используют комментарий стиля C++ (//), а комментарий стиля C (/* */) применяют для временного отключения больших участков программы. Следует помнить, что комментарии должны пояснять, не *что* это за операторы, а *для чего* они здесь используются.

Программа, записанная на языке C/C++, обычно состоит из одной или нескольких функций. *Функция* – это самостоятельная единица программы, созданная для решения конкретной задачи, которая может оперировать данными и возвращать значение. Структура программы представлена на рисунке 5.1.

Каждая программа на языке C++ начинается с *директивы препроцессора #include*, которая подключает *заголовочный файл* (*.h), содержащий *прототипы* функций, которые сообщают компилятору информацию о синтаксисе функции, например,

include <iostream.h>

Заголовочный файл обычно содержит определения, предоставляемые компилятором для выполнения различных операций. Заголовочные файлы записаны в формате ASCII, их содержимое можно вывести для просмотра с помощью любого текстового редактора из каталога INCLUDE. *Препроцессор* просматривает программу до компилятора, подключает необходимые файлы, заменяет символические аббревиатуры в программе на соответствующие директивы и даже может изменить условия компиляции.

Каждая программа на C++ содержит, по крайней мере, одну функцию – *main()*, которая автоматически вызывается при запуске, может вызывать другие имеющиеся в программе функции и обычно имеет вид:

```
void main ( )
#include <имя.h>                    →   Подключение заголовочных файлов
function1 ( )                       →   Определение функций программы
{
  оператор;
}
function2 ( )
{
  оператор;
}
void main ( )                       →   Определение главной функции
{                                   →   Начало тела главной функции
  оператор 1;                       →   Операторы функции
    …
  оператор N;
function1 ( );                      →   Вызов функций программы
function2 ( );
…
}                                   →   Конец тела главной функции
```

Рисунок 5.1 – Структура программы на языке C++

Обычную функцию необходимо вызывать (обращаться к ней) программно, в ходе выполнения кода. Функция *main()* вызывается операционной системой, и обратиться к ней из кода программы невозможно. Слово *void* служит признаком того, что программа не возвращает конкретного значения. В случае возврата значения операционной системе перед функцией *main()* указывается слово *int*, а в конце тела этой функции помещается выражение *return()* или *return0*.

После определения главной функции следуют операторы программы, которые заключены в группирующие фигурные скобки *{ }*. Каждый оператор оканчивается точкой с запятой (;), указывающей на его завершение. Программа выполняется по строкам, в порядке их расположения в исходном коде, до тех пор, пока не встретится вызов какой-нибудь функции, тогда управление передается строкам этой функции. После выполнения функции управление возвращается той строке программы, которая следует за вызовом функции [2, 5, 10].

Лекция №6. Представление данных в языке С++. Оператор присваивания. Арифметические операции. Директивы препроцессора

Цель – получить представление о стандартных типах данных, порядке выполнения операций, изучить особенности оператора присваивания и использования препроцессора.

Определяя данные, необходимо предоставить компилятору информацию об их типе, тогда ему будет известно, сколько места нужно выделить (зарезервировать) для хранения информации и какого рода значение в ней будет находиться. В С++ определены пять базовых типов данных: символьные (*char*), целые (*int*), вещественный с плавающей точкой (*float*), вещественный с плавающей точкой двойной длины (*double*), а также пустой, не имеющий значения тип (*void*). На основе перечисленных типов строятся все остальные.

Простейшим приемом является использование модификаторов типа, которые ставятся перед соответствующим типом: знаковый (*signed*), беззнаковый (*unsigned*), длинный (*long*) и короткий (*short*). В таблице Г.3 приведены все возможные типы с различными комбинациями модификаторов с указанием диапазона изменения и занимаемого размера в байтах. При многократном использовании в программе типов данных с различными комбинациями модификаторов, например, *unsigned short int*, легко сделать синтаксические ошибки, во избежание которых в С++ предусмотрена возможность создания псевдонима (синонима) с помощью ключевого слова *typedef*. Например, строка *typedef unsigned short int USHORT;* создает новый псевдоним *USHORT*, который может использоваться везде, где нужно было бы написать *unsigned short int*.

Переменная – это имя, связанное с областью памяти, которая отведена для временного размещения хранимого значения и его последующего

извлечения. Для длительного (постоянного) хранения значений переменных используются базы данных или файлы. В C++ все переменные должны быть объявлены до их использования. Объявление предполагает наличие имени переменной и указание ее типа. Однако следует иметь в виду, что нельзя создать переменную типа *void*. Основная форма объявления переменных имеет вид

тип <список_переменных>;

В этом объявлении: *тип* – один из существующих типов переменных; <*список_переменных*> может состоять из одной или нескольких переменных, разделенных запятыми. Например,

int x, e, z; float radius; long double integral;

Можно объявлять переменные и одновременно присваивать им начальные значения, т.е. *инициализировать* их. Например, *int min=15; float p1=1.35;*

Переменная называется *глобальной*, если она объявлена вне каких-либо функций, в том числе функции *main()*. Такая переменная может использоваться в любом месте программы (за исключением глобальных статических переменных), а при запуске программы ей присваивается нулевое значение. Переменная, объявленная внутри тела функции (одного блока), является *локальной* и может использоваться только внутри этого блока. Вне блока она неизвестна. Важно помнить, что:

- две глобальные переменные не могут иметь одинаковые имена;
- локальные переменные разных функций могут иметь одинаковые имена;
- локальные переменные в одном блоке не могут иметь одинаковые имена.

Данные в языках программирования могут представляться также в виде *констант*. Константы используются в тех случаях, когда программе запрещено изменять значение какой-либо переменной. Для определения константы традиционным способом используется *#define*, которая просто выполняет текстовую подстановку. Например,

#define StudentsOfGroup 15

В данном случае константа *StudentsOfGroup* не имеет конкретного типа и каждый раз, когда препроцессор встречает имя *StudentsOfGroup*, он заменяет его литералом 15. Поскольку препроцессор запускается раньше, компилятор никогда не увидит константу, а будет видеть только число 15.

Наиболее удобным способом определения констант является следующий:

const тип имя_константы = значение_константы;

Этот способ облегчает дальнейшее сопровождение программы и предотвращает появление ошибок. Так как определение константы содержит тип, компилятор может проследить за ее применением только по назначению (в соответствии с объявленным типом). Например,

const int Diapazon=20;

Литеральные константы (*литералы*) – это значения, которые вводятся непосредственно в текст программы. Поскольку после компиляции нельзя изменить значения литералов, их также называют константами. Например, в выражении *int MyAge=19*; имя *MyAge* является переменной типа *int*, а число 19 – литеральной константой, которой нельзя присвоить никакого иного значения.

Символьная константа состоит из одного символа, заключенного в апострофы: ‘q’, ‘2’, ‘$’. Например,

const char month=’December’; .

К символьным константам относятся специальные символы (в том числе управляющие, список приведен в таблице Г.1).

Строковые константы состоят из последовательности символов кода ASCII, заключенной в кавычки, оканчивающейся нулевым байтом. Конец символьной строки (нулевой байт) обозначается символом NULL ('\0').

Перечислимые константы позволяют создавать новые типы данных, а затем определять переменные этих типов, значения которых ограничены набором значений константы. Для создания перечисляемой константы используется ключевое слово *enum*, а запись имеет вид:

enum имя_константы {список_значений_константы};

Значения константы в списке значений разделяются запятыми. Например,

enum COLOR {RED, BLUE, GREEN, WHITE, BLACK};

Каждому элементу перечисляемой константы соответствует определенное значение. По умолчанию, первый элемент имеет значение **0**, а каждый последующий - на единицу большее. Каждому элементу константы можно присвоить произвольное значение, тогда последующие инициализируются значением на единицу больше предыдущего. Например,

enum COLOR {RED=100, BLUE, GREEN=200, WHITE=300, BLACK};

В этом примере значение *BLUE=101, BLACK=301*.

Существует механизм явного задания типов констант с помощью суффиксов. Для констант целого типа в качестве суффиксов могут использоваться буквы *u, l, h, L,H,* а для чисел с плавающей точкой – *l, L, f, F*. Например,

12h 34H	-	short int
23L -273l	-	long int
23.4f 67.7E-24F	-	float
89uL 89Lu 89ul 89 LU	-	unsigned short

Выражение в языке C++ представляет собой некоторую допустимую комбинацию *операций* и *операндов* (констант, переменных или функций). Перечень операций языка C++ приведен в таблице Г.4. Все перечисленные операции выполняются традиционным способом, за исключением операции деления. Особенность операции деления заключается в том, что если оба операнда целого типа, то она даст целый результат, например, *3/2* даст *1*. Для

получения действительного результата необходимо иметь хотя бы один действительный операнд, например, *3/2.0* даст *1.5*.

Для каждой операции языка определено количество операндов:

а) один операнд – *унарная* операция, изменяющая знак, например, унарный минус *–x*;

б) два операнда – *бинарная* операция, например, операция сложения *x+y;*

в) три операнда – операция *условие ?:*, она единственная.

Каждая операция может иметь только определенные типы операндов. Каждая бинарная операция имеет определенный порядок выполнения: слева направо или справа налево. Наконец, каждая операция имеет свой приоритет. Приоритет и порядок выполнения операций приводятся в таблице Г.4.

Часто в выражениях используются математические функции языка С++, которые находятся в библиотеке *math*. Чтобы воспользоваться этими функциями в начало программы необходимо включить заголовочный файл *<math.h>*. Основные математические функции приводятся в таблице Г.5.

Все выражения являются *операторами*, которые в языке предназначены для описания действий. Любой оператор может быть помечен меткой. Операторы отделяются друг от друга точкой с запятой (;). В любом месте программы, где может быть размещен один оператор, можно разместить *составной* оператор, называемый *блоком*. Блок содержит несколько операторов, которые выполняются как одно выражение, ограничивается фигурными скобками {}, но не заканчивается точкой с запятой (;).

Объявление переменной в программе означает всего лишь выделение места в памяти компьютера для ее размещения. Программа же должна позволять оперировать данными. В этом процессе наиболее важна *операция присваивания*, которая выглядит следующим образом: *переменная = выражение*. Операция присваивания заменяет значение операнда, расположенного слева от знака «=», значением, вычисляемым справа от него. При этом могут выполняться неявные преобразования типа. Знак «=» в С/С++ - это знак присваивания, а не равенства.

В отличие от других языков, где присваивание – оператор по определению, в С/С++ существуют понятия «операция присваивания» и «оператор присваивания». Операция «превращается» в оператор, если в конце выражения поставить точку с запятой, например, *++x* – это выражение, а *++x* **;** - это оператор. Оператор присваивания удобно использовать при инициализации переменных, например, *j=k;*. Кроме того, в С/С++ операция присваивания может использоваться в выражениях, которые включают в себя операторы сравнения или логические операторы, например, *if ((x=x+5)>0) cout<<"Вывод";*.

Еще одной особенностью использования операции присваивания в С/С++ является возможность *многократного* присваивания, которое выполняется справа налево. Например, для того, чтобы присвоить значение *2*k* нескольким переменным, можно воспользоваться операцией: *x=y=z=2*k*.

В языке C/C++ имеются *дополнительные* операции присваивания **+=, -=, *=, /=** и **%=**. При этом величина, стоящая справа, добавляется (вычитается, умножается, делится или делится по модулю) к значению переменной, стоящей слева. Например, вместо оператора *x=x+5;* можно записать *x+=5;*. Причем, операция *x+=5* выполняется быстрее, чем операция *x=x+5*.

Очень часто в программах к переменным добавляется (или вычитается) единица. Увеличение значения на 1 называется *инкрементом* (++), а уменьшение на 1 - *декрементом* (--). Например, оператор *c=c+1;* эквивалентен оператору *c++;*, оператор *c=c-1;* эквивалентен оператору *c--;*.

Операторы инкремента и декремента существуют в двух вариантах: *префиксном* и *постфиксном*.

Префиксные операции увеличивают (уменьшают) значение переменной на единицу, а затем используют это значение. Например, оператор *x=++y;* эквивалентен выполнению двух операторов *y=y+1; x=y;*. В этом примере сначала происходит увеличение на единицу значения переменной **y**, а затем присваивание этого значения переменной *x*.

Постфиксные операции сначала используют значение переменной, после чего увеличивают (уменьшают) его. Например, оператор *x=y--;* эквивалентен выполнению двух операторов *x=y; y=y-1;*. В этом примере переменная *x* получает значение *y*, после чего значение *y* уменьшается на единицу.

Вообще в выражениях лучше использовать операнды одного типа, но C++ допускает *преобразование типов*, то есть если операнды принадлежат к разным типам, то они приводятся к некоторому общему типу. Приведение выполняется в соответствии со следующими правилами:

а) автоматически производятся лишь те преобразования, которые превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном значений, т.к. это происходит без какой-либо потери информации;

б) выражения, не имеющие смысла (например, число с плавающей точкой в роли индекса), не пропускаются компилятором еще на этапе трансляции;

в) выражения, в которых могла бы потеряться информация (например, при присваивании длинных целых значений более коротким или действительных значений целым), могут вызвать предупреждение (warning), но они допустимы.

В отличие от других языков программирования в C++ для любого выражения можно явно указать преобразование его типа, используя унарный оператор, называемый *приведением типа*. Выражение приводится к указанному типу по перечисленным правилам конструкцией вида

(имя типа) выражение;

Например, *(int) i=2.5*3.2;* .

Однако пользоваться этим оператором можно лишь в том случае, если вполне осознаются цель и последствия такого преобразования [2, 5, 10].

Лекция №7. Функции ввода/вывода. Основные конструкции языка С++

Цель – получить представление о функциях «ввода-вывода», используемых в С++, а также ознакомиться с особенностями использования основных конструкций языка.

В любой достаточно сложной программе можно выделить *линейные* фрагменты. Фрагмент программы имеет *линейную* структуру, если все операции в нем выполняются последовательно, друг за другом, и может содержать операторы присваивания, математические функции, арифметические операции, функции «ввода-вывода» данных и другие операторы, не изменяющие общего порядка следования операторов.

