CSci 365: Organization of Programming Languages

Report 1

Max Forst & Kenneth Jahnke

9/29/2024

**Fortran 1:**

FORTRAN (FORmula TRANslation) was the first high-level programming language. It’s main purpose was to simplify complex mathematical and scientific computations. It used a compiler that would break down the high-level language into computer level language, which would run much more efficiently.

Source: <https://numat.net/fortran/unfp/ch1-1.html>

**Fortran 1 ® Fortran II:**

Program modules could now be compiled individually instead of all together. The ability to integrate assembly language modules with FORTRAN modules was also added, and was called ‘linked loading.’

Source: <https://numat.net/fortran/unfp/ch1-1.html>

**Fortran II ® Fortran IV:**

Both the COMMON and EQUIVALENCE statements were improved by making them more efficient on memory. It also became less machine-dependent, meaning that it could now be more portable and reliable across different machines and programs.

Source: <https://numat.net/fortran/unfp/ch1-1.html>

**Fortran IV ® Fortran 77:**

Added DO loops with a decreasing control variable, giving more flexibility with loops. Added block IF THEN ELSE statements, which replaced IF GOTO statements. Pretests were also added to DO loops, which now meant that loops had the ability to loop zero times instead of the mandatory first loop in previous versions. A new data type called a CHARACTER was implemented. The STOP statement is also no longer needed in the Main program to terminate.

Source: <https://numat.net/fortran/unfp/ch1-1.html>

**Fortran 77 ® Fortran 90:**

Added dynamic memory allocation, allowing programs to allocate memory at runtime. CASE and DO WHILE control structures were also added, improving conditional and loop handling. Array notation was implemented, making it easier to perform operations on entire arrays. The addition of modules, allowed for better code organization and reusability. It also added derived types and operator overloading, allowing for user-defined data types and customization of operators.

Source: <https://numat.net/fortran/unfp/ch1-1.html>

**Fortran 90 ® Fortran 95:**

Fortran 95 is noted as the standard. It added the HPF (High Performance Fortran) extension, which focuses on parallel computing. This allowed for the language to spread computations across multiple processors, allowing for overall faster computations. The FORALL statement was implemented, which allowed for efficient array operations. In addition, other features that were previously marked as obsolete in Fortran 90 were removed.

Source: <https://www.admin-magazine.com/HPC/Articles/Modern-Fortran-Part-2>

**Fortran 95 ® Fortran 2003:**

Added more features for object-oriented programming (OOP), which enhanced code modularity and reusability. It also introduced type-bound procedures, allowing procedures to be tied to derived types. Inheritance and polymorphism were also added, making it possible to create hierarchical relationships between types and use dynamic type dispatching. It also added dynamic type allocation, providing more flexibility in handling memory and data types at runtime.

Source: <https://www.admin-magazine.com/HPC/Articles/Modern-Fortran-Part-2>

**Fortran 2003 ® Fortran 2008:**

Added concurrent computing via Coarray Fortran (CAF), which was a set of extensions developed for Fortran 95 and 2003, allowing users to run separate pieces of code at the same time. It also added the DO CONCURRENT construct, allowing loops to be executed in parallel for better performance. The BLOCK construct was also added, which can contain declarations of objects. It also added the CONTIGUOUS attribute, which specifies storage layout restrictions. In addition to these new features, it also revised many corrections and clarifications from the previous version.

Source: <https://www.admin-magazine.com/HPC/Articles/Modern-Fortran-Part-3>

**Fortran 2008 ® Fortran 2015:**

Improved interoperability with C, which allowed for better integration of Fortran and C code. New additions to coarrays were added, called “teams,” “events,” “atomics,” and “collectives.” Teams are groups of images in a coarray that work together to complete a specific task. Events allow one image to tell a different image that it has finished a task. Atomics existed in Fortran 2008 also, but this version added operations for ADD, AND, CAS, OR, and XOR. Collectives allowed for performing operations across entire teams, affecting every image in that team.

Source: <https://www.admin-magazine.com/HPC/Articles/Modern-Fortran-Part-3>

**Fortran IV ® BASIC:**

One of the biggest differences is that BASIC is interpreted, unlike Fortran, which is compiled. BASIC is also more portable, meaning that it was designed to run on a wider variety of machines. This is mainly due to how it is a more general programming language than Fortran, which focused on scientific calculations. This also made BASIC more simple and easier to learn for programmers.

