История

Этапы развития

В 1930—1940 годах, А. Чёрч, А. Тьюринг, А. Марков разработали математические абстракции (лямбда-исчисление, машину Тьюринга) - для формализации алгоритмов.

В это же время, в 1940-е годы, появились электрические цифровые компьютеры и был разработан язык, который можно считать первым высокоуровневым языком программирования для ЭВМ — “Plankalkül”, созданный немецким инженером К. Цузе в период с 1943 по 1945 годы.

Программисты ЭВМ начала 1950-х годов, в особенности таких, как `UNIVAC` и `IBM 701`, при создании программ пользовались непосредственно машинным кодом, запись программы на котором состояла из единиц и нулей и который принято считать языком программирования первого поколения (при этом разные машины разных производителей использовали различные коды, что требовало переписывать программу при переходе на другую ЭВМ).

Первым практически реализованным языком стал в 1949 году так называемый “Краткий код”, в котором операции и переменные кодировались двухсимвольными сочетаниями. Он был разработан в компании `Eckert–Mauchly Computer Corporation`, выпускавшей `UNIVAC`-и, созданной одним из сотрудников Тьюринга, Джоном Мокли. Мокли поручил своим сотрудникам разработать транслятор

математических формул, однако для 1940-х годов эта цель была слишком амбициозна. Краткий код был реализован с помощью интерпретатора.

Вскоре на смену такому методу программирования пришло применение языков второго поколения, также ограниченных спецификациями конкретных машин, но более простых для использования человеком за счёт использования мнемоник (символьных обозначений машинных команд) и возможности сопоставления имён адресам в машинной памяти. Они традиционно известны под наименованием языков ассемблера и автокодов. Однако, при использовании ассемблера становился необходимым процесс перевода программы на язык машинных кодов перед её выполнением, для чего были разработаны специальные программы, также получившие название ассемблеров. Сохранялись и проблемы с переносимостью программы с ЭВМ одной архитектуры на другую, и необходимость для программиста при решении задачи мыслить терминами “низкого уровня” - ячейка, адрес, команда. Позднее языки второго поколения были усовершенствованы: в них появилась поддержка макрокоманд.

С середины 1950-х начали появляться языки третьего поколения, такие как Фортран, `Lisp` и `Cobol`. Языки программирования этого типа более абстрактны (их ещё называют “языками высокого уровня”) и универсальны, не имеют жёсткой зависимости от конкретной аппаратной платформы и используемых на ней машинных команд. Программа на языке высокого уровня может исполняться (по крайней мере, в теории, на практике обычно имеются ряд специфических

версий или диалектов реализации языка) на любой ЭВМ, на которой для этого языка имеется транслятор (инструмент, переводящий программу на язык машины, после чего она может быть выполнена процессором).

Обновлённые версии перечисленных языков до сих пор имеют хождение в разработке программного обеспечения, и каждый из них оказал определённое влияние на последующее развитие языков программирования. Тогда же, в конце 1950-х годов, появился `Algol`, также послуживший основой для ряда дальнейших разработок в этой сфере. Необходимо заметить, что на формат и применение ранних языков программирования в значительной степени влияли интерфейсные ограничения.

В период 1960-х - 1970-х годов были разработаны основные парадигмы языков программирования, используемые в настоящее время, хотя во многих аспектах этот процесс представлял собой лишь улучшение идей и концепций, заложенных ещё в первых языках третьего поколения.

- Язык `APL` оказал влияние на функциональное программирование и стал первым языком, поддерживавшим обработку массивов.
- Язык `PL/1` (`NPL`) был разработан в 1960-х годах как объединение лучших черт `Fortran` и `Cobol`.
- Язык `Snobol`, разработанный и совершенствуемый в течение 1960-х годов, ориентированный на обработку текстов, ввёл в число базовых операций языков программирования

сопоставление с образцом.

