CSC 312

FORMAL METHODS AND SOFTWARE DEVELOPMENT

Topic: Introduction to Imperative Programming Language

DISCLAIMER

The contents of this document are intended for leaning purposes at the undergraduate level. The materials are from different sources including the internet and the contributor do not in any way claim authorship or ownership of them.

Language Vs. Paradigm

Language may refer to any of the following:

language definition1. When referring to a language that computers speaks, binary is the lowest form of language understood by computers.

2. When referring to a language a human can use to communicate with a computer, a programming language is used to communicate instructions to for a computer to perform.

3. When referring to a selection in a program, a language refers to what language the user of the computer speaks or prefers. For example, English may be an option.

Paradigm commonly refers to a new method of thinking about a problem or situation.

What is Imperative Programming?

Imperative Programming (IP) is one of the popular Programming Paradigms which executes a sequence of steps / instructions / statements in some order.

4.1: What Makes a Language Imperative?

- Programs written in imperative programming languages consist of
  - A program state
  - Instructions that change the program state
- Program instructions are “imperative” in the grammatical sense of imperative verbs that express a command

Examples of Imperative programming languages

- Ada
- ALGOL
- Assembly language
- BASIC
- Blue
- C
- C#
- C++
- COBOL
- D
- FORTRAN
- Go
- Groovy
- Java
- Modula
- MUMPS
- Nim
- Oberon
- OCaml
- Pascal
- Perl
- PHP
- PROSE
- Python
- Ruby
- Rust
- Julia
- Lua
- MATLAB

Von Neumann Machines and Imperative Programming

- Commands in an imperative language are similar to the native machine instructions of traditional computer hardware – the von Neumann-Eckley model.
- John von Neumann: first person to document the basic concepts of stored program computers.
- Von Neumann was a famous Hungarian mathematician; came to US in 1930s & became interested in computers while participating in the development of the hydrogen bomb.

The “von Neumann” Computer

- A *memory* unit: able to store both data and instructions
  - Random access
  - Internally, data and instructions are stored in the same address space & and are indistinguishable
- A *calculating unit* (the ALU)
- A *control unit,* (the CPU)
  Stored program → an instruction set
- Duality of instructions and data → programs can be self modifying
- Von Neumann outlined this structure in a document known as the “First Draft of a Report on the EDVAC”
  June, 1945

The von Neumann Computer – Historical Background

- Earlier computers had fixed programs: they were hardwired to do one thing.
- Sometimes external programs were implemented with paper tape or by setting switches.
- Eckert and Mauchly considered stored program computers as early as 1944
- During WW II they designed & built the ENIAC (although for simplicity the stored program concept was not included at first)

The von Neumann Computer – Historical Background

- Later (with von Neumann), they worked on the EDVAC
- First stored program electronic computer: the Manchester ESSM (Baby)
  - Victoria University of Manchester
  - Executed its first program June 21, 1948
- A number of other stored program machines were under development around this time

History of Imperative Languages

- First imperative languages: assembly languages
- 1954-1955: Fortran (FORmula TRANslator) John Backus developed for IBM 704
- Late 1950's: Algol (ALGOrithmic Language)
- 1958: Cobol (COmmon Business Oriented Language) Developed by a government committee; Grace Hopper very influential.

