

# COMP1200 - Perspectives on Computing

These slides were based on those of C Johnson, E McCreath and W Liang.

# Outline

- Programming Languages
- History
- Traditional Concepts
- Procedural Units
- Implementation

# History

- Programs consist of sequences of instructions encoded using digits
- Such an encoding is machine language
- Writing programs in machine language is error-prone (and tedious!)
- In the 1940's programming was simplified by the developing ways to implement instructions in mnemonic form (rather than numeric form)

4056  $\iff$  MOV R5, R6

# History

```
156C          LD R5 Price
166D          LD R6 Tax
5056         ADDI R0, R5 R6
306E         ST R0, TotalPrice
C000         HLT
```

- Descriptive names have been used to refer to memory locations
- Such descriptive names are often called *identifiers*

# History

- Use an *assembler* program to translate programs in mnemonic form into machine language
- Mnemonic system is called *Assembly Language*
- At the time, this was considered so revolutionary it was referred to as a second-generation language!
- Assembly languages do not provide the ultimate programming environment
- Assembly language and machine code differs only in syntax

# History

- An assembly language program is machine dependent
  - so cannot be transported to another computer
- Programmers are still forced to think in terms of “small steps”
- Better to think in terms of higher level primitives
- These can be translated into lower level primitives

# History

- Third-generation programming languages differed in that they consisted of higher level primitives and were machine independent
- Examples include
  - FORTRAN - FORMula TRANslator
  - COBOL - COMmon Business Oriented Language
- In general, identify a collection of high-level primitives in which software could be developed
- Each designed so that it could be executed as a sequence of low-level primitives

# History

- Example

**assign** TotalPrice **the value** Price + Tax

- Identify a collection of high-level primitives
- Use a *translator* to translate these into machine language
  - Similar to an assembler
  - Often has to compile several machine instructions into short sequences
  - Therefore referred to as *compilers*

# History

- *Interpreters* (as opposed to compilers) execute instructions as they are translated
- a program in a third-generation language can (theoretically) be used on any machine (by applying the appropriate compiler)
- In practice, minor modifications need to be made to the program in order to move it from one machine to another

# Programming Paradigms

- Programming languages have developed along different paths as alternative approaches have surfaced
- These approaches are known as *programming paradigms*
  - The imperative (procedural) paradigm
  - The declarative paradigm
  - The functional paradigm
  - The object-oriented paradigm

# Programming Paradigms

- The imperative (procedural) paradigm
  - The traditional approach to the programming process
  - Defines the programming process to be the development of a sequence of commands that, when followed, manipulate data to achieved the desired result
  - Find an algorithm to solve the problem and express that algorithm as a sequence of commands

# Programming Paradigms

- The declarative paradigm
  - Asks a programmer to describe the problem
  - Applies a pre-established, general-purpose, problem-solving algorithm to solve problems
  - Need to develop a precise statement of the problem
  - How do you design a general-purpose, problem-solving algorithm ?
  - e.g. Prolog

# Programming Paradigms

- The functional paradigm
  - A program accepts inputs and produces outputs
  - (Mathematically these are called functions)
  - Construct a program by connecting smaller, pre-defined functions
  - e.g. LISP

```
Diff(Sum(Sum(Credits) OldBalance) Sum(Debits))
```

# Programming Paradigms

- The object-oriented paradigm
  - A software system is viewed as a collection of units called *objects*
  - an objecting is capable of
    - performing actions immediately related to itself
    - requesting actions of other objects
  - Objects interact to solve the problem at hand
  - e.g.(i) developing a GUI - e.g.(ii) a list of items
  - the descriptions of the data and the methods within an object are called a *class*

# Programming Concepts

- Statements within a programming language fall into three categories
  - Declarative
    - define customised terminology such as names used to reference data items
  - Imperative
    - describe steps underlying algorithms
  - Comments
    - enhance the readability of a program

# Programming Concepts

- Memory locations are referenced by descriptive names called *variables*
- Variables must be identified by a declarative statement before they are used
- The *type* of data to be stored in a variable must also be declared
- These are known as *data types* and encompass
  - the manner in which the data is encoded
  - the operations that can be performed on that type