Source: <https://oozden.wordpress.com/2013/02/08/fortran-vs-basic/>

**BASIC ® Quick BASIC:**

Quick BASIC gained the ability to be compiled instead of just interpreted. This allowed for much faster execution of code as well as it being more efficient. The addition of structued syntax is also important because it added things like procedures, functions, and local variables, which can make the readability of the code much easier.

Source: <https://www.q7basic.org/History%20of%20BASIC.pdf>

**Quick BASIC ® Visual BASIC:**

Visual BASIC’s main difference over the previous version is that it incorporates a visual design environment for creating GUIs for applications. This means that users can drag and drop buttons or text to create a GUI. It also incorporated event-driven programming, which would execute a portion of code based on a user’s clicks on a GUI. It also developed some features of OOP, like classes and objects.

Source: [https://www.researchgate.net/publication/371834735](https://www.researchgate.net/publication/371834735_Comparative_Analysis_of_Visual_Basic_and_QBasic_A_Comprehensive_Study)

**Visual BASIC ® Visual Basic.NET:**

It now supports OOP fully instead of just a few capabilities of it, including inheritance, encapsulation, and polymorphism. It can now create a wider variety of applications such as console, web, and windows applications. It also added multithreading capabilities, which allowed for programs that could handle multiple tasks at the same time.

Source: <https://www.geeksforgeeks.org/difference-between-vb-net-and-visual-basic/>

**Fortran IV ® PL/I:**

PL/I had more flexible block structures, meaning that statements like IF GO TO and DO had similar meanings to Fortran IV, but varied slightly as they didn’t focus only on arithmetic expressions. Asynchronous procedures were added, allowing control to pass to the next statement before the asynchronous one has even been executed. More advanced input/output features were also implemented because it handles more than just arithmetic expressions.

Source: <http://www.bitsavers.org/pdf/ibm/360/pli/SC20-1637-3_A_Guide_to_PL_I_for_FORTRAN_Users_May68.pdf>

**Fortran 1 ® ALGOL 58:**

It added block structures, which could act like a single line of code, but executed many lines of code. This allowed for a more organized way of writing complex programs. It also aimed to be machine independent, meaning that it could run on a wider variety of machines.

Source: <https://courses.cs.washington.edu/courses/cse505/99au/imperative/algol.html>

**ALGOL 58 ® ALGOL 60:**

ALGOL 60 refined block structures with clear scoping rules, allowing for better organization of code. It also supported nested functions and recursion, meaning that a function could call itself for more complex algorithms. It also supported both pass-by-value and pass-by-name parameter passing methods.

Source: <https://bulldogjob.com/readme/why-algol-was-an-important-programming-language>

**ALGOL 60 ® ALGOL W:**

It removed the need for stropping, which is special marking of keywords, making the syntax simpler. It also added new data types such as strings, bit strings, complex numbers, and records. It also added the while loop, adding more flexibility than other loop options. It also incorporated a larger standard library including built in input and output functions.

Source: <https://try-mts.com/algol-w-introduction/>

**ALGOL W ® PASCAL:**

PASCAL had a simpler syntax, making it easier for beginners to learn. It was also strongly typed, meaning that it was better at detecting errors at compile time. It also made structured programming through procedures and functions better by more clearly defining scopes and parameter passing.

Source: <https://www.tutorialspoint.com/pascal/pascal_overview.htm>

**PASCAL ® MODULA-2:**

Modula-2 introduced the module system, allowing for better organization of code by separating interfaces from implementations. It also allowed for separate compilation of modules, meaning that programs could be developed in smaller chunks instead of all at once. It also added support for multiprogramming, which allowed for the execution of multiple processes at the same time.

Source: <https://dl.acm.org/doi/pdf/10.1145/948290.948293>

**MODULA-2 ® MODULA-3:**

Modula-3 added support for object-oriented programming features, including classes, inheritance, and polymorphism. It also added support for automatic garbage collection, which would automatically deallocate memory when it was no longer being used. It also added exception handling, which improved the management of errors.

Source: <https://www.cs.purdue.edu/homes/hosking/m3/reference/intro.html#:~:text=Modula%2D3%20supports%20interfaces%2C%20objects,in%20unsigned%20numeric%20data%20type>.

**MODULA-3 ® Python:**

Python’s main goal was to improve readability and simplicity, so it was developed with a simple and easy to understand syntax. Python is an interpreted language instead of being compiled, like MODULA-3, which allows for more interactive development. It also introduced dynamic typing, meaning that variables don’t need to be explicitly typed when created. Also, everything is OOP in Python, unlike MODULA-2, where OOP was available, but not required.

