ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФИЗИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
www.phys-math.ru

ДЕКАДА НАУКИ 2006

МАТЕРИАЛЫ СЕКЦИИ-СЕМИНАРА

Языки программирования и технологии «Оберон»: перспективы для индустрии и образования

(сокращённая версия, исправлены ссылки, 14.05.2013)

30 марта 2006 г.

ООО «ОРЛОВСКИЙ ЦЕНТР ПРОГРАММНОЙ ИНЖЕНЕРИИ

«МЕТАСИСТЕМЫ»
www.metasystems.ru,
blackbox.metasystems.ru,
sources.metasystems.ru

СОДЕРЖАНИЕ СБОРНИКА

Программная инженерия вчера и сегодня: эволюция императивного программирования

Ермаков И.Е.

> Но существует одно качество, которое нельзя купить, — это надежность. Цена надежности — погоня за крайней простотой. Это цена, которую очень богатому труднее всего заплатить.
>
> Ч-А. Хоар

I. Обзор. Цель данной статьи − краткий обзор состояния, проблем и перспектив современной программной инженерии (ПИ). Под ПИ будем понимать создание технических систем программного обеспечения (ПО). На сегодняшний день в ПИ существует несколько фундаментальных, ортогональных друг другу методологий программирования (МТП). В основе МТП лежит какая-либо модель алгоритма, абстрактная машина. Основные МТП: *императивная* (машина Тьюринга), *функциональная* (лямбда-исчисление), *логическая* (логика предикатов)[1]. Выбор МТП определяет стиль мышления, инструментарий и набор доступных разработчику приемов. Выбор диктуется особенностями задачи, но в целом можно говорить как о достаточной универсальности всех основных МТП, так и об их взаимопроникновении.

Очевидно, что наиболее распространена в индустрии императивная МТП. Однако последние 15 лет острие лабораторных исследований направлено именно на альтернативные МТП – основные надежды (часто чрезмерные) возлагаются на декларативные подходы, в частности, 4GL[2], и может создаться впечатление, что в области императивных 3GL ничего нового не происходит. На самом деле это не так, что мы и постараемся показать далее. Таким образом, мы будем рассматривать современную ПИ с позиций императивной методологии.

II. Эволюция императивного программирования (ИПР). Императивный алгоритм представляется в виде последовательности команд для некоторого исполнителя, которые переводят данный исполнитель из одного состояния в другое. Выполнение алгоритма − это прохождение последовательности состояний, в процессе которого решается задача − вычисление, обработка данных, управление некоторым процессом[3].

Обычный императивный исполнитель имеет ограниченный набор простых команд и работает с неструктурированными двоичными данными. Преимущество языков программирования высокого уровня (ЯВУ) в том, что они позволяют при написании алгоритма оперировать более высокоуровневыми понятиями (*абстракциями*), нежели машинные. В идеале хотелось бы сократить до минимума разрыв между понятиями предметной области (ПрО) и абстракциями, которые предоставляет ЯВУ. Эволюция ИПР – это в первую очередь развитие абстракций ЯВУ.

В этом развитии особую роль сыграли три ключевых нововведения: зарождение ЯВУ (50-е гг.), структурный подход (70-е гг.) и объектный подход (80-е гг.). После появления каждого из них эффективность ПИ резко возрастала, поскольку в распоряжении разработчиков оказывались некоторые новые абстракции для конструирования программного кода. В первых ЯВУ этими абстракциями стали: **оператор** − минимальная логическая *единица выполнения* (абстрагирует нас от нескольких физических единиц выполнения), **переменная** − *именованная единица данных* (абстрагирует от физического размещения и представления в памяти), **подпрограмма** – *составная единица выполнения* (позволяет явно выделить логически цельную последовательность операторов). Видим, что уже присутствует составная единица выполнения, однако отсутствуют составные единицы данных − насущной потребности в них не было, так как на заре ИТ преобладали вычислительные задачи. Нет никаких абстракций для *управления выполнением*: арифметический IF[4], GOTO и CALL первых ЯВУ эквивалентны соответствующим машинным командам. Построение кода на прямых передачах управления получило название «стиль спагетти» − он порождал огромные трудности при написании больших программ, зачастую делал невозможным их сопровождение и развитие. Низкая производительность программистов и крайне низкая надежность ПО привели к **первому кризису программирования** (60-е

---

[1] Есть и некоторые другие, например, автоматная. См. [3].

[2] 4th Generation Programming Languages – «языки 4-го поколения», позволяют пользователю описывать (в т.ч. графически) требуемый результат без программирования алгоритма для его достижения.
3GL – «языки 3-го поколения» - классические универсальные языки программирования.

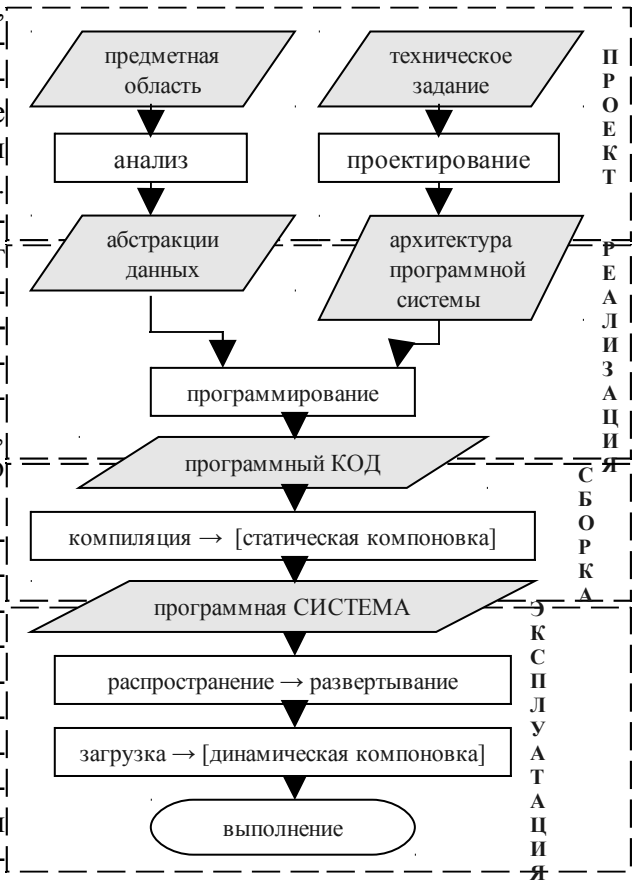
[3] Актуальные сегодня параллельные вычисления предполагают взаимодействие многих исполнителей, каждый из которых может проходить свой граф состояний одновременно с другими.

[4] Арифметический IF передавал управление по одной из 3-х возможных веток в зависимости от значения числового параметра: $x < 0$, $x=0$, $x>0$.

гг.). Очередным прорывом явились идеи *структурного программирования* (СП), провозглашенные Э. Дейкстрой, Н. Виртом и Ч. Хоаром (70-е гг.)[1].

Основные принципы СП: **декомпозиция** – разбиение решаемой задачи на набор подзадач, их *независимая* реализация и затем организация взаимодействия; **блочность** – организация кода в виде отдельных блоков для каждой подзадачи, каждый блок – это подпрограмма (**процедура**) с одним входом и одним выходом, минимальное использование глобальных переменных − сокрытие данных внутри процедур; **структурная организация кода** − отказ от GOTO, использование для управления выполнением исключительно *составных операторов*. СП вводит абстракцию для управления выполнением – **составной оператор**. Вводятся три составных оператора: *последовательный*, *выбора*, *повторения*. Любой алгоритм может быть записан в терминах этих и только этих операторов. В этом случае любой блок кода имеет ровно один вход и один выход. Можно определить *предусловия*/*постусловия*, истинность которых ожидается на входе/выходе, и *инварианты* – условия, которые истинны на протяжении выполнения данного блока. Это дает возможность математически доказывать правильность алгоритмов. Такое доказательство трудоемко и не всегда оправдано, но его облегченный вариант постоянно выполняется опытным программистом в уме, и это позволяет строить действительно надежные программы.

Еще одним китом СП становятся **структуры данных**. Как мы уже отмечали выше, ранее в ЯВУ отсутствовали составные единицы данных, однако к концу 60-х специфика разнообразных прикладных задач потребовала их введения. Структурные языки позволяют конструировать составные **типы данных** из элементарных и таким образом непосредственно выражать в коде понятия из ПрО. Программная система в СП представляется как набор процедур, обрабатывающих и передающих друг другу **потоки структурированных данных**. Одним из важнейших принципов СП является **сильная типизация** – отсутствие любых неявных преобразований типов, контроль за корректностью использования переменных как на этапе компиляции, так и на этапе выполнения – без этого невозможно гарантировать надежность ПО[2].



III. Модульное ПО. В конструировании любой техники важнейшим является принцип модульности: система разрабатывается и собирается из набора независимых модулей. Модули взаимодействуют друг с другом только через небольшое количество «каналов» (в ПИ набор таких «каналов» называется *интерфейсом* модуля) по строго определенным правилам. Это позволяет нескольким группам конструкторов работать над разными модулями одновременно. Более того, в дальнейшем это облегчает эксплуатацию и обслуживание техники, так как модули разных производителей, даже различающиеся по своему внутреннему исполнению, будут взаимозаменяемы (самый яркий пример – архитектура IBM PC). Другая сторона модульности – это **инкапсуляция**, представление модулей «черными ящиками», обладающими опре-

[1] Эти идеи были реализованы сначала в Алголе (1968 г.), а затем в завоевавшем всемирное признание Паскале Н.Вирта (1970 г.).Появившийся в то же время «высокоуровневый ассемблер» С в основных аспектах также ориентирован на структурный стиль программирования.

[2] Н. Вирт: «Исключительного использование сильно типизированного языка является фактором, в наибольшей степени определяющим возможность проектирования сложной системы в короткий срок». Идеально сбалансированный структурный язык – Модула-2 (Вирт), наследница Паскаля (активно используется в российских военно-космических проектах). Богатейший набор типов предоставляет язык Ада (стандарт военной отрасли США). Оба языка сильно типизированы. Линейка С-образных языков придерживается обратного подхода – «все, что не запрещено, то разрешено», что приводит к низкому качеству и большим затратам на отладку – тестирование – сопровождение программных систем.

деленным интерфейсом и поведением, но полностью скрывающими свое внутреннее устройство от доступа извне.

Этапы жизненного цикла ПО изобразим следующим образом (см. рис.)[1]. На этапе анализа выделяются сущности ПрО, которые формулируются в типах данных (ТД) ЯВУ (если информация о внутреннем устройстве ТД скрывается внутри модуля, то он называется *абстрактным* ТД). Процессы обработки данных ПрО формулируются в терминах процедур ЯВУ. На этапе проектирования принимается решение об архитектуре, выделяется набор модулей, описываются их интерфейсы и протоколы взаимодействия. На основе полученных **абстракций** и **архитектуры** выполняется непосредственно программирование.

Неоспоримым преимуществом является возможность проектировать и реализовывать систему в терминах одного и того же языка. Современный ЯВУ должен позволять непосредственно выражать архитектурные решения. Именно этот критерий в настоящее время является главным, т.к. в плане «программирования в малом» все современные императивные ЯВУ приблизительно одинаковы. Эталоном модульного языка стала Модула-2 (1980). В ней модуль является единицей ПО во всех отношениях – единицей написания кода, инкапсуляции, компиляции, импорта другими модулями. В языке Ада модули носят название пакетов. В С-семействе (включая C#) в принципе нет единой абстракции модуля[2]. В совокупности с трудночитаемым синтаксисом это затрудняет проектирование в терминах ЯВУ. Вводятся дополнительные нотации проектирования, например, Буча или UML, однако переход от одного представления к другому отнимает время и вносит ошибки[3]; контроль соответствия кода архитектуре нельзя переложить на компилятор.

В то же время удачность единой абстракции модуля позволила Вирту в языке Оберон-1 (1989) и одноименной ОС сделать прорыв в новое измерение – к **компонентному программированию** (КП). Модуль становится единицей не только разработки, но и распространения, развертывания и загрузки в память, двоичным кирпичиком как ПО, так и ОС в целом. Фаза статической компоновки модулей была исключена, Модули динамически загружаются, связываются, выгружаются, заменяются в любой момент прямо во время работы ПО. Механизм **метапрограммирования** позволяет выполнять динамический анализ модулей и работать с типами, переменными, процедурами, неизвестными на этапе компиляции. Для КП обязательна строгая типизация, динамический контроль типов и границ массивов, сборщик мусора – в противном случае организовать надежное взаимодействие многих модулей от разных производителей просто невозможно[4]. На основе идей Оберона была создана Java (1994). Сегодня мы имеем три промышленных компонентных языка: Java, Компонентный Паскаль (Оберон-2) и C#.

IV. Модульность + ООП: Оберон-парадигма. Суть объектно-ориентированного (ОО) подхода – в рассмотрении ПрО как множества взаимодействующих **объектов**. Объект обладает состоянием (полями) и поведением (методами). **Класс** − множество объектов со сходной структурой и поведением. Чистые ОО-языки (Smalltalk) не предоставляют никаких абстракций кроме классов и объектов («все есть объект»). Однако многие механизмы ПО лучше выражаются в процедурном, нежели в ОО-стиле. Общепринятый подход − надстройка структурных ЯВУ абстракциями класса, объекта и метода. Но (по мнению Вирта) в ООП нет почти ничего нового по сравнению со СП, кроме названий, вносящих путаницу: класс = тип данных, объект = запись, метод = процедура. Модула-2 позволяет писать программы в «почти ОО» стиле. Единственно новое в ООП – это наследование, которое Вирт называет **расширением типа**. В Обероне-1 ООП реализовано на основе расширяемых записей, полиморфизм процедур – на основе динамической проверки типа (x IS SomeType), виртуальные методы – на основе полей процедурного типа (это позволяет каждому экземпляру типа иметь особое поведение). В Обероне-2 добавлены **связанные процедуры**, которые позволяют оформлять код в привычном стиле ООП (object.-

[1] Отметим, что схема отражает техническую сторону процесса ПИ, но не организационную – организационно выделенные этапы обычно проходят не последовательно, а одновременно и/или циклически.

[2] В C++ пространства имен являются всего лишь логической единицей, единицей написания – текстовые файлы, единицей компиляции и импорта – объектные файлы (сборки в .NET), единицей инкапсуляции – классы. Это порождает путаницу и не позволяет разделить систему на полностью непересекающиеся части.

[3] Б. Мейер: «От модели UML до реального программирования дистанция огромного размера». Н. Вирт: «Это даже шаг назад - на заре программирования было принято записывать программы в виде графов. Такая схема приводит к слишком большому количеству ошибок, как только программа усложняется».

[4] Новейший процессор Эльбрус e2k (разработан в России), благодаря использованию аппарата тегов, поддерживает в полной мере защищенное программирование (для модулей единого адресного пространства).