# Data Types

- Integers
  - Numerical data consisting of whole numbers
  - Operations consist of arithmetic operations and comparisons
  - `int Width;`
  - `int Width, Height;`
  - `int Width = 100;`

# Data Types

- Reals/Floats
  - Numerical data not consisting of whole numbers
  - Operations consist of arithmetic operations and comparisons
  - `float Width;`
  - `float Width, Height;`
  - `float Width = 100.0;`

# Data Types

- Characters
  - Data consisting of symbols
  - Operations consist of comparison, testing, concatenation
  - `char Letter;`
  - `char Letter, Digit;`
  - `char Letter = 'w';`

# Data Types

- Booleans
  - Data that can take the value True/False
  - Operations consist of enquiries
  - Bool TooBig;
  - Bool TooBig = 1;
  - **if** (TooBig) **then** (...) **else** (...)
- The data types int, char, float and bool are known as *primitive data types*

# Data Structures

- Variables are often associated with *data structures* - the conceptual shape or arrangement of data
- Text may be viewed as a long string of characters

T	h	i	s		i	s		s	o	m	e		t	e	x	t
---	---	---	---	--	---	---	--	---	---	---	---	--	---	---	---	---

- Records might be viewed as a table of integers


# Homogeneous Arrays

- Can be one-dimensional, two-dimensional, multi-dimensional
- `int MyArray [2] [9];` (2 rows and 9 columns)
- Reference cells using indices (Language dependent)

			★					

- ★ = `MyArray[1][3]` in C
- ★ = `MyArray[2][4]` in FORTRAN

# Heterogeneous Arrays

- Different elements can have different data types

```
struct {char Name[25];  
        int Age;  
        float Rating;  
    } Employee;
```

- Reference using individual components using

```
Employee.Age = 25
```

# Constants and Literals

- Sometimes fixed pre-determined values are used in programs (*literals*)

```
EffectiveAlt = Altimeter + 543
```

- Literals consisting of text are delimited using quotation marks

```
FirstName = 'John'
```

```
FirstName = John
```

- The use of literals is not good programming practice
  - makes programs difficult to understand
  - makes programs difficult to modify

# Constants and Literals

- *Constants* allow descriptive names to be assigned specific non-changeable values

```
const int AirportAlt = 645
```

```
final int AirportAlt = 645
```

- makes programs easier to understand
- makes programs easier to modify

```
EffectiveAlt = Altimeter + AirportAlt
```

# Assignment Statements

- Once the special terminology has been introduced we can begin to describe the algorithms involved
- This is done using imperative statements
- The most basic imperative statement is the *assignment statement*

$X = Y + Z$  C

$X := Y + Z$  Ada

$X \leftarrow Y + Z$  APL

# Assignment Statements

- Any algebraic expression may be used in conjunction with  $+$ ,  $-$ ,  $*$  and  $/$
- $2 * 4 + 6 / 2 = 7$  (evaluated left to right)
- $2 * 4 + 6 / 2 = 14$  (evaluated right to left)
- Rules of *operator precedence* resolves ambiguity
- $*$  and  $/$  have precedence over  $+$  and  $-$
- $2 * 4 + 6 / 2 = 11$
- Multiple use of an operating symbol is called *overloading* e.g. “comp” + “1200” gives “comp1200”