Source: <https://www.educative.io/blog/python-versions-history>

**Python ® Python 2.0:**

This version added list comprehensions, which provided a concise way to create lists, replacing the old loop based methods. It also updated garbage collection capabilities to improve memory management. It also introduced full support for Unicode characters, allowing for better handling of foreign characters.

Source: <https://www.geeksforgeeks.org/python-version-history/>

**Python 2.0 ® Python 3.0:**

In Python 3.0, the print statement was changed to a function to improve consistency and flexibility. The syntax of the division operator was changed to include both a true division operator and a floor division operator. In addition, features that were depreciated were removed to simplify the language.

Source: <https://www.geeksforgeeks.org/python-version-history/>

**MODULA-2 ® Oberon:**

Oberon reduced the complexity of MODULA-2 by removing a lot of extra features that otherwise wouldn’t affect the overall goal of the language, which was to teach coding. It also introduced the type extension, which implemented something called type inclusion, which allowed new types to be derived from existing ones, such as integer being derived from longint.

Source: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6e455efb54c17c25c67b6bea09cf854525e83bb7>

**PASCAL ® Ada 83:**

Ada 83 is more complex than PASCAL in a variety of different ways, but the most notable is how Ada 83 uses a package system instead of modules like PASCAL. This system included both data and procedures, allowing for better modularization. It also has strong typing, meaning that it will have less runtime errors than PASCAL would have.

Source: <https://www.researchgate.net/publication/220459730_A_Comparison_of_Pascal_and_Ada>

**Ada 83 ® Ada 95:**

Ada 95 added full support of object-oriented programming, which was previously not fully supported in Ada 83. It also improved the separate compilation facility, which allowed for better modular programming and also made it easier to make incremental builds of your code. It also added support for real-time systems, meaning that developers could specify time restrictions in their program.

Source: <https://www.adacore.com/about-ada/timeline-of-ada>

**Ada 95 ® Ada 2005:**

Ada 2005 added abstract interface types, which allowed for inheritance of behavior. It also implemented concurrency features with object-oriented programming, allowing for parallel processes to be part of a class. Many new generic containers and packages were added to the predefined library as well.

Source: <https://www.adacore.com/about-ada/timeline-of-ada>

**Ada 2005 ® Ada 2012:**

Ada 2012 added contract-based programming, which allowed programmers to specify conditions that must be met before or after a piece of code is called. It also made enhancements to the real-time programming features, such as scheduling specific tasks. Improvements were also made to the multicore support, meaning that parallel programming would be handled more efficiently.

Source: <https://www.adacore.com/about-ada/timeline-of-ada>

**Ada 83 ® Eiffel:**

Eiffel was designed entirely with object-oriented programming in mind, so it focuses on classes and inheritance. It also implements automatic garbage collection, allowing for less errors regarding memory. It also implements Design by Contract, which is a unique mechanism that ensures that the code will have less errors by following ‘the rules.’

Source: <https://www.eiffel.com/resources/faqs/eiffel-language/>

**ALGOL 60 ® ALGOL 68:**

ALGOL 68 added parallel programming, which allowed multiple processes to run at the same time. It also developed a new two-level context-sensitive grammar, known as the Van Wijngaarden grammar, which enabled more expressive and flexible syntax rules. It also added support for data pointers, which allowed for direct manipulation of memory and dynamic data structures.

Source: <https://dl.acm.org/doi/pdf/10.5555/1074100.1074115>

**ALGOL 60 ® SIMULA I:**

SIMULA I added certain features of object-oriented programming, like objects and classes. It also introduced something called co-routines, which allows certain functions to call each other while stopping their executions temporarily to simulate the processes running at the same time. SIMULA I’s whole deal was that it was meant to simulate real world systems, which was not a focus in ALGOL 60.

Source: <https://dl.acm.org/doi/pdf/10.1145/365813.365819>

**SIMULA I ® SIMULA 67:**

It added full support for object-oriented programming, meaning that concepts like inheritance and polymorphism were also available, not just objects and classes. A new concept called virtual methods was also introduced, which acted like polymorphism, allowing for methods to be overridden in lower classes.

Source: <https://twobithistory.org/2019/01/31/simula.html>

**SIMULA 67 ® Eiffel:**

While SIMULA-67 implemented full support for object-oriented programming, Eiffel’s entire language was written with OOP in mind, everything was an object. Eiffel also introduced a concept called repeated inheritance, which is where a class can inherit from another class through two or more paths. Eiffel also had a goal to be platform independent, unlike SIMULA 67, which could only run on a few select machines due to machine requirements.

