7. A few notes on computer history, C++ and other programming languages

CS138 Winter 2017

Prof. Mike Godfrey
University of Waterloo

A short history of programming languages

- **1940s-50s:**
  - Programming is done directly in machine language and (later) assembly language

- **1950s:**
  - First "high level languages": FORTRAN, LISP, and COBOL
    - All are still in common use!
    - FORTRAN designed for science, LISP (Scheme/Racket) for CS, COBOL for business ("data processing")

- **1960s:**
  - First attempt to define clean, abstract imperative languages
    - esp. Algol, which has many descendants (Pascal, Turing)
  - First object-oriented language: Simula67 (for simulations)

Machine and assembly language

- The CPU of your computer (cell phone, toaster, etc.) has a set of very low level instructions that it knows how to execute directly, e.g.:
  - Load/store contents of memory address to/from a register
  - Perform arithmetic operations on values in registers
  - Skip (branch) to another place in the current program, ...

- Each family of CPUs (Intel x86, ARM) has its own instruction set (aka *machine language*)
  - *Assembly language* is a slightly easier-to-use abstraction of a given machine language; it is closely tied to the particular machine architecture e.g., Intel x86-64 assembly language

Motorola 6800 assembly language example

[Wikipedia]
Assembly language

- Programming directly in assembly language is possible, but very tedious and error prone
  - Usually, we do this only in special cases, like writing parts of device drivers where it's important for efficiency / memory footprint reasons to get things just so

- Instead, software developers usually write in a high-level language like C/C++, Java, Python, etc. and use a compiler or interpreter to automatically generate machine language instead
  - Java (and C#) additionally uses a technique called a virtual machine that makes it easier to write high level source code that, in principle, performs identically on any platform

A short history of programming languages

- 1940s-50s:
  - Programming is done directly in machine language and (later) assembly language

- 1950s:
  - First "high level languages": FORTRAN, COBOL, and LISP
    - All are still in common use!
    - FORTRAN designed for scientific calculations, LISP (Scheme/Racket) for CS, COBOL for business ("data processing")

- 1960s:
  - First attempt to define clean, abstract imperative languages
    - esp. Algol, which has many descendents (Pascal, Turing)
  - First object-oriented language: Simula67 (for simulations)

FORTRAN

- FORTRAN == (Mathematical) FORMula TRANslation
  - First "high-level programming language"
  - John Backus, 1957, IBM
    - "Much of my work has come from being lazy. I didn't like writing low-level programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories [in assembly language], I started work on a programming system to make it easier to write programs."
  - Intended for scientific computations, now still very widely used in the scientific and engineering community to do floating-point ("real") calculation
  - Language features:
    - Procedural/imperative style: DO-loops, IF-stmts, arithmetic, ints, floats, arrays
    - No {} blocks, use GOTOs instead (in early versions)

Example code - FORTRAN IV or 66

```
C THE TRK ALGORITHM
C FORTRAN IV STYLE
DIMENSION A(11)
FUN(T) = SQRT(ABS(T) + 5.)*T**3
READ (5,1) A
1 FORMAT(5F10.2)
DO 10 J = 1, 11
     I = 11 - J
     Y = FUN(A(I+1))
     IF (400.0-Y) 4, 8, 8
     WRITE (6,5) I
5    FORMAT(110, 'too large')
     GO TO 10
8    WRITE (6,9) I, Y
     FORMAT(110, F12.6)
10   CONTINUE
C STOP
C END
```

[British Computer Society]
Fortran punch card for the single statement:  

\[ Z(1) = Y + W(1) \]

All photos from Wikimedia Commons, used under Creative Commons license except as noted

http://www.w3.org/2010/Talks/01-08-steven-ten-euro-computer/
My high school computer: IBM 1130 mainframe


Commodore Pet: The box that changed my life (1977 model)

http://www.commodorecomputerclub.com/commodore-computer-club-101010-nerd-holiday-party/

Gaming in 1980: Star Trek on an Apple ][

http://www.theregister.co.uk/2013/05/03/antique_code_show_star_trek/
Punch card for Hollerith tabulating machine for storing US census data; Hollerith's company later morphed into IBM.