# Control Statements

- A “major villain” : goto

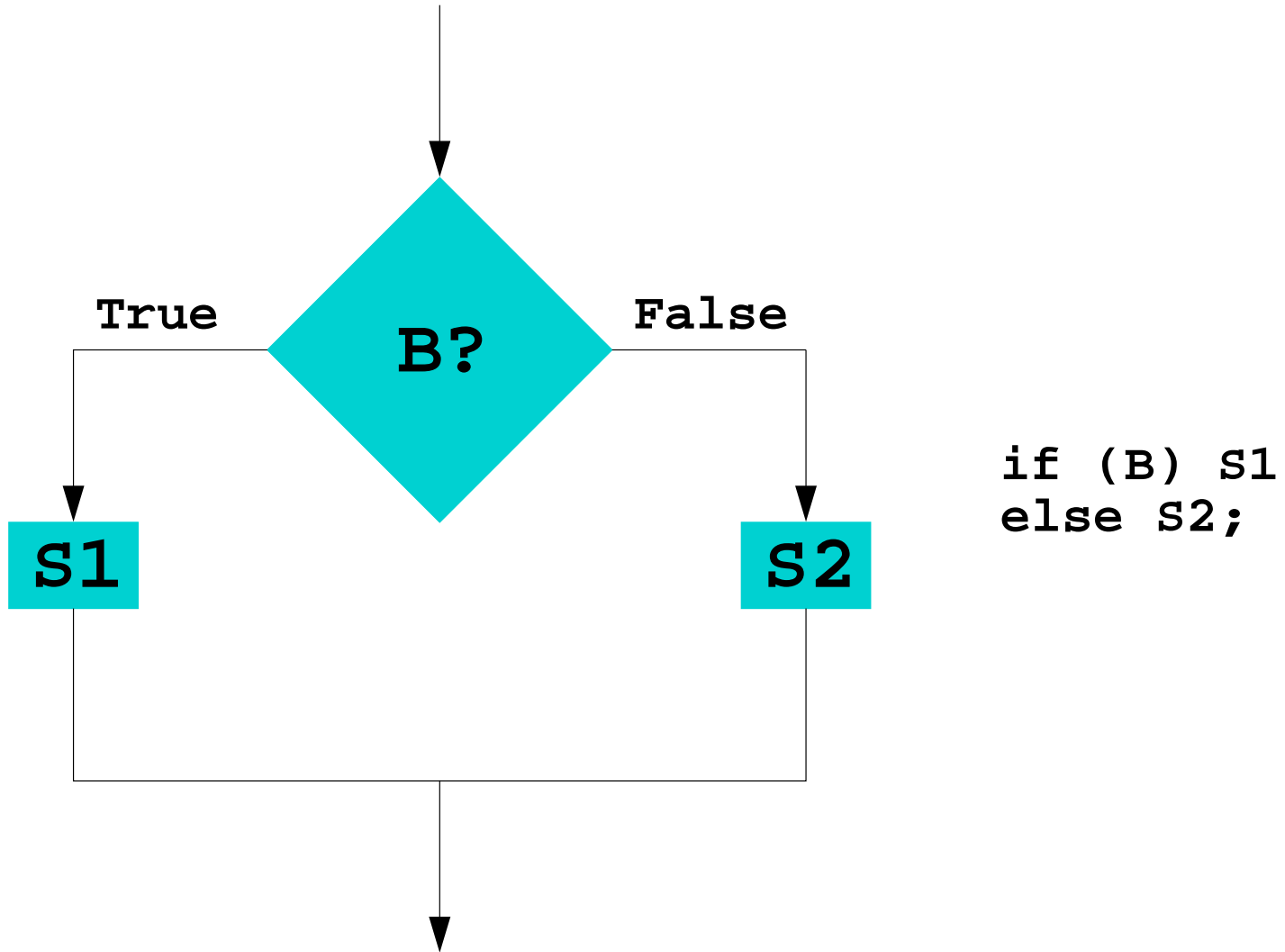
```
        goto 40
20    do something else
        goto 70
40    if (condition) then goto 60
        goto 20
60    do something
70    ...
```