Source: <https://www.eiffel.com/resources/faqs/eiffel-language/>

**SIMULA 67 ® Smalltalk 80:**

Smalltalk 80’s entire language was written with OOP in mind, so everything was an object, unlike SIMULA 67, which had OOP capabilities, but they were optional. Smalltalk also added support for dynamic typing, so the type of a variable didn’t have to be declared before it was used. Smalltalk was also developed specifically with GUIs in mind, so it was designed to build interactive graphical user interfaces.

Source: <https://dl.acm.org/doi/pdf/10.5555/273>

**Smalltalk 80 ® Ruby:**

Ruby was designed to be easy for programmers to understand, so it had a more concise and readable syntax compared to Smalltalk 80. It focused on the syntax feeling similar to natural-language. Ruby is also used a lot in web based application programming, unlike Smalltalk 80. Ruby also focused on standardizing its library to make scripting easier, unlike Smalltalk 80, which didn’t have as robust of a library.

Source: <https://dev.to/kopylov_vlad/why-i-love-ruby-44g9>

**Ruby ® Ruby 1.8:**

Ruby 1.8 wasn’t much of an upgrade, it mainly improved performance in a few areas, as well as changed how certain functions work or added new features to functions. For example, the automatic garbage collection is noted to be faster, as well as string concatenation. As mentioned, some functions were updated and some new functions were added to the library as well.

Source: <https://www.ruby-lang.org/en/news/2003/08/04/ruby-180-released/>

**Ruby 1.8 ® Ruby 1.9:**

The progression to Ruby 1.9 had significant performance upgrades compared to previous versions. It also added a new feature called Fibers, which allowed for manual control of pausing and resuming execution of code. Is also added a new hash syntax, which made hashes more concise and readable.

Source: <https://historytimelines.co/timeline/ruby>

**ALGOL 60 ® BASIC:**

BASIC was an interpreted language instead of a compiled one, like ALGOL 60. This made it much easier to develop a program interactively. It also had a much more simplistic design than ALGOL 60, which made it easy for students to pick up and learn quickly. A feature that BASIC did not have was recursion, which ALGOL 60 did include.

Source: <https://www.techtarget.com/whatis/definition/BASIC-Beginners-All-purpose-Symbolic-Instruction-Code>

**ALGOL 60 ® PL/I:**

PL/I had a nice error handling mechanisms that made debugging code much easier, unlike ALGOL 60, which lacked error handling capabilities. It also allowed for concurrent programming, which made it possible to run multiple processes at once. It also had the ability to pass arguments to functions by either the reference or the value, unlike ALGOL 60, which could only pass arguments by reference.

Source: <https://academic.oup.com/comjnl/article-pdf/17/4/325/1215152/17-4-325.pdf>

**FLOW-MATIC:**

It was one of the first languages to use english-like syntax, making it easier for non-technical users to program machines. It included keywords like ADD, SUBTRACT, and MULTIPLY to make it easier for programmers to understand how the code worked.

**FLOW-MATIC ® COBOL:**

COBOL was designed to be machine-independent, meaning that it can be run on a variety of different operating systems as well as machines. This is in contrast to FLOW-MATIC, which was dependent of the type of machine it ran on. COBOL was also designed specifically for business applications, so it focused on records and data management. It also added a few new control structures like the DIVISION, SECTION, and PARAGRAPH structures, which improved the overall flow of the code.

Source: <https://www.ibm.com/topics/cobol>

**COBOL ® PL/I:**

PL/I offers a wider range of data types when compared to COBOL, including things like arrays. It also offered concurrent programming so that multiple tasks to run at the same time. PL/I was also more flexible when compared to COBOL, which was mainly designed for business applications. PL/I had a more simple structure, which made it more flexible when it came to developing diverse programs.

Source: <https://dl.acm.org/doi/pdf/10.1145/960118.808389>

**PERL ® Ruby:**

Unlike PERL, Ruby is entirely object-oriented programming, meaning that everything in Ruby is an object, while PERL supports OOP, but to a lesser degree. Ruby also has a more natural, human-like syntax, which makes it more readable, unlike PERL, which doesn’t have as natural of syntax when compared to Ruby. Ruby also has a much larger number of libraries than PERL.