Редкая программа обходится без операций «ввода-вывода». В языке С были реализованы две новаторские идеи: средства «ввода-вывода» были отделены от языка и вынесены в отдельную библиотеку *stdio* (стандартная библиотека «ввода-вывода»), а также была реализована концепция процесса «ввода-вывода», независимого от устройств. Именно поэтому язык С++, унаследовавший черты своего «прародителя», имеет большой набор функций «ввода-вывода»данных различных типов.

Для реализации форматного «ввода-вывода» часто используются две функции *printf* и *scanf*, подключаемые с помощью заголовочного файла *<stdio.h>*. Функцию *printf* можно использовать для вывода любой комбинации символов, целых и вещественных чисел, строк, беззнаковых целых, длинных целых и беззнаковых длинных целых. Она описывается следующим образом: *printf (“управляющая_строка”, список_аргументов);*.

Список аргументов – это последовательность констант, переменных или выражений, значения которых выводятся на экран в соответствии с форматом *управляющей строки*, которая определяет количество, тип аргументов и обычно содержит следующие объекты: обычные символы, выводимые на экран без изменений; *спецификации преобразования*, каждая из которых вызывает вывод на экран значения очередного аргумента из последующего списка аргументов; *управляющие символьные константы* (список наиболее используемых констант приводится в таблице Г.1). Спецификация преобразования начинается с символа *%* и заканчивается символом преобразования (таблица Г.6), между которыми могут записываться:

- знак «минус», указывающий на то, что выводимый текст выравнивается по левому краю, по умолчанию, выравнивание происходит по правому краю;
- строка цифр, задающая минимальный размер поля вывода;
- точка, являющаяся разделителем;
- строка цифр, задающая точность вывода;
- символ *l*, указывающий, что соответствующий аргумент имеет тип *long*.

Например, *printf("\nВозраст Эрика - %d. Его доход $%.2f", age, income);*

Предполагается, что целой переменной *age* (возраст) и вещественной переменной *income* (доход) присвоены какие-то значения. Последовательность символов *\n* переводит курсор на новую строку, поэтому последовательность символов *«Возраст Эрика»* будет выведена с начала новой строки. Символы *%d* являются символами преобразования формата (спецификацией) для переменной *age*. Затем следует литерная строка *«Его доход $»* и символы *%.2f* - спецификация для значения переменной *income*, а также указание формата для вывода только двух цифр после десятичной точки.

Функция ввода данных *scanf* описывается аналогично функции *printf*:

scanf (“управляющая_строка”, список_аргументов);

Аргументы функции *scanf* должны быть *указателями* на соответствующие значения (более подробно указатели будут рассмотрены позже), для чего перед именем переменной записывается символ *&*. Как и в функции *printf*, управляющая строка содержит спецификации преобразования и используется для установления количества и типов аргументов. В ней допустимо использование пробелов, символов табуляции и перехода на новую строку, которые игнорируются при вводе. В управляющей строке спецификации преобразования должны быть отделены теми же разделителями, что и при вводе с клавиатуры. Например, *scanf (“%d %f %c”, &i, &a, &ch);*

Программы на C++ не имеют дело ни с устройствами, ни с файлами - они работают с *потоками.* Ввод информации осуществляется из *входного потока,* вывод производится в *выходной поток*, которые связаны с устройством или с файлом. В C++ концепция независимого от устройств «ввода-вывода» получила дальнейшее развитие в виде объектно-ориентированной библиотеки «ввода-вывода» *iostream*, в которую входят объект *cout* для вывода данных на экран и объект *cin*, используемый для ввода информации [7, 8].

Оператор вывода *cout* выглядит следующим образом:

cout<<переменная1<<...<<переменнаяN;

Знак “**<<**” называется *операцией вставки*, которая вставляет символы в выходной поток. Для перевода курсора в начало следующей строки в операторе *cout* часто используется символ *endl* (конец строки), например, *cout<<xl<<endl;*. В операторе вывода можно использовать специальные символы, которые должны быть заключены в одинарные кавычки, если они используются самостоятельно, например: *cout<<'\a'<<"Звонок";*. Если же специальные символы используются внутри двойных кавычек, то дополнительно заключать их в апострофы не нужно.

Например,

cout<<"Вывод\tx="<<x<<endl;

Для организации форматного вывода чисел с помощью оператора *cout* можно, например, воспользоваться функциями *cout.width, cout.fill, cout.put,*

модификатором *setw*, манипулятором *setprecision*, включив в программу заголовочный файл <*iomanip.h*>.

Оператор ввода *cin* имеет вид:

cin>>переменная1>>...>>переменнаяN;

Знак “>>” называется *операцией извлечения* (эта операция извлекает данные из входного потока, присваивая значение указанной переменной). Значения вводимых переменных должны соответствовать типам переменных, указанным в операторах объявления, а при вводе с клавиатуры отделяются друг от друга хотя бы одним пробелом, например, *1.5 2.15 -1.1 25*.

Значения символьных переменных и строк при вводе записываются во входном потоке без апострофов или кавычек. Для их ввода С++ предоставляет дополнительно две функции: *cin.get* и *cin.getline*.

Для оформления внешнего вида программы можно воспользоваться функциями, которые описываются в заголовочном файле <*conio.h*>.

Для изменения естественной последовательности выполнения операторов (передачи управления) в С/С++ содержится ряд специальных конструкций, относящихся к *конструкциям принятия решений* и по своему смыслу совпадающих с аналогичными конструкциями алгоритмов.

К операторам передачи управления относятся оператор *безусловного перехода*, оператор *условного перехода*, оператор *выбора* (*варианта*), которые имеют аналоги и в других языках программирования (например, в Pascal), а также *троичный условный* оператор.

Оператор безусловного перехода имеет вид:

goto метка;

Метка перехода указывает в программе оператор, которому следует передать управление. При выполнении оператора *goto* переход осуществляется без проверки каких-либо условий. Поскольку такие переходы разрушают связи между структурой программы и структурой вычислений, что приводит к потере ясности программы и затрудняет задачу верификации, оператор безусловного перехода следует использовать только в исключительных ситуациях.

Оператор условного перехода предназначен для выбора одного из двух вариантов развития решения задачи в зависимости от значения некоторого проверяемого условия, и его полная форма имеет вид:

if (условие) оператор1; else оператор2;

В качестве *условия* используется некоторое произвольное выражение, задающее условие выбора выполняемого оператора; *оператор1* и *оператор2* могут быть как простыми, так и составными. Если условие истинно (TRUE или любое ненулевое значение), то выполняется *оператор1*, если же условие ложно (FALSE или нуль), то выполняется *оператор2*. Краткая форма оператора условного перехода имеет вид:

if (условие) оператор;

В этом случае, если условие истинно, то выполняется оператор, если же условие ложно, то управление передается следующему оператору программы.

Чаще всего условие представляет собой логическое выражение, состоящее из операндов и знаков операций. В качестве операций в логическом выражении, прежде всего, используются операции сравнения *(*==, *!*=, <, >, <=, >=). Кроме операций сравнения, для построения логических выражений можно использовать логические операции *(!*, ||, &&). Значение логического выражения вычисляется путем выполнения указанных в нем операций с учетом их приоритета и расставленных круглых скобок, например,

(abs(x)<=2) - значение *x* по модулю не превышает *2*;

((x>=1)&&(x<=2)) - точка принадлежит отрезку [*1*,*2*];

*(x*x+y*y<1)* - точка с координатами (*x, y*) принадлежит единичному кругу с центром в начале координат.

Поскольку в C/C++ истина представляется как ненулевое значение, а ложь как нуль, то возможно и другое использование оператора условного перехода: *x = значение; if (x) оператор;*

В этом случае условие в операторе *if* представляет собой не логическое выражение, а переменную, которой предварительно присваивается какое-либо значение. Если значение переменной отлично от нуля, то условие истинно, если значение переменной равно нулю, то условие ложно.

Внутри оператора *if* допустимо использование вложенных конструкций:

```
if (выражение1) оператор1;
        else if (выражение2) оператор2;
        ...
          else if (выражениеN) операторN;
             else оператор_по_умолчанию;          // необязательная часть
```

Во избежание неоднозначного толкования программы нужно пользоваться фигурными скобками для выделения вложенностей в единые блоки, так как в подобных конструкциях *else* связывается с ближайшим предыдущим *if*.

Если в программе необходимо выбрать один из многочисленных вариантов, то вместо вложенной конструкции *if* целесообразнее применять оператор-переключатель *switch*, называемый оператором выбора или варианта, который имеет вид:

```
switch (выражение)
{case n1 : оператор1; break;
case n2  : оператор2; break;
case nK : операторK; break;
default  : операторN; break;}
```

Выполнение оператора варианта начинается с вычисления значения выражения (*селектора*). Затем оператор выбора передает управление тому оператору, перед которым стоит константа, совпадающая с вычисленным значением переключателя *switch*. Если совпадений не обнаружено, выполняется оператор, стоящий после *default*. Допускается конструкция *switch*, в которой оператор *default* может отсутствовать. Оператор *break*, расположенный в каждой ветви оператора варианта, предписывает завершить

выполнение текущего оператора и передать управление следующему оператору программы. Отсутствие оператора *break* предписывает программе продолжать выполнение до первого оператора *break* либо до конца оператора *switch*.

Альтернативой оператору условного перехода является *троичный условный оператор ?:* - единственный оператор, который работает с тремя операндами. Этот оператор получает три выражения и возвращает значение:

(выражение_1) ? выражение_2: выражение_3;

Выполнение оператора начинается с вычисления *выражения_1*. Если выражение истинно, то результатом будет являться значение *выражения_2*, в противном случае результатом будет значение *выражении_3*. Например, оператор *max = (x>y) ? x: y;* определяет наибольшее из двух чисел *x* и *y*.

Алгоритм, в котором предусмотрено многократное выполнение одной и той же последовательности действий, называется алгоритмом *циклической структуры*, а эта последовательность – *циклом*. Циклический алгоритм позволяет существенно сократить объем программы. В С++ используются операторы циклов трех видов: с предварительным условием, с последующим условием и с параметром.

Оператор цикла с предварительным условием (с *предусловием*) относится к базовым конструкциям, выполняет повторяющиеся действия до тех пор, пока заданное условие истинно, и имеет вид:

while (условие)
{ оператор1;
оператор2;
операторN;} ,

где *условие* представляет собой любое простое или сложное выражение языка С++; *оператор* – любой допустимый оператор или блок операторов (тело цикла).

Цикл выполняется до тех пор, пока *условие* принимает значение TRUE. При ложном значении (FALSE) цикл завершается, и программа передает управление следующему оператору программы. Так как проверяемым условием в цикле *while* может быть любое допустимое выражение, то, в принципе, возможно использование и выражения *true*. Однако в этом случае цикл не завершится никогда, став бесконечным, что приведет к зависанию компьютера. Бесконечные циклы следует использовать очень осторожно и тщательно проверять. Если анализируемое условие с самого начала ложно, ни один из операторов, образующих тело цикла, не будет выполнен ни одного раза, т.е. цикл будет полностью пропущен.

Если необходимо гарантировать выполнение операторов цикла хотя бы один раз, следует воспользоваться *оператором цикла с последующим условием* (с *постусловием*), синтаксис которого имеет вид:

do
{ оператор1;
оператор2;

операторN;}
while (условие);

Если в теле цикла содержится только один оператор, то операторные скобки пишутся для устранения неоднозначности, обусловленной наличием *while*. Действия, определяемые оператором, выполняются до тех пор, пока *условие* не станет ложным или равным нулю. Назначения *условия* продолжения цикла и операторов аналогичны оператору *while*. Тело цикла с постусловием, как и в случае с *while*, обязательно должно содержать действия, влияющие на результат выполнения выражения, которое является условием входа/выхода из цикла, иначе цикл станет бесконечным.

Наконец, используемый в C/C++ *оператор цикла с параметром* имеет вид:

for (инициализация; проверка; приращение)
оператор;

Оператор *инициализация* устанавливает начальное значение счетчика. Оператор *проверка* – это любое выражение C++, результат которого проверяется на каждой итерации: если результат TRUE, то выполняется тело цикла. После изменения счетчика на величину *приращения* (по умолчанию, увеличение на 1), действия повторяются. Все эти выражения не являются обязательными.

Инициализирующее выражение, если оно есть, будет выполняться всегда. Вычисление конечного выражения (*проверка*) может не производиться, если условие ложно с самого начала. *Приращение*, как правило, определяет закон изменения параметра цикла, но и это необязательно. Более того, цикл *for* в C/C++ не является классическим циклом с параметром: параметр цикла может быть вещественным значением или изменяться по указанному закону.

В цикле *for* можно опустить одно или более выражений, но при этом нельзя опускать символы «;». Необходимо только включить в тело цикла несколько операторов, которые приведут к завершению его работы [7, 8].

Если цикл не инициализирует параметр цикла, не содержит управляющего выражения и не выполняет никаких действий, он называется *открытым*.

Существует несколько способов прервать выполнение цикла или изменить порядок следования операторов тела цикла. Иногда необходимо перейти к началу цикла до завершения выполнения всех операторов тела цикла. С этой целью используется оператор *continue;*, который осуществляет переход в начало цикла, пропуская все оставшиеся операторы. В ряде случаев требуется выйти из тела цикла еще до проверки условия продолжения цикла. Для этого используется оператор *break;*, который осуществляет выход из цикла, пропуская все, вплоть до закрывающейся фигурной скобки.

Операторы *continue* и *break* следует использовать очень осторожно, т.к. их свободное применение способно запутать даже простой цикл *while* и сделать его нечитабельным. Это наиболее опасные команды после *goto* [10].

Лекция №8. Сложные типы данных: массивы. Одномерные и многомерные массивы. Организация алгоритмов сортировки

Цель – получить представление о сложных типах данных, изучить методы обработки и особенности использования массивов.

В C/C++ имеется возможность строить на основе базовых типов сложные типы данных. К ним относятся массивы, структуры, объединения и файлы.