**if** (condition) **then** (do something)  
**else** (do something else)

# Control Statements

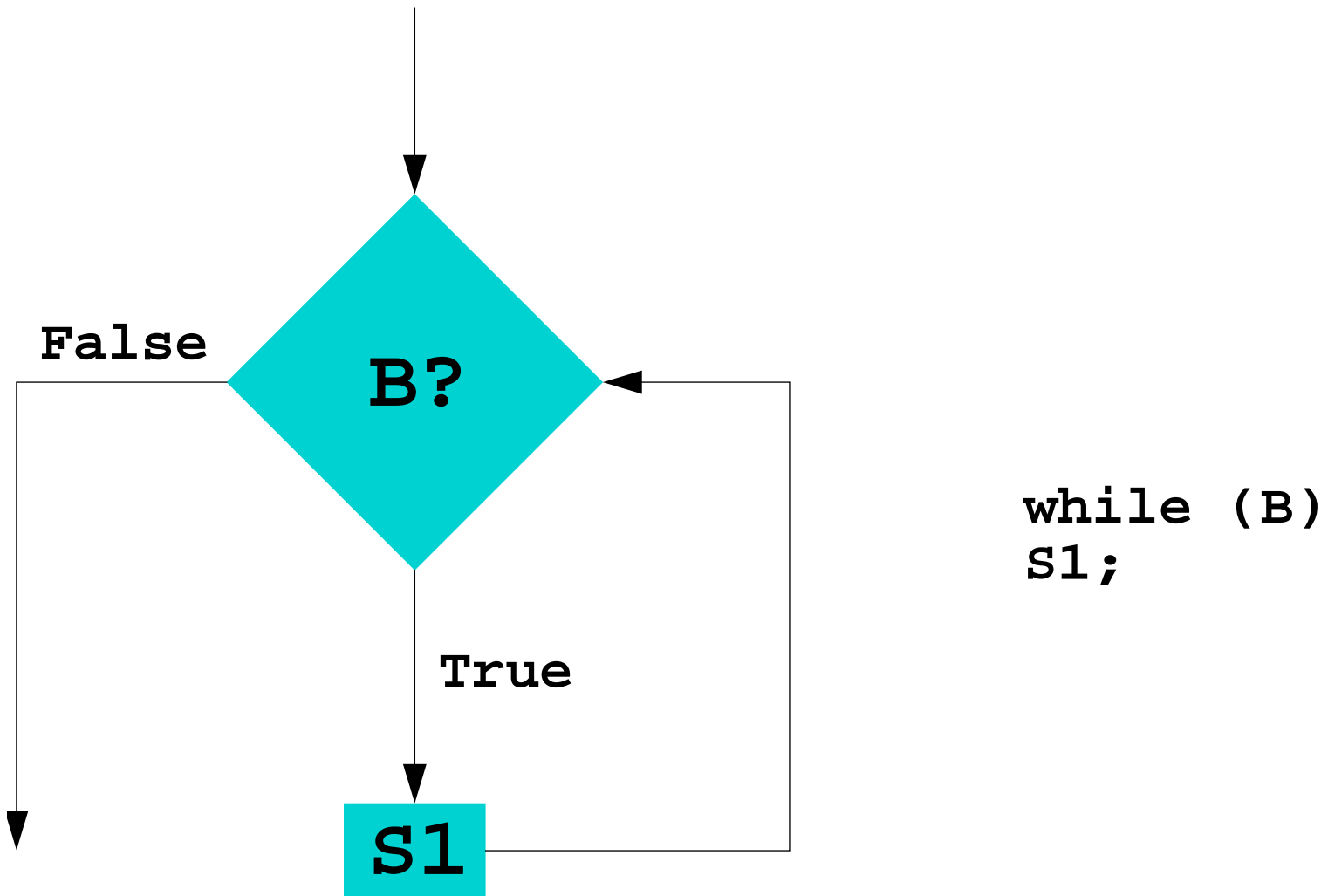
- To avoid such complexities modern languages are designed with control statements that allow an entire branching structure to be expressed as a single statement
- Aim to provide languages that allow algorithms to be expressed in a readable format
- Aim to assist programmers in attaining such readability
- The result is known as *structured programming*