Source: <https://www.geeksforgeeks.org/difference-between-perl-and-ruby/>

**ANSI C (C89) ® PYTHON:**

Python is completely based on object-oriented programming, meaning that everything is an object, unlike ANSI C, which supports OOP capabilities, but is not entirely OOP. Python is also an interpreted language, not compiled like ANSI C, which makes it easier to develop as you go. Python also has automatic garbage collection, unlike ANSI C, in which memory must be managed manually.

Source: <https://www.geeksforgeeks.org/ansi-c-c89-standard/>

**Prolog**

A portmanteau of *pro*gramming and *log*ic, Prolog began development in 1972 and reached full implementation in 1975. In contrast to many other programming languages, Prolog is a declarative language. It implements logic in the form of facts and rules. These facts and rules can be specified as truths. These truths can then be used as the basis of queries to determine the likeliness of truth or falsehood of other statements. This process of determining the likeliness of truth of a matter based on a set of known information makes this language ideal in the field of artificial intelligence.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 77, 78)

<https://www.geeksforgeeks.org/prolog-an-introduction/>

<https://builtin.com/software-engineering-perspectives/prolog>

**APL**

APL (*A* *P*rogramming *L*anguage) is a dynamically typed language with dynamic storage allocation created circa. 1960 Kenneth E. Iverson at IBM. APL has a massive collection of operators, each used for a specific purpose. Because of such a large set of operators, APL programs are difficult to read and write. The consequence of this difficulty is that APL quickly fell out of use.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 69, 70)

**Lua**

The Portugeuse word for “moon”, Lua was designed as an efficient, lightweight, and embeddable scripting language. As far as interpreted languages go, Lua is the fastest. Lua was designed for use with Windows and Unix-like operating systems. One very useful feature of Lua is its ability to extend programs written in several other languages, such as C, C++, C#, Java, Fortran, Perl, and Ruby.

Source(s):

https://www.lua.org/about.html

**Lisp**

Lisp (meaning “list processing”) was the first functional programming language (where functions are applied to arguments). The languages available to John McCarthy in 1958 did not fit his needs, those being dynamic storage allocation, conditional expressions, and recursion. This led to the first iteration of Lisp, what is sometimes called “pure Lisp”. The data types in “pure Lisp” consist of lists and atoms. The lists are either made up of atoms or sublists. Lisp was the programming language of choice for those exploring the area of artificial intelligence for about a quarter century.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 45-48)

**Lisp ® Scheme**

Scheme treats functions as first-class entities. This means a function's return can directly be assigned to variables, passed as parameters of other functions, or assigned as elements of a list. Scheme allows for static scoping. Because of its simple syntax, it is a decent introductory language for new programmers. It is also an excellent introduction to functional programming.

Source(s):

Concepts of Programming Languages, 12th Edition (pg. 49)

**Scheme ® Common Lisp**

Lisp had many different dialects, including Scheme. The development intention of Common Lisp was to condense the best of each of these dialects into a common operating schema. It kept the syntax and primitive functions of “pure Lisp” but expanded greatly in size. Common Lisp has over 800 built-in functions. It has native support for several common data types, such as arrays, complex numbers, and character strings. Although variables are static by default, programmers can choose to specify them as dynamic.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 49, 50)

<https://courses.cs.umbc.edu/331/fall15/03/scheme/scheme1.pdf>

**Lisp ® ML**

ML (“MetaLanguage”) is primarily a functional language. However, much of ML’s syntax (e.g., the parenthesized functional syntax) is replaced with more imperative syntax, such as that that can be seen in C or Java. Thus, ML does support some degree of imperative programming. In ML, the assignment of values to variables is bound to the environment of the variable. Thus, it is impossible for ML functions to have an effect on variables outside its own environment. The data type of variables and expressions are determined at compile time. Types are tied to objects instead of names.

Source(s):

Concepts of Programming Languages, 12th Edition (pg. 50)

<https://www.geeksforgeeks.org/difference-between-functional-and-imperative-programming/>

<https://courses.cs.washington.edu/courses/cse341/02wi/functional/basics.html>

**Pascal ® ML**

Whereas ML inherits its functional nature from Lisp, its syntax was more heavily inspired by Pascal.

Source(s):

<https://cvr.ac.in/cse/stud/NOTES/PPL/PPL.pdf>

**ML ® Miranda**

Miranda was developed by David Turner in 1986. It is a purely functional language (no variables, no assignment statements, no side effects). Miranda is a “lazy language”, meaning its functions are non-strict, it supports infinite data structures, and its expressions are not evaluated until its value is required. Miranda was also capable of handling list comprehensions and algebraic data types. A program written in Miranda can is typically upwards of five times shorter than if it were written in in an imperative language such as Java or C. Miranda saw widespread academic and commercial use in the 90s.

