The Imperative Paradigm

Joseph Spring

7COM1023 Programming Paradigms
Discussion

• What makes a Language Imperative?
• Procedural Abstraction
• Expressions and Assignment
• Library Support for Data Structures
• Imperative Programming and C (Next Week)
The Imperative Paradigm

• Said to be the oldest and most well developed programming paradigm
  – Emerged with the first computers (1940’s)
  – It’s elements directly reflect the architectural characteristics of modern computers
• In mid 1940’s it was recognised (von Neumann et al) that a program and it’s data could reside in main memory – consistent with Turing’s research in 1936
  – Led to an enormous increase in the potential power and versatility of Computers
  – Previously computers had stored their programs outside of main memory using a wire and plug board
The Imperative Paradigm

• The architecture (von Neumann – Eckert Model) is the basis for the imperative paradigm

• Memory contains:
  – Program Instructions (*the Program Store*)
  – Data values (*the Data Store*)

• At the heart of this architecture:
  – Is the idea of *assignment*
    • changing the value of a memory location and destroying its previous value
      sum = sum + num;
  – All imperative languages include assignment as a central concept
The Imperative Paradigm

• In addition to the central idea of assignment, the imperative paradigm supports:
  – Variable declarations
  – Expressions
  – Conditional statements
  – Loops
  – Procedural abstraction
The Imperative Paradigm

- Variable declarations
  - Assign names to memory locations
  - Associate types with stored values

- Expressions
  - Interpreted by:
    - retrieving the current value of named variables from their respective memory locations
    - Computing a result from the above values
    - Given any reference to a variable $x$, the memory returns the current value in the location associated with $x$
The Imperative Paradigm

• Conditional statements and Loops
  – Commands normally executed in the order that they appear in memory
  – However:
    • Conditional and unconditional branching statements can interrupt this normal flow of execution
  – Due to the extensive use of branching
    • early imperative programs were often modelled using flowcharts
  – Exercise
    • Construct a flowchart to compute Fibonacci Numbers
The Imperative Paradigm

• Originally:
  – Commands in imperative languages were simple abstractions of instructions in standard von Neumann – Eckert machines

• These included:
  – Assignment statements
    • Provide ability to dynamically update value stored in memory location
  – Conditional statements and Branching statements
    • If combined allow statements to be skipped or repeatedly executed
**Turing Complete**

- Definition - An imperative language is said to be Turing Complete if it provides an effective basis for implementing any algorithm that can be designed.

- Imperative languages containing:
  - Integer variables and values
  - Basic arithmetic operations
  - Assignment statements
  - Memory based statement sequencing
  - Conditionals and Branching statements

Are said to be Turing Complete.
Turing Complete

• We Note:
  – Excessive use of the Branching statement (the *go to* statement) was considered harmful to the development of reliable programs (Dijkstra 1968)
  – It may be shown that any statement sequence in an imperative program that includes a branching statement can be written equivalently using only conditionals and while loops (Böhm-Jacopini theorem 1966)
Imperative Programming Language

• An imperative programming language is:
  – one which is Turing Complete
  – and supports certain common features that have emerged with the development of the imperative paradigm
    • Control structures
    • Input/output
    • Error and exception handling
    • Procedural abstraction
    • Expressions and assignment
    • Library support for data structures

These features appear in a variety of languages
Procedural Abstraction

• In the Imperative Programming Paradigm programs are modelled as
  – Algorithms plus Data Structures (Wirth 1976)

• Algorithms are developed into programs using two complementary ideas
  – Procedural Abstraction
  – Stepwise Refinement

• **Definition** Procedural abstraction
  – allows programmer to be concerned with the interface between functions and what they compute
  – Ignoring details of how computations are achieved
Stepwise Refinement

• **Definition** (Stepwise refinement)
  – utilises procedural abstraction by developing an algorithm from its most general form into a specific implementation

• **Example** (Sorting Function)
  – Program requires algorithm to sort an array of numbers while ignoring details of how sort is to be achieved
  – This has the advantage that any given implementation of the sorting algorithm can be replaced without affecting the original interface
Stepwise Refinement

• Example (Sorting Function - continued)
  – Interface for sort routine is of the form:

\[
\text{sort( list, len)}
\]

where \text{list} denotes the array of numbers to be sorted and \text{len} contains the number of numbers in the list

  – \textbf{First approximation} to the function:

\[
\text{for each } i \text{ in the sequence of indices of list}
\{ \\
\quad \text{list}[i] = \text{minimum element in remainder of list}
\}
\]
Stepwise Refinement

– Second approximation to the function:

for each $i$ in the sequence of indices of list

\{

\hspace{1em} for each $j > i$ in the sequence of indices of list

\hspace{2em} \{

list[i], list[j] = min, max of list[i], list[j]

\hspace{2em} \}

\hspace{1em} \}

\}
Stepwise Refinement

– Third (possible) approximation to the function:

\[
\text{for each } i \text{ in the sequence of indices of list}
\]

\[
\text{each } j > i \text{ in the sequence of indices of list}
\]

\[
\text{if list[j] < list[i]}
\]

\[
\text{swap (list[i], list[j])}
\]
Stepwise Refinement

– The third (possible) approximation is now in a form that can be easily coded in an imperative language.