Массив — это упорядоченный набор элементов одинакового типа, имеющих общее имя. Все элементы массива пронумерованы. Порядковый номер элемента в массиве называется *индексом.* Массивы должны быть обязательно описаны перед использованием в программе. Различают одномерные (*векторы)*, двумерные *(матрицы)* и многомерные массивы. Общий вид объявления массива:

тип_элементов имя_массива [N1][N2]...[Nk];

Количество индексов *[N1][N2]...[Nk]* определяет размерность массива. При объявлении массива указывается общее число элементов массива. Индексация элементов массива в C/C++ по умолчанию начинается с нуля. Например, первый элемент одномерного массива имеет индекс [0], двумерного — [0][0] и т.д. Размер массива может задаваться константой или константным выражением. Нельзя задать массив переменного размера, для этой цели используется отдельный механизм - *динамическое выделение памяти*.

При объявлении массива возможна его инициализация. В этом случае первоначальные значения элементов массива указываются между левой и правой фигурными скобками, следующими за знаком равенства. Например,

int a[3]={10, 20, 30}, b[ ]={10, 20, 30};

float c[2][3]={5, 10,15, 20, 25, 30};

Если массив объявлен без указания размерности (массив *b*), то компилятор сам определяет необходимое количество элементов массива. При объявлении массива с неизвестным количеством элементов можно не указывать размер только в самых левых квадратных скобках, например,

float d[ ][3]={{5, 10, 15},{20, 25, 30}};

Если при инициализации задано меньше начальных значений массива, чем их общее количество, остальные элементы инициализируются нулевыми значениями.

К элементам массива применимы две операции: *присваивание* и *выборка*.

Для выборки элементов массива чаще всего используются *циклы*. Например, для ввода (вывода) элементов одномерного массива *a[100]* с клавиатуры можно использовать цикл с параметром

```
for (i=0; i<100; i++)
cin >> a[i];                //при выводе используется оператор  cout << a[i];
```

Для организации ввода (вывода) значений элементов многомерного массива используются *вложенные* циклы (их количество равно количеству индексов). Например,

```
for (i=0; i<10; i++)   //внешний цикл, выборка организована по строкам
for (j=0; j<5; j++)    //внутренний (вложенный) цикл, по столбцам
cin >> a[i][j];        //при выводе используется оператор  cout << a[i][j];
```

Для формирования массива случайным образом используется генератор случайных чисел. Например,

```
#include <stdlib.h>//подключение заголовочного файла, содержащего
...                //функцию random
randomize();       //инициализация генератора случайных чисел
...
for (i=0; i<M; i++)
{x[i]=random(100);      //формирование массива из M элементов,
cout<<x[i]; }           //содержащего целые значения в диапазоне 0 – 99
```

Для получения с равной вероятностью положительных и отрицательных значений элементов из сгенерированного значения следует вычесть число, равное половине аргумента функции *random*. Например, *x[i]=random(100)-50;*

При использовании характерных приемов программирования при работе с массивами также организуются циклы. На практике реализация основных приемов (таблица Г.7) сводится к следующим действиям: до открытия цикла задается начальное значение накапливаемого или предположительное значение искомого параметра; накапливание или поиск осуществляется непосредственно внутри цикла.

Во многих задачах возникает необходимо переставить элементы массива таким образом, чтобы они располагались в порядке возрастания или убывания. Такие массивы называются *упорядоченными,* а процесс их получения — *сортировкой*.

Самым простым алгоритмом является *сортировка простым выбором*. В сортируемом массиве находится минимальный элемент, который обменивается местами с элементом в начале массива. Оставшаяся часть массива рассматривается как самостоятельный массив, в котором также производится поиск наименьшего элемента и его обмен с первым элементом укороченного массива. Действия повторяются до тех пор, пока массив не укоротится до одного элемента [7, 8].

Пример 8.1 – Отсортировать массив методом сортировки простым выбором.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
{ const int N=6; int mas[N], i, j, i_min, j_min, min;
  randomize();
  clrscr();
```

```
 for (i=0; i<N; i++)
        { mas[i]=random(100)-50;  printf (“%4d”, mas[i]); }
 printf (“\n”);
 for (i=0; i<N; i++)
        { min=mas[i];  i_min=i;
          for (j=i+1; j<N; j++)                  // перебор в уменьшенном массиве
                if (mas[j]<min) { min=mas[j]; j_min=j; }
          mas[j_min]=mas[i]; mas[i]=min; }
 for (i=0; i<N; i++)
        printf (“%4d”, mas[i]);
 printf (“\n”); }
```

Под *пузырьковой* сортировкой понимают целый класс алгоритмов сортировки. В своем простейшем варианте пузырьковая сортировка выполняется крайне медленно, поэтому обычно применяют пузырьковую сортировку с элементами оптимизации. Все алгоритмы пузырьковой сортировки имеют общую особенность – обмен элементов производится между двумя соседними элементами массива.

Пример 8.2 - Отсортировать массив из 6 элементов пузырьковым методом.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
{ const int N=6; int mas[N], i, j, wrk;
 randomize();
 clrscr();
 for (i=0; i<N; i++)
        { mas[i]=random(100)-50; printf (“%4d”, mas[i]); }
 printf (“\n”);
 for (i=0; i<N-1; i++)                    // счетчик проходов
        for (j=0; j<N-1; j++)
                if (mas[j]<mas[j+1])
                { wrk=mas[j]; mas[j]=mas[j+1]; mas[j+1]=wrk; }
  for (i=0; i<N; i++)
  printf (“%4d”, mas[i]);
printf (“\n”); }
```

Можно уменьшить количество проходов сортировки, выполняя их не $(N-1)^2$ раз, а пока массив не будет отсортирован. Определить этот факт достаточно просто: если массив уже отсортирован, то в процессе прохода в нем не происходит никаких перестановок. Перед началом просмотра нужно установить признак отсутствия перестановок (*флаг*). В случае, если производится хотя бы одна перестановка, флаг изменяет свое значение. Если к моменту завершения прохода значение флага осталось первоначальным, значит, массив отсортирован и дальнейшие проходы не нужны.

Пример 8.3 - Отсортировать массив пузырьковым методом с оптимизацией по количеству проходов.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
{ const int N=6; int mas[N], j, wrk, flag;
  randomize();
  clrscr();
  for (j=0; j<N; j++)
        { mas[j]=random(100)-50; printf ("%4d", mas[j]); }
  printf ("\n");
  flag=1;                                  // оператор нужен для входа в цикл
  while (flag!=0)
        { flag=0;
          for (j=0; j<N-1; j++)
                if (mas[j]<mas[j+1])
                { flag++; wrk=mas[j]; mas[j]=mas[j+1]; mas[j+1]=wrk; }}
   for (j=0; j<N; j++)
   printf ("%4d", mas[j]);
printf ("\n"); }
```

Для оптимизации метода пузырьковой сортировки по времени выполнения каждого прохода можно использовать то, что после первого прохода наибольший элемент окажется в конце массива на предназначенном ему месте. В процессе выполнения второго прохода то же самое произойдет со следующим по величине элементом и т. д. Поэтому при последующих проходах можно уменьшать длину просмотра массива, что существенно сократит общее время выполнения алгоритма. Если объединить этот метод оптимизации с проверкой признака завершения сортировки, то получится алгоритм обменной сортировки с признаком завершения [6, 10].

Пример 8.4 - Отсортировать массив методом обменной сортировки с признаком завершения.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
{ const int N=6; int mas[N], j, wrk, flag, k;
  randomize();clrscr();
  for (j=0; j<N; j++)
        { mas[j]=random(100)-50; printf ("%4d", mas[j]); }
  printf ("\n");
  flag=1;                                  // оператор нужен для входа в цикл
  k=1;                                     // k –уменьшение длины просмотра
```

```
  while (flag!=0)
        { flag=0;
          for (j=0; j<N-k; j++)
                if (mas[j]<mas[j+1])
                { flag++; wrk=mas[j]; mas[j]=mas[j+1]; mas[j+1]=wrk; }
        k++; }
   for (j=0; j<N; j++)
   printf ("%4d", mas[j]);
printf ("\n"); }
```

Лекция №9. Сложные типы данных: символьные массивы. Обработка символьных данных

Цель – получить представление об особенностях обработки и использовании символьных массивов

Строка символов в C/C++ представляет собой массив значений типа *char*, завершающийся нулевым байтом *'\0'* - признаком конца строки, что следует учитывать при объявлении строки, т.е. указывать не *N*, а *N+1* элемент. Например, строка из 10 символов будет объявлена: *char str [11];*.

При инициализации строк используются традиционные методы объявления. Например, *char My_symbols[ ]="ABCD";*

или его эквивалент *char My_symbols[ ]={'A', 'B', 'C', 'D'};*

Для определения константы, равной длине инициализированной строковой переменной можно воспользоваться функцией *sizeof()*, например,

char Stroka_symb[ ]="ABCDEFG";

#define Length_s(sizeof(Stroka_symb))

При использовании *строковых констант* - литер, заключенных в двойные кавычки, - нулевой байт '\0' в конце не ставится.

Ввод строк с клавиатуры удобно производить с помощью функций *scanf()*, которая позволяет ввод до первого пробельного символа, или *gets()*, которая позволяет вводить строки, содержащие пробелы. Ввод оканчивается нажатием клавиши *Enter*. Обе функции автоматически ставят в конец строки нулевой байт. В качестве параметра используется просто имя массива.

Вывод производится с помощью функций *printf()* или *puts()*. При использовании всех перечисленных функций требуется подключить заголовочный файл <*stdio.h*>. Обе функции выводят содержание массива до первого нулевого байта. Причем, функция *puts()* добавляет в конце выводимой строки символ новой строки, а при использовании функции *printf()* переход на новую строку надо предусмотреть в строке формата.

Для работы со строками существует специальная библиотека, описание которой находится в файле <*string.h*>.

Наиболее часто используются следующие функции:

- функция *strlen(str);* определяет длину строки *str* (в символах), при этом завершающий нулевой байт не учитывается;

- функция *strcpy(str1, str2);* – копирует строку *str2* в строку *str1*. Массив *str1* должен быть достаточно большим, чтобы в него поместилась строка *str2*. При этом результирующая строка автоматически завершается нулевым байтом. Если места мало, компилятор не выдает указания на ошибку или предупреждения, это не прервет выполнения программы, но может привести к порче других данных или самой программы и неправильной работе программы в дальнейшем;

- функция *strcat (str1, str2);* присоединяет строку *str2* к строке *str1* и помещает её в массив, где находилась строка *str1*, при этом строка *str2* не изменяется. Нулевой байт, который завершал строку *str1*, будет заменен первым символом строки *str2*. Результат записывается в строку *str1*, причем результирующая строка также автоматически завершается нулевым байтом;

- функция *strcmp(str1, str2);* сравнивает строки *str1, str2*; возвращает 0, если строки равны (т.е. содержат одно и то же число одинаковых символов), число меньше нуля, если *str1<str2*; и число больше 0, если *str1>str2*. Сравнение строк производится посимвольно до первого несовпадающего символа, и та строка считается большей, в которой первый несовпадающий символ имеет больший код. (A<a, a<я);

- функция *strlwr(str);* осуществляет преобразование прописных символов в строчные (A -> a);

- функция *strupr(str);* осуществляет преобразование строчных символов в прописные (a ->A);

- функция *strset(str,'*');* организует заполнение строки str заданным символом *;

- функция *strstr(ss,w);* позволяет отыскать в строке *ss* подстроку *w*;

- функция *int(a);* позволяет получить код символа *a*.

Помимо функций ввода символьных переменных и строк, описанных выше, C++ предоставляет пользователю дополнительно еще две функции *cin.get* и *cin.getline*. В том случае, если необходимо вводить по одному символу с клавиатуры, можно воспользоваться функцией *cin.get*, например, *c1=cin.get()*. В том случае, если необходимо ввести целую строку, можно воспользоваться функцией *cin.getline*. Например, *cin.getline(c1,21)*, где c1 – имя строки, 21 – количество символов, записанных в символьную строку.

При обработке символьных данных используются те же способы и характерные приемы, что и при обработке числовых массивов. Строку, как и любой другой массив, можно обрабатывать либо традиционным методом как массив с использованием операции индексации, либо через указатели с использованием операций адресной арифметики.

При работе со строкой как с массивом нужно иметь в виду, что длина строки заранее неизвестна, поэтому циклы целесообразно организовывать не со счетчиком, а до появления признака конца строки [7, 8].

Лекция №10. Сложные типы данных: структуры и объединения

Цель – получить представление об особенностях обработки и использовании структур и объединений.

К сложным типам данных, кроме массивов, относятся структуры и объединения.

Структура (*struct*) представляет собой группу связанных компонентов (элементов) разных типов, количество которых фиксировано.

Прежде, чем создать объект структуры, следует определить ее формат, то есть объявить структуру. Переменные, составляющие структуру, называются ее *элементами* (*членами* или *полями*). *Имя* структуры – это *спецификатор типа*. Ключевое слово *struct* означает начало объявления структуры. Таким образом, описание типа *struct* имеет следующий вид:

```
struct   имя_структуры
{тип1 имя_элемента1;
...
 типN имя_элементаN; }
имя_переменной_типа_структуры;
```

К отдельным элементам структуры доступ осуществляется с помощью оператора «*точка*». Чтобы обратиться к элементу структуры, нужно перед его именем поставить имя структурной переменной и оператор «точка». Общий формат доступа имеет следующий вид:

Имя_структурной переменной . имя_элемента

Рассмотрим пример, который позволяет записать в структуре даты каких-либо событий следующим образом:

```
struct date
{int   day; char   month[9]; int year; }d;
```

В примере описана структура *date* и переменная *d*, принадлежащая типу *date*. Переменная *d* представляет собой структуру, состоящую из трех элементов, каждый из которых содержит следующие данные:

- элемент *day* — целое число, обозначающее день (от 1 до 31);
- элемент *month[9]* — строка символов с названием месяца;
- элемент *year* — целое число, обозначающее год.