Source(s):

Concepts of Programming Languages, 12th Edition (pg. 50)

<https://www.cs.kent.ac.uk/people/staff/dat/miranda/short.html>

<https://www.cs.kent.ac.uk/people/staff/dat/miranda/3.8.html>

<https://www.cs.kent.ac.uk/people/staff/dat/miranda/nancypaper.pdf>

**Miranda ® Haskell**

Put simply, Haskell is just a more complex version of Miranda. The core cause of the jump in complexity has to do with Haskell’s typing system which introduces type classes.

Source(s):

<https://courses.cs.washington.edu/courses/cse341/06au/miranda/>

<https://haskell.dev/article/Understanding_Haskells_type_system.html>

<https://courses.cs.washington.edu/courses/cse341/06au/miranda/miranda-basics.html>

<https://www.cs.kent.ac.uk/people/staff/dat/miranda/3.8.html>

**CPL**

The Combined Programming Language (sometimes referred to as Cambridge Programming Language or “Cambridge Plus London” because of the development collaboration with the University of London) or CPL, was developed at Cambridge University in the early 1960s. The language was heavily inspired by ALGOL 60. The premise of development was to take the useful features of that language (such as the block structure) but exclude the features that would cause inefficiencies or make implementation more difficult than need be. There was also to be a heavy emphasis on mathematical operations.

Source(s):

v

<https://www.ithistory.org/resource/cpl-programming-language>

<https://www.cl.cam.ac.uk/~mr10/strachey100.pdf>

<https://www.cs.ox.ac.uk/strachey100/slides/3-MR.pdf>

**CPL ® BCPL**

CPL proved cumbersome for system-programming at scale. Hence a simpler version, BCPL or Basic CPL, was created. Many of CPLs features were stripped down, but the most significant improvement was that BCPL allowed only one data structure – the bit-pattern, hence it was not a typed language. BCPL is where we get the idea of a "Hello, World!” program from. BCPL also introduced the idea of using double forward slashes (//) to indicate comment lines.

Source(s):

<https://archive.org/details/richards1979bcpl/page/n167/mode/2up>

<https://archive.org/details/richards1979bcpl/page/n13/mode/2up>

<https://archive.org/details/richards1979bcpl/page/n7/mode/2up>

**BCPL ® B**

The B language can best be described as an intermediary between BCPL and C. It is “typeless” like BCPL but provided the foundation of syntax for C. It appears that B was developed to address known issues with BCPL (such as the syntactical handling of floating-point numbers) that Ken Thompson of Bell Laboratories did not want present when working within the UNIX operating system. “B, compared to BCPL, is syntactically rich in expressions and syntactically poor in statements.” There are seemingly no other differences of note between BCPL and B.

Source(s):

<https://web.archive.org/web/20150611114427/https://www.bell-labs.com/usr/dmr/www/kbman.pdf>

Concepts of Programming Languages, 12th Edition (pg. 75)

**ALGOL 68 ® C**

Elements C ripped from ALGOL 68:

* `for` statement
* `switch` statement
* The languages treatment of pointers
* Assignment operators

Source(s):

Concepts of Programming Languages, 12th Edition (pg. 75)

**B ® C**

In 1972, Dennis Ritchie invented programming when he introduced the C language. C addressed many of the performance issues present in B. Compared to assembly language, B was incredibly slow. C was implemented in such a way that it ran closer to performance of assembly, but did not match it. Machines, such as the PDP-11 were quickly approaching the ability to efficiently perform floating point arithmetic. There were flaws in the implementation of B that hindered its ability to take advantage of these advancements in hardware. Ritchie ensured this would not be the case in C. B created overhead when dealing with pointers; these were corrected in C. The biggest change between B and C was C’s implementation of typing. Now, specified data types (such as `int` and `char`) were present. All the improvements of C over B made C more portable than B as a happenstance consequence.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 75, 76)

<https://www.bell-labs.com/usr/dmr/www/chist.html>

**C ® Objective-C**

* Whereas C utilizes no concept of classes inherently (making it procedure-oriented), Objective-C does (making it object-oriented).
* Objective-C utilizes both pointers and references, whereas C only uses pointers.
* Objective-C allows default arguments within function definitions.
* Whereas C requires function calls for memory allocation/deallocation (`malloc()`, `calloc()`, `free()`), Objective-C performs these functions with simple operators.
* Objective-C offers `null` pointers, making it more secure.
* Objective-C is a more high-level language than C.
* Objective-C offers exception handling, whereas C does not.