- Язык `Simula`, появившийся примерно в это же время, впервые включал поддержку объектно-ориентированного программирования. В середине 1970-х группа специалистов представила язык `Smalltalk`, который был уже всецело объектно-ориентированным.
- В период с 1969 по 1973 годы велась разработка языка Си, популярного и по сей день и ставшего основой для множества последующих языков, например, столь популярных, как `C++` и `Java`.
- В 1972 году был создан `Prolog` - наиболее известный (хотя и не первый, и далеко не единственный) язык логического программирования.
- В 1973 году в языке `ML` была реализована расширенная система полиморфной типизации, положившая начало типизированным языкам функционального программирования. Каждый из этих языков породил по семейству потомков, и большинство современных языков программирования в конечном счёте основано на одном из них.

Кроме того, в 1960—1970-х годах активно велись споры о необходимости поддержки структурного программирования в тех или иных языках. В частности, голландский специалист Э. Дейкстра выступал в печати с предложениями о полном отказе от использования инструкций `GOTO` во всех высокоуровневых языках. Развивались также приёмы, направленные на сокращение объёма программ и повышение продуктивности работы программиста и пользователя.

Более подробно о развитии, классивикации языков можно почитать в `WiKi` статье Языки программирования

Парадигмы программирования

Императивное программирование

Императивное программирование - это парадигма программирования (стиль написания исходного кода компьютерной программы), для которой характерно следующее:

1. в исходном коде программы записываются инструкции (команды);
2. инструкции должны выполняться последовательно;
3. при выполнении инструкции данные, полученные при выполнении предыдущих инструкций, могут читаться из памяти;
4. данные, полученные при выполнении инструкции, могут записываться в память.

Императивная программа похожа на приказы (англ. `imperative` - приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках, то есть представляют собой последовательность команд, которые должен выполнить компьютер.

При императивном подходе к составлению кода (в отличие от функционального подхода, относящегося к декларативной парадигме) широко используется присваивание. Наличие операторов

присваивания увеличивает сложность модели вычислений и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном подходе.

Основные черты императивных языков:

1. использование именованных переменных;
2. использование оператора присваивания;
3. использование составных выражений;
4. использование подпрограмм;

История:
Первыми императивными языками были машинные инструкции (коды) - команды, готовые к исполнению компьютером сразу (без каких-либо преобразований). В дальнейшем были созданы ассемблеры, и программы стали записывать на языках ассемблеров. Ассемблер - компьютерная программа, предназначенная для преобразования машинных инструкций, записанных в виде текста на языке, понятном человеку (языке ассемблера), в машинные инструкции в виде, понятном компьютеру (машинный код). Одной инструкции на языке ассемблера соответствовала одна инструкция на машинном языке. Разные компьютеры поддерживали разные наборы инструкций. Программы, записанные для одного компьютера, приходилось заново переписывать для переноса на другой компьютер. Были созданы языки программирования высокого уровня и компиляторы - программы, преобразующие текст с языка программирования на язык машины (машинный код). Одна инструкция языка высокого уровня соответствовала одной или нескольким инструкциям языка машины, и

для разных машин эти инструкции были разными. Первым распространённым высокоуровневым языком программирования, получившим применения на практике, стал язык `Fortran`

Декларативное программирование

Декларативное программирование - это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат. В общем и целом, декларативное программирование идёт от человека к машине, тогда как императивное - от машины к человеку. Как следствие, декларативные программы не используют понятия состояния, то есть не содержат переменных и операторов присваивания (см. также ссылочная прозрачность).

Наиболее близким к “чисто декларативному” программированию является написание исполнимых спецификаций (см. соответствие Карри - Ховарда). В этом случае программа представляет собой формальную теорию, а её выполнение является одновременно автоматическим доказательством этой теории, и характерные для императивного программирования составляющие процесса разработки (проектирование, рефакторинг, отладка и др.) в этом случае исключаются: программа проектирует и доказывает сама себя.

К подвидам декларативного программирования также зачастую

относят функциональное и логическое программирование - несмотря на то, что программы на таких языках нередко содержат алгоритмические составляющие, архитектура в императивном понимании (как нечто отдельное от кодирования) в них также отсутствует: схема программы является непосредственно частью исполняемого кода.