Turing Completeness

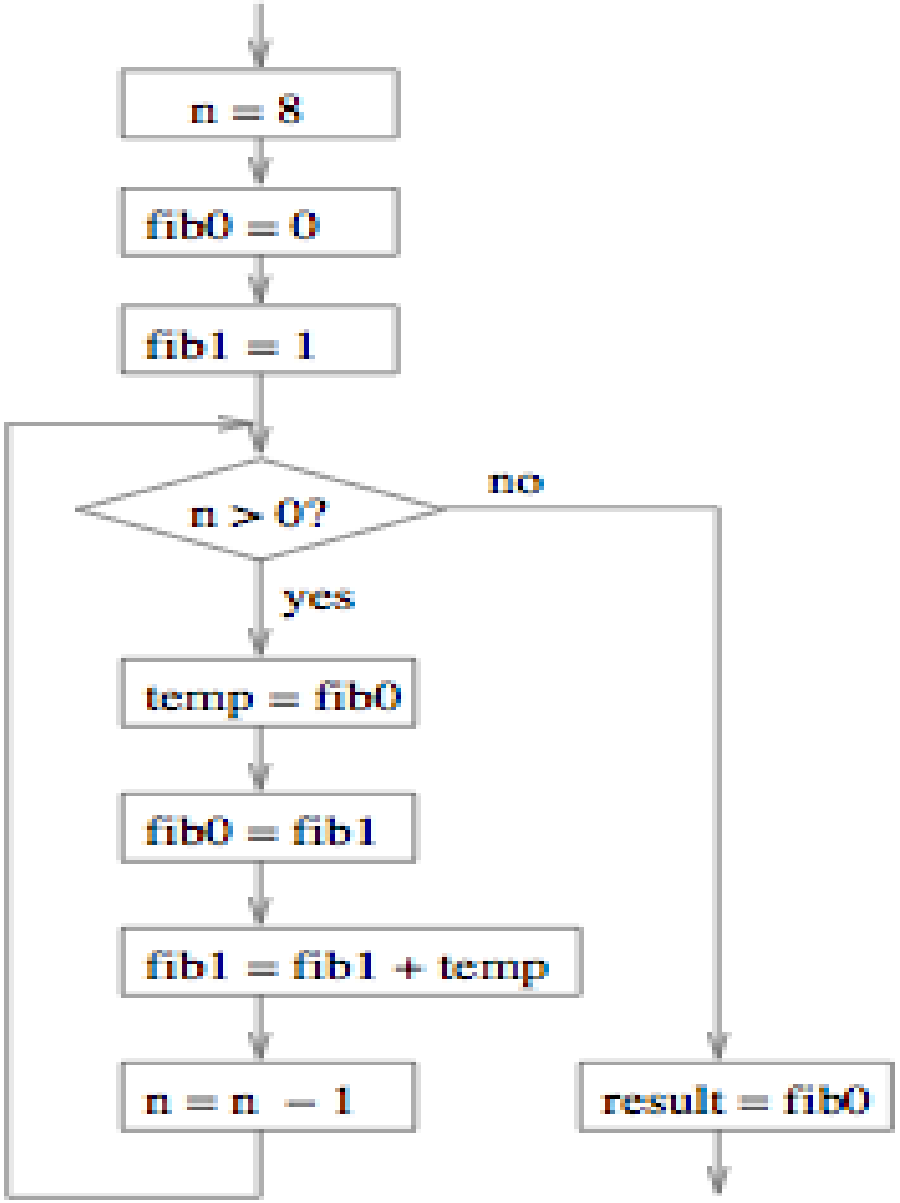
- A language is Turing complete if it can be used to implement any algorithm.
- Central to the study of computability
- Alan Turing: A British mathematician, logician, and eventually computer scientist.

Imperative Programming

- Imperative languages are Turing complete if they support integers, basic arithmetic operators, assignment, sequencing, looping and branching.
- Modern imperative languages generally also include features such as
  - Expressions and assignment
  - Control structures (loops, decisions)
  - I/O commands
  - Procedures and functions
  - Error and exception handling
  - Library support for data structures

Flowchart

- Used to model imperative programs
- Based on the three control statements that are essential to have Turing machine capability
- It's a forerunner of UML and other modern techniques
- Originated to describe process flow in general



Imperative versus Declarative Languages

- Imperative programming languages (Java, C/C++)
  - specify a sequence of operations for the computer to execute.
- Declarative languages (SQL, Haskell, Prolog)
  - describe the solution space
  - provide knowledge required to get there
  - don't describe steps needed to get there
- Functional languages and logic languages are declarative.

Imperative Language design techniques

4.2 Procedural Abstraction

- Nicholas Wirth described [imperative] programs as being “algorithms plus data structures”.
- Algorithms become programs through the process of procedural abstraction and stepwise refinement.
- Libraries of reusable functions support the process (functions = procedures)
- Imperative programming + procedures = procedural programming.

Procedural Abstraction

- Procedural abstraction allows the programmer to be concerned mainly with the interface between the function (procedure) and what it computes, ignoring the details of how the computation is accomplished.
- Abstraction allows us to think about *what* is being done, not *how* it is implemented.

Stepwise Refinement

- Stepwise refinement (also called functional decomposition) uses procedural abstraction by developing an algorithm from its most general form [the abstraction] into a specific implementation.
- Programmers start with a description of what the program should do, including I/O, and repeatedly break the problem into smaller parts, until the sub-problems can be expressed in terms of the primitive states and data types in the language.