Do). Аналогично вводится ООП в Ада-95. При таком подходе идеология и синтаксис языка становятся гораздо проще, стройнее и логичней. К тому же в языках типа C++ из-за отсутствия модулей на класс возлагается двойная нагрузка – он становится и ТД, и единицей инкапсуляции, «модулем» ПО. Однако модуль и ТД – принципиально различные вещи. ТД – это абстракция ПрО, модуль – элемент архитектуры ПО. Для системного ПО типичны группы классов, которые должны взаимодействовать напрямую *в обход инкапсуляции*. Модульность дает элегантное решение проблемы − под некоторую группу тесно связанных ТД выделяется один «кукловод»-модуль, который инкапсулирует их поведение и взаимодействие, при этом внутри модуля нет сокрытия информации между типами[1]. Оберон-парадигма гармонично сочетает ООП, структурный стиль и модульность: ПО строится как набор модулей, связанных процедурными шинами, по которым передаются объектно-ориентированные данные.

V. В условиях очередного **кризиса программирования** очевидно, что основной целью является **надежность** ПО, и достичь ее можно только стремлением к предельной простоте и стройности систем. В противовес американской ПИ, попавшей в тупик избыточной сложности и неуправляемого калейдоскопа бизнес-технологий, в традициях европейской и российской школ программирования – стремление к системному мышлению и поиску простых решений. Огромным потенциалом идей для преодоления кризиса ПИ обладает швейцарская школа Н. Вирта, в частности Оберон-направление.

Литература

1. http://oberon2005.ru − Европейский центр программирования.
2. http://blackbox.metasystems.ru/ – русский портал по BlackBox и Оберонам.
3. Одинцов И.О. Профессиональное программирование. Системный подход − 2-е изд-е, − СПб.: БХВ-Петербург, 2004.
4. Szyperski C. Component Software. Beyond Object-Oriented Programming, Addison Wesley Longman, 1998.
5. Pfister C., Component Software − Русский перевод: Ермаков И.Е., http://blackbox.metasystems.ru/, раздел “Статьи”.

[1] В системном ОО-программировании много тесно связанных групп классов, для которых приходится обходить инкапсуляцию – в этом одна из причин того, для написания ОС рекомендуется чистый C, а не C++. В то же время на Оберонах в ОО-стиле реализовано много ОС, в том числе две – жесткого реального времени.

Критерии эффективности языков программирования. Сравнение Pascal- и C-семейства языков

Темиргалеев Е.Э.

Введение

В настоящее время существует огромное число языков программирования (ЯП). Мы будем рассматривать в основном императивные языки, которых сейчас большинство. Одни из них разрабатывались как универсальные, предназначенные для решения широкого круга задач; другие - как специализированный инструмент для решения задач в конкретной области.

В любом случае часто мы можем наблюдать тенденцию универсализации языка, связанную с человеческим фактором. Если язык оказался удачным, завоевал любовь пользователей, его (как любимый и привычный) стараются приспособить к решению и других задач, которым он может соответствовать не очень хорошо. Пишутся специальные библиотеки и/или вносятся модификации в язык для расширения области его применения, при этом дополнительные средства не всегда оказываются столь эффективными, как в других языках, нацеленных на решение тех же задач.

Особенно пагубно эта тенденция отражается на более универсальных языках, т. к. сфера их использования больше и шире теоретически решаемого круга задач. Зачастую пользователи считают свой любимый язык панацеей от всех бед и просто отказываются признавать право других языков на существование/применение.

Т. о. важный вопрос выбора более эффективного языка для решения широкого круга задач может оказаться непростым, т. к. выбор богат и навскидку трудно определить, какой из языков является более подходящим.

Цели данной статьи: во-первых, выделить основные параметры, по которым можно сравнить эффективность языков. Во-вторых, на основе этих критериев сравнить широко распространенные Pascal- и C-семейства языков.

Эффективность

Что понимать под эффективностью? Если говорить о каких-либо целенаправленных действиях, то можем определить ее так: более эффективными будут действия, затрачивающие меньший объем ресурсов для достижения одного и того же результата.

Программирование – человеческая деятельность, целью которой является разработка/сопровождение программы или набора программ – программного продукта (ПП), предназначенного для решения определенных задач и обладающего заданными свойствами. *Разработка* – проектирование "с нуля" и реализация. *Сопровождение* – исправление выявленных в результате эксплуатации ПП дефектов и добавление новых функциональных возможностей в ПП с сохранением его архитектуры.

Выполнение ПП – "деятельность" компьютера, направленная на решение задачи.

Главный ресурс для выполнения ПП – *машинный*: время вычислений, объемы оперативной и внешней памяти и т. п. Его затраты определяются:

- способами реализации ПП (более быстрый алгоритм и т. п.);
- надежностью (при сбоях результаты работы могут быть потеряны и ее придется делать заново).

Основным ресурсом программирования можно назвать *человеко-часы* – время, которое понадобится программистам для реализации ПП. Его затраты определяются такими параметрами, как:

- квалификация программистов;
- требования, предъявляемые к ПП (надежность, используемые машинные ресурсы);
- используемые инструменты – средства разработки (в основном – ЯП).

Если говорить о сопровождении ПП, то сюда добавятся также параметры, влияющие на машинный ресурс:

- способы реализации ПП (гибкость архитектуры ПП – насколько ПП расширяем функционально в пределах исходной архитектуры);
- надежность (меньше сбоев – меньше выявлять и исправлять ошибок; защита кода от ошибок – если произошел сбой, как трудно выявить ошибку).

Т. о., говоря об *эффективности ПП*, мы подразумеваем меньшую его требовательность к машинным ресурсам. *Эффективность программирования* – меньшая затрата времени на раз-

работку ПП и большая эффективность созданного ПП, т. к. последняя влияет на сложность сопровождения.

Критерии эффективности

Итак, предположим что нам требуется выбрать универсальный язык для решения широкого круга задач. Этот вопрос актуален, например, для фирмы, производящей программное обеспечение, или для исследовательского коллектива, т. к. использование единого языка для всех проектов позволяет объединить усилия разработчиков и добиться синергетического эффекта в работке команды.

Рассматривая конкретный язык, будем отмечать, как он влияет на остальные параметры эффективности программирования:

(A) Требуемую квалификацию программиста. Основной вопрос – подготовка (переподготовка) специалиста, если он не обладает необходимыми знаниями.

(1) Знание базовых средств языка – необходимо.

(2) Знание дополнительных средств языка. Некоторые возможности можно не использовать, поэтому их знание не обязательно. Если же они используются, то программист должен разбираться в них основательно.

(3) Специальные знания. Некоторые средства языка могут требовать знаний в других областях информационных технологий.

(B) Требуемое время:

(1) разработки ПП;

(2) сопровождения ПП.

(C) Эффективность ПП с точки зрения:

(1) меньших затрат машинного ресурса;

(2) большой надежности.

При указании влияния на параметры групп B и C будем пользоваться следующими обозначениями: +параметр – положительное влияние, -параметр – отрицательное влияние. Например, +B1, -B2. При указании только буквы подразумевается вся группа, например +C <=> +C1, +C2.

Параметры группы A будут указываться как необходимое условие. Если программист обладает необходимыми знаниями и опытом, то положительный эффект от используемого средства будет увеличиваться, а отрицательный уменьшаться. В противном случае будет требоваться время на обучение, плюс отсутствие опыта и недостаточная подготовка (которая часто имеет место) будут усиливать отрицательный эффект и ослаблять или сводить на нет положительный эффект, либо вообще трансформировать его в отрицательный.

Анализировать языки будем в следующих категориях:

(1) История создания.

1. *Для какой области задач разрабатывался язык?* Очевидно, что наиболее эффективным будет естественное использование языка.

2. *Как разрабатывался язык?* Существовал ли проект, целью которого был сам язык, или язык появился как инструмент решения какой-то другой задачи? Более вероятно, что специально спроектированный язык будет обладать лучшими качествами.

3. *Существует ли строгое формальное описание (стандарт) языка?* Наличие стандарта способствует стандартизации средств разработки, что положительно влияет на мобильность ПП. Часто ли подвергается стандарт изменениям и/или исправлениям, что сказывается на изменчивости языка, и отклонении существующих средств разработки от стандарта: этот факт отрицательно сказывается на мобильности ПП.

(2) Средства разработки. Выбор средства разработки полностью влияет на эффективность программирования.

1. *Доступность.* Как много реализаций существует, какие платформы охватывают, способ распространения, производители.

2. *Соответствие стандарту* (напрямую влияет на мобильность ПП). Большинство стандартов оставляют специфические (не поддающиеся стандарти-

зации) моменты на усмотрение реализации. Реализация таких моментов. Расширения стандарта.

(3) Средства языка. Различные средства языка (конкретизация при сравнении).

1. *Стандартные*.
2. *Специфические, расширения* (для рассмотренных в 2.2 реализаций).

Сравнение Pascal- и C-семейства языков

Среди большого числа ЯП выделяются два обширных семейства: Pascal и C. Общие черты языков этих семейств можно назвать "противоположными":

- синтаксис языка: криптованный, сокращенный – стремление уменьшить размеры лексем языка (C) и человекоподобный – стремление приблизить синтаксис к "разговорному" языку (Pascal);
- автоматический контроль корректности кода во время выполнения (проверка правильности индексов массивов, соответствия типов): отсутствует – вся ответственность за безошибочность кода перекладывается на программиста (C) и присутствует – с генерацией компилятором дополнительного кода и ограничением возможности программиста полностью определять исполняемый код (Pascal);
- стиль программирования "можно все, что не запрещено" (C) и "можно только то, что разрешено" (Pascal).

Такая противоположность постоянно вызывает "вооруженные столкновения" между фанатиками с обоих сторон, порождает горячие споры, в которых каждый превозносит свое семейство и изничтожает противоположное.

Я хотел бы постараться дать объективную характеристику положительных и отрицательных особенностей каждого семейства. В качестве представителей C-семейства (очевидно) выберем C и C++. В основном C отдельно рассматриваться не будет, так как он практически полностью является подмножеством C++. В качестве представителей Pascal-семейства рассмотрим языки Компонентный Паскаль (КП) (как последний на данный момент этап эволюции Паскаля) и Ада (как один из наиболее мощных и богатых возможностями).

Приводимая оценка базируется на личном опыте, данных из различных источников, отчасти на стандартах языка, поэтому не может рассматриваться как полностью объективная и точная. Наиболее объективно было бы провести сравнение стандартов языков, однако стандарты Ады, C и C++ очень объемные, и одному человеку провести такую работу в приемлемые сроки не представляется возможным.

1. История создания

Рассмотрим категории 1.1-1.3 для каждого языка по отдельности.

C

1. Язык C разрабатывался как универсальное средство системного программирования.

2. В 1970 г. в фирме AT&T Bell Laboratories был разработан язык B как модификация языка BCPL (Basic Combined Programming Language) для написания ранней версии операционной системы UNIX для компьютера PDP-11.

Язык C получился из B устранением недостатков, выявленных в процессе его использования (1972 г., Д. Ритчи, фирма AT&T Bell Laboratories). Основным улучшением было введение типов данных, которые в B отсутствовали.

C получил широкое распространение (в частности, как основной ЯП в операционных системах UNIX).

3. Первая редакция международного (ISO) стандарта языка C была выпущена в 1990, далее было два исправления 1994 и 1996 гг. Действующая версия стандарта: ISO/IEC 9899:1999 (объем – 538 с., платная 281$), к ней выпущено два исправления: 2001, 2004.

C++

1. C++ разрабатывался как универсальный язык программирования, поддерживающий ООП, абстракцию данных и сохраняющий возможности системного программирования – язык C как свое подмножество.

2. C++ начал развиваться с 1980 года в исследовательской группе Б. Страуструпа как "C с классами", для написания программ моделирования, управляемых прерываниями. Все работы (проектирование, реализация и документирование) выполнялись в соответствии с текущими потребностями авторов и велись вокруг двух положений:

- сохранение языка C как подмножества;
- включение в язык возможностей, обеспечивающих абстракцию данных и ООП. Основные идеи касательно ООП были заимствованы из SIMULA-67.

В июле 1983 г. C++ впервые вышел за пределы исследовательской группы и получил свое настоящее название. Число пользователей языка увеличилось, и дальнейшее развитие в основном было связано с преодолением проблем, с которыми столкнулись пользователи. В результате такого развития, язык пополнился и новыми возможностями, не связанными непосредственно с ООП. Наиболее значительным можно назвать введение шаблонов функций и классов (приблизительно 1988 г.) – механизма поддержки обобщенного программирования. Частично идеи были позаимствованы из языков Ада и CLU.

Лавинообразный рост числа пользователей и появление нескольких производителей компиляторов в итоге привели к необходимости стандартизации языка. Работы были начаты в 1987 г.

3. Первая редакция международного стандарта C++ была издана в 1998. В 2003 году – вторая, действующая на данный момент: ISO/IEC 14882:2003 (757 с., 291$). При этом следует учитывать, что текущий стандарт базируется на стандарте языка C 1990 года.

Сейчас (как и последние лет десять) бурное развитие языка идет в области шаблонов. Это отрицательно влияет на стабильность языка.

КП

1. Цель создания КП – увеличить контроль в плане безопасности за проектируемыми свойствами сложной компонентной системы. КП можно считать универсальным ЯП за исключением в общем случае низкоуровневого программирования – наличие этих средств не требуется стандартом языка.

2. КП представляет собой модификацию языка Оберон-2, сделанную (1997 г.) в Oberon microsystems, Inc.

Появился КП в итоге нескольких десятилетий исследовательской работы и эволюции нескольких языков, начиная с Паскаля и заканчивая Обероном-2. Более подробно см. [2].

3. Стандартом языка является его определение 1997 г. (32 с, бесплатно).

Ада

1. Ада – универсальный ЯП. Из руководства к языку: "Язык Ада был разработан с учетом трех взаимно перекликающихся концепций: надежность программирования и сопровождения, программирование как человеческая деятельность и эффективность".

2. В 1975 г. Министерство Обороны США разработало и распространило список требований к языку, который должен был служить средством разработки ПО для встроенных компьютерных систем (например, систем наведения ракет). Ни один из существующих языков не удовлетворял этим требованиям, и было предложено создать новый.

Этот новый язык был предметом конкурсного пересмотра в широких индустриальных и академических кругах. Из большого числа было отобрано четыре, затем из них два. В финале выбрали проект представленный компанией Cii-Honeywell Bull, разработкой данного проекта руководил Жан Ишбиа. Язык получил название Ада.

Официальная дата появления языка – 1983 г., когда был утвержден национальный стандарт США (ANSI/MIL-STD-1815A-1983).

3. Первый международный стандарт – 1987 г. В 1995 году выпущена вторая редакция стандарта, в основном связанная с поддержкой ООП, и действующая сейчас – ISO/IEC 8652:1995 (511 с., 281$), в 2001 - исправления.

Итог

	C	**C++**	**Ada**	**КП**
Универсален стандартный	да*	да	да	нет**

язык?				
* Язык предназначен для системного программирования и многие высокоуровневые возможности в нем просто отсутствуют (абстракция данных, ООП, и т. д.). Теоретически его можно использовать для решения любых задач, но практически для многих задач решения будут неоправданно сложными. ** Отсутствуют низкоуровневые средства.				
Появление языка - специальный проект?	да	нет	да	да
Стандарт:				
доступен бесплатно[1]	281$	291$+281$*	281$	да
объем (страниц)	538	757+538*	511	32
трудно изучить	трудно	очень трудно	трудно	легко
изменчивость	да	да	да	нет
* Стандарт C++ базируется на стандарте C 1990 г, однако поскольку C является подмножеством C++ (за малым числом исключений), то должна быть выполнена доработка (чтобы текущий стандарт C++ базировался на текущем стандарте C), и добавочно указаны цифры, соответственно текущему стандарту C (1999 г.).				