Переменная типа структуры *d* может быть объявлена вне описания структуры при помощи имени, которое называют *структурным тэгом*. Тэг структуры может начинаться со слова *struct*:

```
struct   date   d;
```

К элементу структуры применима операция присваивания.

Например, чтобы переслать в переменную *d* дату *14 апреля 1930 г.*, программа должна содержать следующий фрагмент:

```
d.day=14;
d.month="апреля";
d.year=1930;
```

Структура может передаваться в функцию по имени. Например,

d.day=14;

strcpy(d.month,“апреля”);//использование функции копирования значения

d.year=1930;

Объединения (*union*) отличаются от структур способом хранения информации. В каждый момент времени объединение хранит значение только одного элемента. Память распределяется для хранения наибольшего элемента объединения. Описание типа *union* имеет следующий вид:

union имя_объединения

{тип1 имя_элемента1;

...

типN имя_элементаN;

} имя_переменной_типа_объединения;

Например, объединение для хранения единиц измерения времени можно представить следующим образом:

union time

{int hour; long second;} t;

В этом примере описано объединение *time* и переменная *t*, принадлежащая типу *time*. Переменная *t* описана как объединение, состоящее из двух элементов: *hour* и *second*, которые содержат следующие данные:

элемент *hour* — целое число типа *int*, обозначающее время в часах;

элемент *second* — целое число типа *long*, обозначающее время в секундах.

Так как наибольший элемент (*second*) имеет тип *long,* под элементы объединения будет отведено 4 байта памяти.

Переменная объединения может быть объявлена вне описания объединения, например:

union time

{int hour; long second;};

...

time t;

При обращении к элементу объединения нужно указать имя переменной и имя элемента объединения, разделив их точкой, аналогично обращению к элементу структуры:

t.hour=10;

Использование объединений в программах позволяет экономить память компьютера. Еще большей экономии памяти в программах на языках C/C++ можно достичь, если использовать *анонимные* объединения. В анонимных объединениях нет имени, переменная объединения не объявляется:

union { тип1 имя_элемента1;

...

типN имя элементаN; } ;

К элементам анонимного объединения обращаются по имени (без точки), как к обычным переменным в программе [6-8].

Рассмотрим пример использования анонимного объединения в программе.

```
#include<iostream.h>
void main(void)
{
union {int hour; long second; };
hour=720;
cout<<"В  месяце  "<<hour<<" часов. "<<endl;
second=2592000;
cout<<"В  месяце  "<<second<<" секунд."<<endl; }
```

В приведенном примере значение элемента *hour* анонимного объединения будет уничтожено в момент присваивания значения элементу *second*.

Лекция №11. Сложные типы данных: файлы. Файловые операции над массивами и структурами

Цель – получить представление об особенностях обработки и использовании файлов при работе с другими сложными типами данных.

Еще одним сложным типом данных, который часто используется в программах, являются файлы. *Файл* представляет собой последовательность элементов одного и того же типа, имеющих общее имя. Число элементов (длина файла) не ограничено. Файлы обычно создаются в оперативной памяти компьютера или на внешних устройствах.

Для работы с файлами можно использовать стандартные библиотечные функции, при этом в программу необходимо включить заголовочный файл <*fstream.h*>. Рассмотрим основные функции.

Запись данных в файл возможна в двух режимах:

1) создание нового файла (или перезапись существующего, ранее созданного файла);

2) добавление данных в существующий файл.

Для *открытия* файла в режиме создания используется оператор

ofstream out_file("имя_файла.расширение");

В круглых скобках оператора *ofstream* можно указать второй параметр из списка:

ios::out — открыть файл для вывода;

ios::trunc — перезаписать содержимое существующего файла;

ios::noreplace — если файл существует, не открывать файл;

ios::nocreate — если файл существует, не создавать файл;

ios::ate — расположить файловый указатель в конце файла.

Например, оператор *ofstream out_file("file1.txt",ios::out|ios::noreplace);*

открывает файл для вывода с именем *file1.txt* без перезаписи существующего файла в текущем каталоге на диске.

Вывод в файл осуществляется с помощью оператора вставки (*«*).

Для закрытия файла используется функция *close().*

Для *открытия* файла в режиме добавления используется оператор

ofstream out_file ("имя_файла. расширение", ios::app);

Параметр *ios::app* предписывает открыть файл в режиме добавления, располагая файловый указатель в конце файла. Например,

```
#include<fstream.h>
void main(void)
{ofstream out_file("file1.txt", ios::app);
out_file <<"Режим добавления. "<<endl;
out_file.close (); }
```

В результате выполнения этой программы в текущем каталоге на диске будет открыт файл *file1.txt*, в который будет добавлена строка: *"Режим добавления."* Таким образом, в файле *file1.txt* окажутся три строки:

Запись в файл.
Режим создания.
Режим добавления.

Ввод (чтение) данных можно выполнять не только с клавиатуры, но и из файла. Для открытия файла в *режиме ввода* используется оператор

ifstream in_file("имя_файла.расширение");

В операторе *ifstream* можно указать второй параметр:

ios::in - открыть файл для ввода.

Ввод из файла осуществляется с помощью оператора извлечения (*»*). Например,

```
#include<iostream.h>
#include<fstream.h>
void main(void)
{
char a[64], b[64], c[64];
ifstream in_file ("file1.txt");
in_file>>a>>b>>c;
cout<<"первая строка файла file1.txt: "<<a<<" "<<b<<" "<<c<<endl;
in_file.close(); }
```

Допустим, что в текущем каталоге на диске заранее создан файл *file1.txt*, в котором записаны три строки:

Запись в файл.
Режим создания.
Режим добавления.

Тогда в результате решения программы на экран дисплея будет выведено:

Первая строка файла file1.txt: Запись в файл.

Сначала будет прочитана и выведена на экран дисплея первая строка из файла *file1.txt*. Пробелы между словами *"Запись в файл."* будут восприняты компилятором как символы конца строки, а переменные *a, b* и *c* получат значения: *"Запись", "в"* и *"файл."* соответственно.

Для того чтобы прочитать из файла filel.txt все три строки, можно воспользоваться функцией *getline*. Например,

```
#include<iostream.h>
#include<fstream.h>
void   main(void)
char   a[64], b[64], c[64] ;
ifstream   in_file("file1.txt") ;
        in_file.getline(a,sizeof(a));
        in_file.getline(b,sizeof(b));
        in_file.getline(c,sizeof(c));
cout<<a<<endl;
cout<<b<<endl;
cout<<c<<endl;
in_file.close();}
```

В результате работы этой программы на экран будет выведено:

Запись в файл.
Режим создания.
Режим добавления.

В том случае, когда неизвестно, сколько строк содержится в файле, можно обеспечить чтение информации до тех пор, пока не встретится конец файла. Достижение конца файла проверяется с помощью функции *eof***,** которая возвращает значение 0 (ложь), пока не встретится конец файла, и значение 1 (истина), если встретился конец файла. Рассмотрим пример чтения содержимого файла *file1.txt* до тех пор, пока не встретится конец файла.

```
#include<iostream.h>
#include<fstream.h>
void main (void) {
char  a[64];
ifstream in__file ("file1. txt") ;
while (! in file.eof())
{       in_file.getline(a,sizeof(a));
        cout<<a<<endl; }
in_file.close(); }
```

Для того чтобы считать содержимое файла *file1.txt* по одному слову до тех пор, пока не встретится конец файла, в тело цикла *while* следует вместо оператора *in_file.getline(a,sizeof(a));* вставить оператор *in_file>>a;*

Чтобы выполнить чтение содержимого файла *file1.txt* по одному символу до тех пор, пока не встретится конец файла, при объявлении переменной вместо *char a[64];* в приведенном выше примере следует использовать *char a;*

Файловые операции можно эффективно использовать при работе с массивами и структурами. Для записи в файл и чтения из файла не символьных строк, а сложных типов данных таких, как массивы и структуры, используются функции *write* (запись) и *read* (чтение).

В приводимых ниже примерах функции write и read в качестве одного из параметров используют оператор приведения типов (char *), представляющий собой указатель на символьную строку [6-8].

Пример 11.1. Запись массива в файл.

```
#include<fstream.h>
void main (void)
{ int i, x[3]={10,20,30};
ofstream out_file("file2.dat") ;
for (i=0;i<3;i++)
out_file.write ((char *)&x[i], sizeof(x[i]));
out_file.close () ;}
```

Пример 11.2. Чтение массива из файла и вывод на экран.

```
#include<iostream.h>
#include<fstream.h>
void  main (void)
{ int   i, x[3];
ifstream   in_file("file2.dat") ;
for (i=0;i<3;i++)
{ in_file.read((char *)&x[i], sizeof(x[i]));
cout<<x[i]<<"   ";}
in_file.close();}
```

В результате последовательного выполнения этих двух программ на экран дисплея будет выведено: 10 20 30

Пример 11.3. Запись структуры в файл.

```
#include<fstream.h>
void main (void)
{ struct  date{int day; char month[9]; int year;}d={25,"ноября",1998};
ofstream   out_file("file3.dat") ;
out_file.write((char *) &d,sizeof(date));
out_file.close (); }
```

Пример 11.4. Чтение структуры из файла и вывод на экран.

```
#include<iostream.h>
#include<fstream.h>
void main (void)
{ struct date { int  day; char month[9]; int year; }d;
ifstream   in_file("file3.dat");
in_file.read((char *) &d,sizeof(date));
```

```
cout<<d.day<<"  "<<d.month<<"  "<<d.year<<endl;
in_file.close () ; }
```

В результате последовательного выполнения этих двух программ на экран дисплея будет выведено: 25 ноября 1998.

Лекция №12. Функции и их параметры. Рекурсия

Цель – получить представление об особенностях разработки и использования пользовательских функций.

Кроме стандартных функций, расположенных в заголовочных файлах, которые подключаются к программе с помощью оператора *#include<имя.h>*, языки C/C++ позволяют пользователю формировать свои собственные функции. Эти функции целесообразно создавать, если при решении задач возникает необходимость проводить вычисления по одним и тем же алгоритмам многократно. Применение функций позволяет разделить программу на простые, легко контролируемые части, а также скрыть несущественные для других частей программы детали ее реализации. Таким образом, *функция* - это логически завершенный и оформленный в соответствии с требованиями языка фрагмент программы.

Функции размещаются в программе в любом порядке и считаются глобальными для всей программы. При использовании функций необходимо различать *описание* функции и *оператор вызова* функции. Структура функции похожа на структуру программы *main*. Описание функции содержит заголовок функции, объявления переменных и операторы:

```
тип_функции  имя_функции (список  формальных_параметров)
{ объявления переменных;
оператор1;
...;
операторN; }
```

Здесь: *тип_функции* - тип возвращаемого в основную программу результата; *имя_функции* - уникальное имя, соответствующее по смыслу операции, которую выполняет функция (например, *max* - определение максимального из двух чисел); *список_формальных_параметров* - перечень формальных параметров и их типов.

Формальные параметры - это наименования переменных, через которые передается информация из основной программы в функцию.

Для вызова функции достаточно указать ее имя со списком фактических параметров в любом выражении вызывающей программы:

```
имя_функции (список_фактических_параметров);
```

Здесь: *список_фактических_параметров* - перечень фактических параметров. *Фактические параметры* - это наименования переменных, значения которых при обращении к функции, присваиваются соответствующим формальным параметрам.

Для возвращения вычисленного значения в основную программу в функциях используется оператор *return(результат);*

Рассмотрим примеры использования функций в программах.

Пример 12.1 - Функция show не содержит параметров и не возвращает значение в основную программу.

```
#include <iostream.h>
#include <conio.h>
void show (void)
{cout<<"Функция show"<<endl; }
void main (void)
{ clrscr();
cout<<"ВЫЗОВ функции show"<<endl;
show();
cout<<"Возврат в основную программу"<<endl;
getch();}
```

Пример 12.2 - Функция max содержит формальные параметры и не возвращает значение в основную программу.

```
#include <iostream.h>
#include <conio.h>
void  max (float x, float y)                    //x, y - формальные параметры
{if    (x>y) cout<<x<<">"<<y<<endl;
else  cout<<y<<">"<<x<<endl;}
void  main (void)
{ clrscr();
float  a, b;
cout<<”введите a и b “<<endl;  cin>>a>>b;
max (a,b);
getch();}
```

Пример 12.3 - Функция max содержит формальные параметры и возвращает значение в основную программу. Возвращаемое значение имеет тип float.

```
#include <iostream.h>
#include <conio.h>
float  max (float x, float y)
{ float  result;
if  (x>y)  result=x; else  result=y;
return (result);}
void  main (void)
{clrscr();
float  a, b;
cout<<”введите a и b “<<endl;  cin>>a>>b;
```

cout<<"Наибольшее из двух чисел "<<a<<" и "<<b<<":" <<max (a, b) <<endl;

cout<<"Наибольшее из двух чисел 2.71 и 3.14:" <<max(2.71,3.14)<<endl;

getch();}

В приведенных примерах описание функции предшествовало обращению к ней. В противном случае в начало программы нужно поместить *прототип* функции – т.е. объект, который содержит информацию об имени функции, типе возвращаемого значения, количестве и типе формальных параметров, например, *float max (float, float);*

Значения формальных параметров функции могут использоваться *по умолчанию*. Например,

```
#include <iostream.h>
void show (int a=l, int b=2, int c=3)
{ cout<<a<<" "<<b<<" "<<c<<endl;}
void main (void)
show () ;
show (4) ;
show (5,6);
```

В заголовке функции *show* происходит присваивание значений формальным параметрам *a, b* и *c*. При первом обращении к функции фактические параметры отсутствуют, при втором - указывается значение для параметра *a*, при третьем - значения для параметров *a* и *b*. Из примера видно, если опущено значение одного параметра, то опускаются значения и всех последующих параметров. В результате работы программы на экран будет выведено: *123 423 563.* Таким образом, если при вызове функции отсутствуют значения каких-то фактических параметров, то функция использует вместо них значения формальных параметров, назначенных пользователем по умолчанию.