На повышение уровня декларативности нацелено языково-ориентированное программирование.

“Чисто декларативные” компьютерные языки зачастую не полны по Тьюрингу - примерами служат `SQL` и `HTML` - так как теоретически не всегда возможно порождение исполняемого кода по декларативному описанию. Это иногда приводит к спорам о корректности термина “декларативное программирование” (менее спорным является “декларативное описание решения” или, что то же самое, “декларативное описание задачи”).

Структурное программирование

Структурное программирование - методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Предложена в 1970-х годах Э. Дейкстрой и др.

В соответствии с данной методологией любая программа строится без использования оператора `goto` из трёх базовых управляющих структур: последовательность, ветвление, цикл; кроме того,

используются подпрограммы. При этом разработка программы ведётся пошагово, методом `сверху вниз`.

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственно, усложнения программного обеспечения. В 1970-е годы объёмы и сложность программ достигли такого уровня, что традиционная (неструктурированная) разработка программ перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать. Поэтому потребовалась систематизация процесса разработки и структуры программ.

Методология структурной разработки программного обеспечения была признана `самой сильной формализацией 70-х годов`.

По мнению Бертрана Мейера, “Революция во взглядах на программирование, начатая Дейкстрой, привела к движению, известному как структурное программирование, которое предложило систематический, рациональный подход к конструированию программ. Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование”.

Цель структурного программирования - повысить производительность труда программистов, в том числе при разработке больших и сложных программных комплексов, сократить число ошибок, упростить отладку, модификацию и сопровождение программного обеспечения.

Функциональное программирование

Функциональное программирование - раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия “функции” в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных.

Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов - чистые функции).

Лямбда-исчисление являются основой для функционального программирования, многие функциональные языки можно рассматривать как “надстройку” над ними.

Пример(`Erlang`):

```
proc(Function, List, Number) ->
    process_flag(trap_exit, true),
    Supervisor = self(),
    spawn_link(combinat, Function, [List, Number, fun(R)->Sup
ervisor!R end]),
    loop([]).

loop(Total) ->
    receive
    {'EXIT', Worker, normal} ->
```

```
        io:format("~w~n", [Total]),
        unlink(Worker);
    Result ->
        loop(Total ++ [Result])
    end.
```

Логическое программирование

Логическое программирование - парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Самым известным языком логического программирования является `Prolog`.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Идеологически ООП - подход к программированию как к моделированию информационных объектов, решающий на новом

уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что в свою очередь особенно важно при реализации крупных проектов.

Управляемость для иерархических систем предполагает минимизацию избыточности данных (аналогичную нормализации) и их целостность, поэтому созданное удобно управляемым - будет и удобно пониматься. Таким образом через тактическую задачу управляемости решается стратегическая задача - транслировать понимание задачи программистом в наиболее удобную для дальнейшего использования форму.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

1. абстрагирование для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счете - контекстное понимание предмета, формализуемое в виде класса;
2. инкапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды “что делать”, без одновременного уточнения как именно делать, так как это уже другой уровень управления;

3. наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя все остальное, учтенное на предыдущих шагах;
4. полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот - собрать воедино.

То есть фактически речь идет о прогрессирующей организации информации согласно первичным семантическим критериям: “важное/неважное”, “ключевое/подробности”, “родительское/дочернее”, “единое/множественное”. Прогрессирование, в частности, на последнем этапе дает возможность перехода на следующий уровень детализации, что замыкает общий процесс.

Обычный человеческий язык в целом отражает идеологию ООП, начиная с инкапсуляции представления о предмете в виде его имени и заканчивая полиморфизмом использования слова в переносном смысле, что в итоге развивает выражение представления через имя предмета до полноценного понятия-класса.

Процедурное программирование

Процедурное программирование - программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

Процедурное программирование является отражением архитектуры традиционных ЭВМ, которая была предложена Фон Нейманом в 1940-х годах. Теоретической моделью процедурного программирования служит абстрактная вычислительная система под названием машина Тьюринга.

