CSC215
Lecture

Introduction

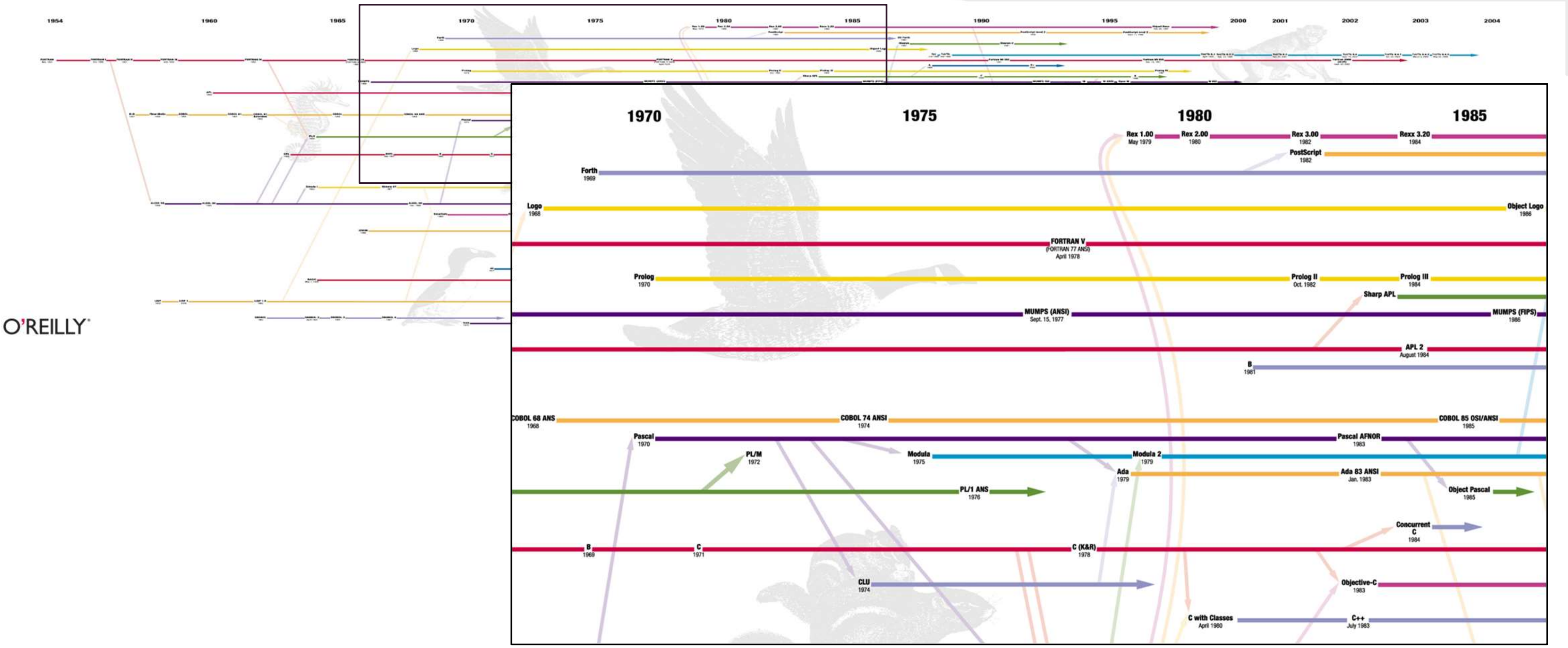
Outline

- Programming Languages
  - Object Oriented Programming
  - Procedural Programming
- What is C?
  - Short history
  - Features, Strengths and weaknesses
  - Relationships to other languages
- Writing C Programs
  - Editing
  - Compiling
- Structure of C Programs
  - Comments
  - Variables
  - Functions: main, function prototypes and functions
  - Expressions and Statements

Programming Languages

- ❑ Many programming languages exist, each intended for a specific purpose
  - ○ Over 700 programming language entries on wikipedia
  - ○ Should we learn all?
- ❑ Which is the best language? None!
- ❑ Choose the right tool for the job based on:
  - ○ problem scope,
  - ○ target hardware/software,
  - ○ memory and performance considerations,
  - ○ portability,
  - ○ concurrency.

Programming Languages



Object Oriented Programming

- ❏ Very useful to organize large software projects
- ❏ The program is organized as classes
- ❏ The data is broken into ‘objects’ and the sequence of commands becomes the interactions between objects:
  - ○ decide which classes you need
  - ○ provide a full set of operations for each class
  - ○ and make commonality explicit by using inheritance.
- ❏ Covered in CSC111 and CSC113

Procedural Programming

- The program is divided up into subroutines a.k.a procedures a.k.a functions ...
- Allows code to become structured
- The programmer must think in terms of actions:
  - decide which procedures and data structures you want
- Procedural languages include:
  - Fortran
  - BASIC
  - Pascal
  - C
  - :

Differences

- ❑ Think about:
    - ○ Basic program unit
    - ○ Design approach
    - ○ Extending functionality
    - ○ Security and visibility of program components
    - ○ Relationship to real world
    - ○ Level of abstraction
    - ○ Implementation of code reusability

What is C?