С целью улучшения наглядности и удобочитаемости пользовательских программ используется *перегрузка* функций, которая позволяет использовать одно и то же имя для нескольких функций, решающих одну и ту же задачу, но имеющих разное количество и разные типы формальных параметров. Для того чтобы осуществить перегрузку функций, в программе просто описываются функции с одинаковыми именами. При решении программы компилятор самостоятельно, без каких-либо дополнительных указаний определяет нужную функцию на основании списка фактических параметров в операторе вызова функции. Рассмотрим пример использования перегрузки функций в программе.

```
#include <iostream.h>
int sum(int a, int b)
{ return(a+b);}
float sum (float a, int b, int c)
```

{return (a+b+c);}
void main (void)
{cout<<sum(2, 3) <<endl;
cout<<sum (2.5,3,4) <<endl;}

В примере определены две функции sum с разным количеством и типами формальных параметров. В результате работы программы сначала осуществляется обращение к первой функции *sum*, которая складывает два значения (2+3=5) типа *int*. Затем происходит обращение ко второй функции *sum*, которая складывает три значения (2.5+3+4=9.5) и возвращает в основную программу результат типа *float*.

Одной из особенностей использования функций в программах является объявление локальных и глобальных переменных. *Локальные переменные* объявляются внутри функции точно так же, как и внутри главной функции *main*. Локальные переменные действуют *только внутри* функции, в которой они объявлены. *Глобальные переменные* объявляются в начале программы вне какой-либо функции. Глобальные переменные доступны для любой функции в программе.

Если локальная и глобальная переменные имеют одинаковые имена, то переменная в функции воспринимается компилятором C/C++ как локальная переменная. Если внутри функции нужно использовать глобальную переменную, совпадающую по имени с локальной переменной, то в этом случае нужно воспользоваться *глобальным оператором разрешения*:

:: имя_переменной

Например,

#include <iostream.h>
int xg=l; //объявление глобальной переменной
void show (int xg)
{ cout<<"Локальная переменная xg: "<<xg<<endl;
cout<<Глобальная переменная xg: "<< : :xg<<endl;}
void main (void)
{ int y=0;
show(y); }

В результате работы программы на экран дисплея будет выведено:

Локальная переменная xg:0
Глобальная переменная xg:1

Поскольку значения глобальных переменных могут быть легко изменены любой функцией, их использование в программах не рекомендуется.

При решении вычислительных задач часто используются *рекурсивные* функции (функции, вызывающие сами себя). Рекурсивные функции, которые прямо вызывают сами себя, называются *включительно рекурсивными*. Если две функции вызывают друг друга, они называются *взаимно рекурсивными*.

Решение рекурсивной задачи обычно разбивается на несколько этапов. Одним из этапов является решение базовой задачи, т.е. простейшей задачи,

для которой написана вызываемая функция. Если задача оказывается более сложной, чем базовая, она делится на две подзадачи: выделение базовой задачи и всего остального. При этом часть, не являющейся базовой задачей должна быть проще, чем исходная задача, иначе рекурсия не сможет завершиться. Процесс продолжается до тех пор, пока не сведется к решению базовой задачи [6-8].

Каждый вызов рекурсивной функции называется *рекурсивным вызовом* или *шагом рекурсии*. Целесообразно использовать рекурсию для вычисления факториала, возведения в степень, вычисления числа Фибоначчи и др. В качестве примера рассмотрим рекурсивное вычисление факториала. По определению факториала: *n!=n*(n-1)*(n-2)*…*2*1*, причем *0!=1, 1!=1.*

Рекурсивное решение заключается в многократном вызове функции вычисления факториала из самой функции. Например, при вычислении *5!* вычисление *1!* – базовая задача, а также признак завершения рекурсии. Таким образом,

1!=1 – *возвращает 1*
*2!=2*1!= 2*1=2* – *возвращает 2*
*3!=3*2!=3*2=6* – *возвращает 6*
*4!=4*3!=4*6=24* – *возвращает 24*
*5!=5*4!=5*24=120* – *возвращает 120*

Лекция №13. Указатели и ссылки

Цель – получить представление об особенностях переменных типа указатель и ссылка, а также их использовании.

Чрезвычайно мощным инструментом в программировании считаются *указатели*. С их помощью можно упростить и повысить эффективность работы программ. Например, с помощью указателей можно изменять значения переменных внутри функции, при этом переменные могут передаваться в функцию в качестве параметров. Кроме того, указатели можно использовать для динамического выделения памяти, а это означает, что можно писать программы, которые могут обрабатывать практически неограниченные объемы данных. Указатели похожи на метки, которые ссылаются на места в памяти.

Представим сейф в банке с депозитными ячейками различного размера. Каждая ячейка имеет уникальный номер, который связан только с этой ячейкой, чтобы можно было быстро идентифицировать нужную ячейку. Эти номера аналогичны адресам ячеек компьютерной памяти. Допустим, клиент хранит все свои ценности в сейфе. При этом, чтобы обезопасить все свои сбережения, он решил завести меньший сейф, в который будет лежать карта, показывающая местоположение большого сейфа, где хранятся реальные драгоценности и его пароль. В результате, один сейф с картой будет хранить расположение другого сейфа. Эта организация сбережения драгоценностей эквивалентна понятию указателя в языках C/C++.

Указатели - это переменные, которые хранят адреса других переменных в памяти. Зная адрес переменной, можно перейти по этому адресу и получить данные, хранящиеся в нем (рисунок Д.1). Если требуется передать большое количество данных в функцию, намного проще передать адрес в памяти, по которому хранятся эти данные, чем скопировать каждый элемент.

Принцип объявления указателей такой же, как и принцип объявления переменных. Визуально отличие заключается только в том, что перед именем ставится символ звёздочки *****: *тип_данных *имя_переменной-указателя;*

При объявлении указателей компилятор выделяет несколько байт памяти, в зависимости от типа данных отводимых для хранения некоторой информации в памяти. Кроме того, если в одной строке объявляется несколько указателей, каждый из них должен предваряться символом звездочки *****.

Прежде чем использовать указатель, его необходимо *инициализировать* для того, чтобы он ссылался на определенный адрес памяти. В противном случае, указатель будет ссылаться на какой угодно участок памяти, что может привести к крайне неприятным последствиям. Более того, операционная система просто помешает программе получить доступ к неизвестному участку памяти, так как знает, что в программе не выполняется инициализация указателя, в результате это приведет к краху программы.

Например, чтобы поместить в указатель *ptrvar* адрес переменной *var* требуется выполнить инициализацию:

```
int *ptrvar;        // объявление указателя
ptrvar = &var;      // инициализация указателя
```

Чтобы получить значение, записанное в некоторой области, на которое ссылается указатель, следует воспользоваться *операцией разыменования* указателя ***** или *косвенной адресации*. С этой целью необходимо поставить звёздочку перед именем и получить доступ к значению указателя, например,

**ptrvar.*

Указатели можно сравнивать, поскольку адреса могут быть меньше или больше относительно друг друга, а при использовании операции разыменования можно сравнивать и значения хранящихся переменных. Кроме того, над указателями можно выполнять арифметические операции, например, сложения, вычитания, инкремента и декремента.

Разыменованные указатели на переменные объявленных типов позволяют производить над ними те же, что и над переменными соответствующих типов.

Унарный оператор взятия адреса & возвращает адрес объекта в памяти, поэтому его можно использовать для вычисления адресов переменных и функций, но нельзя – для вычисления выражений и символических констант, объявленных с помощью *#define*, так как эти объекты не имеют адреса.

Поскольку указатели ограничены заданным типом данных, при компиляции производится проверка соответствия типов. Например,

```
float *dPtr; char b;
```

dPtr = &b; // вызовет ошибку компиляции

В C++ существуют *нулевые* указатели – указатели, которые в данный момент не адресуют никакого допустимого значения в памяти. Значение нулевого указателя равно нулю. Это – единственный адрес, к которому указатель не имеет доступа. Поэтому в программах целесообразно использовать проверки вида: *if (dPtr != NULL) оператор;*

В этом случае указанный в выражении оператор будет выполняться лишь при *dPtr*, который адресует достоверные данные. Присваивание *dPtr=NULL* или *dPtr=0* является инициализацией и может защитить от ошибок.

Кроме нулевых, в языке C++ можно встретить указатели типа *void* – *обобщенные* указатели для адресации данных, без необходимости заранее определять их тип: *void *SomethingPtr;*

При этом указатель на *void* может быть нулевым. Их основное применение – адресация буферов, заполнение блоков памяти, чтение аппаратных регистров.

Указатели могут ссылаться на другие указатели. При этом в ячейках памяти, на которые будут ссылаться первые указатели, будут содержаться не значения, а адреса вторых указателей. Число символов ***** при объявлении указателя показывает его порядок. Чтобы получить доступ к значению, на которое ссылается указатель, его необходимо разыменовывать соответствующее количество раз. Например,

int var = 123; // инициализация переменной var числом 123
*int *ptrvar = &var; // указатель на переменную var*
*int **ptr_ptrvar = &ptrvar; // указатель на указатель на переменную var*

Таким образом, логика n-кратного разыменования заключается в том, что программа последовательно перебирает адреса всех указателей вплоть до переменной, в которой содержится значение.

Ссылки – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Ссылка может быть объявлена как другим именем, так и как псевдоним переменной, на которую ссылается. Объявление ссылки имеет следующий вид:

*/*тип*/ &/*имя_ссылки*/ = /*имя_переменной*/;*

При объявлении ссылки перед её именем ставится символ &, сама же ссылка должна быть инициализирована именем переменной, на которую она ссылается. Тип данных, на который указывает ссылка, может быть любым, но должен совпадать с объектом, на который ссылается, - с типом данных *ссылочной переменной*. Любое изменение значения, содержащегося в ссылке, повлечёт за собой изменение этого значения в переменной, на которую она ссылается. Например,

int value = 15;
int &reference = value; // объявление и инициализация ссылки

Основное назначение указателя – это организация *динамических объектов*, размер которых может меняться (увеличиваться или уменьшаться). Тогда как ссылки предназначены для организации *прямого доступа* к тому

или иному объекту. Главное отличие состоит во внутреннем механизме работы. Указатели ссылаются на участок в памяти, используя его адрес. А ссылки ссылаются на объект по его имени (тоже своего рода адрес).

Ссылки, как правило, в большинстве случаев используют в функциях как *ссылки-параметры* или *ссылки-аргументы*. Когда происходит *передача по значению*, то передаваемые данные нужно сначала скопировать, а при большом объёме данных на передачу затрачивается большое количество времени и ресурсов. В этом случае необходимо использовать передачу по ссылке**,** а данные копировать нет необходимости, так как к ним обеспечен прямой доступ. Однако это нарушает безопасность данных, хранимых в ссылочных переменных, так как к ним открывается прямой доступ [1, 5, 7].

Лекция №14. Динамическое распределение памяти. Использование указателей при решении задач

Цель – ознакомиться с функциями выделения и освобождения динамической памяти и их использованием, получить представление о возможностях использования указателей при работе со сложными типами данных (массивами, структурами, файлами) и функциями.

При работе с программой возникает необходимость в распределении доступной памяти между объектами программы. При этом для глобальных и локальных переменных, которые называются *статическими* и объявляются непосредственно в тексте программы, память резервируется на этапе компиляции на все время выполнения программы.

Для объектов, размер которых не представляется возможным определить заранее (например, изображений, файлов, структур), используется *динамическое распределение памяти*. *Динамическая память* представляет собой совокупность различных видов памяти, которая *выделяется* и *освобождается* по указанию программиста во время выполнения программы, в любом месте, в соответствии с алгоритмом решения задачи. Для доступа к динамической памяти используют указатели. При динамическом распределении памяти объекты размещаются в «*куче*» (*heap*) – специально организованной области памяти переменного размера. Причем, при конструировании объекта указывается размер запрашиваемой под объект памяти, и, в случае успеха, *выделенная* область памяти, условно говоря, «изымается» из «кучи», становясь недоступной при последующих операциях выделения памяти. Динамические переменные также называются *адресуемыми указателями.* По мере создания в программе новых объектов, количество доступной памяти уменьшается. Отсюда вытекает необходимость постоянно *освобождать* ранее выделенную память. В идеальной ситуации программа должна полностью освободить всю память, которая потребовалась для работы.

Некорректное распределение памяти приводит к ее «*утечкам*», т.е. ситуациям, когда выделенная память не освобождается. Многократные утечки памяти могут привести к исчерпанию всей оперативной памяти и нарушить работу операционной системы. Поэтому операция, противоположная по смыслу операции выделения памяти под объект, - это *освобождение* занятой ранее под какой-либо объект памяти: освобождаемая память, условно говоря, возвращается в «кучу» и становится доступной при дальнейших операциях выделения памяти.

Поскольку при динамическом выделении памяти память резервируется не на этапе компиляции, а на этапе выполнения программы, это дает возможность выделять память более эффективно, в основном это касается массивов. При динамическом выделении памяти нет необходимости заранее задавать размер массива, тем более что не всегда известно, какой размер должен быть у формируемого массива.

Для выделения и освобождения динамической памяти в языках C/C++ используются специальные функции.

Функция *malloc() – memory allocation* - определена в заголовочном файле *alloc.h* и используется для инициализации указателей необходимым объемом памяти. Память выделяется из сектора оперативной памяти доступного для любых программ, выполняемых на данном компьютере. Аргументом функции *malloc()* является количество байт памяти, которую необходимо выделить, возвращает функция - указатель на выделенный блок в памяти. Поскольку различные типы данных имеют разные требования к памяти, объем адресуемой памяти вычисляется с помощью операции *sizeof()*. Например, при выделении указателю *ptrVar* необходимого объема памяти для хранения значения типа *int* можно использовать оператор

*int *ptrVar = malloc(sizeof(int));*

Размер выделяемой памяти можно определять, передавая пустой указатель. Например, *int *ptrVar = malloc (sizeof (*ptrVar));*

Операция *sizeof(*ptrVar)* оценит размер участка памяти, на который ссылается указатель, а так как *ptrVar* является указателем на участок памяти типа *int*, то *sizeof()* вернет размер целого числа.