2. Средства разработки

Рассмотрим категории 2.1-2.2 для каждого языка по отдельности.

C

1. Существует большое число реализаций от разных производителей под большое число платформ, как коммерческие[2], так и некоммерческие. Стоит отметить компилятор GNU CC, распространяемый по лицензии GPL (бесплатно, в открытых кодах).

2. Многие реализации соответствуют стандарту (про все сказать невозможно), и часто предоставляют различные расширения языка/стандартной библиотеки.

Однако, следует отметить факт, что для таких компиляторов как GNU CC, Borland, Microsoft говорится о соответствии национальному стандарту США (ANSI), а не международному (ISO).

C++

1. Аналогично C, за исключением того, что из-за большей сложности языка число реализаций, поддерживаемых платформ меньше.

2. Поскольку выпускаться компиляторы различными производителями стали гораздо раньше (порядка 15 лет) появления стандарта, то с появлением такового здесь сложилась интересная ситуация.

Во-первых, многие реализации поддерживают стандарт не полностью (или далеко не полностью) - в основном это касается шаблонов. К тому же, некоторые производители, стремясь как можно быстрее ввести некоторое нововведение (или приблизить свою реализацию к стандарту), делают это зачастую нерациональным способом (например, модель жадного инстанцирования фирмы Borland).

Во-вторых, практически каждая реализация содержит различные дополнения, которые оказываются расширениями стандарта либо из-за того, что они в него не были включены, либо добавленные после выпуска стандарта с целью облегчить жизнь программиста (при этом никто не гарантирует, что эти изменения в последствии станут частью стандарта). Например, оператор typeof в GNU CC.

---

[1] При желании, многие ISO-стандарты можно найти и бесплатно. Однако стремление международного комитета по стандартизации действовать в роли коммерческой организации и наживаться на стандартах, не соответствует целям, ради которых он существует.

[2] Стоит отметить, что многие фирмы делают значительные скидки учебным заведениям на свои коммерческие продукты.

КП

1. На данный момент существуют две реализации КП: BlackBox Component Builder (Windows, Mac; производитель - Oberon Microsystems, Inc), Gardens Point Component Pascal (.NET, выполнена в австралийском Queens University).

Система Blackbox является профессиональным инструментом; версия для Windows распространяется бесплатно в открытых кодах по лицензии BlackBox Open Source License, совместимой с GPL.

2. BlackBox полностью поддерживает стандарт КП. Существуют различные расширения, для поддержки платформенно-специфичных средств. Например, подсистема SYSTEM (низкоуровневые операции), расширение компилятора Direct-to-COM для поддержки COM-программирования под Windows.

Ада

1. Существует немало реализаций, под широкое число платформ. Большое число производителей, распространение в основном коммерческое. Свободно доступны компиляторы ObjectAda от Aonix (правда со значительными ограничениями по сравнению с коммерческой версией) – для Windows и Solaris, и GNAT 3.15p от Ada Core Technologies (под лицензией GPL) – для большого числа платформ. К сожалению, AdaCore закончила выпуск публичных бесплатных версий на 3.15p.

2. Имя "Ada" зарегистрировано как торговая марка. Поэтому, вы не имеете права распространять компиляторы языка программирования Ада до тех пор, пока они не пройдут тестирование на совместимость (позднее эти требования были ослаблены, и теперь защищенное название звучит как "Validated Ada"). Таким образом все версии компиляторов Ады полностью соответствуют стандарту, а все расширения идут в рамках стандарта (те пункты, которые оставляются на усмотрение реализации).

Итог

<table>
<tr><th></th><th>C++</th><th>Ada</th><th>КП</th></tr>
<tr><td>Число платформ</td><td>много</td><td>много</td><td>мало</td></tr>
<tr><td>Соответствие реализаций стандарту</td><td>не полное для большинства реализаций</td><td>полное</td><td>полное ?*</td></tr>
<tr><td colspan="4">* Всего две реализации: BlackBox Component Builder (полное), Gardens Point Component Pascal (неизвестно).</td></tr>
<tr><td>Мобильность (программ соответствующих стандарту)</td><td>проблематично*</td><td>почти полная**</td><td>полная</td></tr>
<tr><td colspan="4">* Поскольку многие реализации не полностью соответствуют стандарту, то перенос с одной реализации на другую может быть проблематичен. Хороший пример (того насколько эта задача может оказаться сложной) – проект STLPort, цель которого – реализация единой стандартной библиотеки шаблонов (STL) для большого числа реализаций.
** Переносимость не будет полной, если будут использоваться платформенно-специфичные средства которые полностью не описаны в стандарте и оставлены на усмотрение реализации.</td></tr>
<tr><td>Доступность бесплатных полнофункциональных средств разработки</td><td>да</td><td>нет*</td><td>да</td></tr>
<tr><td colspan="4">* Хотя GNAT 3.15p – полнофункциональное средство, оно будет устаревать, и его использование будет все менее и менее оправданным.</td></tr>
</table>

3. Средства языка

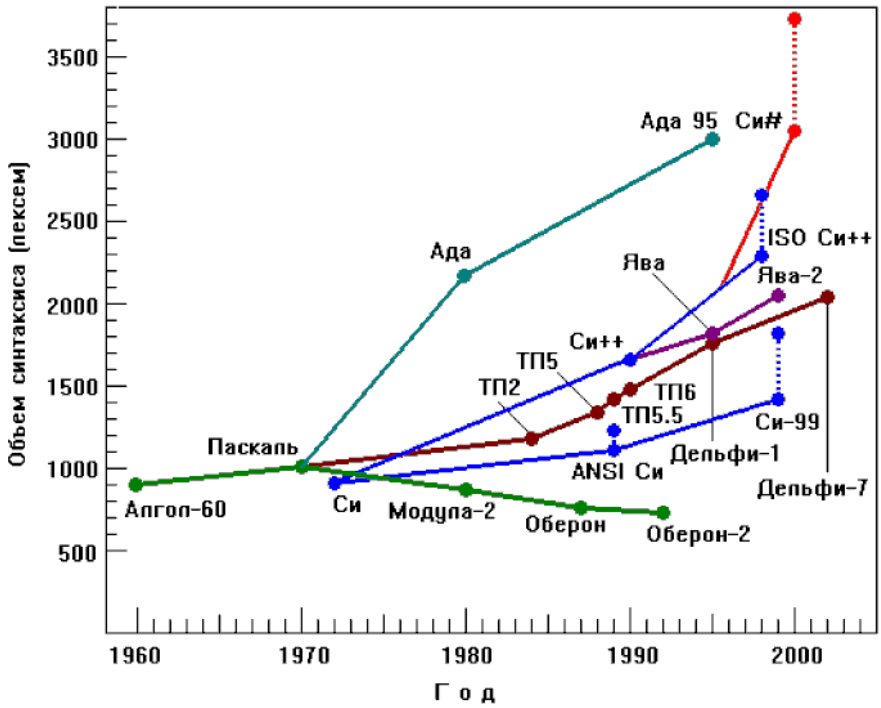
При рассмотрении средств языка, будем сравнивать специфические особенности следующих реализаций: GNU CC 2.8.1 (C++), GNAT 3.15p (Ada), BlackBox 1.5 (КП).

Богатство средств языка

Влияние: A1, A2, +B, +C.

Широкие возможности, предоставляемые языком, с одной стороны могут способствовать решению более широкого круга задач. Однако, поскольку многие задачи пересекаются, эти возможности порождают неоднозначность способов решения. Т. о. выбор оптимального будет требовать большей квалификации программиста. Начинающиеся и неопытные будут путаться в этих способах и почти всегда будут использовать не самый лучший, что негативно отразится на B, C.

Ниже приведена схема, отображающая сложность (число лексем) в основных ЯП (из доклада Свердлова С. З. на II Всероссийском совещании по ИТ-образованию, [2]). Из нее видно, что в основном развитие языков идет за счет их усложнения, с добавлением новых (возможно многочисленных) возможностей. Противоположностью являются языки Оберон-семейства, в которых наряду с включением новых средств исключаются малоиспользуемые и несущественные; пока это дает в результате более простой язык, однако нет гарантий, что эту тенденцию удастся сохранить.



Читабельность и самодокументируемость кода

Влияние: +B, +C.

Насколько понятны для программиста будут действия, описанные языком программирования (*читабельность*)? Насколько просто можно будет понять алгоритм, реализуемый фрагментом программы без дополнительной документации или комментариев (*самодокументируемость*)?

Хорошо написанный и понятный программный код может значительно сократить время сопровождения (+B2), время разработки (+B1) – особенно если речь идет о крупном проекте, в котором задействовано большое число программистов. Уменьшается вероятность ошибок, проще реализовать сложный алгоритм (+C).

Читабельность в первую очередь зависит от языка – какие средства доступны программисту, и во вторую – от того, как эти средства используются программистом. Самодокументируемость в большей степени зависит от программиста, т. к. один и тот же алгоритм на любом языке можно записать достаточно большим числом способов, и сложность выявления алгоритма непосредственно зависит от выбранного способа.

Рассмотрим различные особенности, влияющие на читабельность и самодокументируемость.

	C++	Ada	КП

1. Синтаксис языка	криптованный	человекоподобный	человекоподобный
2. Препроцессор	есть	нет	нет
3. Неявные преобразования	присутствуют	отсутствуют	только точные
4. Побочные эффекты операций	есть	нет	нет
5. Неоднозначность базовых средств	значительно	незначительно	незначительно

1. **Криптованный синтаксис** позволяет сократить тексты программ путем минимизации языковых конструкций, в частности за счет интенсивного использования спец. символов; такие конструкции обычно интуитивно малопонятны. Негативное влияние на читабельность проявляется для:

- начинающих и не опытных программистов, которые еще просто не привыкли к специфике синтаксиса;
- любых программистов, т. к. чисто с психологической точки зрения проще воспринимать более привычные (человеку) словесные формы (хоть и более длинные), нежели короткие и сокращенные.

2. **Наличие препроцессора** сказывается на читабельности в общем случае негативно. Мы видим не тот код, который будет компилироваться, а тот, который будет обработан препроцессором. Т. о., когда применяются сложные методы макроподстановок и условной компиляции, то понять (стороннему наблюдателю, а возможно - и самому разработчику через некоторое время), каков будет результирующий код, либо очень трудно, либо просто невозможно (без наличия и изучения дополнительной документации о применяемых макросредствах). Про читабельность и самодокументируемость в этом случае говорить не приходится.

3. В Ada **преобразования** из одного типа в другой могут быть только явными.

В КП допускаются неявные преобразования, если результирующий тип является надмножеством исходного. Например, вещественному можно присвоить целое, но не наоборот.

В C++ неявные преобразования допускаются для всех базовых типов, для некоторых производных, плюс круг таких преобразований может быть расширен за пользовательскими типами. Т. о. чтение программы требует повышенной внимательности, и сильно увеличивается вероятность неправильного понимания. Это сильно ухудшает читабельность.

4. **Операции** (записываемые в ЯП при помощи знаков) – по сути математические функции, которые прежде всего имеют определенный результат. Это их естественный смысл, который легко воспринимается.

В C++ в категорию операций входят операции присваивания (простого и вычисляемого) и инкремента/декремента, которые прежде всего изменяют свои аргументы (это их основной смысл) и как операции обладают результатом (основной смысл операций). К побочному эффекту тут можно отнести и то и другое – в зависимости от того, что трактовать как основное понятие: *действие присваивания* – операция или *операция* присваивания.

Это, во-первых, вызывает путаницу и частые ошибки начинающих. Во-вторых, благодаря такому средству можно так "навернуть" "мощное" вычислительное выражение, в котором помимо многочисленных вычислений изменяются значения различных переменных, что и опытный программист далеко не сразу разберется что к чему. При этом порядок вычислений (в случае изменения переменных) не всегда регламентируется стандартом, т. е. в таких случаях смысл выражения вообще зависит от реализации компилятора.

Все это – большой минус для читабельности.

5. **Неоднозначность средств** подразумевает, что множества результатов, которые можно получить за счет использования различных средств, пересекаются либо включают друг друга. Т. е. к одному и тому же результату можно прийти разными путями, причем не все пути будут одинаковыми по сложности и наглядности.

В общем случае этот вопрос рассмотрен в пункте "Богатство средств языка". Здесь мы остановимся на базовых (простейших) средствах ЯП. Неоднозначность этих средств, когда проявляется в большом числе возможных вариантов, негативно сказывается как на читабельности так и на самодокументируемости, т. к. у каждого программиста складывается свой стиль реализации базовых алгоритмических конструкций (последовательные действия, ветвление, циклы). Чем больше неоднозначностей, тем больше усилий приходится приложить, чтобы понять код.

Последовательные действия

КП

Представлены синтаксической конструкцией "операторная последовательность", которая может быть включена в тело процедуры/функции или в какой-либо составной оператор.

Ада

Аналогично КП.

C++

Представлены составным оператором, который может использоваться в составных операторах и выступает в качестве тела функции.

Неоднозначность возникает для последовательных вычислений, которые могут быть записаны как последовательностью операторов-выражений, так и одним оператором-выражением с использованием операции запятая (последования):

```
x = a + b;
y = c * d - x;
```

или

```
x = a + b, y = c * d – x;
```

При этом конструкции с операцией *запятая* могут быть включены в любое место, где требуется некоторое выражение:

```
x = a * b;
if (0 <= x && x <= 9) …
```

или

```
if (x = a * b, 0 <= x && x <= 9) …
```

Ветвление

КП

Представлено операторами IF, CASE и WITH.

IF позволяет указать произвольное число операторных последовательностей, одна из которых будет выполнена, если предшествующее ей логическое выражение истинно, и (необязательно) одну операторную последовательность, которая выполняется, если все выражения ложны. Пример:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = "'") OR (ch = '"') THEN ReadString
ELSE SpecialCharacter
END
```

CASE выполняет определенную операторную последовательность в зависимости от значения выражения (которое можно считать целочисленным: целые типы и символьные, коды которых также целые). Каждой последовательности соответствует уникальный набор значений. Последней можно указать последовательность, выполняемую, когда значение выражения не попадает ни в один указанный выше набор значений. Пример:

```
CASE ch OF
  "A" .. "Z": ReadIdentifier
| "0" .. "9": ReadNumber
| "'", '"': ReadString
ELSE SpecialCharacter
END
```

WITH похож на CASE, выполняет операторную последовательность в зависимости от динамического типа переменной, причем внутри операторной последовательности переменная обрабатывается в соответствии со своим динамическим типом (т. е. не требуется приведение типа). Например, типы A1 и A2 являются расширениями типа A (производными от A типами), x – переменная типа A.