Автоматное программирование

Автоматное программирование - это парадигма программирования, при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального автомата. Известна также и другая “парадигма автоматного программирования, состоящая в представлении сущностей со сложным поведением в виде автоматизированных объектов управления, каждый из которых представляет собой объект управления и автомат”. При этом о программе, как в автоматическом управлении, предлагается думать как о системе автоматизированных объектов управления.

В зависимости от конкретной задачи в автоматном программировании могут использоваться как конечные автоматы, так и автоматы с более сложным строением.

Определяющими для автоматного программирования являются следующие особенности:

временной период выполнения программы разбивается на шаги автомата, каждый из которых представляет собой выполнение определённой (одной и той же для каждого шага) секции кода с

единственной точкой входа; такая секция может быть оформлена, например, в виде отдельной функции и может быть разделена на подсекции, соответствующие отдельным состояниям или категориям состояний
передача информации между шагами автомата осуществляется только через явно обозначенное множество переменных, называемых состоянием автомата; между шагами автомата программа (или её часть, оформленная в автоматном стиле) не может содержать неявных элементов состояния, таких как значения локальных переменных в стеке, адреса возврата из функций, значение текущего счётчика команд и т. п.; иначе говоря, состояние программы на любые два момента входа в шаг автомата могут различаться между собой только значениями переменных, составляющих состояние автомата (причём такие переменные должны быть явно обозначены в качестве таковых). Полностью выполнение кода в автоматном стиле представляет собой цикл (возможно, неявный) шагов автомата.

Название автоматное программирование оправдывается ещё и тем, что стиль мышления (восприятия процесса исполнения) при программировании в этой технике практически точно воспроизводит стиль мышления при составлении формальных автоматов (таких как машина Тьюринга, автомат Маркова и др.)

Сфера применения

Автоматное программирование широко применяется при построении лексических анализаторов (классические конечные автоматы) и синтаксических анализаторов (автоматы с магазинной памятью).

Кроме того, мышление в терминах конечных автоматов (то есть разбиение исполнения программы на шаги автомата и передача информации от шага к шагу через состояние) необходимо при построении событийно-ориентированных приложений. В этом случае программирование в стиле конечных автоматов оказывается единственной альтернативой порождению множества процессов или потоков управления (тредов).

Часто понятие состояний и машин состояний используется для спецификации программ. Так, при проектировании программного обеспечения с помощью `UML` для описания поведения объектов используются диаграммы состояний (`state machine diagrams`). Кроме того, явное выделение состояний используется в описании сетевых протоколов (см., например, `RFC 793`.

Мышление в терминах автоматов (шагов и состояний) находит применение и при описании семантики некоторых языков программирования. Так, исполнение программы на языке Рефал представляет собой последовательность изменений поля зрения Рефал-машины или, иначе говоря, последовательность шагов Рефал-автомата, состоянием которого является содержимое поля зрения (произвольное Рефал-выражение, не содержащее переменных).

Механизм продолжений языка `Scheme` для своей реализации также требует мышления в терминах состояний и шагов, несмотря на то что сам язык `Scheme` никоим образом не является автоматным. Тем не менее, чтобы обеспечить возможность “замораживания” продолжения,

приходится при реализации вычислительной модели языка `Scheme` объединять все компоненты среды исполнения, включая список действий, которые осталось выполнить для окончания вычислений, в единое целое, которое также обычно называется продолжением. Такое продолжение оказывается состоянием автомата, а процесс выполнения программы состоит из шагов, каждый из которых выводит следующее значение продолжения из предыдущего.

Александр Оллонгрен в своей книге описывает так называемый Венский метод описания семантики языков программирования, основанный целиком на формальных автоматах.

В качестве одного из примеров применения автоматной парадигмы можно назвать систему `STAT`; эта система, в частности, включает встроенный язык `STATL`, имеющий чисто автоматную семантику.