Hollerith keypunch [photo likely staged]

Gavioli fairground organ, which used punch cards to store programmed music (c. 1896)
The Jacquard loom, which uses punch cards to store long and complex weaving patterns, first demonstrated

COBOL

- COBOL == Common Business-Oriented Language
  - Adm. Grace Hopper (US Navy), 1959
    - Annual Grace Hopper Celebration of Women in Computing Conference (GHC) is named in her honour
  - Designed for "data processing" (i.e., IT)
    - Manipulating HR data, computing payrolls, etc.
    - Banks, insurance companies, government agencies all have LOTS of old COBOL systems even today
  - Language features:
    - Procedural/imperative style: "structs", GOTOS (make logic hard to follow), code reads like (awkward, verbose) prose
    - e.g., MOVE X TO Y; ADD Y TO X GIVING Z; IF X GREATER THAN Y
Lecture 22

CS138 W17

(Review) A short history of programming languages

• 1940s-50s:
  – Programming is done directly in machine language and (later) assembly language

• 1950s:
  – First “high level languages”: FORTRAN, COBOL, and LISP
    • All are still in common use!
    • FORTRAN designed for scientific calculations, LISP (Scheme/Racket) for CS, COBOL for business (“data processing”)

• 1960s:
  – First attempt to define clean, abstract imperative languages
    • esp. Algol, which has many descendents (Pascal, Turing)
  – First object-oriented language: Simula67 (for simulations)
LISP (+ Scheme/Racket)

- **LISP == LISt Processing**
  - John McCarthy, MIT, 1958
  - Based on Lambda Calculus of Alonzo Church in the 1930s, but not "implemented" back then since there were no computers!
  - Simple, elegant language designed by a mathematician
  - ... rather than an engineer who wanted to "get stuff done"
  - Functional programming! Roughly, you provide a set of function definitions and then apply the functions to arguments
    - (print "Hello world!") → Hello world!
    - (+ 3 5) → 8
  - Language features:
    - Functional (not procedural/imperative): recursion instead of loops, function application instead of storing values in variables

LISP example (REPL)

```lisp
(defun fact (n)
  (cond
   ((= 0 n)   1)
   (t         (* n (fact (- n 1))))))

-> (fact 5)
120
-> (fact 0)
1
```

Joke from 1990

"Through some clever security hole manipulation, I have been able to break into all of the government's computers and acquire the Lisp code to SDI. Here is the last page [...] of it to prove that I actually have the code:


A short history of programming languages

- **1940s-50s:**
  - Programming is done directly in machine language and (later) assembly language

- **1950s:**
  - First "high level languages": FORTRAN, LISP, and COBOL
    - All are still in common use!
    - FORTRAN designed for science, LISP (Scheme/Racket) for CS, COBOL for business ("data processing")

- **1960s:**
  - First attempt to define clean, abstract imperative languages
    - esp. Algol, which has many descendents (Pascal, Turing)
  - First object-oriented language: Simula67 (for simulations)
A short history of programming languages

• Early 1970s, on big time-sharing computers:
  – C and Unix co-developed at AT&T Bell labs
    • Ritchie, Thompson, Kernighan, …
  – C is a breakthrough! (as is Unix)
    • Can write very efficient code directly
    • Can think of C as being fairly portable high-level assembly language
  – C is a simple but very powerful language, but:
    • No support for user-defined higher-level abstractions
    • DIY memory management; primary cause of insecure code
    • Very easy to make hard-to-find bugs

Meanwhile, late 1970s on the first "micros" (proto-PCs):
  – BASIC (designed in 1960s, but came into use in 1970s)
  – Small, simple procedural language designed:
    • to be very easy to learn
    • to run on small, cheap computers with little RAM, slow CPU
  – Usually interpreted, not compiled
  – A breakthrough in the burgeoning new micro market of the late 1970s (pre IBM-PC/Microsoft), esp. games
    • Altair, Apple ][, Commodore PET, TRS-80, etc
    • Microsoft’s first product was a BASIC interpreter written largely by Bill Gates and Paul Allen … they had to buy an OS (QDOS) from someone else when IBM wanted something to run on their new PC

• Early 1980s:
  – C++ at AT&T Bell labs [Stroustrup]
    • Create a powerful and efficient OO language
      • ... that is (mostly) backward compatible with ANSI C (unlike Objective-C)
      • Can define your own strongly typed abstractions!
      • Built on experience with other OO languages (Simula, Smalltalk)
  – C++ is a big, powerful, and very complicated language
    • It’s the Swiss army knife of OOP
    • Writing a good compiler for it is very, very hard
    • Most programmers use only a subset of the features
    • "Shoot yourself in the foot less often ... but ..."
A short history of programming languages

• Early 1990s:
  – Java at Sun Microsystems [Gosling]
  – Idea:
    • An OO language that will work “everywhere”: rich, portable RTE
    • Take best ideas of C++, but remove complicated and error prone features
    • Remove many low-level decisions from them programmer; give them to the compiler or RTE
  – Portability is achieved through a “virtual machine”, like a special OS that sits on top of the real OS
    • Need to port the VM to multiple platforms, but only once
      – ... no need to port the application code
      – Java programs should work the same on any VM
    • But performance is always a concern with virtualization, esp. in early days

• Late 1990s:
  – Microsoft .NET and C#
    • Roughly, try to do the same thing as Java, but with several years of hindsight and experience in watching Java
    • Emergence of the idea of platform / framework as the deployment target, not the raw OS and hardware
    • Successor to this idea: The web as the OS, web services, cloud, etc.