```
WITH
  x: A1 DO … (* x рассматривается как переменная типа A1 *)
| x: A2 DO … (* x рассматривается как переменная типа A2 *)
ELSE …
END
```

Операторы WITH и CASE являются частными случаями оператора IF.

Ада

Представлены операторами if и case, полностью аналогичными этим операторам в КП; разница только в синтаксисе. Пример:

```
if (ch >= 'A') and (ch <= 'Z') then ReadIdentifier;
elsif (ch >= '0') and (ch <= '9') then ReadNumber;
elsif (ch = ''') or (ch = '"') then ReadString;
else SpecialCharacter;
enf if;

case ch is
  when 'A' .. 'Z' => ReadIdentifier;
  when '0' .. '9' => ReadNumber;
  when ''' | '"' => ReadString;
  when others => SpecialCharacter;
end case;
```

Оператор case является частными случаями оператора if.

Стоит отметить (хотя так никто делать не станет), что можно реализовать case и общий if при помощи if с одним условием без else и оператора goto. Т. е. в общем случае неоднозначность средств есть.

C++

Представлены операторами if и switch.

if аналогичен этим операторам в КП и Аде. Отличие – нельзя указать несколько условий (нет elsif). Пример:

```
if (ch >= 'A' && ch <= 'Z') ReadIdentifier();
else if (ch >= '0' && ch <= '9') ReadNumber();
else if (ch == '\'' || ch == '"') ReadString();
else SpecialCharacter();
```

switch схож только тем, что также рассматривает значения целочисленных выражений. В общем же случае механизм оператора switch отличается от case в Аде и КП. Он представляет собой последовательность операторов, каждый из которых может быть помечен меткой варианта вида case *целая_константа* (константы должны быть уникальны, нельзя указать диапазон значений), либо меткой default. Если значение выражения совпадает с одной из констант, выполняется **вся** последовательность операторов, начиная с помеченного оператора и до конца; либо, если не было ни одного совпадения, выполнение начнется с оператора, помеченного default (если есть). Для прерывания выполнения этой последовательности используется оператор break. Пример:

```
switch (ch) {
  case 'A': case 'B': … case 'Y': case 'Z': ReadIdentifier(); break;
  case '0': case '1': … case '8': case '9': ReadNumber(); break;
  case '\'': case '"': ReadString(); break;
  default: SpecialCharacter();
}
```

Этот пример эквивалентен примеру выше с if. Однако, если убрать break, то полученная конструкция может быть получена только с использованием if и goto (также, как и весь switch можно реализовать при помощи if с goto). Т. е. switch не является частным случаем if, if и switch неоднозначны.

Иногда такое поведение switch может быть полезно, однако в большинстве случаев оно не нужно и может вызвать лишнюю путаницу и ошибки, особенно у начинающих.

Замечание: gcc расширяет стандартный switch, позволяя указывать диапазон значений в метке варианта (значения разделяются троеточием):

```
… case 'A' ... 'Z': ReadIdentifier(); break;
  case '0' ... '9': ReadNumber(); break; …
```

Циклы

В качестве общей циклической конструкции можно рассматривать *безусловный цикл* с возможностью выхода в любой точке тела. При составлении алгоритмов часто применяются два частных случая: *цикл с предусловием* (один выход в начале) и *цикл с постусловием* (один выход в конце).

КП

Представлены операторами LOOP, WHILE, REPEAT, FOR.

Оператор LOOP – безусловный цикл, выход из которого осуществляет оператор EXIT. Пример (считывание и вывод неотрицательных целых чисел):

```
LOOP
  ReadInt(i);
  IF i < 0 THEN EXIT END;
  WriteInt(i)
END
```

Оператор WHILE – цикл с предусловием выхода из цикла если заданное логическое выражение ложно. Пример (поиск элемента в списке):

```
WHILE (p # NIL) & (p.x # value) DO
  p := p.next
END
```

Оператор REPEAT – цикл с постусловием выхода из цикла если заданное логическое выражение истинно. Пример (вывод последовательности чисел введенных пользователем, начиная с заданного числа, до 0):

```
REPEAT
  WriteInt(i);
  ReadInt(i)
UNTIL i = 0
```

Оператор FOR – цикл, выполняющийся заданное число раз, – с управляющей целочисленной переменной, которая пробегает указанный диапазон значений с заданным шагом. Пример (смещение к началу элементов одномерного массива из 10 элементов):

```
FOR i := 9 TO 1 BY -1 DO
  a[i-1] := a[i]
END
```

Оператор FOR является частным случаем оператора WHILE. WHILE и REPEAT в свою очередь – частные случаи LOOP.

Ада

Представлены операторами loop, while, for.

Оператор loop – безусловный цикл; выход осуществляют операторы exit (безусловный выход) и exit when (выход по условию). Пример:

```
loop
  ReadInt(i);
  if i < 0 then exit end if;
  WriteInt(i);
end loop;
```

Оператор while – аналогичен КП. Пример:

```
while (p /= null) and then* (p.x /= value) loop
  p := p.next;
end loop;
```

* В Аде две версии операций "логическое и" и "логическое или". Операции and и or всегда вычисляют все свои операнды. Операции and then и or else вычисляются по правилу короткого замыкания. (В КП и C++ эти логические операции присутствуют в одном экземпляре и вычисляются только по правилу короткого замыкания.)

Специального оператора для цикла с постусловием нет. Для этого непосредственно используется loop. Пример:

```
loop
  WriteInt(i);
  ReadInt(i);
  exit when i = 0;
end loop;
```

Оператор for – аналогичен КП за тем исключением, что допустимый шаг +1 или -1. Пример:

```
for i in reverse (a'First+1) .. a'Last loop
  a(i-1) := a(i);
end loop;
```

Оператор for является частным случаем оператора while. while мог бы считаться частным случаем loop. Однако, теоретически, из циклов while и for можно выйти при по-

мощи goto (из loop тоже). На практике же не многие станут это делать (как и использовать goto и if для организации циклов).

Т. о. неоднозначность средств есть, но с теоретической точки зрения.

C++

Представлены операторами while, do-while, for.

Здесь присутствует как теоретическая неоднозначность (учитывая goto) так и практическая (операторы break и continue). break позволяет прервать выполнение любого оператора цикла в любой точке тела, continue – прервать выполнение текущей итерации и перейти к следующей; break и continue интенсивно используются на практике. Т. о. если в КП и Аде (не учитывая goto) самым общим является безусловный цикл, то здесь любой оператор цикла можно считать общим. Т. е. мы имеем три разных оператора, которые фактически позволяют делать одно и тоже.

Оператор while – цикл с "предусловием" выхода, если заданное логическое выражение ложно (цикл может оказаться также и циклом с постусловием или еще каким-то, в зависимости от того, будет ли и где будет использован оператор break в теле цикла). Пример:

```
while (!p && p->x != value)
  p = p->next;
```

Оператор do-while – цикл с "постусловием" выхода из цикла если заданное логическое выражение ложно. Пример:

```
do {
  WriteInt(i);
  ReadInt(i);
} while (i != 0);
```

Оператор for – обобщение оператора while.

```
Общий синтаксис:
for (expr1; expr2; expr3) op
expr1, expr2, expr3 - не обязательны, по умолчанию expr2 == true, это эквива-
лентно:
expr1;
while (expr2) {
  op;
  next: expr3; // сюда будет выполнять переход continue
}
```

Примеры:

```
for (i = 9; i >= 1; --i)
  a[i-1] = a[i];
или
for (i = 9; i >= 1; ) {
  a[i-1] = a[i];
  --i;
}
или
for (i = 9; i >= 1; a[i-1] = a[i], --i);
```

Оператора безусловного цикла нет – вместо него используется любой оператор со всегда ложным условием выхода. Примеры:

```
while (true) {
  ReadInt(i);
  if (i < 0) break;
  WriteInt(i);
}
или
for (;;) { … }
или
do { … } while (true);
```

Парадигмы программирования

Какие парадигмы программирования поддерживает язык? Возможность более точно и просто реализовать архитектуру ПП средствами ЯП напрямую влияет на +В.

Общая картина выглядит так:

	C++	Ada	КП
1. Структурное	±	±	+
2. Модульное	±	+	+
3. Объектно-ориентированное	+	+	+
4. Компонентное	-	-	+

1. Структурное программирование поддерживается всеми языками. В каждом присутствуют составные операторы (последовательный, ветвления, циклов), подпрограммы, структуры данных.

В КП отсутствуют средства, которые не соответствуют принципам структурного программирования, т. е. их использование исключено. В Аде (goto) и C++ (goto; частично break и continue) ответственность за непременение таких средств перекладывается на программиста.

Отсутствие вложенных функций в C++ может затруднить декомпозицию (GNU CC поддерживает эту возможность для языка C как расширение).

Стоит отметить, что написание хорошей структурированной программы реально зависит именно от программиста. Приведем пример на КП (взят из [2]) двух процедур, делающих одно и тоже, первая из которых плохо структурирована, вторая – хорошо:

```
PROCEDURE (obj: setGuider) generate*, NEW;
  VAR exit, reset: BOOLEAN; i: INTEGER;
BEGIN
  exit:=FALSE; reset:=FALSE; i:=LEN(obj.content)-1;
  WHILE exit = FALSE DO
    IF i >= 0 THEN
      obj.content[i].checkFull();
      IF obj.content[i].full = FALSE THEN
        obj.content[i].add();
        exit := TRUE
      ELSE
        reset:=TRUE;
        obj.content[i].reset();
        DEC(i)
      END;
      IF (reset = TRUE) & (exit = TRUE) THEN
        obj.setMax(i);
      END
    ELSE
      exit := TRUE;
      obj.end := TRUE
    END
  END
END generate;