Существуют также предложения по использованию автоматного программирования в качестве универсального подхода к созданию компьютерных программ вне зависимости от предметной области. Так, авторы статьи Туккель Н.И., Шалыто А.А. “Программирование с явным выделением состояний” утверждают, что автоматное программирование способно сыграть роль легендарной серебряной пули.

История языка Си

Язык был разработан в лабораториях `Bell Labs` в период с 1969 по 1973 годы. Согласно Ритчи, самый активный период творчества

пришёлся на 1972 год. Язык назвали “Си” (`C` - третья буква английского алфавита), потому что многие его особенности берут начало от старого языка “Би” (`B` - вторая буква английского алфавита). Существует несколько различных версий происхождения названия языка Би. Кен Томпсон указывает на язык программирования `BCPL`, однако существует ещё и язык `Bon`, также созданный им, и названный так в честь его жены Бонни.

Существует несколько легенд, касающихся причин разработки Си и его отношения к операционной системе `UNIX`, включая следующие:

1. Разработка Си стала результатом того, что его будущие авторы любили компьютерную игру, подобную популярной игре `Asteroids` (Астероиды). Они уже давно играли в неё на главном сервере компании, который был недостаточно мощным и должен был обслуживать около ста пользователей. Томпсон и Ритчи посчитали, что им не хватает контроля над космическим кораблём для того, чтобы избегать столкновений с некоторыми камнями. Поэтому они решили перенести игру на свободный `PDP-7`, стоящий в офисе. Однако этот компьютер не имел операционной системы, что заставило их её написать. В конце концов, они решили перенести эту операционную систему ещё и на офисный `PDP-11`, что было очень тяжело, потому что её код был целиком написан на ассемблере. Было вынесено предложение использовать какой-нибудь высокоуровневый портируемый язык, чтобы можно было легко переносить ОС с одного компьютера на другой. Язык Би, который они хотели

сначала задействовать для этого, оказался лишён функциональности, способной использовать новые возможности `PDP-11`. Поэтому они и остановились на разработке языка Си.

2. Самый первый компьютер, для которого была первоначально написана `UNIX`, предназначался для создания системы автоматического заполнения документов. Первая версия `UNIX` была написана на ассемблере. Позднее для того, чтобы переписать эту операционную систему, был разработан язык Си.

   К 1973 году язык Си стал достаточно силён, и большая часть ядра `UNIX`, первоначально написанная на ассемблере `PDP-11/20`, была переписана на Си. Это было одно из самых первых ядер операционных систем, написанное на языке, отличном от ассемблера; более ранними были лишь системы `Multics` (написана на ПЛ/1) и `TRIPOS` (написана на `BCPL`).

В 1978 году Брайан Керниган и Деннис Ритчи опубликовали первую редакцию книги “Язык программирования Си”. Эта книга, известная среди программистов как `K&R`, служила многие годы неформальной спецификацией языка. Версию языка Си, описанную в ней, часто называют `K&R C`. Вторая редакция этой книги посвящена более позднему стандарту `ANSI C`, описанному ниже.

`K&R` ввёл следующие особенности языка:

1. структуры (тип данных `struct`);
2. длинное целое (тип данных `long int`);

3. целое без знака (тип данных `unsigned int`);
4. оператор `+=` и подобные ему (старые операторы `=+` вводили анализатор лексики компилятора Си в заблуждение, например, при сравнении выражений `i =+ 10` и `i = +10`).

`K&R C` часто считают самой главной частью языка, которую должен поддерживать компилятор Си. Многие годы даже после выхода `ANSI C` он считался минимальным уровнем, которого следовало придерживаться программистам, желающим добиться от своих программ максимальной переносимости, потому что не все компиляторы тогда поддерживали `ANSI C`, а хороший код на `K&R C` был верен и для `ANSI C`.