Source(s):

<https://www.geeksforgeeks.org/difference-between-c-and-objective-c/>

**Smalltalk 80 ® Objective-C**

Smalltalk introduced the idea of object-oriented programming. This concept crossed with C became Objective-C. This is how Objective-C came to have Smalltalk-style messaging syntax.

Source(s):

<https://www.geeksforgeeks.org/difference-between-c-and-objective-c/>

<https://www.geeksforgeeks.org/what-is-objective-c/>

<https://medium.com/chmcore/a-short-history-of-objective-c-aff9d2bde8dd>

**Objective-C ® Swift**

* Objective-C is to dynamic typing as Swift is to static typing.
* Whereas Objective-C has only classes, Swift has structures and classes.
* Objective-C relies on the data types found in C. Swift expands its set of available data types, (including but not limited to tuples).
* Swift offers proper boolean data types (`true` and `false`) whereas Objective-C relies on indirect boolean types, such as 1 and 0.

Source(s):

<https://www.geeksforgeeks.org/difference-between-swift-vs-objective-c/>

<https://medium.com/swift-and-beyond/objective-c-vs-swift-an-exhaustive-list-of-differences-0551ef637afa>

<https://cyberogism.com/swift-vs-objective-c-ios/>

**C ® C++**

We’ve all heard it: “C++ is C with classes”. That’s the main point. A few more things C++ offers that C does not:

* Memory allocation/deallocation operators
* Reference variables
* Inheritance
* User-defined data types
* Exception handling

Source(s):

<https://www.geeksforgeeks.org/difference-between-c-and-c/>

**SIMULA 67 ® C++**

SIMULA 67 was the inspiration for “C with classes”. A goal of this derivative of C was to have the organizational structure of SIMULA 67 implemented in C, that is, with the implementation of classes and inheritance.

Source(s):

Concepts of Programming Languages, 12th Edition (pg. 86)

**C++ ® Java**

The problem with C/C++ is that programs compiled on Machine A cannot be simply transferred over to Machine B and run as expected. The program would need to be compiled on Machine B to run on Machine B. Sun Microsystems introduced Java in 1990 to address this main issue. Any machine running a Java Virtual Machine (JVM) can run the program that has been converted to Java bytecode. Another issue with C++ code is that it can get quite large, which is not ideal when running on small consumer machines.

Source(s):

<https://www.geeksforgeeks.org/why-is-java-write-once-and-run-anywhere/>

**Java ® Java 5.0**

Features new to Java 5.0:

* Generics
* Enhanced `for` loop
* Autoboxing/unboxing
* Typesafe enumerations
* Varargs
* Static importing support
* Metadata

Source(s):

<https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html#lang>

**Java 5.0 ® Java 6.0**

Features new to Java 6.0:

* Performance improvements
* Scripting language support
* Java compiler API
* New garbage collection algorithms
* New annotation features
* An assortment of Java-specific development tools

Source(s):

<https://howtodoinjava.com/series/java-versions-features/>

**Java 6.0 ® Java 7.0**

Features new to Java 7.0:

* JVM support for dynamic languages
* Timsort chosen over merge sort (“Let’s go, Timsort!”)
* APIs for graphics features
* `switch` statements offered instead of if-else statements
* Improved exception handling
* Support for emerging network protocols such as SCTP

Source(s):

<https://howtodoinjava.com/series/java-versions-features/>

**Java 7.0 ® Java 8.0**

Features new to Java 8.0:

* New date/time API
* Lambda expression support in APIs
* Streaming API
* Function interface offered with default methods
* A few niche features

Source(s):

<https://howtodoinjava.com/series/java-versions-features/>

**Java ® C#**

Features present in C# that are not present in Java:

* Structure and union support
* `goto` statement support
* Conditional compilation
* Use of pointers is an option
* Operator overloading

Source(s):

<https://www.geeksforgeeks.org/java-vs-c-sharp/>

**C++ ® C#**

* C++ is to manual memory management as C# is to automatic memory management (garbage collection
* C++ : compilation requirement on specific platform :: C# : Windows centric (with slight exceptions
* C++ : low-level :: C# : high-level
* C++ : pointers anywhere :: C# : pointers unsafe
* C++ : multiple-inheritance :: C# : no multiple inheritance
* C++ : limited objected-oriented programming :: C# : fully object-oriented programming

Source(s):