– Example (C style encoding)

```c
void sort(Type list, int len){
    for ( int i = 0; i < len; i++)
        if (list[j] < list[i] ) {
            Type t = list[j];
            list[j] = list[i];
            list[i] = t;
        }
}
```
Expressions and Assignment

• Fundamental to all imperative languages
  – the assignment statement
  – General form (most popular):
    target = expression    (Fortran style)
    target := expression   (Algol style)
  – Example
    sum = sum + count;
  – Semantics (referred to as copy semantics) involves:
    • Expression is evaluated to a particular value
    • Value is copied to the Target
    • Expressions involve standard math and logic operators, with calls to standard functions provided by language library
Expressions and Assignment

• Some standard mathematical functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sine of x</td>
<td>sin(x)</td>
<td>sin(x)</td>
</tr>
<tr>
<td>Cosine of x</td>
<td>cos(x)</td>
<td>cos(x)</td>
</tr>
<tr>
<td>Tangent of x</td>
<td>tan(x)</td>
<td>tan(x)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td>Arcsine of x</td>
<td>asin(x)</td>
<td>asin(x)</td>
</tr>
<tr>
<td>$x^y$</td>
<td>X**y</td>
<td>pow(x,y)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>....</td>
</tr>
</tbody>
</table>

Note: in general the Fortran operators are a subset of the set of C operators which are a subset of the set of C++ operators
### Expressions and Assignment

- Some standard function libraries in C

<table>
<thead>
<tr>
<th>Functions Provided</th>
<th>C Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mathematical functions</td>
<td>math.h</td>
</tr>
<tr>
<td>Character classification functions</td>
<td>ctype.h</td>
</tr>
<tr>
<td>String manipulation functions</td>
<td>string.h</td>
</tr>
<tr>
<td>Utility functions</td>
<td>stdlib.h</td>
</tr>
<tr>
<td>Date and time functions</td>
<td>time.h</td>
</tr>
</tbody>
</table>
Library Support for Data Structures

• The basic data structures in the imperative paradigm are **arrays** and **records** (also known as a **structure**). The following example is from C:

```c
struct employeeType {
    int id;
    char name[26];
    int age;
    float salary;
    char dept;
};

struct employeeType employee;
```
Library Support for Data Structures

• Every field within a structure can be a different data type
• Every field name within a structure must be unique
• Assignments are achieved by statements of the form:
  
  \[ \text{employee}.age = 45; \]

  \text{employee} being a struct containing the int field age

Libraries

• Extensive libraries of useful functions are to be found in modern programming languages
• Allow programmers to reuse useful code for functions or data structures
Library Support for Data Structures

• Standard C++ library contains following main parts:
  – Functions for defining and manipulating data structures
  – Input /Output functions
  – A string class
  – A complex number class
  – Framework for fitting programs to execution environments
    • E.g. implementation details for each elementary data type on a particular architecture
  – Memory allocation and de-allocation functions
  – Exception handling functions
  – A class optimised for matrix arithmetic
Library Support for Data Structures

• Prior to the current C++ library being adopted in late 1990’s
  – The parts described on the previous slide were known as the Standard Template Library (STL)
  – This portion of the Standard C++ library contains
    • data structures and
    • functions to manipulate the data structures
  – The library may therefore be considered as being designed for imperative rather than OO programming
  – Programmers that prefer the imperative rather than the OOP can easily use the functionality offered by the standard C++ library
Library Support for Data Structures

• Main elements provided by Standard C++ Library
  – Iterators
  – Vectors
  – Lists
  – Stacks, queues, dequeues and priority queues
  – Sets and Multisets
  – Maps
  – Graphs
  – Strings
  – Complex numbers
  – Overloading
  – Generics
Library Support for Data Structures

• Some major functions used to manipulate data structures
  – Subscripting a vector
  – Resizing vectors and lists
  – Inserting and removing elements from vectors, lists and maps
  – Vector, list, set, map, graph and string searching
  – Vector and list sorting
  – List, set and map insertion and deletion
  – Queue and stack operations (push, pop, …)
  – Graph functions (shortest path, …)
  – String functions (substring, insertion, …)
  – Arithmetic, comparison, and I/O functions for complex numbers
    (See Josuttis, 1999)
Imperative Programming and C

• Next week
References


4) Josuttis, Nicolai *The Standard C++ Library*, Addison Wesley, 1999