После публикации `K&R C` в язык было добавлено несколько возможностей, поддерживаемых компиляторами `AT&T` и некоторых других производителей:

1. функции, не возвращающие значение (с типом `void`), и указатели, не имеющие типа (с типом `void*`);
2. функции, возвращающие объединения и структуры;
3. имена полей данных структур в разных пространствах имён для каждой структуры;
4. присваивания структур;
5. спецификатор констант (`const`);
6. стандартная библиотека, реализующая большую часть функций, введённых различными производителями;
7. перечислимый тип (`enum`);
8. дробное число одинарной точности (`float`).

Более подробно о истории создания, эволюции и особенностей версий стандарта языка Си можно прочитать в `WiKi` статье Си(Язык программирование)

Стандарты языка

Всего, на данный момент существует как минимум три стандарта языка: `C89` он же `ANSI C`, `C99`, `C11`.

`C89`

В 1983 году Американский национальный институт стандартов (`ANSI`) сформировал комитет для разработки стандартной спецификации Си. По окончании этого долгого и сложного процесса в 1989 году он был наконец утверждён как “Язык программирования Си” `ANSI` X3.159-1989. Эту версию языка принято называть `ANSI C` или `C89`. В 1990 году стандарт `ANSI C` был принят с небольшими изменениями Международной организацией по стандартизации (`ISO`) как `ISO/IEC 9899:1990`.

Одной из целей этого стандарта была разработка надмножества `K&R C`, включающего многие особенности языка, созданные позднее. Однако комитет по стандартизации также включил в него и несколько новых возможностей, таких, как прототипы функций (заимствованные из `C++`) и более сложный препроцессор.

`ANSI C` сейчас поддерживают почти все существующие компиляторы. Почти весь код Си, написанный в последнее время, соответствует `ANSI`

`C`. Любая программа, написанная только на стандартном Си, гарантированно будет правильно выполняться на любой платформе, имеющей соответствующую реализацию Си. Однако большинство программ написаны так, что они будут компилироваться и исполняться только на определённой платформе, потому что:

1. они используют нестандартные библиотеки, например, для графических дисплеев;
2. они используют специфические платформо-зависимые средства;
3. они рассчитаны на определённое значение размера некоторых типов данных или на определённый способ хранения этих данных в памяти для конкретной платформы.

`C99`

После стандартизации в `ANSI` спецификация языка Си оставалась относительно неизменной в течение долгого времени, в то время как `C++` продолжал развиваться (в 1995 году в стандарт Си была внесена Первая нормативная поправка, но её почти никто не признавал). Однако в конце 1990-х годов стандарт подвергся пересмотру, что привело к публикации `ISO 9899:1999` в 1999 году. Этот стандарт обычно называют `C99`. В марте 2000 года он был принят и адаптирован `ANSI`.

Некоторые новые особенности `C99`:

1. подставляемые функции (`inline`);
2. объявление локальных переменных в любом операторе

программного текста (как в `C++`);
3. новые типы данных, такие, как `long long int` (для облегчения перехода от 32- к 64-битным числам), явный булевый тип данных `_Bool` и тип `complex` для представления комплексных чисел;
4. массивы переменной длины;
5. поддержка ограниченных указателей (`restrict`);
6. именованная инициализация структур: `struct { int x, y, z; } point = { .y=10, .z=20, .x=30 }`;
7. поддержка однострочных комментариев, начинающихся на `//`, заимствованных из `C++` (многие компиляторы Си поддерживали их и ранее в качестве дополнения);
8. несколько новых библиотечных функций, таких, как `snprintf`;
9. несколько новых заголовочных файлов, таких, как `stdint.h`.

C11

8 декабря 2011 опубликован новый стандарт для языка Си (`ISO/IEC 9899:2011`). Основные изменения:

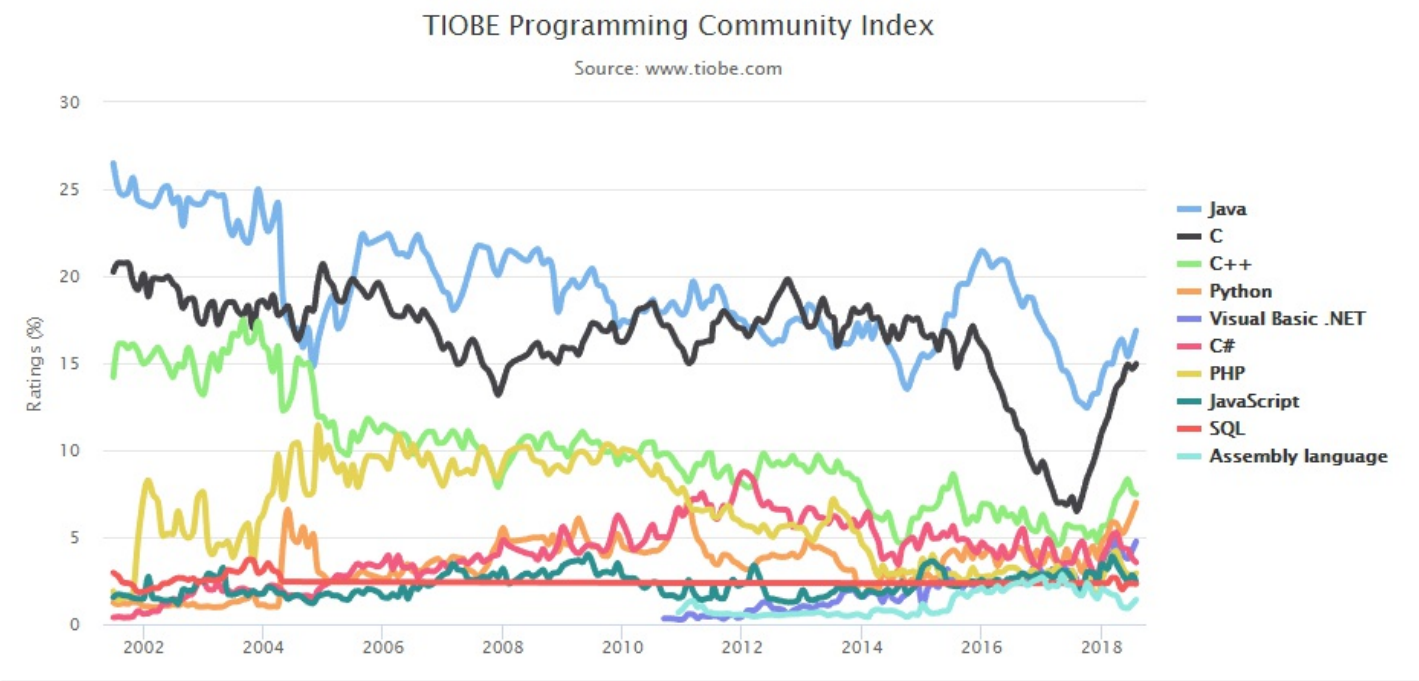
1. поддержка многопоточности;
2. улучшенная поддержка `Unicode`;
3. обобщённые макросы (`type-generic expressions`, позволяют статичную перегрузку);
4. анонимные структуры и объединения (упрощают обращение ко вложенным конструкциям);
5. управление выравниванием объектов;
6. статичные утверждения (`static assertions`);
7. удаление опасной функции `gets` (в пользу безопасной

`gets_s`);

8. функция `quick_exit`;
9. спецификатор функции `_Noreturn`;
10. новый режим эксклюзивного открытия файла.

Почему Си `C89`

На данный момент, язык Си является наиболее популярным и востребованным языком программирования высокого уровня. Компания `TIOBE` публикует индекс популярности языков на рынке.



Популярность языка обусловлена:

1. простотой синтаксиса;
2. жесткой стандартиризацией;
3. наличием стандартной библиотекой;
4. наличие механизмов прямого обращения к памяти;

5. простотой написания компилятора. How I wrote a self-hosting C compiler in 40 days;

В настоящее время язык претерпевает следующую волну популярности из-за использования его в качестве основного языка ЯПВУ для `Embedded` платформ. Большиство компаний пишут собственные реализации компилятора для своих микроконтроллеров, в простейшем случае стандарта `C89`.