A short history of programming languages

• Industrial programming today:
  – Old days: Design a monolithic system to run on top of a desired OS on particular hardware
    • Porting to new deployment platforms takes real work!
  – These days: Design a piece of a component that interacts with a bunch of other services, platforms, and peer components across various networks and “the cloud”
    • Underlying OS is (mostly) irrelevant!
    • Web services, rich frameworks, scripting languages
    • Interaction is assumed via APIs, services, JSON, XML, http

"Scripting" languages:
JavaScript, Python, PHP, Scala, Ruby, ...

• In the old days:
  – Monolithic programs did heavy lifting
  – Co-ordination / interaction is ad hoc, often using "shell scripts" (BASH and similar) to move files around, start processes, etc.
• These days, "scripting" languages are much more powerful, support OO and modules
  – Easier to create medium-sized programs (100+ KLOC)
  • Much of the game Civilization is in Python!
  – But are typically dynamically-typed (can be hard to debug) and slow ...
  – They are popular because many "script-like" tasks can be done much more quickly and simply than a traditional HLL like C or Java
We're studying C++ cos ...

- The C/C++/Java memory model dominates all others in industry
- C++ supports full object-based and object-oriented styles of programming (as does Java)
- C++ is, roughly speaking, a superset of C, so it's less stressful for you to learn
- C++ supports lots of interesting language design features
  - Raw pointers, multiple inheritance, templates
  - Java has limited or so support for these by design

A quick word about C++ standards

- Over the years, there have been several official standard versions of the C++ language, each with its own formal definition:
  - ARM-90, C++98, C++03, C++11, C++14
- Differences include changes to the core language plus new elements in the C++ standard library
  - The Boost library project (www.boost.org) is used by the C++ community to field test new libraries; can often find high-quality, robust implementations of useful ideas there

A word about C++ compilers

- The original AT&T Unix command-line compiler was called simply "c++"; there are many other commercial compilers out there also, such as MS-Visual Studio
- g++ is an open source C++ compiler from the GCC project
  - It is "mature", high quality, and supports all of C++11
- clang++ (which sits on top of the LLVM back end) is another open source C++ compiler that is overtaking g++ in popularity
  - LLVM/clang++ is much newer and its design is more modern
  - clang++ is also high quality, and implements most if not all of C++11
  - clang++ seems to be faster to compile, produce faster object code, and give nicer error messages than g++

Open source politics

- g++/GCC
  - is entirely "free software"; it is released under the 3rd version of the GNU Public License (GPL3)
  - GPL3 requires that all "derived works" be released under the GPL3 too
  - GCC was designed to be hard to break into chunks that could be used as a part of a non-GPL project
- clang++/LLVM
  - is also "open source", but the license is different and more "permissive" of what you can do with the LLVM codebase
  - RMS would say that it is less "free"
  - LLVM was designed to be easy to take apart + build new tools around
  - Apple abandoned GCC for LLVM largely because of the licensing; they are big contributors to the LLVM project
• It’s common to design a compiler with an explicit "front-end" and "back-end"
  – The **front-end** translates source code into a set of intermediate data structures that are (usually) largely independent of both the source language (e.g., C++) and the target platform (e.g., Linux, MacOS Darwin, Win64)
  – The **back-end** takes the intermediate data structures and generates binary executables (and other binaries) that are tied to the target OS

  e.g., g++ is the C++ front-end for GCC
  clang++ is the C++ front-end for LLVM
  Visual-C++ is the C++ front-end (+IDE) for Microsoft Visual Studio

  [More about compilers coming in CS241!]

- Often, the front- and back-ends can be teased apart into separate components
  – This allows front-ends for different source languages to use a common back-end
  – Both the GCC and LLVM compiler systems support multiple source languages in this way:
    e.g., C, C++, Objective-C, Java, Ada, FORTRAN,

![Diagram of compiler architecture](image)

7. A few notes on computer history, C++ and other programming languages

CS138 Winter 2017

Prof. Mike Godfrey
University of Waterloo