# Control Structures

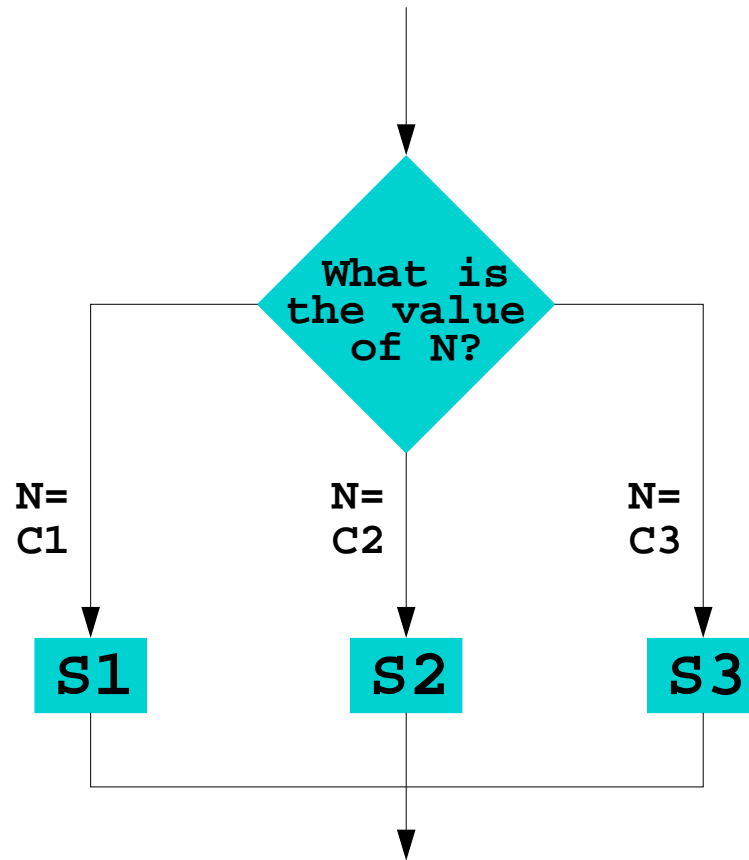


```
if (B) S1  
else S2;
```

# Control Structures

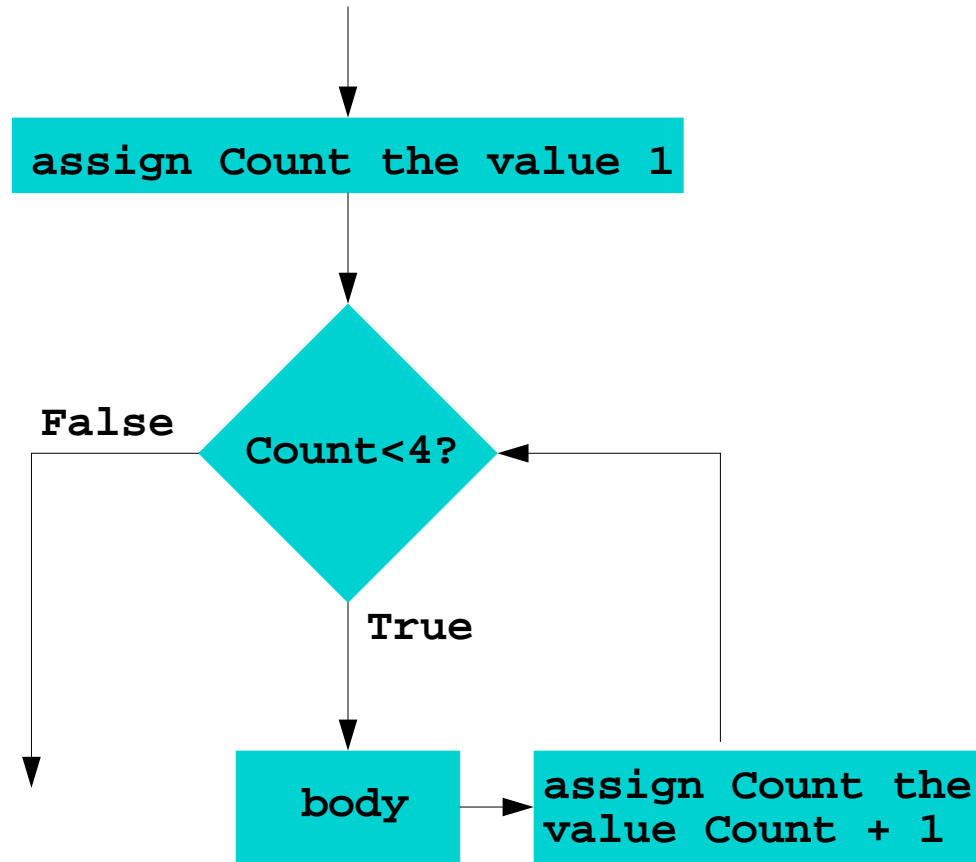


# Control Structures



```
switch (N)
{
  case C1: S1; break;
  case C2: S2; break;
  case C3: S3; break;
};
```

# Control Structures



```
for (int Count = 1; Count < 4; Count++)  
    body;
```