PROCEDURE (sg: SetGuider) Next*, NEW;
  VAR
    i: INTEGER; c: POINTER TO ARRAY OF Set;
BEGIN
  c := sg.content;
  i := LEN(c) - 1;
  WHILE (i >= 0) & c[i].IsFull() DO
    c[i].Reset(); i := i - 1
  END;
  IF i < 0 THEN
    sg.end := TRUE
  ELSE
    c[i].Inc();
    IF i < LEN(c) - 1 THEN sg.SetMax(i) END
  END
END Next;
```

2. В Ada и КП поддерживается явно, в C++ – нет.

В Ada модуль (единица обладающая определенным интерфейсом и обеспечивающая инкапсуляцию) – пакет, его интерфейс и реализация разносятся по разным единицам компиляции (т. е. раздельно компилируются); инкапсуляция обеспечивается за счет приватной части пакета.

В КП модуль – модуль (также единица написания, распространения и загрузки). Интерфейс обозначается метками экспорта прямо в теле модуля, и файл, содержащий описание интерфейса, генерируется по ним автоматически. Инкапсуляция обеспечивается автоматически – все не экспортированные средства модуля не доступны.

В C++ модуль может быть реализован на основе класса, который прежде всего является типом данных. Для класса может быть полностью определен интерфейс, который импортируется как единое целое; раздельной компиляции нет – интерфейс (определение класса) должно полностью включаться в каждую единицу компиляции, которая его использует. Инкапсуляция обеспечивается за счет приватной и защищенной частей класса.

3. Полностью поддерживается всеми языками. Имеются средства, предоставляющие наследование и полиморфизм.

C++ допускает множественное наследование (КП и Ада – нет).

4. Компонентное программирование поддерживается только КП. Более подробно см. [2].

Средства расширения языка

Сюда прежде всего относятся средства, позволяющие организовать работу с пользовательскими типами данных так же, как с предопределенными в ЯП типами.

В C++ и Аде – это механизм совмещения (перегрузки) имен процедур/функций (1) и знаков операций (2). В КП эти средства отсутствуют.

Первое дает возможность определять процедуры/функции с теми же именами, что у предопределенных процедур/функций или функций из стандартной библиотеки. Например, процедуры Put/Get из пакетов ввода-вывода Ады.

Второе позволяет определять и использовать предопределенные знаки операций с типами пользователя. Например, определив тип матрица и перегрузив знак операции сложения, можно будет писать a+b, для переменных a, b типа матрица.

Влияние этого средства двойственное. Если рассматривать с точки зрения разработчика библиотек типов данных, то для него процесс усложняется: A2, -B. Чтобы новый тип данных действительно расширял язык, нужно строго следить за перегружаемыми средствами – их поведение не должно отличаться от поведения встроенных средств. В этом случае для пользователя библиотеки процесс разработки упрощается, улучшается читабельность: +B.

Однако, никто не может гарантировать что новый тип обладает поведением, аналогичным встроенным типам. Например, можно знак операции "+" использовать для выполнения вычитания (это гротескный пример, более опасны случаи, когда разница в поведении незначительная). Т. о. пользователь библиотеки интуитивно ожидает одного, а при выполнении получит другое: -читабельность, -B, -C.

Отдельно отметим случай высокой требовательности перегруженной операции к машинному ресурсу (что не характерно для встроенных типов). Завуалированность таких операций (даже при правильном поведении типа) будет приводить к менее внимательной проработке вычислительных алгоритмов, и может повлиять на их качество: -C1.

Обобщенное программирование

Обобщенное программирование прежде всего дает возможность повторного использования кода. Если некоторый алгоритм применим к множеству типов данных, обладающих определенными свойствами, то его достаточно описать один раз (в основном это применимо к математическим алгоритмам). Далее меняется уже не сам алгоритм, а конкретно используемый тип. Например, (один и тот же) алгоритм вычисления минимума из двух значений, применим к любому типу, который описывает упорядоченное множество (определена операция "меньше").

C++ и Ада (но не КП) предоставляют средства, которые позволяют описать такой обобщенный алгоритм один раз, а затем настроить его (конкретизировать) для использования с заданным типом данных.

Влияние этого средства также двойственное.

Использование обобщенных алгоритмов может значительно сократить время разработки за счет повторного использования кода (+B), и повысить надежность (+C2), т. к. применяются уже проверенные и отработанные алгоритмы. Однако, оно требует большей квалификации, опыта и внимания: A2. Как от использующего такие алгоритмы – пользователь должен хорошо

разбираться в требованиях к "параметрам" обобщенного алгоритма, чтобы правильно оценить возможность его эффективного (с точки зрения С1) применения, иначе: -С1. Так и от разработчика (причем в гораздо большей мере), поскольку приходится оперировать с абстракциями более высокого уровня и отслеживать большое число нюансов (-В); больше вероятность ошибки (-С2). К тому же за счет обобщения приходится отбрасывать особенности, что сказывается на производительности алгоритма: -С1.

Ада позволяет создавать обобщенные подпрограммы и пакеты, их параметрами могут быть: типы, константные значения, подпрограммы, обобщенные пакеты.

С++ позволяет создавать обобщенные подпрограммы (шаблоны функций) и типы данных/модули (шаблоны классов) (см. "Парадигмы программирования", класс – фактически аналог пакета Ады). Параметрами могут быть: типы данных, константные значения и шаблоны классов. Также С++ позволяет частично конкретизировать шаблоны классов (частичная специализация), что дает возможность лучше учитывать особенности (+С1) при сохранении подхода обобщенного программирования.

Интерфейс с другими языками

Повторное использование кода (написанного на другом языке) или реализация каких-либо подзадач (на более подходящем языке) сократит время разработки и повысит эффективность: +В1, +С1.

Это требует дополнительных знаний (А23); нарушает целостность проекта, что может усложнить сопровождение и снизить надежность: -В2, -С2.

Стандарты языков не требуют интерфейса с другими языками:

C++	**Ада**	**КП**
Интерфейс ограничивается языком С, который является подмножеством С++.	Стандарт описывает интерфейсы с языками ассемблера, С и COBOL. Однако поддержка этих интерфейсов не обязательна.	Нет

Реализации:

GCC	**GNAT**	**Blackbox (Windows)**
?	ассемблер, C/C++, COBOL	C

Низкоуровневое программирование

Средства низкоуровневого программирования требуют непосредственного доступа к машинным командам и обоснованно применимы в особых случаях, когда высоки требования к затратам машинного ресурса; использование этого средства дает +С1.

Однако требует специальных знаний (А3), более детальной проработки (-В), наиболее высока вероятность ошибок (-С2).

Стандарты:

C++	**Ада**	**КП**
Ассемблерная вставка - ключевое слово asm.	Либо вставка ассемблерного кода, либо интерфейс с ассемблером.	Нет.

Реализации:

GCC	**GNAT**	**Blackbox (Windows)**
Обширные возможности по вставке кода с указанием используемых машинных средств, что позволяет применять к такому коду оптимизацию, осуществляемую компилятором.		Кодовые процедуры (встраиваемый ассемблерный код).

Надежность и безопасность

Человеку – свойственно ошибаться. И он ошибается. Поэтому любая программа (которую пишет человек) потенциально содержит в себе ошибки. Зачастую эти ошибки не просто

сказываются на работе конкретных пользователей, но и приводят к просто катастрофическим последствиям и многомиллионным убыткам.

Как пример рассмотрим распространения вирусов-червей, связанных с так называемой ошибкой переполнения буфера, возникающей из-за отсутствие контроля корректности индексов массивов:

Вирус	**Обнаружен**	**Поражал**	**Механизмы распространения**	**Машин заразил**
Code Red I/II	2001, июль	ISS	Переполнение буфера в ISS	1000000
Nimda	2001, сентябрь	ISS	Переполнение буфера в ISS, слабые пароли и т. п.	2200000
Slammer	2003, январь	MS SQL	Переполнение буфера SQL	300000
Love San	2003, август	NT/2000/XP/2003	Переполнение буфера в DCOM	более 1000000

И это – далеко не полный список…

Т. о. любой современный язык просто обязан иметь в наличии средства, повышающие надежность и безопасность кода, за счет выявления человеческих ошибок или исключения возможности появления оных.

Среди этих средств можно выделить:

1. *Статические* (+B, +C2) – задействуемые во время компиляции и никак не влияющие на программный код. Здесь можно выделить *строгую типизацию* и сведение к минимуму *неявных преобразований*. Это помогает избегать алгоритмических ошибок, связанных с неправильными значениями, когда, например, к переменной одного типа обращаются как к переменной другого, или когда упускают возможность потери значащих цифр при преобразовании из длинного целого в короткое.
2. *Динамические* (+B, -C1, +C2) – задействуемые во время выполнения – специальный код генерируется компилятором автоматически. Надежность повышается, но ценой дополнительных затрат машинного ресурса (времени для выполнения и оперативной памяти для хранения дополнительных инструкций). Сюда входят *контроль корректности индексов массивов* (не допускающий, например, переполнения буфера) и *контроль типов* (применяемый к объектам, тип которых не известен во время компиляции, например, полиморфные сущности в ООП) – позволяет избегать нарушения типизации во время выполнения.
3. *Автоматическое управление памятью* (+B, +C2) – более не используемая динамическая память освобождается автоматически (так называемая сборка мусора). Этот механизм исключает одни из наиболее опасных (для стабильной работы ПП) и трудновыявимых ошибок – утечки памяти и висячие ссылки (указатели).

<table>
<tr><th></th><th></th><th>C++</th><th>Ada</th><th>КП</th></tr>
<tr><td rowspan="4">1</td><td>Строгая типизация</td><td>да*</td><td>да</td><td>да</td></tr>
<tr><td colspan="4">* Средства C нарушают строгую типизацию. Например, передача функции произвольного числа аргументов – их типы могут быть любыми – необязательно теми, что ждет вызываемая функция.</td></tr>
<tr><td>Неявные преобразования</td><td>есть*</td><td>нет</td><td>ограничено**</td></tr>
<tr><td colspan="4">* Неявные преобразования определены для многих базовых типов, и расширяемы за счет преобразований, определяемых пользователем.
** Неявные преобразования ограничены случаями, в которых невозможна потеря точности (когда результирующий тип включает исходный как подмножество).</td></tr>
<tr><td rowspan="2">2</td><td>Контроль индексов массивов</td><td rowspan="2">вручную</td><td rowspan="2">автоматически, отключаемо (выборочно)</td><td rowspan="2">автоматически, не отключаемо</td></tr>
<tr><td>Контроль типов</td></tr>
</table>

3	*Автоматическое управление памятью*	нет	элементы*	есть**
	* Если время жизни типа закончено (для локально определенных типов), то динамическая память, занимаемая переменными таких типов, освобождается автоматически. Компилятор гарантирует, что ссылок на эти переменные не будет (статическая проверка); однако эта проверка может быть отключена… ** Требования стандарта языка к среде выполнения.			

Итог

В свете рассмотренного выше понимания эффективности, позволю себе сделать вывод, что в **общем случае**, рассматривая широкий круг задач и учитывая то, что многие из них не требуют системного программирования и не критичны к скорости выполнения, программирование на C/C++ менее эффективно. Т. к. при равных требованиях к ПП, в сравнении с языками Pacal-семейства:

- требуется гораздо более детальная проработка кода;
- больше усилий со стороны программиста;
- повышен риск ошибок;
- и, в итоге, требуется большая квалификация и опыт программиста для достижения того же результата.

Совет начинающим

Программирование на C++ далеко не так просто, как может показаться на первый взгляд.

Это сложный язык, поэтому научиться программированию с нуля – именно программированию, а не писанию программ на C++, крайне трудно. Программирование прежде всего подразумевает умение просчитать всевозможные варианты и составить алгоритм в уме, выполнить абстракцию данных и разбить поставленную задачу на подзадачи, выстроить схему всего ПП. Для начала лучше выбрать более простой язык и развивать эти умения, нежели сложный, в котором придется большую часть сил отдавать на усвоение особенностей языка. Приведу цитату от создателя C++ [1]:

"Вопрос "Как пишут хорошие программы на C++" очень похож на вопрос "Как пишут хорошую английскую прозу?" Есть два вида ответов: "Знайте, что вы хотите сказать" и "Практикуйтесь. Подражайте хорошему языку." Оба совета оказываются подходящими к C++ в той же мере, сколь и для английского – и им столь же трудно следовать."

Продолжая эту цитату, скажу: лучше сначала научиться вообще писать прозу, а лишь потом переходить к прозе на английском. Тоже справедливо и для C++. (Рассматривая все варианты, придется заметить, что этот пример не подойдет для тех, у кого английский язык является родным.)

Еще цитата:

"Напомню, что большую часть программирования можно легко и очевидно выполнять, используя только простые типы, структуры данных, обычные функции и небольшое число классов из стандартной библиотеки. Весь аппарат, входящий в определение новых типов, не следует использовать за исключением тех случаев, когда он действительно нужен."

Возникает вопрос: зачем сразу изучать множество сложностей и нюансов, если они, не пригодятся вам, как минимум, на начальных этапах? Лучше больше времени уделить навыку программирования вообще, нежели усваивать специфику возможностей C++. Без знания этой специфики, нормально программировать на C++ вы вообще не сможете, однако знание этой специфики никак не улучшит ваших общих навыков программирования.

Выбирая C++ своим первым языком, вы рискуете пройти вдвое больший путь до профессионального программиста, следуя противоречивому характеру развития языка. В нем новые техники программирования (привнесенные в соответствии с потребностями времени) сочетаются с давно устаревшими и ненужными (которые оставлены в основном для совместимости с уже написанным кодом). Такое сочетание средств часто требует нетривиальных решений в языке, что сильно его усложняет. И после сложной работы (которая ведется и сейчас) по усовершенствованию языка и обогащению его новыми средствами, ведутся работы и по его упрощению :). Так, например, ведутся работы [9] по выделению подмножества **стандартного C++** путем исключения некоторых возможностей, для повышения читабельности и надежности кода. При этом, если вы захотите следовать этим рекомендациям, весь контроль возлагается именно на вас – человека (которому свойственно (говорю про всех людей вообще) ошибаться и делать себе поблажки). А не лучше ли в таком случае сразу выбрать язык более простой и строгий, в котором компилятор будет следить за соблюдением всех правил? …

Заключение

В заключение хочу еще раз подчеркнуть, что не смотря на все претензии на объективность, при сравнении я приводил прежде всего свое мнение. Мнение человека, который в течение нескольких лет изучения и использования C++ был его ярым фанатиком (в основном из-за того, что считал этот язык единственным, поддерживающим средства обобщенного программирования)… Но человеку свойственно ошибаться, как и стремиться использовать в работе более удобные инструменты, поэтому в последствии мое мнение изменилось…

В любом случае, никто не будет спорить с утверждением, что профессиональный программист должен знать несколько языков, и выбирать наиболее подходящий для конкретной задачи. В контексте этого утверждения ни про один язык нельзя сказать, что он не востребован и вообще не нужен...

ПРИМЕЧАНИЕ РЕДАКТОРА СБОРНИКА: в настоящее время выпуск бесплатных версий комплятора Ada GNAT возобновлен.

Литература

1. Страуструп Б. Язык программирования C++. Второе дополненное издание.
2. Информатика-21. Международный общественный научно-образовательный проект. http://www.inr.ac.ru/~info21
3. Трой Д. Программирование на языке Си для персонального компьютера IBM PC: Пер. с англ.- М.: Радио и связь, 1991. – 432 с.
4. Гавва А. "Адское" программирование. Ada-95. Компилятор GNAT. (электронный документ) V – 0.4w, май 2004.
5. Вандевурд Д., Джосаттис Н. М. Шаблоны C++: справочник разработчика. Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 544 с.
6. International Standard ISO/IEC 14882:1998(E): Information technology – Programming languages – C++.
7. Ada 95 Language Reference Manual ANSI/ISO/IEC 8652:1995 http://www.adapower.com/rm95/index.html
8. Документация к компилятору GNU CC 2.8.1.
9. The Programming Research Group. High-Integrity C++ Coding Standard Manual. Version 2.2. http://www.programmingresearch.com
10. Ермаков И. Е. "Программная инженерия вчера и сегодня: эволюция императивного программирования".

Язык программирования Компонентный Паскаль и среда разработки BlackBox Component Builder

Ермаков И.Е., Рюмшин Б.В.

I. Компонентный Паскаль – язык компонентно-ориентированного программирования

Язык Компонентный Паскаль (далее - КП) является кульминацией десятилетий исследовательской работы швейцарской школы программирования. Его родословная выглядит следующим образом: Алгол – Паскаль – Модула-2 – Оберон – Оберон-2 – КП. КП родился в 1997 году, когда швейцарская компания Oberon Microsystems[1] сделала небольшие добавления к языку Оберон-2. В язык были добавлены некоторые средства, для повышения безопасности компонентных каркасов (component frameworks)[2] и контроля целостности больших компонентных систем. На сегодняшний день КП является одним из трех ЯП, ориентированных на компонентное программирование: Java, КП и C#.

Основные принципы компонентно-ориентированного программирования (КОП) были сформулированы авторами КП Клеменсом Шиперски[3] («Компонентное программирование: за пределами ООП» [1]) и Куно Пфистером [2]. Наметим здесь основные идеи КОП и свойства, характерные для компонентных ЯП. КОП является дальнейшим развитием идеи модульных систем. Модульные системы строятся из набора модулей - «черных ящиков», взаимодействующих через строго определенные интерфейсы и инкапсулирующих в себе детали реализации. Современные среды выполнения (в порядке появления: Оберон, Java, CORBA, COM, .NET) поддерживают динамическую загрузку/связывание/выгрузку модулей, что позволяет расширять программную систему прямо во время ее выполнения. КОП предполагает совместное использование динамической модульности и ООП, результатом чего является прорыв в новое измерение в инженерии ПО. Единственным вводимым ограничением на ООП является *запрет межмодульного наследования реализаци*. Такое наследование приводит к проблеме *хрупкого базового класса* [1,2], суть которой в том, что разработчик базового класса не может изменять в новых версиях его реализацию из боязни нарушить работу классов-потомков, созданных пользователями его компонента. Это противоречит требованиям безопасности и расширяемости системы.

Технически компонентное ПО можно создавать на любом современном языке программирования – используя некоторую языково-независимую объектную модель[4] (Windows DLL, COM, CORBA, .NET), однако на практике некоторые языки подходят для этого гораздо больше других, а три выше перечисленных признаны непосредственно *компонентными языками*. Их сильные стороны касаются программирования «в большом», то есть, возможности эффективно воплощать архитектурные решения в программном коде. Основным качеством таких языков является *безопасность* – в среде, состоящей из множества взаимодействующих компонент, которые разработаны независимо друг от друга, она выходит на первый план. *Статическая безопасность* нацелена на то, чтобы предотвратить ошибку или опечатку программиста на этапе написания кода – она включает в себя строгий контроль типов, запрет неявных преобразований, сведение к минимуму всякого рода «умолчаний» в языковых конструкциях[5], прозрачный и недвусмысленный синтаксис, изоляцию низкоуровневых средств в отдельном секторе языка/среды (в КП – псевдомодуль компилятора SYSTEM). *Динамическая безопасность* подразумевает в первую очередь целостность памяти, то есть, гарантии того, что один компонент не может разрушить память другого. В традиционных операционных системах (Windows, Unix) для этого используются аппаратные механизмы защиты, при которых каждое приложение работает в своем адресном пространстве и не имеет доступа к памяти другого. В то же время сама

---

[1] Oberon Microsystems отпочковалась от Университета ETH, состоит в основном из учеников Н.Вирта, который также входит в состав директоров.

[2] Компонентный каркас – набор модулей, предоставляющих расширяемые классы для конкретной предметной области.

[3] К. Шиперски долгое время был ведущим разработчиком Oberon Microsystems, после чего был приглашен в Microsoft Research и принял непосредственное участие в разработке платформы .NET и языка C#.

[4] Под объектной моделью понимаются стандарты и механизмы загрузки, связывания и взаимодействия компонентов в оперативной памяти (в общем случае – в распределенной сетевой системе).

[5] Проще говоря, если программист по невнимательности упустил какую-либо явную директиву, компилятор языка не должен делать никаких предположений, а выдать сообщение об ошибке.

идея КОП требует, чтобы модули работали в едином адресном пространстве, взаимодействуя друг с другом напрямую. Поэтому обеспечить безопасность в рамках одного адресного пространства может лишь система времени выполнения самого языка. Это достигается запретом адресной арифметики (то есть, с указателем нельзя делать никаких операций кроме обращения по нему), контролем границ массивов[1], динамическим контролем типов объектов. Отметим, что некоторые новейшие модели процессоров (например, отечественная разработка Эльбрус-Е2К), поддерживают эти механизмы на аппаратном уровне. И, наконец, важнейшей частью компонентного языка является автоматическое освобождение памяти, так называемая *сборка мусора*. В компонентной среде ни один модуль не способен сам определить момент, когда память можно освободить. Это может сделать только механизм среды, проверив отсутствие во всех модулях указателей на освобождаемый объект.

Статическая безопасность всегда была характерна для языков семейства Паскаля. Динамическая безопасность долгое время была свойственна только динамическим, то есть, интерпретируемым, языкам (например, Smalltalk). Среди компилируемых языков динамическая безопасность (контроль типов, сборка мусора) впервые появилась в языке Оберон (1989), который наглядно показал, что надежный язык с богатыми динамическими возможностями может эффективно компилироваться и по быстродействию выходного кода не уступать C/C++[2]. Осознание потребности в надежных языках привело к созданию Java (1995) и C# (2000), которые объединяют концепции Паскаль- и С-семейств (во многих аспектах объединяют довольно неуклюже).

К. Пфистер в [2] отмечает еще один важный аспект компонентных языков: сокрытие информации на более высоком уровне, нежели классы, то есть, наличие в непосредственно в языке понятия модуля (пакета). Обычно в программной системе присутствуют группы классов, совместно выполняющие некоторые функции. Они должны свободно взаимодействовать друг с другом, минуя инкапсуляцию. Модули позволяют выстроить общий интерфейс для таких групп, в то же время не устанавливая ненужных барьеров между классами одной группы. В то же время большинство объектно-ориентированных языков ограничиваются сокрытием информации на уровне класса, не предоставляя более глобальных абстракций. К сожалению, язык C# также по-прежнему не является модульным, так же, как и его предшественник C++. В то же время Компонентный Паскаль уже по самому своему происхождению - модульный язык, в котором модуль является глобальной единицей написания кода, инкапсуляции, компиляции, распространения и загрузки в память.

Если сравнивать КП и Java, то эти языки эквивалентны по своим возможностям и совместимы на двоичном уровне – существуют компиляторы КП в байт-код виртуальной машины Java (JVM). Однако КП гораздо более строен, последователен и прост для освоения, он продолжает традицию Оберонов – предельно лаконичных языков, описание которых занимает 16-30 страниц. В то же время Java, хотя и позиционировалась как простой язык, содержит достаточное количество подводных камней, язык сложно изучать поэтапно, так как каждый из его аспектов требует одновременного рассмотрения многих других. То же можно сказать и про C#, который во многом является ответом Microsoft на Sun'овкую Java.

Важным аспектом является то, что Java как технология ориентирована на выполнение на виртуальной машине. Это порождает проблемы, связанные с быстродействием. Также играет свою роль то, что, во-первых, все экземпляры классов в Java размещаются в динамической памяти, - это дает чрезмерную нагрузку на диспетчер памяти и сборщик мусора; во-вторых, в Java отсутствуют как указатели, так и передача параметров процедур по ссылке. Последнее вынуждает разработчика даже для такой элементарной операции, как передача массива по ссылке, прибегать к различного рода ухищрениям, например, классам-оберткам. Все эти проблемы, несмотря на заявляемую безопасность языка, сильно ограничивают его использование в системах реального времени[3]. Напротив, КП, как и все Обероны, особенно эффективен для задач систем-

[1] Отсутствие обязательного контроля границ массивов в языках С-семейства (точнее, пренебрежение программистов соответствующими опциями компиляторов) сделало возможным самую распространенную хакерскую атаку – так называемую «атаку на переполнение буфера». Миф о том, что контроль границ массивов якобы сильно снижает быстродействие, до сих пор препятствует созданию надежного ПО.

[2] А в некоторых реализациях – превосходить большинство известных компиляторов (см. XDS – разработка новосибирской компании Excelsior, http://excelsior-usa.com).

[3] Sun в документации к своей Java-платформе заявляет, что их продукт не предназначен для использования в тех сферах, где сбой системы может повлечь тяжкие последствия.

ного программирования. На Оберон-2 и КП написаны многие операционные системы, их них три - промышленные жесткого реального времени: XO/2 (на ее основе швейцарским федеральным правительством сертифицирована и принята к развертыванию автоматизированная система контроля за дорожным движением), XOberon (разработка Цюрихского института робототехники) и JBed (см. [3], раздел Ссылки). О последней стоит сказать особо. JBed разработана на основе BlackBox Component Builder компанией Esmertec – дочерней от Oberon Microsystems. Она предназначена в первую очередь для встроенных систем, в том числе жесткого реального времени, используется также в различной бытовой технике европейского производства, в мобильных телефонах. Одна из особенностей ОС JBed в том, что под ней могут выполняться модули, написанные как на КП, так и на Java; более того – такие модули работают в едином адресном пространстве и с общим сборщиком мусора. На КП в среде BlackBox компания Oberon Microsystems разработала для Borland JIT-компилятор Java; систему мониторинга для крупнейшей ГЭС на Амазонке в Бразилии (под UNIX-64). Эти примеры убедительно доказывают успешность КП для сложных задач системного программирования.

II. Экскурсия по языку

Программирование «в малом». КП поддерживает стандартный набор управляющих конструкций структурного программирования: *IF*, *CASE*, *WHILE*, *REPEAT*, *FOR* и *LOOP* (цикл с явным прерыванием по *EXIT*). Разумеется, такие неструктурные операторы, как *GOTO*, *BREAK* и *CONTINUE* в язык не включены. Приведем пример алгоритма на КП (быстрая сортировка):