Imperative Language design techniques.

Structured Programming

- A disciplined approach to imperative program design.
- Uses procedural abstraction and top-down design to identify program components (also called modules or structures)
- Program structures combined in a limited number of ways, so understanding how each structure works means you understand how the program works
- Program control flow is based on decisions, sequences, loops, but…
- Does not use `goto` statements
- Modules are developed, tested separately and then integrated into the whole program.

Defining Characteristics of Imperative Languages

- They contain Sequence of Statements.
- Order of execution of Statements is very important.
- They use both Immutable and Mutable Data.
- Stateful Programming Model.
- They directly change the state of Program.
- They represent state with Data Fields.
- They may have state Side-effects.

Characteristics cont…

- They are usually "typed“ either statically or dynamically.
  - Basic data types (e.g.,int, float, boolean, char)
  - Compound data types (structs, arrays).
- Statement types:
  - Declarations, Assignment, Conditionals, Loops . . .
- I/O and error handling mechanisms.
- A method of grouping all of the above into a complete program - (program composition).
  - Procedural abstraction, step-wise refinement, function mechanisms.

4.3:Expressions and Assignment

- Recall: imperative languages operate by changing program state. This is done using destructive assignment statements.
- General format:

```
target = expression
```

- Assignment operators: `=` or `:=`
- Based on machine operations such as `MOV` or `STOP`

Assignment Semantics

- Evaluate expression to get a single value
- Copy the expression value to the target.
- Pure imperative programs implement copy semantics (as opposed to the reference semantics used in object-oriented languages)

Expressions

- Expressions represent a value and have a type.
- Understanding expressions means understanding operator precedence, operator overloading, casting and type conversion, among other issues.
- Simple arithmetic expressions are based on machine language (ML) arithmetic operators (`DIV, MUL`, etc)

Expressions

- Logical operators are based on similar ML instructions (`AND, XOR`, …)
- Machine language supports data types indirectly through different instructions:
  integer `add`/`sub`/`mul`/`div`
  versus
  floating point `add`/`sub`/`mul`/`div`, for example.
- Internally, no visible way to distinguish integers from floats from characters.
  - Languages from Fortran on have provided some protection from type errors, and some ability to define new types.

Functional programming versus imperative programming

- Functional programming is programming without effects and sequencing
  - Only output from a procedure is its return value
  - Procedures behave like clauses in English (or functions in math)
  - Computation is achieved by nesting procedure calls
  - We think about execution in terms of call and response, transformation, and the other metaphors we discussed last quarter

- Imperative programming
  - Output of a procedure is its effect on the computer
  - Computation is achieved by sequencing effects
  - We think about execution in terms of changes and motion

Advantages of imperative programming

1. It is written in a step-by-step function, smaller programs written this way are very easy to follow.
2. Easy to maintain, as each procedure/function can be debugged in isolation from the rest, allowing for easy isolation of problems, in contrast to OOP which can often take very long to find the problem code.
3. Since it is written for a very specific purpose the code often gets you extremely efficient and high-performance applications.

Disadvantages include:

1. Imperative coding tends to get very difficult to maintain the larger the code gets. When the lines of code needed start ending up in the thousands it is very difficult for a team of people, or even one person to maintain.
2. Unlike in OOP, portions of the code are so interdependent that the code in one application will not be useable in another, meaning despite being somewhat similar the code for one program will not able to be carried to a new one, which OOP can do.
3. Imperative code is difficult to relate with real world objects.

Examples of imperative PL – as time permits

- C
- Ada
- Perl

4.5: Imperative Programming & C

- “C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. ... C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.” *The C Programming Language*, Kernigan & Ritchie, 1978

C: History and Influences

- Rooted in development of Multics, an advanced OS being developed at Bell Labs
- When Bell Labs pulled out of the project, Thompson and Ritchie proposed development of a simpler (hence UNIX) OS which would be platform independent.
- Initial development efforts were informal
- Platforms: minicomputers such as PDP-11.

C: History and Influences

- Multics was written in PL/1, a full-featured high-level language, rather than an assembly language, as was traditional.
  - high-level language supports platform independence.
- UNIX followed this pattern; its developers also developed the C programming language and used it for virtually all of the OS and its utilities, including the C compiler.