Оператор резервирования может отказать в случае, если куча уже заполнена или оставшаяся память меньше, чем требуемое количество байтов. В этой ситуации функция *malloc()* возвращает нулевое значение.

Автоматически выделенный участок памяти становится недоступным для других программ. Следовательно, после того, как выделенная память станет ненужной, её нужно явно высвободить. Высвобождение памяти выполняется с помощью функции *free()* из той же библиотеки. Например, *free(ptrVar);*

Задавать размер высвобождаемой памяти не нужно, т.к. он запоминается в нескольких байтах, примыкающих к каждому зарезервированному блоку, и совпадает с объемом первоначально зарезервированной памяти, с которой связан указатель.

После освобождения памяти хорошей практикой является сброс указателя в нуль, то есть присвоить **ptrVar = 0;*. Если указателю присвоить *0*, то он становится нулевым, другими словами, он уже никуда не указывает. В противном случае, даже после высвобождения памяти, указатель все равно будет указывать на неё, а, значит, можно случайно нанести вред другим программам, которые, возможно будут использовать эту память.

Для резервирования памяти вместо функции *malloc()* можно использовать функцию *calloc()*, прототип которой также объявлен в заголовочном файле *alloc.h.* Эта функция тоже резервирует память в куче, но требует два аргумента: количество объектов, которое необходимо разместить, и размер одного объекта. Например, чтобы зарезервировать память для 10 значений типа *long* и присвоить указателю *sPtr* адрес первого значения, можно использовать оператор *long *sPtr = calloc(10, sizeof(long));*

Параметры функции *calloc()* могут задаваться в любом порядке, т.е. *calloc(num,size)* и *calloc(size,num)* резервируют память объемом *num*size* байтов. Кроме выделения памяти, эта функция устанавливает каждый зарезервированный ею байт равным нулю, т.е. инициализирует память.

В языке C++ также имеются операторы для выделения и освобождения динамической памяти, которых не было в C:

- *new* - автоматически создает объект соответствующего размера в памяти и возвращает его адрес; если выделение памяти не произошло, возвращается нулевой указатель. Например, *int *iPtr = new int;*

Кроме того, при резервировании памяти посредством оператора *new* допускается присвоение значения созданному объекту сразу:

*float *fPtr = new float (17.245);*

- *delete* - освобождение зарезервированной ранее памяти, при этом размер освобождаемой памяти задавать не требуется: *delete iPtr;*

При работе с операторами *new* и *delete* подключать заголовочный файл *alloc.h* не нужно.

Следует помнить, попытка освободить одну и ту же память более одного раза, приводит к системной ошибке. Частично или полностью эти проблемы в C++ решаются созданием классов из библиотек *STL* и *Boost*, реализующих «*умные указатели*».

Указатели могут адресовать переменные всех видов – от простых целочисленных переменных типа *int* до сложных типов таких, как массивы и структуры. Кроме того, они не заменимы при работе с функциями.

В языках C/C++ при работе с массивами очень удобно использовать указатели, поскольку с их помощью можно организовать экономный и быстрый доступ к любому элементу массива. На рисунке Д.2 показана связь между элементами массива и указателями. Здесь указатель *mPtr* позволяет обратиться к элементу *m[0]*, *mPtr+1* указывает на следующий элемент, а *mPtr+i* на *i-элемент* массива M.

Использование указателей при работе с массивами позволяет экономить память. Если размер массива точно не определен или требуется использовать

только часть элементов уже существующего массива, удобно воспользоваться динамической памятью с применением указателей. С этой целью следует:

1) объявить указатель требуемого типа данных;

2) затем в программе вызвать функцию *malloc()* для выделения памяти в куче для массива или *calloc()* для инициализации значений массива нулями;

3) работать с массивом как обычно, а после его использования освободить память с помощью функции *free().*

Например, для выделения памяти под массив из 100 элементов типа *double* следует выполнить действия:

```
double *BigArrayPtr;
...
BigArrayPtr = malloc(100*sizeof(double));
```

или `BigArrayPtr = calloc(100, sizeof(double)); // инициализация нулевыми значениями`

```
...
for ( int i = 0; i < 100; i++ )
      BigArrayPtr [i] = random (50);
...
free (BigArrayPtr);
```

Для резервирования памяти под массив и ее последующего освобождения можно применять операторы *new* и *delete*:

```
double *BigArrayPtr = new double(100);
...
delete [ ] BigArrayPtr;
```

Следует обратить внимание, что при удалении временного массива оператором *delete* необходимо указать квадратные скобки, поскольку инициализировать динамический массив невозможно и результат может оказаться непредсказуемым.

При работе с двумерными массивами можно представлять их в памяти как одномерные, а для выделения памяти воспользоваться следующей формулой: $i\_текущее = Nx * iY + iX,$

где *i_текущее* - текущее значение индекса в одномерном массиве;

Nx – количество столбцов в исходной матрице;

iY, iX – индексы текущего элемента в исходной матрице.

Строковые указатели являются адресами, которые определяют местонахождение первого символа строки, сохраненной в памяти. Строковые указатели объявляются как *char** или, чтобы операторы не могли изменить адресуемые им данные, как *const char**. Поскольку строки представляют собой символьные массивы, поэтому над ними допустимо выполнение тех же действий, что и над массивами. При выделении строке памяти в куче можно использовать функции *malloc()* и *free().*

Для экономного размещения в памяти больших структур также целесообразно использовать кучу, а не глобальные или локальные

переменные. Для этого нужно объявить структуру, а затем указатель на структурный тип:

*struct имя_структуры *имя_указателя;*

Например,

```
struct date
{ int   day; char   month[9]; int year; }d;
...
struct date *dat_Ptr;
```

Затем с помощью функции *malloc()* выделить память в куче и присвоить адрес указателю:

указатель = (struct тип_структуры) malloc (sizeof (struct тип_структуры));*

Например, *dat_Ptr = (struct data*) malloc (sizeof (struct data));*

Размер адресуемого указателем блока памяти соответствует размеру одной структуры указанного типа (в примере, типа *data*). Однако использовать привычный способ доступа к полям структуры нельзя, поэтому при организации доступа в программе необходимо вместо обращения

тип_структуры . имя_элемента = значение;

следует применить обращение

тип_структуры -> имя_элемента = значение;

Оператор доступа к полю структуры **->** указывает на его расположение в памяти относительно начала структуры.

При обычном вызове функции компилятор C/C++ создает копию значений фактических параметров и помещает эту копию в стек - участок памяти компьютера для временного хранения информации. Значения параметров из стека присваиваются формальным параметрам функции. После завершения работы функции происходит очистка стека. Таким образом, значения фактических параметров в основной памяти компьютера после завершения работы функции остаются без изменений.

В случае, когда нужно изменить значения фактических параметров, т.е. осуществить присваивание значений формальных параметров при выходе из функции фактическим параметрам основной программы, следует использовать оператор адреса (признак &) и переменные-указатели (признак *).

Оператор адреса предназначен для передачи адресов значений фактических параметров из основной программы в функцию. Оператор адреса имеет вид: *имя_функции (&фактич_параметр1, ... , &фактич_параметр N);*

Чтобы «объяснить» компилятору C++, что значения фактических параметров будут передаваться из основной программы в функцию с помощью адреса (в стек будут записываться адреса фактических параметров, а не их значения), в заголовке функции объявляются переменные-указатели (формальные параметры) с признаком *****:

*тип_функции имя_функции (тип *форм_пар1, ... , тип *форм_парN)*

Переменные (формальные параметры), которые используются в операторах функции, должны быть также помечены звездочкой (*****).

```
#include <iostream.h>
```

```
void change (int *x, int *y)
{ *x=100; *y=200; }
void main (void)
{ int a,b; a=10; b=20;
cout<<"Параметры a и b до обращения к функции:"<<a<<" "<<b<<endl;
change(&a,&b);
cout<<"Параметры b после обращения к функции:"<<a<<" "<<b<<endl;}
```

В результате после завершения работы функции фактические параметры изменили свои значения, а на экран будет выведено:

Параметры a и b до обращения к функции: 10 20

Параметры a и b после обращения к функции: 100 200

Значения фактических параметров могут быть изменены в программе и при помощи ссылок, которые позволяют создавать *псевдонимы* для переменных, используемых в качестве параметров функции. Для объявления ссылки применяется знак амперсанда (&) после типа параметра:

тип& имя_ссылки=имя_переменной;

После объявления ссылки в программе можно использовать имя переменной или имя ссылки. В качестве иллюстрации изменения значений фактических параметров при помощи ссылок рассмотрим пример:

```
#include <iostream.h>
void change (int& x, int& y)
{ x=100; y=200; }
void main (void)
{ int a,b;
int& as=a;          //объявление ссылки as — псевдонима переменной a
int& bs=b;          //объявление ссылки bs — псевдонима переменной b
a=10; b=20;
cout<<"параметры a и b до обращения к функции:"<<a<<" "<<b<<endl;
change(as,bs);
cout<<"параметры a и b после обращения к функции:"<<a<<" "<<b<<endl;}
```

В результате работы программы на экран будет выведено:

Параметры a и b до обращения к функции: 10 20

Параметры a и b после обращения к функции: 100 200

Таким образом, результат работы программы полностью совпадает с результатом работы программы, рассмотренной в предыдущем примере. Однако не следует увлекаться частым использованием ссылок, так как это затрудняет понимание программы [1, 5, 7].

Указатели также могут ссылаться на функции. Имя функции, как и имя массива, само по себе является указателем, то есть содержит адрес входа:

```
/*тип данных*/ (* /*имя указателя*/) (/*список аргументов функции*/);
```

Тип данных определяется в соответствии с типом, возвращаемым функцией, на которую будет ссылаться указатель. Символ указателя и его имя берутся в круглые скобки, чтобы показать, что это - указатель, а не функция, возвращающая указатель на определённый тип данных. После имени указателя в круглых скобках через запятую перечисляются все аргументы функции, как в объявлении прототипа функции.

И, наконец, указатель на файл (*file pointer*) – это указатель на информацию, которая определяет его характеристики (имя, статус, текущую позицию), а именно: на структуру типа *FILE*, которая определена в заголовочном файле *stdio.h*. Для объявления указателя на файл используется оператор *FILE *fPtr;*

Лекция №15. Использование графических возможностей языка

Цель - изучить возможности и особенности использования графики.

Пакет функций управления экраном делится на две части в соответствии с возможностями компьютера: работа в текстовом режиме (*text mode*) и работа в графическом режиме (*graphics mode*). Все функции управления экраном в текстовом режиме имеют свои прототипы в заголовочном файле *conio.h.* Управление экраном в графическом режиме производится с помощью набора функций, прототипы которых находятся в заголовочном файле *graphics.h.*

Среди функций настройки графического режима следует выделить следующие:

- *void far detectgraph(int far *gdriver, int far *gmode);* - предназначена для определения типа графического адаптера. Эта функция возвращает значения по адресам, указанным первым и вторым параметрами. Здесь *gdriver* - указатель на целое число, содержащее номер графического драйвера. Вторым параметром функции *detectgraph* будет указатель на целое число, содержащее номер графического режима, обеспечивающего максимальную разрешающую способность экрана. Если нет графического адаптера, то функция *detectgraph* присваивает **gdriver = -2*.

- *void far initgraph(int far *gdriver, int far *gmode, char far *pathdriver);* - предназначена для загрузки графического драйвера, имеющего номер **gdriver*, и для установки графического режима номер **gmode*. Если по адресу *gdriver* записан нуль *(*gdriver=DETECT*), то функция вначале обращается к функции *detectgraph*, а затем загружает драйвер, номер которого был установлен функцией *detectgraph*. Необходимая память для загрузки драйвера предоставляется в "куче". В зависимости от графического режима максимальная разрешающая способность экрана изменяется.

Для выгрузки графического драйвера применяется функция, объявленная как *void closegraph(void)*. Эта функция освобождает память из "кучи", занятую графическим драйвером.

Инициализация графического режима производится следующим образом:

```
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
void main()
{ int driver, mode;driver=registerbgidriver(EGAVGA_driver);
driver = VGA; mode = VGAHI;
initgraph (&driver, &mode, "");
closegraph();}
```

После установки графического режима с помощью *initgraph* экран монитора представляет собой прямоугольную область, разбитую на одинаковые прямоугольники - *пиксели*, стороны которых параллельны верхней и нижней границам экрана. *Пиксель* (*pixels*) - это минимальный элемент изображения на экране, состоящий из нескольких (цветных) точек и рассматриваемый в программе как одна точка определенной яркости или цвета. Под *координатами* пикселя подразумеваются целочисленные координаты центров этих прямоугольников, отсчитываемые от координат центра левого верхнего прямоугольника. Чтобы вычислить разрешающую способность экрана по *x* и *y*, применяются функции *int getmaxx(void), int getmaxy(void).* Наиболее часто используется графический режим монитора, при котором поддерживается разрешение 640*480*16. Здесь 16 – это максимальное количество цветов, которое одновременно может присутствовать на изображении.

В отличие от математической системы координат, графический экран выглядит так, как показано на рисунке 15.1.

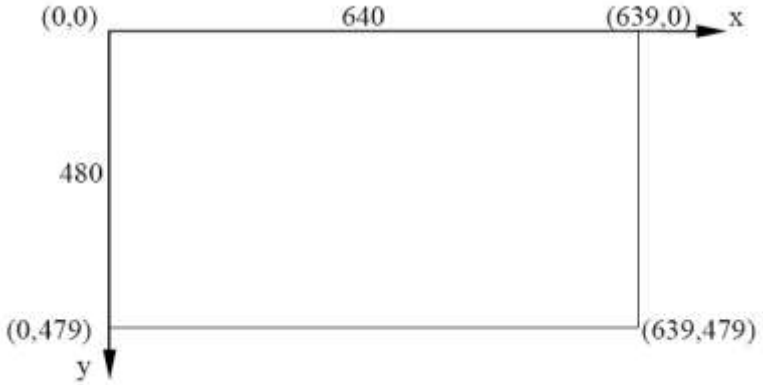


Рисунок 15.1 – Внешний вид графического экрана

В заголовочном файле *graphics.h* определены константы, соответствующие цветам стандартной палитры. Изменение одного из цветов стандартной палитры производится функцией

void setpalette (int index, int color);

где *intindex* – номер из стандартной палитры;

int color – цвет в диапазоне от 0 до 63 (палитра EGA).