```
VAR a: ARRAY 1024 OF Element;

PROCEDURE QuickSort;
  PROCEDURE Sort (l, r: INTEGER);
    VAR i, j: INTEGER;
        w, x: Element;
  BEGIN
    i := l; j := r;
    x := a[(l+r) DIV 2];
    REPEAT
      WHILE a[i] < x DO INC(i) END;
      WHILE x < a[j] DO DEC(j) END;
      IF i <= j THEN
        w := a[i]; a[i] := a[j]; a[j] := w;
        INC(i); DEC(j)
      END
    UNTIL i > j;
    IF l < j THEN Sort(l, j) END;
    IF i < r THEN Sort(I, r) END
  END Sort;
BEGIN
  Sort(0, LEN(a)-1)
END QuickSort;
```

Отметим некоторые моменты: управляющие операторы записываются в лаконичной форме *ОПЕРАТОР ...; ...; ...END* в отличие от двух неудачных форм, которые использовались в старом Паскале: сокращенной *ОПЕРАТОР ...;* и полной *ОПЕРАТОР begin ...; ...; ... end*. Язык стал чувствительным к регистру – имена *x* и *X* теперь являются различными. Все ключевые слова записываются заглавными буквами, что улучшает читаемость программ и позволяет использовать выделение жирностью и цветом для других целей – для расстановки смысловых акцентов. Окончания процедур записываются в виде *END ИмяПроцедуры*, что облегчает поиск потерянного *END*. Введена удобна форма оператора для последовательной проверки многих условий:

IF условие1 THEN ... ELSIF условие2 THEN ... ELSIF... ELSE... END

Массивы нумеруются только от 0, объявляются в виде *ARRAY размер OF...* Поддержива-

ются передача в процедуру по ссылке открытых массивов (неопределенного на этапе компиляции размера), что особенно полезно - многомерных: *PROCEDURE ... (VAR a: ARRAY OF ARRAY OF ...)*. Поддерживаются динамические многомерные массивы: *POINTER TO ARRAY OF ARRAY OF…*.

Модульность. Модули в КП имеют следующий вид:

```
MODULE Module;

  IMPORT StdLog; (* Импортируем необходимые модули *)

  TYPE
    Person* = RECORD
      id: INTEGER;
      name-, surname-: ARRAY 32 OF CHAR;
      salary*: INTEGER
    END;

  PROCEDURE Print* (IN person: Person);
  BEGIN
    StdLog.String(person.name); StdLog.Ln;
    StdLog.String(person.surname); StdLog.Ln;
    StdLog.Int(person.salary); StdLog.Ln
  END Print;

  PROCEDURE Init;
  BEGIN
    …
  END Init;

BEGIN (* Секция инициализации – выполняется при загрузке модуля
*)
  Init
CLOSE (* Секция финализации – выполняется при выгрузке модуля *)
  ...
END Module.
```

Здесь знаком «*» помечаются так называемые *экспортируемые* сущности модуля, т.е. то, что будет доступно снаружи, войдет в *интерфейс* модуля. Знак «-» для переменных и полей записей означает «экспорт только для чтения». Например, у типа *Person* снаружи будут доступны поля *name, surname* – только для чтения и *salary* – для записи. При компиляции данного модуля будет создан кодовый файл, который может быть динамически загружен приложениями, и символьный файл, хранящий всю необходимую информацию об интерфейсе модуля. Текстовое описание интерфейса модуля может быть выведено средой автоматически:

```
DEFINITION Module;
  IMPORT Math;
  TYPE Person = RECORD
    name-, surname-: ARRAY 32 OF CHAR;
    salary: INTEGER
  END;
  PROCEDURE Print (IN person: Person);
END Module.
```

Важным требованием КП является то, что имя любой сущности из импортированного модуля всегда указывается в полном виде: *Модуль.Имя* (как в примере – *StdLog.String*). Это позволяет при чтении кода программы сразу видеть, к какому из модулей относится тот или иной тип/переменная/процедура. Если имя модуля слишком длинное, можно назначить псевдоним: *IMPORT Log := StdLog* и далее работать с ним: *Log.String*.

Работа с памятью. Хотя КП является полностью безопасным языком, в нем, в отличие от

Java, сохранены оба типа выделения памяти под структурные переменные: статическое (в стеке процедур или в глобальной памяти модулей) и динамическое (в куче). Динамически выделенные переменные доступны через *указатели*:

```
TYPE Rec = RECORD … END;
VAR r: Rec (* статически размещенная переменная *)
     pr: POINTER TO Rec; (* указатель на динамически размещенную переменную *)
BEGIN
  NEW(pr); (* Создаем новую переменную в куче *)
```

Освобождать память явно не нужно, да и невозможно – переменная будет уничтожена сборщиком мусора, как только исчезнет последний указатель на нее. Если мы объявим тип следующим образом: *TYPE Type = POINTER TO RECORD ... END*, то экземпляры такого типа могут быть только динамическими, т.е. создаваемыми через *NEW*.

На использование указателей наложены ограничения: отсутствует арифметика указателей; указатель может указывать только на структурные типы, то есть, записи или массивы; указатель может указывать только на переменные, выделенные через *NEW*. В языке отсутствует операция взятия адреса, т.к. она небезопасна (переменная в стеке может стать недоступной после выхода из процедуры, переменная модуля может исчезнуть после выгрузки модуля. Таким образом могли бы появиться висячие указатели). Однако указатель, полученный через *NEW*, никогда не станет висячим, так как переменная в куче будет существовать до тех пор, пока существует хотя бы один указатель на нее.

Однако безопасная операция взятия адреса в языке неявно все же присутствует – это передача параметров в процедуру по ссылке, то есть, давно известные в Паскале *VAR*-параметры, например: *PROCEDURE Proc (IN a: Rec; VAR b: INTEGER; OUT c: BOOLEAN)*. Их в КП три типа: *IN* – входной, доступен только для чтения; *VAR* – переменный, доступен для чтения/записи; и *OUT* выходной, доступен для чтения/записи, позволяет подчеркнуть, что значение параметра на входе в процедуру может быть неопределенным.

Передача параметров по ссылке в совокупности со статическими переменными-записями отчасти обусловливает большую эффективность КП в сравнении с компилируемыми версиями Java – значительная часть небольших объектов в системах на КП размещается и передается статически, через стек, что уменьшает нагрузку на диспетчер памяти и сборщик мусора.

ООП. В семействе Оберонов традиционно принят подход к ООП, заявленный Н. Виртом: *ООП = структурное программирование + расширение типов*. В таком понимании *класс = структура данных*, *объект = переменная структурного типа*, *метод = связанная процедура*, *наследование = расширение типа*. Рассмотрим пример нерасширяемой записи со связанными процедурами:

```
TYPE
  Vehicle = RECORD (* Транспортное средство *)
    x, y, v: REAL;
  END;

PROCEDURE (VAR vh: Vehicle) Go (v: REAL); (* Ехать со скоростью v *)
BEGIN
  vh.v := v
  …
END Go;

PROCEDURE Test;
  VAR v: Vehicle;
BEGIN
  v.Go(15.9)
END Test;
```

Мы видим, что самый обычный тип-запись используется нами в стиле ООП. Связанная процедура *Go* – пример того, как определяются методы в КП. Объект, для которого будет вызы-

ваться процедура, оформляется как явный параметр и записывается в скобках перед именем процедуры. Опять же, такая форма обращения, в сравнении с неявными *this* и *self* в C++ и Object Pascal и их областями видимости, делает код прозрачным, а обучение новичков работе с ООП - гораздо более легким. Запись Vehicle расширена быть не может, для этого потребуется явно указать *EXTENSIBLE RECORD*. Поэтому ООП для обычных, нерасширяемых записей, не вносит никаких накладных расходов – у них нет виртуальной таблицы, и связанные процедуры ничем не отличаются от обычных кроме более удобной формы записи. А вот как выглядит иерархия расширяемых типов:

```
TYPE
  Vehicle = ABSTRACT RECORD (* Транспортное средство *)
    x, y, v: REAL;
  END;
  Car = EXTENSIBLE RECORD (Vehicle) (* Автомобиль *)
    wheels, doors: INTEGER
  END;
  Bus = RECORD (Car) (* Автобус *)
    number: INTEGER
  END;
```

При объявлении типов-записей могут использоваться дополнительные модификаторы: *EXTENSIBLE* (расширяемый), *ABSTRACT* (расширяемый абстрактный, то есть, нельзя создать экземпляры), *LIMITED* (не может быть создан за пределами модуля, в котором объявлен. Внешние модули могут работать с экземплярами типа только через уже выделенные указатели).

Указатель (или *VAR*-параметр процедуры) базового типа можно привести к расширенному с помощью *преобразования типа*, после чего с ним можно работать как с расширенным:

```
PROCEDURE PolymorthDemo (VAR v: Vehicle);
BEGIN
  v(Bus).number := 12
END PolymorthDemo;
```

Однако подсистема времени выполнения проверит динамический тип переменной *v* и допустит данное преобразование только в том случае, если в процедуру действительно был передан параметр типа *Bus*. В противном случае будет сгенерировано исключение.

Динамический контроль типов. Динамическая типизация в КП поддерживаются на основе механизма тегов – каждый экземпляр записи имеет тег своего типа, по которому его можно однозначно идентифицировать и получить о нем всю информацию. Такие проверки выполняет сама подсистема времени выполнения, например, при преобразованиях типов. Однако не менее полезно явное использование программистом этой информации (так называемой RTTI – run-time type information). Простейшая возможность - проверка принадлежности переменной какому-либо типу: *x IS SomeType*. Есть и более удобная форма *IS*, совмещающая проверку с приведением - оператор *WITH*, но на нем мы останавливаться не будем. Конкретные среды КП обеспечивают весь спектр возможностей *метапрограммирования*: позволяют во время выполнения анализировать содержимое подключаемых модулей, работать с неизвестными на этапе компиляции типами, переменными и процедурами и т.д. – эти возможности обычно предоставляются библиотеками.

Реализации языка. Реализаций КП не так много, как его предшественника Оберона-2 (для которого создано несколько десятков компиляторов). Промышленным качеством обладают две их них: BlackBox Component Builder и Gardens Point Component Pascal (GPCP). GPCP – разработка Австралийского технологического университета. Компилирует как для платформы .NET, так и для Java Virtual Machine. Интегрируется со средами разработки Visual Studio и Eclipse. См. [4].

III. Среда разработки BlackBox Component Builder

BlackBox [3] – промышленная среда разработки от Oberon Microsystems. Отличается исключительной гибкостью, стабильностью работы и нетребовательностью к ресурсам. Легла в основу ОС реального времени JBed. Активно используется швейцарскими и российскими ядерными физиками (CERN и ИЯФ РАН, [5]). Существуют версии для Windows и MacOS. В настоящее время в ИЯФ РАН разрабатывается версия под Linux, в том числе для распределенных вы-

числений на кластерах. Активно продвигается в школьном образовании как замена общераспространенному Turbo Pascal (см. проект Информатика-21, [5]). По потенциальным возможностям BlackBox можно сравнивать с .NET, он воплощает те же передовые концепции программной инженерии. Однако, если BlackBox прост и изящен, то .NET продолжает традиции громоздких и перенасыщенных второстепенными возможностями систем. Сам за себя говорит тот факт, что размер среды BlackBox в распакованном виде - 30 Мб, а без интерфейсов автоматизации к MS Office - 17 Мб. Среда устанавливается простым копированием. Итак, что же входит в эти 17 Мб?

1. Компилятор языка Компонентный Паскаль, дающий на выходе быстрый код, не уступающий компиляторам Си.
2. Ядро времени выполнения: динамический контроль типов, сборщик мусора, загрузчик модулей, выполняющий загрузку и связывание "на лету".
3. Компилятор Direct-To-COM - получил в 2000 году на выставке CeBIT приз за технологическое совершенство. Позволяет разрабатывать сервера и клиенты COM на Компонентном Паскале. Полностью скрывает внутренние механизмы COM за системой типов языка. Уникальная особенность: для COM-объектов полностью поддерживается сборка мусора, избавляющая программиста от необходимости вести подсчет ссылок.
4. Система метапрограммирования - позволяет во время выполнения символически обращаться к модулям и разбирать их содержимое, связывать процедуры, размещать переменные и т.п. Возможности аналогичны соответствующим библиотекам .NET.
5. Среда графического интерфейса, построенная на концепции составных документов (расширении OLE). Документы динамически связываются с программными модулями. Документами в BlackBox являются:
   а) визуальные формы и диалоги. Элементы управления связываются с процедурами и переменными-интеракторами "на лету".
   б) текстовые документы, в том числе тексты программ (в текст могут быть встроены и формы, и элементы управления).
6. Возможности текстовой подсистемы BlackBox эквивалентны мощным текстовым процессорам. Важно, что они легко доступны во время выполнения. Это незаменимо для бизнес-приложений, которым необходимо генерировать отчеты.
7. Подсистема SQL обеспечивает доступ к базам данных через ODBC. Можно напрямую работать с переменными-записями в SQL-запросах (SQL-препроцессор основан на мета-механизме среды).
8. Исходные коды системы.
9. Документация, в настоящий момент полностью переведенная участниками проекта Информатика-21 на русский язык.

Приложения, созданные на BlackBox, можно поставлять как в виде отдельных загружаемых модулей с exe-файлом - ядром (80 Кб) либо в статически скомпонованном виде. Можно исключить все ненужные подсистемы (даже ядро со сборщиком мусора) и получить традиционные EXE и DLL размером от 30 Кб. Существует также инструментарий, позволяющий разрабатывать на BlackBox Web-приложения.

Интересная особенность: в BlackBox отсутствует пошаговый отладчик. Его использование в Оберон-сообществе признано плохим стилем программирования. Необходимости в нем практически нет, т.к. Оберон-парадигма делает упор на написание изначально правильного кода. На практике этап отладки значительно сокращается (вместо 40% времени разработки в случае C++ он занимает около 10% для КП) . Вместо пошагового отладчика предлагается так называемый дамп-отладчик, позволяющий после сбоя просмотреть состояние памяти, удобно перемещаясь между статическими и динамическими структурами. Уникальной особенностью является умение дамп-отладчика работать также и с объектами COM. Также в среду входит статистический профилировщик, позволяющий выявлять в программах узкие места по быстродействию.

Литература

1. Szyperski C. Component Software. Beyond Object-Oriented Programming, Addison Wesley Longman, 1998.
2. Pfister C., Component Software − Русский перевод: Ермаков И.Е., см. [3], раздел “Статьи”.

3. http://blackbox.metasystems.ru – BlackBox в российской ИТ-индустрии и образовании.
4. http://www.plas.fit.qut.edu.au – компилятор GPCP.
5. http://www.inr.ac.ru/~info21 – проект Информатика-21.

BlackBox в образовании. Проект Информатика-21

Уже несколько лет в ряде российских ВУЗов и школ BlackBox применяется для обучения программированию. В сентябре 2001 года стартовал проект Информатика-21 [1]. «Его главная цель — координация усилий реальных специалистов науки, образования, аэрокосмической промышленности и ИТ-индустрии на постсоветском пространстве по созданию единой современной системы преподавания фундаментальных основ программирования, опирающейся на наши уникальные образовательные и математические традиции». В настоящий момент участниками проекта Информатика-21, использующими BlackBox в образовательном процессе, являются, в порядке присоединения:

- физический ф-т МГУ, спецкурс "Введение в современное программирование" (Ф.В. Ткачев);
- лицей Научного центра РАН в г. Троицк, эксперимент по преподаванию программирования в формате обычного школьного курса информатики (Н.П. Кучер);
- межшкольная группа «Лидер» для одаренных детей при ГорУО г. Стрежевого Томской области (А.И. Попков);
- Ошский технологический университет (г. Ош, Кыргызская Республика), новаторская программа систематического обучения программированию на основе ETH Oberon под ОС Linux с использованием проектов open source (Кубанычбек Тажмамат уулу);
- Орловский государственный университет, экспериментальный курс "Элементы абстрактной и компьютерной алгебры" для студентов учительских специальностей ОГУ (Б.В Рюмшин);
- лицей №1 им. М.В. Ломоносова г. Орла, спецкурс «Программирование и дискретная математика»;
- Московский авиационный институт (МАИ), факультет прикладной математики и физики. В рамках курса "Системное и прикладное обеспечение", в теме "Методы комплиляции" изучаются Оберон, Компонентный Паскаль и среда BlackBox как образец системы программирования;
- широкомасштабный эксперимент по внедрению Компонентного Паскаля в средних школах Витебской области (Республика Беларусь). Руководитель проекта — А.Б. Кондратович, нач. отдела Витебского областного института повышения квалификации учителей.

Авторский спецкурс «Программирование и дискретная математика» в лицее №1 г. Орла

Ермаков И.Е.

В сентябре 2004 года, я пришел в лицей №1 г.Орла, чтобы вести спецкурс информатики для одаренных школьников. Лицей №1 - одно из лучших учебных заведений г. Орла, ученики которого стабильно занимают призовые места на олимпиадах области, как по естественным, так и по гуманитарным наукам. Не имея возможности и желания брать на себя общеобразовательный курс информатики, я поставил своей целью создать спецкурс для 9-11 классов, который мог бы удовлетворить интерес одаренных школьников к информатике и подготовить их к дальнейшему получению профессии, связанной с программированием.

Моим глубоким убеждением является то, что преподавание любого предмета необходимо строить от основ, от общего к конкретному, а не наоборот. Поэтому я отношусь неодобрительно к попыткам "облегчить" школьный курс исключением "сложных вещей" и добавлением на их место "практически полезных навыков". Любой курс программирования, построенный по принципу "для чайников", облегчая первое знакомство с предметом, на порядок затрудняет серьезное его освоение, по сути, обрекает учащегося оставаться этим самым "чайником". Примером является курс Угриновича, построенный на использовании среды Visual Basic. В этом курсе основной акцент сделан на построении визуальных интерфейсов в среде быстрой разработки. Однако, чтобы эффективно использовать такие среды, необходимо знать хотя бы основы теории, такие, как принципы ООП. Понять суть, преимущества и недостатки ООП невозможно без знакомства со структурным императивным подходом и хотя бы небольшого опыта в написании программ "старыми" методами. Изучение же ООП по языку, который реализует его крайне неполноценно и при этом стимулирует написание программ, нарушающих основные структурные

принципы, представляется большой ошибкой. Алгоритмика же в подобных курсах почти не представлена, то есть, программирование сводится к созданию интерактивной графики. Хотя бы начального понимания того, как технически поддерживается GUI, и какие уровни абстракции от оборудования были выстроены для того, чтобы сделать возможным "программирование для чайников", здесь также не прививается.

Альтернативой "легкому" могли стать два пути: технический и абстрактный. В первом случае можно идти от оборудования, поэтапно показывая, как слои абстракции позволяют решать с помощью компьютера все более сложные задачи. Второй путь делает ставку на использование абстракций языков высокого уровня для решения прикладных задач (например, из области математики, физики), учит использовать компьютер в качестве инструмента познания, вскрывает связи информатики с математикой и естественными науками. У каждого пути есть свои преимущества. В 2004 году мною был выбран первый путь, во многом из-за того, что сам я по складу больше инженер, нежели математик, и занимался разработкой в основном системного ПО. Замечу, что после долгого использования Delphi я перешел на C++, и он некоторое время казался мне удачным и удобным для использования. Однако знакомство с такими языками, как Ada и Oberon, радикально изменило мои предпочтения.

В качестве языка программирования использовался Borland Pascal 7.0. Курс включил в себя основательное изучение архитектуры IBM PC и ее программирования в реальном режиме. При знакомстве со структурами данных акцент, с одной стороны, делался на их абстрактном смысле, с другой - на их физическом представлении в памяти. Понимание того, что прежде чем учить принудительно *нарушать типизацию*, полагаясь на особенности архитектуры, следует научить ее *правильно использовать*, пришло ко мне не сразу. Изучение таких рудиментов, как программирование клавиатуры и экрана через BIOS, теперь уже представляется совершенно излишним, хотя на этом была построена значительная часть практикума, завершавшегося созданием игры "тетрис". Тем не менее, широкому набору алгоритмов сортировки, конечным автоматам как механизму разбора строк, работе с динамическими структурами было уделено внимание. Однако сочетать в рамках спецкурса науку и технику без ущерба для обоих оказалось невозможным. Тем более, стало ясным, что технические нюансы способными школьниками воспринимаются достаточно легко и, несомненно, могут быть освоены при необходимости самостоятельно, а вот теория вызывает либо трудности, либо нежелание ей следовать.

Стала очевидной необходимость кардинально перестроить курс. Основным его содержанием должно стать программирование *в абстракциях*, направленное на решение *алгоритмических задач*.

После знакомства с BlackBox мне стало ясно, что эта среда может стать прекрасной основой для школьного образования. Кроме того, что в ее основе лежит истинно алгоритмический язык - Компонентный Паскаль, она позволяет создавать современное ПО так же легко, как распространенные среды быстрой разработки, но в отличие от них не побуждает прикладного программиста строить структурно неграмотные программы. Также среда, которая обладает уникальной расширяемостью и предоставляется в исходных текстах, не закрывает наглухо от любознательного программиста детали низкоуровневой реализации и позволяет поэтапно углубляться в них. Исходные коды, написанные в безупречном стиле, достойном Вирта и Дейкстры, являются "живым" примером того, как следует правильно создавать большие программные комплексы. Их изучение может принести гораздо больше пользы, чем чтение многих книг "для профессионалов". Как только Оберон получит распространение в высшем образовании и программной индустрии, то неоспоримым преимуществом явится то, что студент и затем разработчик сможет использовать тот же язык и среду, с которых начинал обучение, без необходимости осваивать множество откровенно неудачных инструментов, используемых в бизнес-программировании.

На 2005-2006 учебный год мною поставлена цель апробировать среду в школьном образовании и создать полноценный курс "Программирование и дискретная математика для школьников". Вторая составляющая курса даст не только теоретическое понимание многих проблем, встающих перед программистом, но и алгоритмические задачи для обучения программированию на языке высокого уровня. Были включены разделы математической логики, комбинаторики и теории графов. Следует отметить, что решение многих задач, требующих динамиче-

ского распределения памяти, в BlackBox значительно облегчается наличием автоматического сборщика мусора.

В сентябре 2005 года мной был завершен перевод на русский язык тех частей документации, которые не были переведены участниками проекта "Информатика-21". Наличие русского перевода и возможность использовать русские идентификаторы становится еще одним преимуществом, поскольку не многие среды разработки могут этим похвастать. Польза от англоязычных сред в плане изучения технического английского достаточно относительна. Большинство учащихся не стремится совершенствовать знание языка, а просто-напросто игнорирует англоязычные источники инофрмации. В конце концов, национальная школа информатики должна быть действительно национальной, и преимущество работы с родным языком неоспоримо.

По прошествии шести месяцев уже можно подвести некоторые итоги. BlackBox идеально подошел для преподавания программирования в школе. Стройный и прозрачный язык не содержит никаких подводных камней или непоследовательностей, которые усложнили бы восприятие материала. При том, что Компонентный Паскаль – новейший язык последнего поколения, он оказывается более простым для изучения, чем классический Паскаль. Сама среда позволяет легко создавать графические интерфейсы, **не требуя для этого знания ООП** – это позволяет использовать для ввода-вывода оконные интерфейсы начиная с самых первых занятий. В то же время концепции интерфейса BlackBox предполагают очень четкое разграничение пользовательского интерфейса и основной логики программы. Фактически, программный модуль ничего не знает о том, что к нему будет подключена графическая форма, все связывание и обработку событий выполняет динамически среда. Это позволяет избежать свойственного Visual Basic и Delphi порочного стиля, который так легко усваивают самоучки: «кидание контролов» на форму и затем размазывание кода по обработчикам событий. Удачная концепция ООП как расширения структурного программирования, без введения ненужных понятий-синонимов класса и объекта, сделала возможным объяснение ООП в рамках одного занятия.

Учащимися 11 класса А.Кимом, М.Щербиной и А.Пионтковским был реализован творческий проект на тему трехмерной машинной графики «Система топографического моделирования городской местности». Учащимися были самостоятельно изучены методы трехмерных построений, на основе чего написана система, строящая изометрическую проекцию городских зданий по описанию на языке XML. Библиотеки 3D-графики не использовались, расчет и построение ведется по собственным алгоритмам. Программа была представлена в Москве на Конференции одаренных школьников Intel-Авангард-2006. На городской олимпиаде 2006 года учащиеся лицея заняли 1 место среди 11 классов (А. Ким) и 2 место среди 10 классов (А. Дырив).