<https://www.geeksforgeeks.org/c-vs-c-sharp/>

**C# ® C# 2.0**

Features new to C# 2.0:

* Anonymous methods
* Nullable types
* Generics
* Partial types
* Iterators
* Static classes

Source(s):

<https://www.geeksforgeeks.org/c-sharp-version-history/>

**C# 2.0 ® C# 3.0**

Features new to C# 3.0:

* Anonymous types
* Lambda expressions
* Query expressions
* Partial methods
* Expression trees
* Implicitly typed local variables

Source(s):

<https://www.geeksforgeeks.org/c-sharp-version-history/>

**C# 3.0 ® C# 4.0**

Features new to C# 4.0:

* Named parameters
* Optional parameters
* Dynamic binding

Source(s):

<https://www.geeksforgeeks.org/c-sharp-version-history/>

**C# 4.0 ® C# 5.0**

Features new to C# 5.0:

* Asynchronous programming features
* Covariance/contravariance typing

Source(s):

<https://www.geeksforgeeks.org/c-sharp-version-history/>

**C ® ANSI C (C89)**

Because so many people jumped into the “Let’s create a C compiler game”, there arose compatibility issues across the industry. ANSI C (or C89) was issued by the American National Standards Institue in 1989 as an effort to standardize development of C code. Some key features:

* The C standard library
* Standardized functions for memory allocation/deallocation
  + `malloc`, `calloc`, `free`, `realloc`
* Enhanced support for input/output
* `void` type
* `enum` type
* Function prototyping

Source(s):

<https://www.geeksforgeeks.org/ansi-c-c89-standard/>

**C++ ® ANSI C (C89)**

This link (from pg. 35 of the text) is misleading. Although C89 put out the call to pull the best of C’s derivatives into one standard, C++’s contribution to the standard of the day was not until 1999 with the formalization of C99. The contribution was the double forward slash `//` to indicate comment lines. For the sake of the assignment, let’s say C++ was waiting for the next version of C89 to contribute to the standard.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 35, 76)

**ANSI C (C89) ® C99**

C99 is C89 updated for the year 1999. Some features standardized:

* Complex data types
* Boolean data type
* Double forward slash `//` as the indicator of a comment line

Source(s):

Concepts of Programming Languages, 12th Edition (pg. 76)

**SNOBOL**

SNOBOL is a dynamically typed language with dynamic storage allocation. It was developed in the early 1960s at Bell Laboratories to support text processing and the creation of text editors. Its key feature is a set of powerful string pattern matching operations.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 69, 70)

**SNOBOL ® Icon**

Icon shares many of the capabilities and objectives of SNOBOL, but generalized facilities the facilities of SNOBOL and improved the readability of the syntax. Icon has greater graphical support.

Source(s):

<https://www2.cs.arizona.edu/icon/uguide/faq.htm>

<https://www2.cs.arizona.edu/icon/docs/chump.htm>

**SNOBOL ® AWK**

AWK needed pattern matching and string manipulation capabilities. SNOBOL had these capabilities. Hence, awk ripped these ideas from SNOBOL.

Source(s):

<https://debugstory.com/the-historical-significance-of-snobol-in-pattern-matching/>

**C ® AWK**

C is arguably the most fundamental programming language. As with many languages, AWK’s design was influenced by C. Brian Kernighan, the creator of AWK, coauthored the first book on the C language with Dennis Ritchie, the creator of C. It's safe to say, they were sharing notes on how a programming language should look, act, and feel.

Source(s):

<https://www.gnu.org/software/gawk/manual/html_node/Basic-Data-Typing.html>

**AWK ® Perl**

Perl needs pattern matching and string manipulation capabilities. AWK is already good for these purposes. Hence, Perl utilizes AWK for pattern matching and string manipulation.

Source(s):

<https://commandhunt.com/use-awk-in-perl/>

<https://www.fosslife.org/awk-power-and-promise-40-year-old-language>

**Perl ® PHP**

Advantages of PHP over Perl:

* Can be inserted into HTML
* Requires less setup and code
* Is simpler to utilize

Source(s):

<https://www.geeksforgeeks.org/what-is-the-difference-between-perl-and-php/>

**Perl ® Javascript**

With the advent of the Internet, a language was needed to interface with it. A very popular choice for this just happened to be Perl. The connection between Perl and JavaScript is merely that JavaScript was better suited for this purpose and eventually replaced Perl as the language of choice for Internet-based applications.

Source(s):

Concepts of Programming Languages, 12th Edition (pgs. 93-95)