History and Influences

- Minicomputers during this era were 16-bit machines and had perhaps 32KB of memory
- Thus it was important to generate good, efficient code but the compiler had to be small too.
- Solution: to make C relatively low-level (closer to assembly language than other high-level languages)

History and Influences

- Many C features were adapted directly from hardware instructions
  - Examples: ++ and --
- Although C++ and Java have replaced C in many areas it is still important as a language for writing operating systems, systems with memory or power limitations, and systems that value small code and efficiency over other factors.

C – General Characteristics

- Traditional imperative components:
  - Statements: assignment; if & switch conditionals; for, while and do-while loops; function calls.
  - Data structures: arrays, pointers, structures, and unions
- Introduced: casts
- Lacks:
  - Iterators, exception processing, overloading, generics

“Interesting” Features

- Assignment as an operator, assignment expressions:
  - ```
    void strcpy(char *p, char *q) {
      while (*p++ = *q++);
    ```
- Dynamic array allocation
  - Three versions
    ```

Dynamic Array Allocation

K&R C:

```
int *a;  // size = no. of elements
. . .
a=malloc(sizeof(int) * size);
```

ANSI C:

```
a=(int*)malloc(sizeof(int) * size);
```

or, without the cast,

```
a=malloc(sizeof(int) * size);
```

C++:

```
a = new int[size];
```

Problems

- With `strcpy`: no checks to see if there is enough space in the target string, cryptic code
  - buffer overflow problems in UNIX and on the Internet
- With `malloc`: varying levels of type checking; K&R C doesn't check to see if type specified in `sizeof` matches type of a; ANSI C checks the cast, but not `sizeof`

Summary

- Advantages
  - Relatively small language
  - Can generate very efficient code
  - Runs on virtually all platforms
- Disadvantages
  - Cryptic code http://www.cs.cf.ac.uk/Dave/C/node4.html
  - Not type safe
  - Other problems http://www.andromeda.com/people/ddyer/topten.html

Ada - Background

- Development began in late 1970's by DOD
- Motivation: to save money on software by having one consistent language
- Applications:
  - Large command and control systems
  - Embedded real-time systems
- Developed by committee, competition
- Standardized in 1983

Negative Factors

- Very large – compilers started at 250K lines, compared to average of 8K-12K for Pascal, 15K to 25K for Modula.
  - PCs were just becoming popular – couldn’t host Ada compilers
- OO languages were beginning to revolutionize language design
  - Ada added OO capabilities in Ada 95 version

C/C++ v Ada

- Why are C/C++ and other languages more popular than Ada for embedded systems?
- Ada is a more secure language due to run-time error checking for various errors; e.g.,
  - Out-of-bounds array indexes
  - Buffer overflow
  - Accesses to unallocated memory
- C/C++ is smaller and faster

Current Status

- Ada fell out of favor in the 90's
- No longer mandated in DOD projects
- According to text, there is currently a resurgence of interest in the language due to
  - Unreliability of commercial, off-the-shelf sofware
  - Development of new versions of Ada
    - Spark Ada
    - NYU GNAT (Ada) compiler – part of the GNU collection

Characteristics of Imperative Ada

- Influenced by Pascal and Algol
- Large language: up to 200 production rules in the grammar
- Basic data types: character, integer, floating point, fixed point, boolean, enumeration
- Structured: arrays, strings, records, case-variant records, pointers.
- Supports subtypes and derived types.

Definition

- A derived type definition defines a new (base) type whose characteristics are derived from those of a parent type; the new type is called a *derived type*
- http://archive.adaic.com/standards/83lrm/html/lrm-03-04.html

Ada Characteristics Continued

- Case insensitive
- Unlike C/C++, array indexing errors trapped
- Type safe
- Generics
- Exception handling
- The usual imperative statements, but does not contain iterators

Features - Functions

- Functions: value-returning and otherwise
- Parameters are identified by how they are used (input, output, input-output)
  - The compiler determines which parameter passing mechanism to use (value, result, reference)
- Formal parameters may specify default values
- Function calls can specify parameter and argument; e.g.,

