

НЕКОТОРЫЕ МЕТОДИЧЕСКИЕ ПРИНЦИПЫ ПРЕПОДАВАНИЯ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Н. А. Аленский

*Белорусский государственный университет,
пр. Независимости, 4, 220030, г. Минск, Беларусь, alensky@bsu.by*

Основываясь на многолетнем опыте чтения лекций и проведения различных видов практики на механико-математическом факультете Белгосуниверситета и других учебных заведениях, на конкретных примерах предлагаются следующие методические принципы преподавания алгоритмизации и методов программирования: многоуровневости, сравнения, параллельности, предварительной мотивации и повторения.

Ключевые слова: методы программирования; обучение программированию; методика обучения; принципы обучения.

Несмотря на то, что тематическая линия «Основы алгоритмизации и программирования» школьного предмета «Информатика» изучается шесть лет и изданы хорошие учебники и (или) электронные материалы для них, университетскую дисциплину «Методы программирования» большинство студентов осваивают с трудом. В докладе предлагаются некоторые проверенные на практике преподавания методические принципы, которые позволяют студентам первого курса эффективнее осваивать эту дисциплину.

Принцип многоуровневости связан с тем, что в алгоритмизации и программировании все темы, элементы языка тесно переплетены между собой. Поэтому некоторую трудность представляет определение порядка изучения понятий и конструкций языка, приёмов и методов программирования. Ряд учебников построен таким образом, что сначала рассматриваются элементы выбранного языка программирования (переменные, все их типы и операции над ними, правила построения выражений и т. д.), а потом последовательно изучаются операторы и все их возможности, методы и технологии программирования. При этом, как правило, сразу в одном параграфе приводятся все возможности изучаемого элемента языка, например, все строковые или графические функции, хотя некоторые из них редко используются или сложные, и на начальном этапе можно изучить часто используемые и более простые возможности, которых достаточно для решения обширного класса задач.

Усовершенствование такой общепринятой методики покажу на примере изучения функций языка программирования C++, без которых невозможно освоить современное объектно-ориентированное и визуальное программирование. Студентам эта тема в первом семестре дается нелегко, так как в школьной информатике ограничиваются только использованием стандартных встроенных процедур и функций на базовом уровне, а разрабатывать их по учебному плану должны учить только на повышенном уровне в старших классах, что пока мало где реализовано.

На *начальном этапе* (основы, введение в функции) изучения этого вопроса, используя **принцип сравнения**, по аналогии с учебным школьным языком Паскаль основное внимание надо обратить на то, чтобы студенты поняли, чем отличается написание и вызов функции, отличной от *void* (функция на *Паскале*) и функции типа *void* (процедура на *Паскале*). Для этого надо ответить на следующие вопросы; назначение функции; что дано (входные параметры); результат функции, то есть выходные параметры, которых может не быть; в зависимости от этого, как записать заголовок; алгоритм и особенности текста функции; как вызвать функцию. Результаты функции типа *void* возвращаем с помощью параметров ссылочного типа, которые

для студентов легче, чем параметры-указатели. На старте желательно ограничиться одной функцией (частью программы) в проекте, и предлагается оформлять её без прототипа, как встроенную. Для того, чтобы больше внимания уделить функциям, эти вопросы предлагается изучать раньше, начиная, например, с третьей недели, а операции, операторы и некоторые другие более простые вопросы можно рассматривать одновременно с функциями, используя **принцип параллельности** изучения тем.

После приобретения простых навыков написания и использования функций можно осваивать **второй, более сложный уровень (основной)**. На этом этапе изучаем правила описания функций с прототипом и сравниваем их со встроенными; показываем, как функцию с несколькими результатами оформить как отличную от *void*; как в одном проекте составить и использовать несколько функций, которые вызываются не обязательно из *main*; а также такие возможности функций, как их перегрузка, параметры по умолчанию, простые рекурсивные алгоритмы и сравнение их с не рекурсивными, запрограммированными с помощью циклов. Все эти вопросы можно изучать во второй половине первого семестра.

На **третьем уровне** во втором семестре параллельно с другими темами продолжаем изучать следующие профессиональные сложные для студентов вопросы функций: параметры-указатели; как функцию для работы с одномерным массивом использовать при работе с динамической (обе размерности – константы), частично динамической (одна размерность – константа) и динамической матрицей; как одномерный и двумерный массивы, строки, структуры, объекты и их поля передать в функцию и (или) вернуть из неё; указатели на функции и их использование; сложная рекурсия.

В докладе использование этого принципа рассматривается и на примере изучения других объемных и сложных тем, таких, как указатели, двумерные массивы, классы и некоторые другие.