Настройка палитры EGA осуществляется функцией

void setrgbpalette (int color, int red, int green, int blue);

где *red*, *green* и *blue* изменяются в диапазоне от 0 до 255, причем малым значениям соответствуют темные цвета, большим – более яркие.

Графические функции, с которыми можно работать после установки графического режима, условно делятся на три группы:

1) состоит из функций, которые ничего не выводят на экран, но устанавливают некоторые параметры: например, функция *setcolor* задает номер цвета для дальнейшего вывода линий;

2) состоит из функций, которые осуществляют вывод на экран: например, чтобы вывести точку заданного цвета, применяется функция *void putpixel(int x, int y, int color);*

3) состоит из функций, которые ничего не выводят на экран, но позволяют получить информацию о выведенном изображении: например, для чтения цвета пикселя предназначена функция *unsigned getpixel(int x, int y).*

Имена функций первой группы начинаются со слова *set* (ставить, помещать), а функции третьей группы – со слова *get* (получать, доставать).

Графические функции производят вывод в страницу, которая называется *активной*.

Коэффициентом сжатия экрана называется отношение ширины пикселя к его высоте. Коэффициент сжатия можно узнать с помощью функции *void getaspectratio(int far *xasp, int far *yasp);*

Функция записывает по адресу *yasp* число 10000, а по адресу *xasp* - произведение коэффициента сжатия на 10000. Коэффициент сжатия учитывается при выводе окружностей, дуг окружностей и секторов круга. Коэффициент сжатия устанавливается при инициализации графического режима, исходя из режима, соответствующего максимальной разрешающей способности экрана, и может быть изменен с помощью функции *void setaspectratio(int xasp, int yasp);*.

Группа линий на плоскости образует *контурную* фигуру (отрезок прямой линии, дугу, окружность, прямоугольник, эллипс и т. д.). Кроме формы, фигуры могут отличаться цветом линии (контура), ее толщиной или типом. По умолчанию, в графическом режиме существуют следующие настройки: текущий цвет контура – *WHITE* (белый), толщина – один пиксель, тип – сплошная линия. *Плоскостные* фигуры - это фрагменты плоскости экрана, ограниченные замкнутым контуром. Их можно получить из контурных путем закрашивания области внутри или вне замкнутой сплошной линии, образующей контур. Для отображения наиболее часто используемых фигур, изменения типов линий контура и их параметров можно воспользоваться функциями стандартной графической библиотеки *graphics*.

Вывод текста в графическом режиме можно осуществить с использованием прототипов функций той же библиотеки. Текстовая информация отображается на экране с учетом параметров: цвет, тип шрифта, размер шрифта и направление. Размер символов (по вертикали и горизонтали) определяется как произведение стандартного размера (8×8 пикселей) на

параметр charsize, то есть если значение charsize будет равно 3, то каждый символ, отображающийся на экране, будет вписан в квадрат 24×24 пикселя. Параметр font, задающий стиль шрифта, подключает к программе файлы с расширением *.chr (нестандартные «шрифты»), поэтому необходимо сделать эти файлы доступными (проще всего скопировать их в текущую директорию).

Для вывода текста на экран в графическом режиме можно использовать и функции для текстового режима, однако они имеют ряд недостатков. Например, при использовании функции printf() для вывода текста на каком-либо цветном фоне позади надписи появится ее «фон» (черный прямоугольник, равный длине выводимого текста). Также отсутствует возможность изменения внешнего вида выводимого текста (размера шрифта, стиля и т.д.) [5-8].

Приложение А

Основные этапы развития технологий программирования

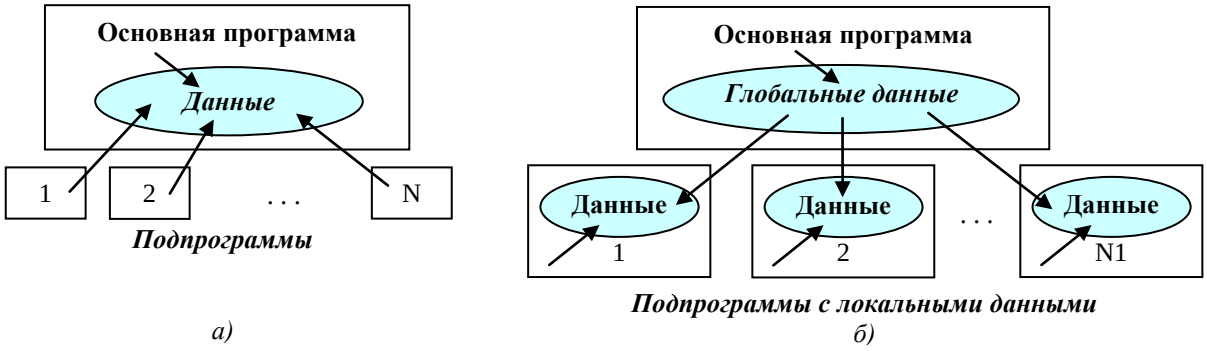


Рисунок А.1 - Архитектура программ с глобальной и локальной областями данных

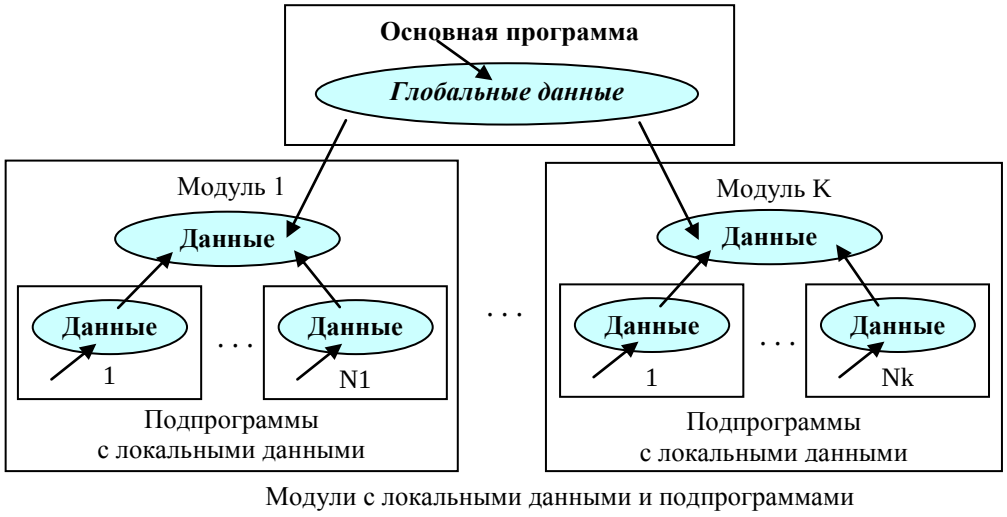


Рисунок А.2 – Архитектура программы при структурном подходе. Модульное программирование

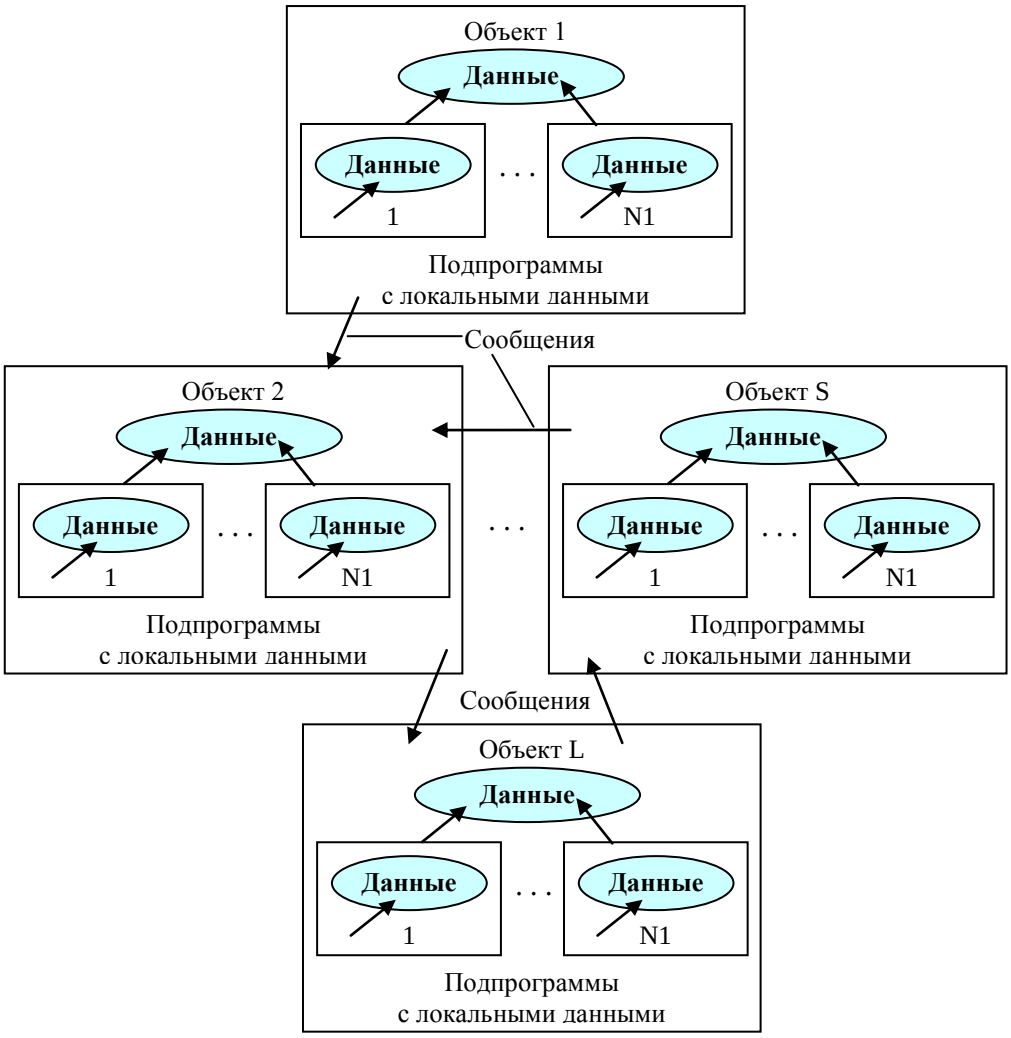


Рисунок А.3 – Объектный подход. Архитектура программы при объектно-ориентированном программировании

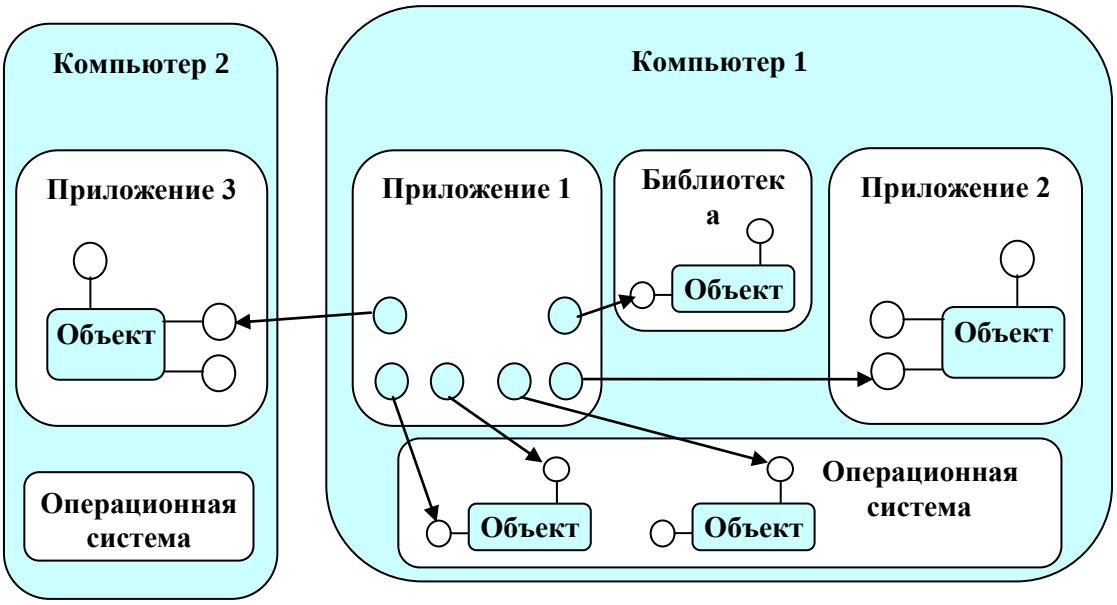


Рисунок А.4 – Компонентный подход. Технология COM

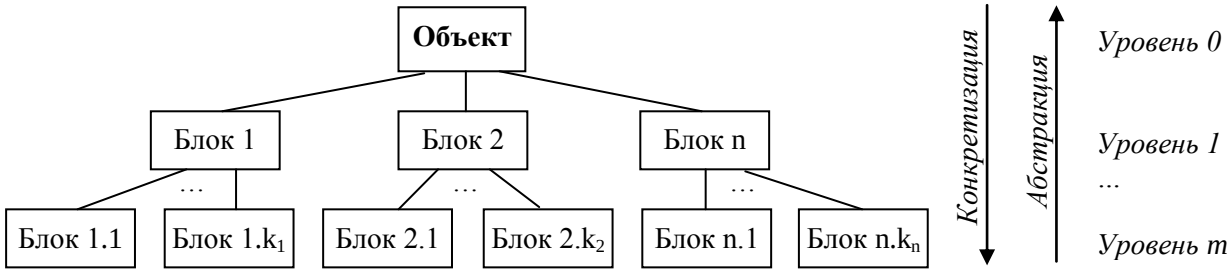


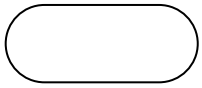
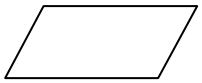
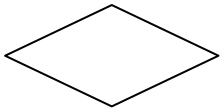
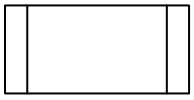
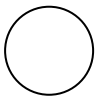
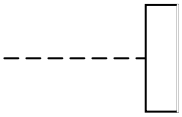
Рисунок А.5 – Соотношение абстрактного и конкретного в описании блоков при блочно-иерархическом подходе