```
sort(list => student_array, length
      =>n);
```

Features – Packages

- Ada **packages** are a way to encapsulate functions and data into a single unit – a form of ADT similar to a class
- A package has a specification and a body
- A specification has a private and public part – it includes subprogram definitions, data declarations, constant definitions, etc.
- The body contains the implementation of the subprograms

Features – Other

- Exception handling
- Overloading
- Generics
  - Type-free subprograms
  - Type supplied when they are needed
  - Purpose: to avoid writing the same function several times
  - Introduced in Ada, adopted in other languages (C++ templates are the same idea)

Features - Generics

Generic – Fig 4.1 is the specification

type `element` is private;

type `list` is array(natural range <>) of `element`;

function ">"(`a,b: element`) return boolean;

package `sort_pck` is procedure `sort` (in out `a` : list);

end sort_pck;

package integer_sort is new generic_sort(Integer, “>”);

```
package sort_pck is
procedure sort (in out a : list) is
begin
     for i in a'first .. a'last - 1 loop
          for j in i+1 .. a'last loop
             if a(i) > a(j) then
                declare t : element;
                begin
                   t := a(i);
                   a(i) := a(j);
                   a(j) := t;
                end;
            end if;
```

Fig 4.2 is the implementation of the previous specification

Perl

- Most modern of the three languages
- Considered to be a scripting language, but is a full-featured general purpose language
  - Early on, scripting languages automated job control tasks; e.g., UNIX Bourne shell scripts
- Most scripting languages are interpreted
- Usually interpreted, Perl is sometimes compiled to a form of byte code and then interpreted

Perl

- Originally designed for text processing, now has many applications
- Supports several paradigms: imperative, object-oriented, functional
  - OO characteristics added as an afterthought

Perl

- Many different ways of saying the same thing:
  - “TIMTOWTDI”: There Is More Than One Way To Do It
- Is this feature good or bad?
  - Complicates readability; perhaps supports writability.

Purpose

- As a scripting language, used to “glue” applications together
  - take output from one application and reformat into desired input format for a different application.
  - Useful for system administration functions
- Also used for Web applications, graphics processing, & many others.
  - “The Swiss Army knife of languages”
- Comparable languages: Python, Ruby, Tcl

General Characteristics

- dynamically typed
  - types: numbers, strings, regular expressions
- implicit conversion from one type to another, based on the operators used
- result is less reliance on operator overloading
  - Separate operators for, e.g., string operations
- String concatenation operation: a period with surrounding white space

Concatenation Example

- Consider
  “abc” . “def”
  which leads to “abcdef”
- Compare to
  123 . 4.56
  which leads to “1234.56”
  - Since the period surrounded by white space means string concatenation, the operands here are converted to strings

Implicit Conversions

String vs. numeric comparisons:

```
10 < 2          # false - numeric
10 < "2"        # false
"10" lt "2"     # true - string
10 lt "2"       # true
```

< is a numeric operator; convert “2” to 2

lt is a string operator; convert 10 to “10”

Designating Types

- Scalar variables (numbers and strings) are prefixed with a dollar sign
  - $barney = ‘hello’;
  - $fred = 23;
  - $barney = $fred - 3; #give barney the value 20
- Arrays are prefixed by @:
  - @a = (2, 3, 5, 7); #0-indexed by default

Array Operations

- Based on the definition
  @a = (2, 3, 5, 7);
  the size of the array is 4, and a[3] is 7.
- What if
  $a[7] = 17;
  is executed?
- Now, the size of the array is 8, and it looks like this: 2, 3, 5, 7, undef, undef, undef, 17

What Perl Doesn’t Have

- Generics
- Exception handling
- Overloading
- User-defined iterators, but there are built in iterators:

```
foreach $rock (@rocks) {
    $rock = “\t$rock”; #put tab before each
}
```

What Perl Does Have

- Assignment operators:

  =

  +=, *=, etc.
- An append that looks like an assignment: $str .= “x”; # append “x” to end of $str
- Lists: a list is an ordered collection of scalars:
  - (1, 2, 3)
  - (“fred”, 4.5)

Features

- Strength: support for regular expressions
- Many irregularities in Perl: for example, regexps are not first class objects, meaning they can't be assigned to variables or passed as parameters.
- First class object: a language element that can be used without restriction – that has all the uses of any other element in that particular language
  - C and C++ don't permit the creation of functions at runtime, so functions aren't considered first-class objects.

Summary

- Imperative programming is the oldest programming paradigm
- It is based on the von Neumann-Eckley model of a computer
- It works by changing the program state through assignment statements
- Procedural abstraction & structured programming are its design techniques.

Assignment

Differences between Imperative Language and the following

- Functional language
- Logic language
- Object Oriented language