- History:
  - 1972 - Dennis Ritchie – AT&T Bell Laboratories
  - 16-bit DEC PDP-11 computer
  - 1978 - Published; first specification of language
  - **1989 - C89 standard (known as ANSI C or Standard C)**
  - 1990 - ANSI C adopted by ISO, known as C90
  - 1999 - C99 standard: mostly backward-compatible
    not completely implemented in many compilers
  - 2007 - work on new C standard C1X announced, improved in 2011 (C11)
  - 2018 - C18, few technical corrections

- In this course: ANSI/ISO C (C89/C90)

What is C?

❑ Features:

- Provides low -level access to memory
- Provides language constructs that map efficiently to machine instructions
- Few keywords (32 in ANSI C)
- Structures, unions – compound data types
- Pointers - memory, arrays
- External standard library – I/O, other facilities
- Compiles to native code
- Systems programming:
  - OSes, like Linux
  - microcontrollers: automobiles and airplanes
  - embedded processors: phones, portable electronics, etc.
  - DSP processors: digital audio and TV systems
  - . . . Macro preprocessor
- Widely used today, extends to newer system architectures

What is C?

- Strengths:
  - Efficiency: intended for applications where assembly language had traditionally been used
  - Portability: hasn’t splintered into incompatible dialects; small and easily written
  - Power: large collection of data types and operators
  - Flexibility: not only for system but also for embedded system commercial data processing
  - Standard library
  - Integration with UNIX

- Weaknesses
  - Error-prone:
    - Error detection left to the programmer
  - Difficult to understand
    - Large programs
    - Difficult to modify
  - Memory management
    - Memory management is left to the programmer

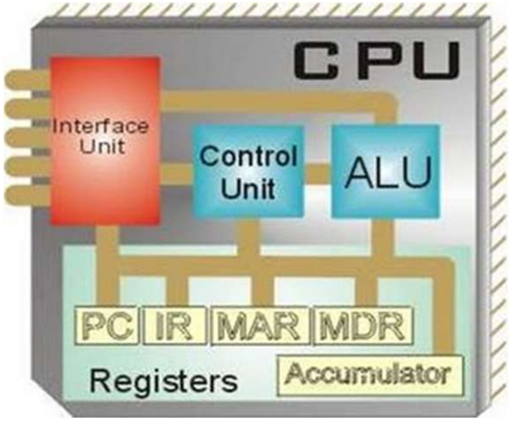
Relationship to Other Languages

- More recent derivatives: C++, Objective C, C#
- Influenced: Java, Perl, Python (quite different)
- In comparison with Java, C lacks:
  - Exceptions
  - Range-checking
  - Memory management and garbage collection.
  - Classes, objects and object-oriented programming
    - Polymorphism, encapsulation, information hiding ...

- Shares with Java:
  - /* Comments */
  - Variable declarations
  - if / else statements
  - for / while loops
  - function definitions (like methods)
  - `main` function starts program

Remember How Processor Works

The Processor



CPU: Central Processing Unit
ALU: Arithmetic and Logic Unit
PC: Program Counter
IR: Instruction Register
MAR: Memory Address Register
MDR: Memory Data Register

❖ How programs are executed:

- ○ CPU fetches instruction from memory
- ○ The instructions is decoded
- ○ If data is needed, it is loaded from memory
- ○ Instruction is executed
- ○ Results are stored back if any
- ○ PC is incremented

- Fetch Decode Execute Cycle in more detail
- https://www.hartismere.com/20398/CPU-Fetch-Decode-Execute-Animation
- http://visual6502.org/JSSim/index.html

C Programs

- Editing:
  - C source code files has .c extension
  - Text files that can be edited using any text editor: Example `product.c`

    ```
    #include <stdio.h>
    main() {
      int a, b, c;
      a = 3; b = 2; c = a * b;
      printf(“The product is %d”, c);
    }
    ```
- Compiling:
  - `gcc -o product product.c`
    - `“-o”` place the output in file product
    - `“product”` is the executable file
  - To execute the program:
    - `product` on windows or `./product` on Linux and Linux-like

C Compilers

- Several compilers
  - Microsoft compiler
  - GNU Compiler Collection (GCC)
  - : (see a List of C compilers)
- How to install GCC on windows:
  - MinGW: from https://nuwen.net/mingw.html
  - Cygwin: from https://cygwin.com/install.html
  - Don’t forget to update the PATH environment variable!
- Compilation options:
  - `gcc -ansi product.c` : check the program compatibility with ANSI C
  - `gcc -Wall product.c` : enables all the warnings that are easy to avoid
  - In this course we will always use:

    ```
    gcc -Wall -ansi -o product product.c
    ```
- Cross Compilation: compiling on one platform to run on another

Structure of .c File

```
/* Begin with comments about file contents */

/* Insert #include statements and preprocessor definitions */

/* Function prototypes and variable declarations */

/* Define main() function {
     Function body
   }
*/

/* Define other function(s) {
     Function body
   }
*/
```

Structure of .c File: Comments

- `/* this is a simple comment */`
- Can span multiple lines

```
/* This comment
   Spans
   multiple lines */
```

- Completely ignored by compiler
- Can appear almost anywhere

```
/* hello.c -
   our first C program
   Created for CSC215 */
```

Structure of .c File: #include Preprocessor

- **`#include`** is a preprocessor:
  - Header files: constants, functions, other declarations
  - `#include`: read the contents of the header file `stdio.h`
- **`stdio.h`**: standard I/O functions for console and files

```
#include <stdio.h>
/* basic I/O facilities */
```

  - `stdio.h` – part of the C Standard Library
- other important header files:

assert.h ctype.h errno.h float.h limits.h locale.h math.h
signal.h setjmp.h stdarg.h stddef.h stdlib.h string.h time.h

- Included files must be on include path
  - standard include directories assumed by default
  - #include "stdio.h" – searches ./ for stdio.h first

Structure of .c File: #Variables and Constants

- Variables: named spaces in memory that hold values
    - Refer to these spaces using their names rather than memory addresses
    - Names selection adheres to some rules
    - Defined with a type that determines their domains and operations
    - Variable must be declared prior to their use
    - Can change their values after initialization

- Constants:
    - Do not change their values after initialization
    - Can be of any basic or enumerated data type
    - Declared by assigning a literal to a typed name, with the use of the keyword `const`

      ```
      const int LENGTH = 10;
      Const char NEWLINE = '\n';
      ```

    - Can also use the `#define` preprocessor

      ```
      #define LENGTH 10
      #define NEWLINE '\n'
      ```

Structure of .c File: Function Prototype

- ❑ Functions also must be declared before use
- ❑ Function's declaration called function prototype
- ❑ Function prototypes:

```
int factorial(int);
int factorial(int n);
```

- ❑ Prototypes for many common functions in header files for C Standard Library
- ❑ General form:

```
return_type function_name(arg1type,arg2type,...);
```

- ❑ Arguments: local variables, values passed from caller
- ❑ Return value: single value returned to caller when function exits
- ❑ void – signifies no return value/arguments int rand(void);

Structure of .c File: Function main

- `main()`: entry point for C program
- Simplest version:
  - no inputs,
  - outputs 0 when successful,
  - and nonzero to signal some error int main(void);

- Two-argument form of main():
  - access command-line arguments int main(int argc, char **argv);
  - More on the `char **argv` notation later

Structure of .c File: Function Definitions

- Function declaration

```
<return_type> <function_name>(<list_of_parameters>){
  <declare_variables;>
  <program_statements;>
  return <expression>;
}
```

- Must match prototype (if there is one)
  - variable names don’t have to match
- No semicolon at end
- Curly braces define a block – region of code
  - Variables declared in a block exist only in that block
  - Variable declarations before any other statements

Structure of .c File: Expressions and statements

- ❑ Expression:
  - ○ a sequence of characters and symbols that can be evaluated to a single data item.
  - ○ consists of: literals, variables, subexpressions, interconnected by one or more *operators*
    - ■ Numeric literals like 3 or 4.5
    - ■ String literals like “Hello”
  - ○ Example expressions:
    - ■ Binary arithmetic

      ```
      x+y , x-y , x*y , x/y , x%y
      ```
- ❑ Statement:
  - ○ A sequence of characters and symbols causes the computer to carry out some definite action
  - ○ Not all statements have values
  - ○ Example statement:

    ```
    y = x+3*x/(y-4);
    ```
  - ○ Semicolon ends statement (not newline)

Console Input and Output

- stdout, stdin: console output and input streams
  - `puts(<string_expression>)`: prints string to stdout
  - `putchar(<char_expression>)`: prints character to stdout
  - `<char_var> = getchar()`: returns character from stdin
  - `<string_var> = gets(<buffer>)`: reads line from stdin into buffer
  - `printf(control_string, arg1, arg2, …)` to be discussed later

Output Statements

```
/* The main ( ) function */
int main (void)/* entry point */ {
  /* write message to console */
  puts( "Hello World!" );
  return 0; /* exit (0 => success) */
}
```

- `puts(<string>)`: output text to console window (stdout) and end the line
- String literal: written surrounded by double quotes
- return 0; exits the function, returning value 0 to caller