Рисунок А.6 – Структура процессов жизненного цикла программного обеспечения

Приложение Б

Элементы блок-схем

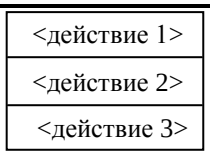
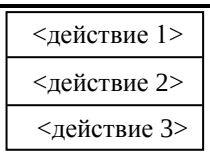
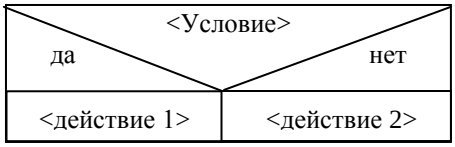
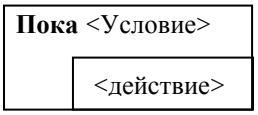
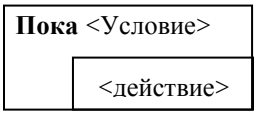
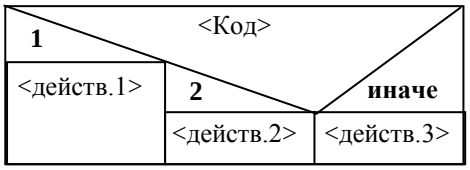
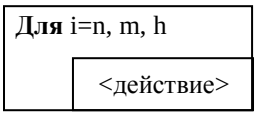
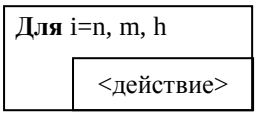
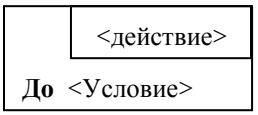
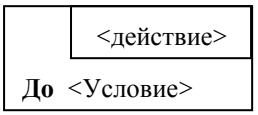
Таблица Б.1 - Основные элементы блок-схем

Наименование	Обозначение	Примечание
Терминатор		Отображает выход во внешнюю среду или вход из внешней среды. Используется для обозначения начала или окончания алгоритма
Данные		Отображает данные, носитель данных не определен. Используется для обозначения операций ввода и вывода данных
Процесс		Отображает функцию обработки данных любого вида. Используется для обозначения операций присваивания
Подготовка		Используется для обозначения заголовка цикла
Решение		Используется для обозначения оператора условного перехода или оператора выбора (варианта)
Предопределен-ный процесс		Отображает предопределенный процесс, состоящий из одной или нескольких операций, которые определены, например, в подпрограмме или модуле. Обычно используется для обозначения подпрограмм.
Соединитель		Отображает выход в часть схемы и вход из другой части этой же схемы и используется для обрыва линии и продолжения ее в другом месте.
Линия		Отображает поток данных или управления. Направления слева направо и снизу вверх обозначаются стрелками. Используется для соединения символов в алгоритме
Комментарий		Используется для добавления описательных комментариев или пояснительных записей с целью объяснений или примечаний.
Примечание – символы могут быть вычерчены в любой ориентации, но предпочтительной является горизонтальная. Внутрь символа помещают обозначения или описания операций. Символы ГСА могут быть отмечены идентификаторами или порядковыми номерами.		

Приложение В

Структурное и «неструктурное» программирование

Таблица В.1 – Соответствие различных способов описания алгоритмов

Струк-тура	**Псевдокоды**	**Flow-формы**	**Диаграммы Насси-Шнейдермана**
Следо-вание	<действие 1> <действие 2>		
Ветвле-ние	**Если** <условие> **то** <действие 1> **иначе** <действие 2> **Все-если**		
Цикл-пока	**Цикл-пока** <условие> <действие> **Все-цикл**		
Выбор	**Выбор** <код> <код 1>: <действие 1> <код 2>: <действие 2> **иначе** <действие 3> **Все-выбор**		
Цикл с парамет-ром	**Для** <индекс> = <n>,<m>,<h> <действие > **Все-цикл**		
Цикл-до	**Выполнять** <действие> **До** <условие>		

Приложение Г

Таблица Г.1 - Специальные и управляющие символы

	Вид	Название	Вид	Название
Специальные символы	+	Плюс	|	Черта
	++	Приращение	||	Логическое ИЛИ
	-	Минус	!	Восклицательный знак
	--	Уменьшение	->	Стрелка
	*	Звездочка	=	Операция присваивания
	/	Наклонная черта	= =	Знак равенства
	\	Обратный слеш	!=	Не равно
	//	Двойной слеш	>	Больше
	.	Точка	<	Меньше
	,	Запятая	<=	Меньше или равно
	:	Двоеточие	<<	Сдвиг влево
	::	Разрешение	>>	Сдвиг вправо
	;	Точка с запятой	< >	Угловые скобки
	‘	Апостроф	( )	Круглые скобки
	“	Кавычки	[ ]	Квадратные скобки
	^	«Крышка»	{ }	Фигурные скобки
	%	Знак процента	/* */	Скобки комментария
	&	Амперсанд	#	Знак номера
	&&	Логическое И	~	Тильда
Управляющие символы	\a	Сигнал динамика	\t	Горизонтальная табуляция
	\b	BS, забой символа	\v	Вертикальная табуляция
	\f	Новая страница	\\	Обратная косая черта
	\n	Новая строка	\0	Нулевой символ (байт)
	\r	Возврат каретки	\000	Восьмеричная константа
	\”	Двойная кавычка	\xhhh	Шестнадцатеричная константа
	\’	Апостроф	\?	Знак вопроса

Таблица Г.2 - Зарезервированные слова в C++

and	char	false	int	private	switch	virtual
and_eq	class	float	long	protected	template	void
asm	compl	for	mutable	public	this	volatile
auto	const	else	namespace	register	throw	while
bitand	continue	enum	new	return	true	xor
bitor	default	explicit	not	short	try	xor_eq
bool	delete	friend	not_eq	signed	typedef	
break	do	goto	operator	sizeof	typename	
case	double	if	or	static	union	
catch	extern	inline	or_eq	struct	unsigned	

Таблица Г.3 – Типы данных с разными комбинациями модификаторов

Тип	Диапазон изменения		Размер в байтах (битах)
	от	до	
void	-	-	0
char (signed char)	-128	127	1 (8)
unsigned char	0	255	1 (8)
wchar_t	0	65535	2 (16)
bool	True (Истина)	False (Ложь)	1 (8)
int (signed int, short int, signed short int)	-32768	32767	2 (16)
unsigned int (unsigned short int)	0	65535	2 (16)
long int (signed long int)	-2147483648	2147483647	4 (32)
unsigned long int	0	4294967295	4 (32)
float	3.4E-38	3.4E+38	4 (32)
double	1.7E-308	1.7E+308	8 (64)
long double	3.4E-4932	3.4E+4932	10 (80)
Примечание – Размер в байтах и диапазон изменения могут варьироваться в зависимости от компилятора, процессора и операционной системы (среды).			

Таблица Г.4 – Перечень операций, их приоритет и порядок выполнения

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

Примечания

1 Наивысший приоритет имеют операторы 1 уровня, низший – 16 уровня.

2 Знак ⇒ обозначает выполнение операций слева направо, а знак ⇐ - выполнение операций справа налево.

3 Унарные операторы (+) и (-), находящиеся на уровне 2, обладают более высоким приоритетом, чем арифметические (+) и (-) с уровня 5. Символ & на уровне 2 - оператор обращения по адресу, а символ & на уровне 9 битовый оператор AND. Символ * на уровне 2 - оператор обращения к указателю, а символ * на уровне 4 – оператор умножения.

4 В отсутствие круглых скобок операторы, находящиеся на одном уровне, обрабатываются согласно их расположению слева направо.

Таблица Г.5 – Основные математические функции

Наименование функции	Функция	Тип		Заголовочный файл
		результата	аргумента	
Абсолютное значение	abs(x) cabs(x) fabs(x)	int double float	int double float	<stdlib.h> <math.h> <math.h>
Арккосинус	acos(x)	double	double	<math.h>
Арксинус	asin(x)	double	double	<math.h>
Арктангенс	atan(x)	double	double	<math.h>
Косинус	cos(x)	double	double	<math.h>
Синус	sin(x)	double	double	<math.h>
Экспонента e^x	exp(x)	double	double	<math.h>
Степенная функция x^y	pow(x,y)	double	double	<math.h>
Логарифм натуральный	log(x)	double	double	<math.h>
Логарифм десятичный	log10(x)	double	double	<math.h>
Корень квадратный	sqrt(x)	double	double	<math.h>
Тангенс	tan(x)	double	double	<math.h>

Таблица Г.6 – Символы преобразования в функциях ввода-вывода

Формат вывода	Значение	Формат ввода	Значение
%c	вывод символа (char)	%c	чтение символа (char)
%d	целое десятичное число (int)	%d	целое десятичное число (int)
%i	целое десятичное число (int)	%i	целое десятичное число (int)
%e (%E)	число (float/double) в виде x.xx e+xx (x.xx E+xx)	%e	чтение числа типа float/double
%f (%F)	число float/double с фиксированной запятой xx.xxxx	%h	чтение числа типа short int
%g (%G)	число в виде f (F) или e(E) в зависимости от значения	%o	чтение восьмеричного числа
%s	строка символов	%s	чтение строки символов
%o	целое число (int) в восьмеричном представлении	%x	чтение шестнадцатеричного числа (int)
%u	беззнаковое десятичное число (unsigned int)	%p	чтение указателя
%x (%X)	целое число (int) в шестнад-цатеричном представлении	%n	чтение указателя в увеличенном формате
%p (%n)	указатель		

Примечание – К форматам можно применять модификаторы **l** и **h**, например, %ld (long в десятичном виде), %lo (long в восьмеричном виде), %lu (unsigned long), %lx (long в шестнадцатеричном виде), %lf (long float с фиксированной точкой), %le (long float в экспоненциальной форме), %lg (long float в виде f или e в зависимости от значения).

Таблица Г.7 – Характерные приемы программирования

Прием программирования	Действия, выполняемые до цикла	Действия, выполняемые в цикле
Накапливание *суммы*	S = 0;	S+=*элемент_массива*;
Накапливание *произведения*	P = 1;	P*=*элемент_массива*;
Накапливание *количества*	k = 0;	k++;
Поиск *максимального* значения	max=*предполаг_знач*;	if (*текущ_элемент*>max) max=*текущ_элемент*;
Поиск *минимального* значения	min=*предполаг_знач*;	if (*текущ_элемент* <min) min=*текущ_элемент*;

Приложение Д

Указатели и ссылки

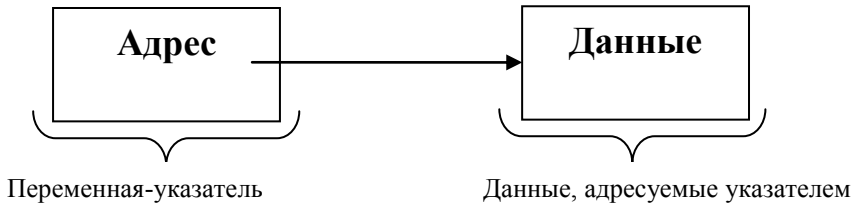


Рисунок Д.1 – Графическая интерпретация указателя

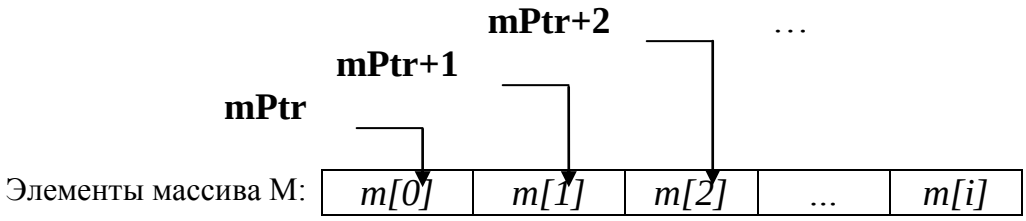


Рисунок Д.2 – Элементы массива и указатели

Список литературы

1 Иванова Г.С. Технология программирования.- М.: «Кнорус», 2013.

2 Страуструп Б. Язык программирования С++. – М., 2012.

3 Потопахин В. Искусство алгоритмизации. - М.: «ДМК Пресс», 2011.

4 Сэджвик Р. Алгоритмы на С++. – М.: «Вильямс», 2011.

5 Павловская Т.А. С/С++. Структурное программирование. – СПб., 2010.

6 Немцова Т.И. Программирование на языке высокого уровня. Программирование на языке С++. - М.: «Форум», 2012.

7 Ашарина И.В. Основы программирования на языках С и С++. - М.: Горячая линия-Телеком, 2012.

8 Аляев Ю.А., Козлов О.А. Алгоритмизация и языки программирования Pascal, C++, Visual Basic: Учебно-справочное пособие. – М.: Финансы и статистика, 2004.

9 Терехов А.Н. Технология программирования. – М.: БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий. - Intuit.ru, 2006.

10 Давыдов В.Г. Технологии программирования С++. – СПб., 2005.

11 Соколов А.П. Системы программирования: теория, методы, алгоритмы: Учебное пособие. - М.: Финансы и статистика, 2004.

Сводный план 2013 г., поз. 256

Наталья Валерьевна Сябина
Лариса Николаевна Рудакова

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Конспект лекций
для студентов специальности 5В070200 - Автоматизация и управление

Редактор Л.Т. Сластихина
Специалист по стандартизации Н.К. Молдабекова

Подписано в печать __. __. __.
Тираж *100* экз.
Объем *4.5* уч.-изд. л.

Формат 60х84 1/16
Бумага типографская №1
Заказ _____. Цена 2250 тг.

Копировально-множительное бюро
некоммерческого акционерного общества
«Алматинский университет энергетики и связи»
050013, Алматы, ул. Байтурсынова, 126