Нами разрабатывается учебный комплекс, состоящий из 3 блоков:

1) Начальный курс алгоритмики на основе собственной алгоритмической среды ШАР (Школьный Алгоритмический Расширяемый). В качестве алгоритмического языка используется подмножество Компонентного Паскаля, в точности соответствующее языку КуМир. Поскольку ШАР создается в среде BlackBox, он является расширяемым, то есть, пользователи смогут дописывать свои модули – исполнители и алгоритмические игры. Проект будет распространяться в исходных кодах, что должно привлечь к его развитию заинтересованную общественность. Основную часть работы по созданию ШАРа будут выполнять старшеклассники лицея №1.

2) Базовый курс «Программирование и дискретная математика» - основа уже заложена в 2004-2006 годах;

3) Курс для 11-классников «Основы программного конструирования». Цель – знакомство с основными аспектами программирования «в большом», формирование инженерного мышления, способности проектировать и реализовывать программные системы.

Ссылки

1. http://www.inr.ac.ru/~info21 – сайт проекта Информатика-21
2. http://blackbox.metasystems.ru – BlackBox в России
3. http://lyceum.metasystems.ru – сайт лицея №1 г. Орла
4. http://webring.metasystems.ru – web-кольцо «Орел: Наука, техника, образование»

Проект "1С:Образование" - критический взгляд

Ермаков И.Е.

Недавно имел возможность ознакомиться с диском "Вычислительная математика и программирование - 10-11 классы", который выпущен фирмой 1С совместно с Microsoft. На нем стоит гриф "Подготовлено по заказу Министерства образования РФ". Как написано в readme-файле, "образовательный комплекс выполнен в рамках конкурсов по федеральной целевой программе "Развитие единой образовательной информационной среды (2001-2005 гг.)", проводимых Министерством образования России, и конкурса, проводимого Национальным фондом подготовки кадров (на заем Всемирного банка), и успешно прошли апробацию в передовых школах. Большинство продуктов серии "1С:Школа" получили гриф "Допущено Министерством Образования РФ в качестве учебного пособия". Эти диски сейчас поставляются во все школы России. Пришел такой и в лицей №1 г. Орла, в котором идет наш спецкурс "Современное программирование и дискретная математика" на Компонентном Паскале.

Первое, что бросилось в глаза и вызвало интерес, - это само название диска. При том, что школьная информатика сегодня выхолощена почти до пользовательского курса, формулировка "Вычислительная математика и программирование" звучала оптимистично. Сейчас в школьном физико-математическом образовании случились некоторые положительные сдвиги - возвращена в школьный курс комбинаторика и классическая теория вероятностей. Это открывает простор для преподавания программирования на новом уровне, в тесной связи с математикой. Поэтому диск от 1С сразу был отложен для внимательного изучения.

1. Платформа 1С "Образование"

Решая поставленную Министерством задачу - создать школьный образовательный пакет, 1С пошла по уже накатанной колее. Была сделана общая платформа "Образование", для которой выпускаются конфигурации по отдельным предметам.

Также на диске поставляется управленческая программа "Хронограф-Школа", конфигурация для 1С"Предприятие", которая предназначена для автоматизации работы директора, завхоза и классных руководителей (нам кажется, что эта конфигурация не приживется в школах, хотя бы в силу нехватки времени и желания у учителей и директоров для освоения громоздкой платформы "Предприятие". Несомненно, нужно было нечто более легкое и доступное).

В программе "Образование" 1С попыталась создать универсальную систему управления содержанием, которая позволяет учителям накапливать, создавать и упорядочивать учебные и методические материалы. Система многопользовательская и сетевая, то есть, работать с ней может несколько учителей и множество учеников со своих рабочих мест. Как написано в документации, при создании был использован "винегрет" технологий: Macromedia Flash - для презентаций и графики, СУБД Firebird, Java Beans, JDBC, SOAP. Тем не менее, работает все достаточно стабильно.

При создании собственных презентаций программа ограничивает учителя заранее созданными шаблонами кадров, то есть, до возможностей MS PowerPoint, который уже прижился в школах, ей далеко.

Прграммисты выполнили свою работу хорошо, и технологически платформа вышла неплохой. На этом положительные впечатления заканчиваются. То, ради чего и создавалась платформа, - содержание, может служить примером еще одной неудачной инновации в школьное образование. Не то, чтобы наполнение выполнено халтурно, - нет, свой замысел авторы воплотили аккуратно и педантично. Но сам замысел явно не учитывал наличие учителей и учеников, которым придется с этим курсом работать.

2. Образовательные материалы

Подход к представлению и оформлению материалов вызывает неприятное ощущение. Слово "мультимедиа" создателями множества подобных курсов трактуется удивительно просто: яркое и утомляющее глаза цветовое оформление, смазанные презентации на градиентном фоне, много картинок, сопровождаемых маленькими блоками текста, которые грустным и задумчивым голосом читаются диктором. (Авторам явно невдомек, что при подаче научного материала лучше всего воспринимаются контрастные черно-белые схемы из прямых линий. Для объяснения же сложных моментов вообще нет ничего лучше доски и мела. Спросите у систем-

ных архитекторов ПО - что они используют при обсуждении новой разработки? Мультимедиа-проектор? Если не доску, то бумагу и карандаш.)

Информационная насыщенность мультимедийного курса на порядок ниже даже не очень хорошего бумажного учебника. Если собрать весь текст, например, по вычислительной математике, едва ли наберется два десятка полноценных страниц. Хорошему учителю в сильном классе будет очень трудно найти применение этим курсам. Разве что подобрать отдельные полезные фрагменты перед уроком. Единственное применение "образовательному комплексу" - заменить плохого учителя и хоть чем-то занять учеников на уроке. Не видел, но боюсь даже представить себе гуманитарные курсы - историю или литературу...

3. "Вычислительная математика и программирование"

Итак, что из себя представляет "образовательный комплекс" по программированию? Содержательно в него входят четыре блока:

- "Вычислительная математика"
- "Алгоритмика"
- "Среда Visual Basic .NET"
- "Среда Turbo Pascal"
- "Среда Borland Delphi"
- "Платформа 1С:Предприятие".

Каждый блок сделан в двух вариантах: А - облегченный и Б - "нормальный" (кавычки во втором случае мы поставили намеренно). Далее эти блоки скомпонованы в четыре курса (каждый курс - в двух версиях: А и Б):

- Гуманитарный (Алгоритмика; Visual Basic)
- Социально-экономический (Алгоритмика; 1С:Предприятие; Visual Basic)
- Естественно-математический (Вычислительная математика; Алгоритмика; Turbo Pascal для облегченного курса и Visual Basic для "нормального").
- Информационно-технологический (Вычислительная математика; Алгоритмика; Turbo Pascal; Visual Basic для облегченного курса и Borland Delphi для "нормального").

Что можно сказать относительно этих материалов? О блоке "Вычислительная математика" - ничего особенного. Кратко изложены особенности приближенных вычислений с вещественными числами, понятие погрешности вычислений, построение графиков функций, решение уравнения $f(x) = 0$ методом половинного деления, численное интегрирование. Много цветной графики, претендующей на наглядность, текстовый материал достаточно беден. Предполагается, что ученики уже познакомились до 10 класса с языком программирования (QBasic).

Блок "Алгоритмика" содержит очень много воды. Поражает, с каким упорством по сей день преподаются блок-схемы, окончательно устаревшие в 70-х годах с рождением структурного программирования. Мало-мальски сложная программа, содержащая более 10 строк, в виде блок-схемы читается гораздо хуже, чем в виде правильно оформленного листинга на структурном языке (например, Паскале). Блок-схемы - это "костыли" для неструктурных языков, которые сегодня уже не используются.

Интересно звучат подобные перлы: "Язык блок-схем настолько четок, что исполнитель, получивший блок-схему алгоритма, ни в каких дополнительных разъяснениях не нуждается. Кроме естественного языка и языка блок-схем, существует еще один способ записи алгоритма – это запись на языке программирования."

"На практике представляют интерес те методы сортировки, которые позволяют экономно использовать оперативную память компьютера при упорядочивании больших массивов и в то же время быстро достигать конечного результата. Это метод выбора, метод вставки и метод обмена (иначе называемый методом пузырька)". В школе преподавание алгоритмов сортировки действительно следует начинать с этих алгоритмов, но такая явная глупость, как утверждение об их практическом интересе в связи быстротой, можно приписать только вопиющей некомпетентности. Нам, например, не известен алгоритм сортировки, более медленный и бесполезный для больших сортировок, нежели метод пузырька... Может быть, честь его изобретения принадлежит авторам курса от 1С?

Краткий обзор языков программирования завершается схемой сегодняшней ситуации с "наиболее распространенными" языками программирования: BASIC => Visual Basic Pascal =>

Borland Delphi Составление алгоритмов на естественном языке => 1С Предприятие. Как говорится, без комментариев...

Блок "Turbo Pascal" содержит обычный курс программирования в этой среде, 14 уроков. Описаны основные операторы (условный, циклов), работа с различными типами. И заключительный штрих - работа с графикой (куда без нее?) и вызовы процедур. Ни словом ни затронуты принципы структурного программирования, записи, работа с файлами. Да и зачем? Школьников ждут гораздо более важные вещи - рисование окошек в Visual Basic и конфигурирование 1С.

Остальные блоки - Visual Basic, Delphi и 1С построены по принципу "щелкните мышкой туда и туда, посмотрите, что получится, запомните, но не думайте, почему это так", потому что научить школьников работать в профессиональных средах за 12 уроков просто невозможно.

На первых 2 урока курса по Visual Basic (не считая самого начала, посвященного открытию среды и компиляции) излагаются: объектно-ориентированный подход и событийно-управляемая модель программирования. Затем описывается работа с формами, и только на 15 уроке изучается оператор присваивания. Как могут понять и научиться использовать ООП школьники, не знакомые со структурным программированием? Не говорим уже о событийно-управляемом программировании... (Видимо, авторы курса просто убеждены, что это - изобретение Microsoft, впервые примененное в Visual Basic.) За этими двумя формулировками стоят десятилетия исследований и технического поиска. Объяснить ООП на первом занятии, может быть, и можно, в самом объектном взгляде на мир нет ничего сложного, но невозможно понять, какую роль оно играет в создании технических систем, каковы его преимущества и недостатки, если перед этим не написать хотя бы несколько небольших программ в структурном стиле. Мы в нашем спецкурсе в Орле даем ООП только в 11 классе, после того, как за спиной у учащихся уже были такие самостоятельные проекты, как простая база данных, игра "Тетрис" и т.п.

Блок по Delphi выстроен аналогично. Единственное отличие в том, что венчают его три урока по работе с базами данных: "Программирование модулей. управляющих базами данных". Нет спору, навык неплохой, но как научить ему за три урока школьников, в предыдущих знаниях которых (благодаря тому же курсу от 1С) к тому есть большие пробелы, если те же три урока придется посвятить общему знакомству с отраслью СУБД и основными понятиеми реляционной модели?

Собственно, обучения построению программ и решению технических задач в курсе вы не найдете. Это - одна большая презентация, за которой ничего нет. Презентация либо на слайдах, либо на примерах, когда учащемуся говорят: "сделай то и это и посмотри, что получилось!" Такой подход не приводил к успеху даже в относительно простом TurboPascal, не говоря уже о современных средах разработки.

Что можно сказать про попытки преподавать в школе Visual Basic и Delphi? На VB даже останавливаться не будем - если сравнивать с Delphi, то у него нет ни одного весомого преимущества. Зато среда и язык прививают плохой стиль, построены совершенно сумбурно. Сколько потом усилий придется потратить в ВУЗах на инженерных специальностях, чтобы переучить юных "рисовальщиков окошек"? Единственный способ переучить пользователя (по другому не назовешь) Visual Basic на профессионального программиста - заставить забыть все, чему научили в школе.

Что касается Delphi, то, при всем нашем уважении к этой замечательной среде, надо сказать, что для начального обучения программированию она не особенно подходит. Единственный способ - начинать с написания консольных приложений, заменив на этом этапе старый TurboPascal. В среде Delphi есть большая концептуальная ошибка - слишком тесная связь логики и интерфейса разрабатываемых программ. Это проявляется, например, в наличии большого количества так называемых невизуальных компонентов - объектов, которые имеют системное назначение, но при этом являются наследниками TComponent и кладутся на формы так же, как, например, кнопки. Но, скажите мне, какое отношение имеет сокетное TCP/IP-соединение к пользовательскому интерфейсу? Это все равно, что монтировать блок развертки телевизора на его переднюю панель...

Для опытного Delphi-программиста это проблемы не представляет, он просто создает те же самые объекты динамически, ему никогда не взбредет в голову кидать их на форму. Но среда почему-то изначально пропагандировала именно такой, по сути своей, неправильный под-

ход. А если еще авторы учебника, вместо того, чтобы подвести к разработке графических приложений поэтапно, только после написания серьезных учебных программ без GUI, в структурном стиле, и вместо объяснения, "что такое хорошо и что такое плохо", сразу же начинают показывать, где надо щелкать мышкой и в какой обработчик события надо вписывать полезный код? А как потом отучить бывших школьников, которые все "ухватили на лету", писать основной код в обработчиках событий и приучить разносить логику программы и ее интерфейс по разным модулям? Спортсмены и учителя музыки знают, что если сразу не научить правильной технике, приобретенные даже способным самоучкой ошибочные привычки могут остаться на всю жизнь.

Но главная проблема, по которой эти среды графической разработки нельзя брать для начального обучения программированию, заключается в том, что даже для самой простой оконной программы требуется знание ООП. А, как мы уже сказали, преподавать ООП без предварительного формирования навыков структурного программирования - безумие. Можно ли решить эту проблему? Действительно, любая современная реализация GUI построена на ООП. Однако при внутреннем объектном устройстве системы можно предложить гораздо более простой подход для того, кто создает на ее основе программы. Такой подход, например, предлагает среда BlackBox.

В целом про курс можно сказать одно: не оставляет ощущение, что его делали бухгалтера из финансового отдела 1С, а не инженеры и тем более не математики.

4. Сложность или сумбурность?

В последние годы завоевал признание тезис, что школьное образование классического типа слишком сложно, и его надо упрощать. При этом происходит парадоксальная вещь: при упрощении и введении новых, "приближенных к практике", "доступных" программ и учебников, сложность восприятия курса для нормального среднего школьника, у которого развита способность анализировать и делать выводы, возрастает. Все больше способных, иногда очень способных, учеников перестают учиться вообще. С чем это связано?

На самом деле, сложность для понимания бывает двух видов: сложность концептуальная, заставляющая человека приложить серьезные усилия для овладения материалом, и сложность от сумбурности, от отсутствия логичности и последовательности как таковой. Многие школьные и университетские курсы сегодня являются примером последней. На самом деле нормальной способностью и стремлением человеческого сознания является обобщение информации, желание уловить самую суть, пусть даже она будет и тяжелее для восприятия, чем набор конкретных навыков. Если же в процессе обучения это естественное стремление не только не удовлетворяется, но подавляется лавиной примитивщины, которая не дает за деревьями увидеть леса, то нормальное сознание либо деградирует, либо отказывается воспринимать подаваемый материал вообще (этакий защитный механизм). Чего стоят школьные программы физики, которые предлагают запоминать набор второстепенных формул и то, куда и в какой момент их следует "приткнуть", вместо того, чтобы научить сначала анализировать задачу качественно, затем на основе базовых формул составлять систему уравнений, описывающую ее количественно, и затем грамотно решать эту систему...

Завершим затянувшиеся размышления высказыванием известного ижевского ученого и преподавателся Н.Н. Непейводы, сказанной, правда, по поводу университетского обучения, но в не меньшей степени применимой к школьному: "Чтобы подготовить хороших специалистов (программистов), нужно готовить аналитиков. Чтобы попасть в цель, нужно целиться выше цели. Все равно не более 20% [учащихся] станут аналитиками, но оставшиеся станут специалистами именно того класса, который нужен российским фирмам".