Принцип **многоуровневости** можно использовать и при подготовке и выдачи вариантов заданий разного уровня сложности как для различных видов практики, так и для контроля и оценки знаний [2].

Многолетний опыт чтения лекций и проведения различных видов практики показал, что в качестве основного принципа изучения алгоритмизации и программирования эффективно использовать **принцип предварительной мотивации** элементов выбранного языка программирования. Это означает, что почти любую тему за редким исключением лучше изучать «от частного к общему», а не наоборот, по следующей схеме:

1. Наглядная простая задача, которая требует для своего решения введения этого элемента, т. е. показывается необходимость изучаемого элемента.
2. Пример использования нового элемента при решении этой задачи.
3. Общий вид конструкции, её формальное описание.
4. Общие примеры на закрепление.
5. Выполнение упражнений и индивидуальных заданий для отладки программ [1].

При этом пункты с первого по третий лучше рассмотреть в форме лекции, а четвертый и пятый – на практических занятиях. При этом простые теоретические сведения желательно давать в сокращённом виде и основные методы, приёмы программирования должны изучаться на простых наглядных примерах. Теоретические вопросы более подробно излагаются в электронном виде, например, в образовательном портале **Moodle**. Следуя этому принципу, не обязательно всегда соблюдать одинаковый порядок форм занятий: лекция – практические занятия – лабораторные работы. При изучении некоторых тем, например, ввод-вывод данных, эффективнее сначала решить задачу с использованием компьютера или выполнить упражнения без ЭВМ, а потом на лекции обобщить, подвести итоги, обратив особое внимание на наиболее сложные вопросы и объяснить новые дополнительные возможности.

При изучении многих тем следует руководствоваться классическим **принципом сравнения**, который предполагает анализ различных алгоритмов и (или) методов программирования

решения одной и той же задачи, выбор из них наилучшего. Этот же принцип можно использовать также для того, чтобы показать необходимость той или иной конструкции языка, быстрее и лучше её понять. Для этого можно записать несколько вариантов некоторого фрагмента программы решения одной и той же задачи, используя разные элементы. При изучении многих систем программирования можно привести такого рода элементы, которые имеет смысл сравнивать:

1. Представление положительных и отрицательных целых чисел в памяти компьютера.
2. Логические операции «&&» (and) и «||» (or).
3. Логические и соответствующие битовые операции.
4. Полная и сокращенная формы оператора if.
5. Оператор if и оператор выбора (switch или case).
6. Операторы цикла while и for, while и repeat, while и do ... while.
7. Процедуры и функции, функции типа void и отличные от void.
8. Встроенные функции и функции с прототипом.
9. Операторы break и return.
10. Параметры-указатели и параметры-ссылки в функциях C++.
11. Динамические и нединамические одномерные и двумерные массивы.
12. Структуры и классы.
13. Бинарные и текстовые файлы.

и т. д. Как видим, этот принцип можно эффективно применять при изучении почти любой темы независимо от используемого языка программирования. Один из вариантов, как правило, «красивее» и, возможно, легче для изучения. С помощью принципа сравнения объективнее можно проверить и оценить знания обучаемого. Одно дело, в экзаменационный билет включить один вопрос из перечисленных выше пар, и совсем другое, если попросить сравнить два каких-нибудь элемента.

Благодаря такому методическому приёму реализуется на практике **принцип повторения**. Один из сравниваемых элементов изучается обычно раньше (например, оператор «if»). Повторение при изучении основ алгоритмизации и программирования в большей степени, чем в математике, играет важную роль. Это связано с тем, что при изучении новой конструкции языка или нового метода программирования, при написании программ по новой теме надо знать много ранее изученных вопросов. Например, операции и выражения, операторы ветвления и цикла, ввода-вывода, а позднее функции будут использоваться при написании программ по любой новой теме.

В докладе рассматриваются также некоторые другие менее важные методические принципы изучения алгоритмизации и программирования.

Библиографические ссылки

1. Аленский Н. А., Травин В. В. Методика преподавания информатики. Уч.-мет. пособие с грифом УМО вузов РБ по естественно-научному образованию. Минск, «Адукацыя і выхаванне». 2019. 104 с.
2. Аленский Н.А., Травин В. В., Филимонов Д. В. Методы программирования [Электронный ресурс]: учеб.-метод. пособие с грифом УМО вузов РБ по естественно-научному образованию. Минск: БГУ, 2023. 1 электрон. опт. Диск (CD-ROM). ISBN 978-985-881-408-3. 192 с. URL: <https://elib.bsu.by/handle/123456789/300305>